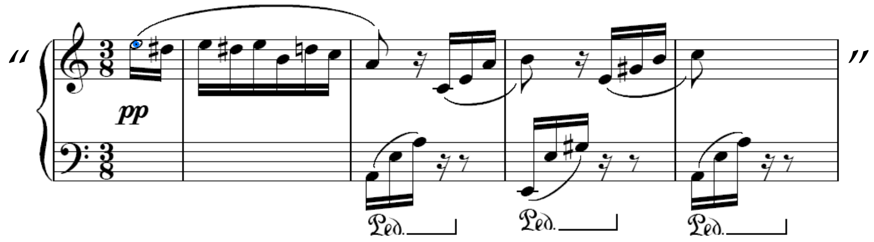
OpenVG Specification

Version 1.0.1

Version 1.0 - August 1, 2005
Amended January 26, 2007

Editor: Daniel Rice, Google, Inc.
rice@google.com



For Ilise – DSR

Table of Contents

1 Introduction....1
 1.1 Feature Set....1
 1.2 Target Applications....1
  SVG Viewers....1
  Portable Mapping Applications....1
  E-book Readers....2
  Games....2
  Scalable User Interfaces....2
  Low-Level Graphics Device Interface....2
 1.3 Target Devices....2
 1.4 Design Philosophy....2
 1.5 Naming and Typographical Conventions....3
 1.6 Library Naming....3
2 The OpenVG Pipeline....4
 2.1 Stage 1: Path, Transformation, Stroke, and Paint....5
 2.2 Stage 2: Stroked Path Generation....5
 2.3 Stage 3: Transformation....5
 2.4 Stage 4: Rasterization....5
 2.5 Stage 5: Clipping and Masking....6
 2.6 Stage 6: Paint Generation....6
 2.7 Stage 7: Image Interpolation....6
 2.8 Stage 8: Blending and Antialiasing....6
3 Constants, Functions and Data Types....7
 3.1 Versioning....7
  OPENVG_VERSION_1_0....7
 3.2 Primitive Data Types....7
  VGbyte....7
  VGubyte....7
  VGshort....7
  VGint....8
  VGuint....8
  VGbitfield....8
  VGboolean....8
  VGfloat....8
 3.3 Floating-Point and Integer Representations....8
  VG_MAXSHORT....9
  VG_MAXINT....9
  VG_MAX_FLOAT....9
 3.4 Colors....9
  3.4.1 Linear and Non-Linear Color Representations....9
  3.4.2 Color Space Definitions....10
  3.4.3 Premultiplied Alpha....12
  3.4.4 Color Format Conversion....13
 3.5 Enumerated Data Types....14
 3.6 Handle-based Data Types....15

VGHandle....15
VG_INVALID_HANDLE....15
4 The Drawing Context....16
4.1 Errors....17
VGErrorCode....18
vgGetError....18
4.2 Manipulating the Context Using EGL....18
4.2.1 EGLConfig Attributes....18
EGL_OPENVG_BIT....19
EGL_ALPHA_MASK_SIZE....19
4.2.2 EGL Functions....19
eglBindAPI....19
eglCreateContext....19
eglCreateWindowSurface....19
eglCreatePbufferFromClientBuffer....20
eglMakeCurrent....20
eglGetCurrentContext....20
eglDestroyContext....20
eglSwapBuffers....21
4.3 Forcing Drawing to Complete....21
vgFlush....21
vgFinish....21
5 Setting API Parameters....22
5.1 Context Parameter Types....22
VGParamType....22
5.2 Setting and Querying Context Parameter Values....23
vgSet....24
vgGet and vgGetVectorSize....25
5.2.1 Default Context Parameter Values....25
5.3 Setting and Querying Object Parameter Values....27
vgSetParameter....27
vgGetParameter and vgGetParameterVectorSize....28
6 Rendering Quality and Antialiasing....30
6.1 Rendering Quality....31
VGRenderingQuality....31
6.2 Additional Quality Settings....31
VGPixelLayout....31
6.3 Coordinate Systems and Transformations....32
6.4 Coordinate Systems....33
6.5 Transformations....33
6.5.1 Homogeneous Coordinates....33
6.5.2 Affine Transformations....34
6.5.3 Projective (Perspective) Transformations....35
6.6 Matrix Manipulation....36
VGMatrixMode....36
vgLoadIdentity....37
vgLoadMatrix....37
vgGetMatrix....38
vgMultMatrix....38
vgTranslate....39

vgScale....39
vgShear....39
vgRotate....40
7 Scissoring, Masking, and Clearing....41
7.1 Scissoring....41
VG_MAX_SCISSOR_RECTS....41
Specifying Scissoring Rectangles....41
7.2 Alpha Masking....42
VGMaskOperation....42
vgMask....44
7.3 Fast Clearing....45
vgClear....45
8 Paths....46
8.1 Moves....46
8.2 Straight Line Segments....46
8.3 Bézier Curves....46
8.3.1 Quadratic Bézier Curves....46
8.3.2 Cubic Bézier Curves....47
8.3.3 G1 Smooth Segments....47
8.3.4 C1 Smooth Segments....48
8.3.5 C2 Smooth Segments....49
8.3.6 Converting Segments From Quadratic to Cubic Form....49
8.4 Elliptical Arcs....49
8.5 The Standard Path Format....50
VG_PATH_FORMAT_STANDARD....50
8.5.1 Path Segment Command Side Effects....51
8.5.2 Segment Commands....52
8.5.3 Coordinate Data Formats....53
VGPathDatatype....54
8.5.4 Segment Type Marker Definitions....54
VGPathAbsRel....55
VGPathSegment....55
VGPathCommand....55
8.5.5 Client-Side Path Example....57
8.6 Path Operations....58
8.6.1 Storage of Paths....58
VGPath....59
8.6.2 Creating and Destroying Paths....59
VGPathCapabilities....59
vgCreatePath....60
vgClearPath....62
vgDestroyPath....62
8.6.3 Path Queries....62
VGPathParamType....62
Path Format....63
Path Datatype....63
Path Scale....63
Path Bias....63
Number of Segments....64
Number of Coordinates....64

8.6.4 Querying and Modifying Path Capabilities....64
vgGetPathCapabilities....64
vgRemovePathCapabilities....64
8.6.5 Copying Data Between Paths....65
vgAppendPath....65
8.6.6 Appending Client-Side Data to a Path....65
vgAppendPathData....65
8.6.7 Modifying Path Data....66
vgModifyPathCoords....67
8.6.8 Transforming a Path....68
vgTransformPath....68
8.6.9 Interpolating Between Paths....69
vgInterpolatePath....69
8.6.10 Length of a Path....70
vgPathLength....70
8.6.11 Position and Tangent Along a Path....71
The Tangents of a Path Segment....71
vgPointAlongPath....72
8.6.12 Querying the Bounding Box of a Path....73
vgPathBounds....74
vgPathTransformedBounds....75
8.7 Interpretation of Paths....75
8.7.1 Filling Paths....75
Creating Holes in Paths....77
Implicit Closure of Filled Subpaths....78
8.7.2 Stroking Paths....79
8.7.3 Stroke Parameters....80
End Cap Styles....80
Line Join Styles....81
Miter Length....82
Dashing....82
8.7.4 Stroke Generation....83
8.7.5 Setting Stroke Parameters....84
VGCapStyle....84
VGJoinStyle....85
VG_MAX_DASH_COUNT....85
Setting the Dash Pattern....86
8.7.6 Non-Scaling Strokes....86
8.8 Filling or Stroking a Path....87
VGFillRule....87
VGPaintMode....88
vgDrawPath....88
Filling a Path....88
Stroking a Path....89
Filling and Stroking a Path....89
9 Paint....90
9.1 Paint Definitions....90
VGPaint....90
9.1.1 Creating and Destroying Paint Objects....91
vgCreatePaint....91

vgDestroyPaint....91
9.1.2 Setting the Current Paint....91
vgSetPaint....91
vgGetPaint....92
9.1.3 Setting Paint Parameters....92
VGPaintParamType....92
VGPaintType....94
9.2 Color Paint....94
Setting Color Paint Parameters....94
vgSetColor....94
vgGetColor....95
9.3 Gradient Paint....96
9.3.1 Linear Gradients....96
Setting Linear Gradient Parameters....97
9.3.2 Radial Gradients....97
Setting Radial Gradient Parameters....98
9.3.3 Color Ramps....99
VG_MAX_COLOR_RAMP_STOPS....100
VGColorRampSpreadMode....100
Setting Color Ramp Parameters....101
Formal Definition of Spread Modes....102
9.3.4 Gradient Examples....103
9.4 Pattern Paint....104
vgPaintPattern....105
9.4.1 Pattern Tiling....105
VGTilingMode....105
Setting the Pattern Tiling Mode....107
10 Images....108
10.1 Image Coordinate Systems....108
10.2 Image Formats....108
VGImageFormat....108
10.3 Creating and Destroying Images....111
VGImage....111
VGImageQuality....111
VG_MAX_IMAGE_WIDTH....112
VG_MAX_IMAGE_HEIGHT....112
VG_MAX_IMAGE_PIXELS....112
VG_MAX_IMAGE_BYTES....112
vgCreateImage....113
vgDestroyImage....113
10.4 Querying Images....114
VGImageParamType....114
Image Format....114
Image Width....114
Image Height....115
10.5 Reading and Writing Image Pixels....115
vgClearImage....115
vgImageSubData....115
vgGetImageSubData....117
10.6 Child Images....118

vgChildImage....118
vgGetParent....119
10.7 Copying Pixels Between Images....119
vgCopyImage....119
10.8 Drawing Images to the Drawing Surface....120
VGImageMode....120
vgDrawImage....120
VG_DRAW_IMAGE_NORMAL....122
VG_DRAW_IMAGE_MULTIPLY....122
VG_DRAW_IMAGE_STENCIL....122
10.9 Reading and Writing Drawing Surface Pixels....123
10.9.1 Writing Drawing Surface Pixels....123
vgSetPixels....123
vgWritePixels....124
10.9.2 Reading Drawing Surface Pixels....125
vgGetPixels....125
vgReadPixels....126
10.10 Copying Portions of the Drawing Surface....128
vgCopyPixels....128
11 Image Filters....129
11.1 Format Normalization....129
11.2 Channel Masks....130
VGImageChannel....130
11.3 Color Combination....131
vgColorMatrix....131
11.4 Convolution....132
VG_MAX_KERNEL_SIZE....132
VG_MAX_SEPARABLE_KERNEL_SIZE....132
VG_MAX_GAUSSIAN_STD_DEVIATION....132
vgConvolve....133
vgSeparableConvolve....134
vgGaussianBlur....136
11.5 Lookup Tables....137
vgLookup....137
vgLookupSingle....138
12 Blending....140
12.1 Blending Equations....140
12.2 Porter-Duff Blending....140
12.3 Additional Blending Modes....141
12.4 Additive Blending....142
12.5 Setting the Blend Mode....142
VGBlendMode....142
13 Querying Hardware Capabilities....143
VGHardwareQueryType....143
VGHardwareQueryResult....143
vgHardwareQuery....143
14 Extending the API....145
14.1 Extension Naming Conventions....145
14.2 The Extension Registry....145
14.3 Using Extensions....146

14.3.1 Accessing Extensions Statically....146
14.3.2 Accessing Extensions Dynamically....146
VGStringID....146
vgGetString....146
eglGetProcAddress....147
14.4 Creating Extensions....147
15 API Conformance....148
15.1 Conformance Test Principles....148
15.1.1 Window System Independence....148
15.1.2 Antialiasing Algorithm Independence....148
15.1.3 On-Device and Off-Device Testing....148
15.2 Types of Conformance Tests....148
15.2.1 Pipeline Tests....149
15.2.2 Self-Consistency Tests....149
15.2.3 Matrix Tests....149
15.2.4 Interior/Exterior Tests....149
15.2.5 Positional Invariance....149
15.2.6 Image Comparison Tests....149
16 The VGU Utility Library....150
VGU_VERSION_1_0....150
VGUErrorCode....150
16.1 Higher-level Geometric Primitives....150
16.1.1 Lines....151
vguLine....151
16.1.2 Polylines and Polygons....151
vguPolygon....151
16.1.3 Rectangles....152
vguRect....152
16.1.4 Round-Cornered Rectangles....153
vguRoundRect....153
16.1.5 Ellipses....154
vguEllipse....154
16.1.6 Arcs....155
VGUArcType....155
vguArc....155
16.2 Image Warping....157
vguComputeWarpQuadToSquare....158
vguComputeWarpSquareToQuad....158
vguComputeWarpQuadToQuad....159
17 Appendix A: Mathematics of Ellipses....160
17.1 The Center Parameterization....160
17.2 The Endpoint Parameterization....161
17.3 Converting from Center to Endpoint Parameterization....161
17.4 Converting from Endpoint to Center Parameterization....162
17.5 Implicit Representation of an Ellipse....165
17.6 Transformation of Ellipses....166
18 Appendix B: Header Files....168
openvg.h....168
vgu.h....179
19 Bibliography....181

20 Document History........182
21 Acknowledgments........183
22 Indices........184
Index of Tables........184
Index of Figures........185
Alphabetical Index........185

1 Introduction

OpenVG is a new application programming interface (API) for hardware-accelerated two-dimensional vector and raster graphics developed under the auspices of the Khronos Group (www.khronos.org). It provides a device-independent and vendor-neutral interface for sophisticated 2D graphical applications, while allowing device manufacturers to provide hardware acceleration where appropriate.

This document defines the C language binding to OpenVG. Other language bindings may be defined by Khronos in the future. We use the term "implementation" to refer to the software and/or hardware that implements OpenVG functionality, and the term "application" to refer to any software that makes use of OpenVG.

1.1 Feature Set

OpenVG provides a drawing model similar to those of existing two-dimensional drawing APIs and formats, such as Adobe PostScript [ADOB99], PDF [ADOB06a], and (originally MacroMedia) Flash [ADOB06b]; Sun Microsystems Java2D [SUN04]; and SVG [SVGF05][SVGT06]. It is specifically intended to support all drawing features required by a SVG Tiny 1.2 renderer, and additionally to support functions that may be of use for implementing an SVG Basic renderer.

1.2 Target Applications

Several classes of target applications were used to define requirements for the design of the OpenVG API.

SVG Viewers

OpenVG must provide the drawing functionality required for a high-performance SVG document viewer that is conformant with version 1.2 of the SVG Tiny profile. It does not need to provide a one-to-one mapping between SVG syntactic features and API calls, but it must provide efficient ways of implementing all SVG Tiny features.

Portable Mapping Applications

OpenVG can provide dynamic features for map display that would be difficult or impossible to do with an SVG viewer alone, such as dynamic placement and sizing of street names and markers, and efficient viewport culling.

E-book Readers

The OpenVG API must provide fast rendering of readable text in Western, Asian, and other scripts. It does not need to provide advanced text layout features.

Games

The OpenVG API must be useful for defining sprites, backgrounds, and textures for use in both 2D and 3D games. It must be able to provide two-dimensional overlays (*e.g.*, for maps or scores) on top of 3D content.

Scalable User Interfaces

OpenVG may be used to render scalable user interfaces, particularly for applications that wish to present users with a unique look and feel that is consistent across different screen resolutions.

Low-Level Graphics Device Interface

OpenVG may be used as a low-level graphics device interface. Other graphical toolkits, such as windowing systems, may be implemented above OpenVG.

1.3 Target Devices

OpenVG is designed to run on devices ranging from wrist watches to full microprocessor-based desktop and server machines. It should be possible to implement OpenVG on any device that is capable enough to support OpenGL ES 1.1. Over time, it is expected that OpenGL ES hardware manufacturers will be able to provide inexpensive incremental acceleration for OpenVG functionality.

Realistically, to obtain the full benefit of OpenVG, a device should provide a display with at least 128 x 128 non-indexed RGB color pixels with 4 or more bits per channel.

1.4 Design Philosophy

OpenVG is intended to provide a hardware abstraction layer that will allow accelerated performance on a variety of application platforms. Functions that are not expected to be amenable to hardware acceleration in the near future were either not included, or included as part of the optional VGU utility library.

Where possible, the syntax of OpenVG is intended to be reminiscent of that of OpenGL, in order to make learning OpenVG as easy as possible for OpenGL developers. Most of the OpenVG state is encapsulated in a set of primitive-valued variables that are manipulated using the **vgSet** and **vgGet** functions. Extensions may add new state variables in order to add new features to the pipeline without needing to add new functions.

Paint, path, and image objects in OpenVG are referenced using opaque handles. This allows implementations to store such objects using their own preferred representation, in whatever form of memory they choose. This is intended to simplify hardware design, and to minimize processing and bus traffic for frequently-used objects.

1.5 Naming and Typographical Conventions

OpenVG uses a consistent set of conventions for API names and symbols. In this document, additional typographic conventions are used to help indicate the type of each symbol, as shown in Table 1 below.

Symbol Type	*Name/Case*	*Type Style*	*Example*
API Function	vgXxxYyy	Boldface	**vgLoadMatrix**
API Function with Varying Parameter Types	vgXxx{f,i,fv,iv}	Boldface	**vgSetfv**
Utility Function	vguXxxYyy	Boldface	**vguRoundRect**
Primitive Datatype	VGxxx	Typewriter	`VGfloat`
Enumerated Datatype	VGXxxYyy	Typewriter	`VGCapStyle`
Enumerated Value	VG_XXX_YYY	Typewriter	`VG_BLEND_MODE`
Utility Enumerated Value	VGU_XXX_YYY	Typewriter	`VGU_ARC_CHORD`
Function Argument	xxxYyy	Typewriter	`paintMode`

Table 1: Naming and Typographical Conventions

1.6 Library Naming

The library name is defined as `libOpenVG`.*z* where *z* is a platform-specific library suffix (*i.e.*, `.a`, `.so`, `.lib`, `.dll`, etc.).

2 The OpenVG Pipeline

This section defines the OpenVG pipeline mechanism by which primitives are rendered. Implementations are not required to match the ideal pipeline stage-for-stage; they may take any approach to rendering so long as the final results match the results of the ideal pipeline within the tolerances defined by the conformance testing process.

Figure 1 below provides an overview of the OpenVG pipeline, focusing on the various steps involved in drawing a thick, dashed line into a scene using a radial gradient paint.

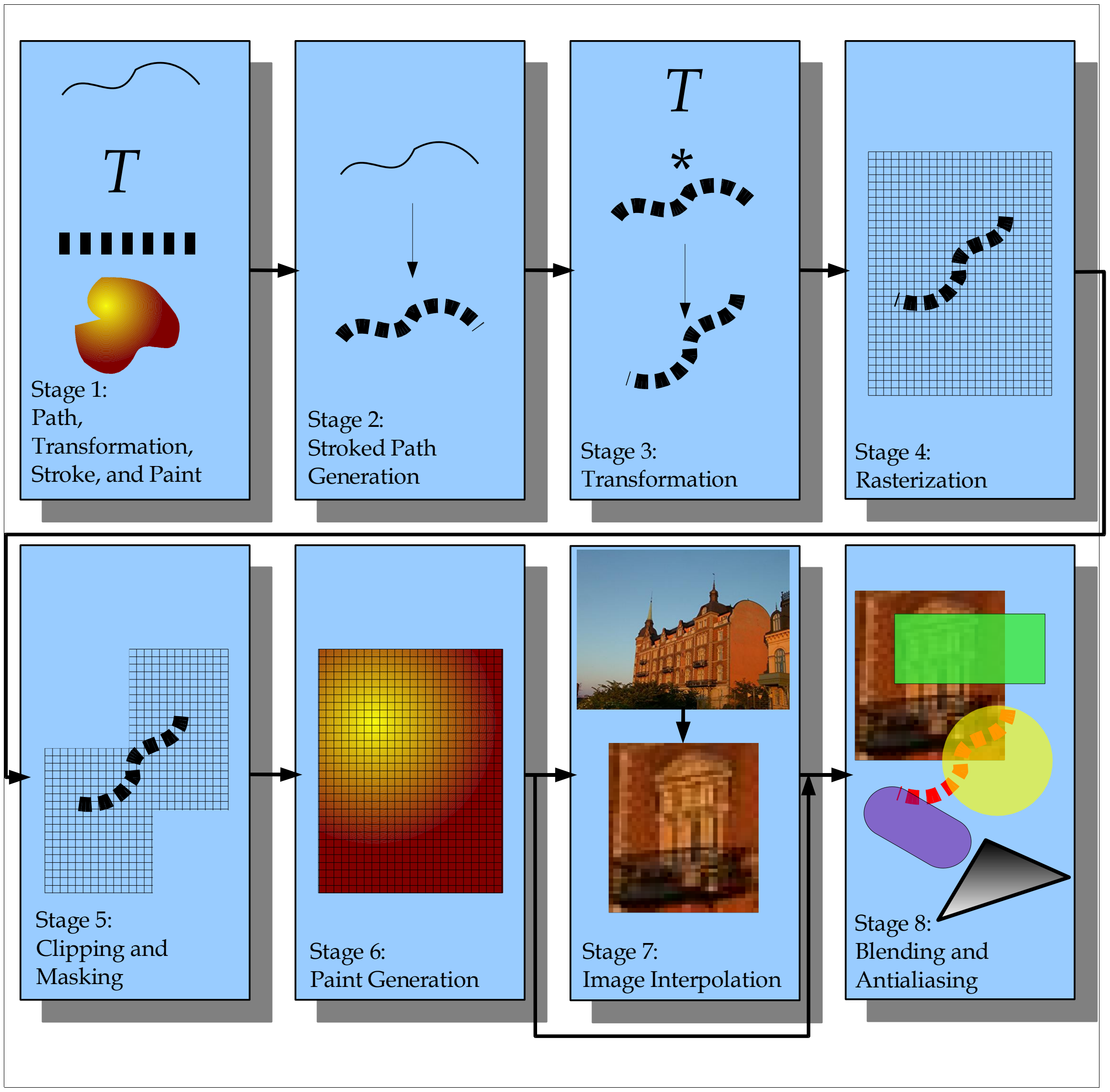


Figure 1: The OpenVG Pipeline

2.1 Stage 1: Path, Transformation, Stroke, and Paint

The application defines the path to be drawn, and sets any transformation, stroke, and paint parameters or leaves them at their default settings. When all parameters have been set, the application initiates the rendering process by calling **vgDrawPath**, indicating whether the path is to be filled, stroked, or both. If the path is to be both filled and stroked, the remainder of the pipeline is invoked twice in a serial fashion, first to fill and then to stroke the path.

If an image is being drawn (via the **vgDrawImage** function), the current path is set to a rectangle bounding the image.

2.2 Stage 2: Stroked Path Generation

If the path is to be stroked, the stroke parameters are applied in the user coordinate system to generate a new path that describes the stroked geometry. This path is then substituted for the original path in the remainder of the pipeline, and the fill rule is set to non-zero.

2.3 Stage 3: Transformation

The current path-user-to-surface transformation is applied to the geometry of the current path, producing drawing surface coordinates. For an image, the outline of the image is transformed using the image-user-to-surface transformation. Non-uniform transformations may result in skewed stroke outlines.

2.4 Stage 4: Rasterization

A coverage value is computed at pixels affected by the current path using a filtering process, and saved for use in the antialiasing step.

Conceptually, a set of sample positions are evaluated for inclusion within the path. At each pixel center that is no more than 1½ pixels away from some portion of the path geometry, a reconstruction filter is applied to the binary inclusion values at nearby sample points to obtain a filtered coverage value for the pixel. If only a single sample per pixel is evaluated, the sample position must be coincident with the pixel center.

Note that for a box filter (a filter that gives equal positive weight to all samples within a rectangle centered on the pixel center, and zero weight elsewhere), this filtering process amounts to estimating the area of the intersection of the path geometry with the filter rectangle.

If antialiasing is disabled, only pixel centers are used as sample points and the reconstruction filter has value 1 at the pixel center and 0 elsewhere.

In the case where a sample point lies exactly on the boundary of a path, the implementation must enforce a consistent “tie-breaking” rule. For any two paths that share a common boundary segment, but whose interiors lie on opposite sides of the segment, a sample point that lies exactly on the boundary must be considered to be included in exactly one of the two paths. If the interiors of the two paths lie on the same side of the common segment, the sample point must belong to both paths, or neither path. Note that the common boundary segment must be specified in exactly the same manner (*i.e.,* with bit-for-bit identical control point values, scale and bias, and transformation matrix settings) for both paths in order for this guarantee to hold.

2.5 Stage 5: Clipping and Masking

Pixels not lying within the bounds of the drawing surface, and (if scissoring is enabled) within the union of the current set of scissor rectangles are assigned a coverage value of 0.

An application-specified alpha mask image is used to modify the coverage values generated by the previous stage. Each coverage value is multiplied by the mask value for the corresponding pixel to obtain a masked coverage value. If the resulting coverage value is zero, the remainder of the pipeline is skipped.

2.6 Stage 6: Paint Generation

At each pixel of the drawing surface, the relevant current paint (depending on whether the original path was to be filled or stroked) is used to define a color and an alpha value. For gradient and pattern paints, the paint-to-user transformation is concatenated with the path-user-to-surface transformation to define the paint transformation that will geometrically transform the paint. Paint generation may be skipped for operations that do not utilize paint values.

2.7 Stage 7: Image Interpolation

If an image is being drawn, an image color and alpha value is computed at each pixel by interpolating image values using the inverse of the current image-user-to-surface transformation. The results are combined with the paint color and alpha values according to the current image drawing mode. If image drawing is not taking place, the results from the preceding stage are passed through unchanged.

2.8 Stage 8: Blending and Antialiasing

At each pixel, the source color and alpha values from the preceding stage are converted into the destination color space and blended with the corresponding destination color and alpha values according to the current blending rule. A special blending rule is used when drawing an image using the "stencil" image drawing mode. The computed coverage value from stage 5 is used to interpolate between the blended result and the previously assigned color at the pixel (preferably in a linear color space) to produce an antialiased result.

3 Constants, Functions and Data Types

OpenVG type definitions and function prototypes are found in an `openvg.h` header file, located in a `vg` subdirectory of a platform-specific header file location. OpenVG makes use of 8-, 16-, and 32-bit data types. A 64-bit data type is not required. If the `khronos_types.h` header file is provided, the primitive data types will be compatible across all Khronos APIs on the same platform.

3.1 Versioning

The `openvg.h` header file defines constants indicating the version of the specification. Future versions will continue to define the constants for all previous versions with which they are backward compatible.

OPENVG_VERSION_1_0

For the current specification, the constant `OPENVG_VERSION_1_0` is defined. The version may be queried at runtime using the **vgGetString** function (see Section 14.3.2).

```
#define OPENVG_VERSION_1_0 1
```

3.2 Primitive Data Types

OpenVG defines a number of primitive data types by means of C `typedef`s. The actual data types used are platform-specific.

VGbyte

`VGbyte` defines an 8-bit two's complement signed integer, which may contain values between -128 and 127, inclusive. If `khronos_types.h` is defined, `VGbyte` will be defined as `khronos_int8_t`.

VGubyte

`VGubyte` defines an 8-bit unsigned integer, which may contain values between 0 and 255, inclusive. If `khronos_types.h` is defined, `VGubyte` will be defined as `khronos_uint8_t`.

VGshort

`VGshort` defines a 16-bit two's complement signed integer, which may contain values between -32768 and 32767, inclusive. If `khronos_types.h` is defined, `VGshort` will be defined as `khronos_int16_t`.

VGint

VGint defines a 32-bit two's complement signed integer. If khronos_types.h is defined, VGint will be defined as khronos_int32_t.

VGuint

VGuint defines a 32-bit unsigned integer. Overflow behavior is undefined. If khronos_types.h is defined, VGuint will be defined as khronos_uint32_t.

VGbitfield

VGbitfield defines a 32-bit unsigned integer value, used for parameters that may combine a number of independent single-bit values. A VGbitfield must be able to hold at least 32 bits. If khronos_types.h is defined, VGbitfield will be defined as khronos_uint32_t.

VGboolean

VGboolean is an enumeration that only takes on the values of VG_FALSE (0) or VG_TRUE (1). Any non-zero value used as a VGboolean will be interpreted as VG_TRUE.

```
typedef enum {
  VG_FALSE = 0,
  VG_TRUE  = 1
} VGboolean;
```

VGfloat

VGfloat defines a 32-bit IEEE 754 floating-point value. If khronos_types.h is defined, VGfloat will be defined as khronos_float_t.

3.3 Floating-Point and Integer Representations

All floating-point values are specified in standard IEEE 754 format. However, implementations may clamp extremely large or small values to a restricted range, and internal processing may be performed with lesser precision. At least 16 bits of mantissa, 6 bits of exponent, and a sign bit must be present, allowing values from $\pm 2^{\pm 31}$ to be represented with a fractional precision of at least 1 in 2^{16}.

Path data (*i.e.*, vertex and control point coordinates and ellipse parameters) may be specified in one of four formats: 8-, 16-, or 32-bit signed integers, or floating-point. Floating-point scale and bias factors are used to map the incoming integer and floating-point values into a desired range when path processing occurs.

Handling of special values is as follows. Positive and negative 0 values must be treated identically. Values of +Infinity, -Infinity, or NaN (not a number) yield unspecified results. Optionally, incoming floating-point values of NaN may be treated as 0, and values of +Infinity and -Infinity may be clamped to the largest and smallest available values within the implementation, respectively. Denormalized numbers may be truncated to 0. Passing any arbitrary value as input to any floating-point argument must not lead to OpenVG interruption or termination.

VG_MAXSHORT

The macro `VG_MAXSHORT` contains the largest positive value that may be represented by a `VGshort`. `VG_MAXSHORT` is defined to be equal to $2^{15} - 1$, or 32,767. The smallest negative value that may be represented by a `VGshort` is given by (–`VG_MAXSHORT` – 1), or -32,768.

VG_MAXINT

The macro `VG_MAXINT` contains the largest positive value that may be represented by a `VGint`. `VG_MAXINT` is defined to be equal to $2^{31} - 1$, or 2,147,483,647. The smallest negative value that may be represented by a `VGint` is given by (–`VG_MAXINT` – 1), or -2,147,483,648.

VG_MAX_FLOAT

The parameter `VG_MAX_FLOAT` contains the largest floating-point number that will be accepted by an implementation. To query the parameter, use the **vgGetf** function with a `paramType` argument of `VG_MAX_FLOAT` (see Section 5.2). All implementations must define `VG_MAX_FLOAT` to be at least 10^{10}.

3.4 Colors

Colors in OpenVG other than those stored in image pixels (*e.g.*, colors for clearing, painting, and edge extension for convolution) are represented as non-premultiplied (see Section 3.4.3) sRGBA [sRGB99] color values. Image pixels may be defined in a number of color spaces, including sRGB, linear RGB, linear grayscale (or *luminance*) and non-linearly coded, perceptually-uniform grayscale, in premultiplied or non-premultiplied form. Color and alpha values lie in the range [0,1] unless otherwise noted. If an alpha channel is present but has a bit depth of zero, the alpha value of each pixel is taken to be 1.

Non-linear quantities are denoted using primed (′) symbols below. [POYN03] contains an excellent discussion of the use of non-linear coding to achieve perceptual uniformity.

3.4.1 Linear and Non-Linear Color Representations

In a linear color representation, the numeric values associated with a color channel value measure the rate at which light is emitted by an object, multiplied by some constant scale factor. Informally, it can be thought of as counting the number of photons emitted in a given amount of time. Linear representations are useful for computation, since light values may be added together in a physically meaningful way.

However, the human visual system responds non-linearly to the light power ("intensity") of an image. Accordingly, many common image coding standards (*e.g.*, the EXIF JPEG format used by many digital still cameras and the MPEG format used for video) utilize non-linear relationships between light power and code values. This allows a larger number of distinguishable colors to be represented in a given number of bits than is possible with a linear encoding. Common display devices such as CRTs and LCDs also emit light whose power at each pixel component is proportional to a non-linear *power function* (*i.e.*, a function of the form x^a where a is constant) of the applied code value, whether due to the properties of analog CRT electronics or to the deliberate application of a non-linear transfer function elsewhere in the signal path. The exponent, or *gamma*, of this power function is typically between 2.2 and 2.5. OpenVG makes use of the non-linear sRGB color specification described below.

Because linear coding of intensity fails to optimize the number of distinguishable values, 8-bit linear pixel formats suffer from poor contrast ratios and banding artifacts; their use with photographic imagery is not recommended. However, synthetic imagery generated by other APIs such as OpenGL ES that make use of linear light may require the use of linear formats. 8-bit linear coding is also appropriate for representing pseudo-images such as coverage masks that are not based on perceptual light intensity.

Although computing directly with non-linear representations may lead to significant errors compared with the results of first converting to a linear representation, it is common industry practice in many imaging domains to do so. Because the cost of performing linearization on pixel values to be interpolated or blended is considered prohibitive for mobile devices in the near future, OpenVG may perform these operations directly on non-linear code values. A future version of this specification may introduce flags to force values to be converted to a linear representation prior to interpolation and blending.

3.4.2 Color Space Definitions

The linear lRGB color space is defined in terms of the standard CIE XYZ color space [WYSZ00], following ITU Rec. 709 [ITU90] using a D65 white point:

$$R = 3.240479\ X - 1.537150\ Y - 0.498535\ Z$$

$$G = -0.969256\ X + 1.875992\ Y + 0.041556\ Z$$

$$B = 0.055648\ X - 0.204043\ Y + 1.057311\ Z$$

The sRGB color space defines values R'_{sRGB}, G'_{sRGB}, B'_{sRGB} in terms of the linear lRGB primaries by applying a gamma (γ) mapping consisting of a linear segment and an offset power function:

If $x \le 0.00304$

$\gamma(x) = 12.92\,x$

else

$\gamma(x) = 1.0556\,x^{1/2.4} - 0.0556$

The inverse mapping γ^{1} is defined as:

If $x \le 0.03928$

$\gamma^{1}(x) = x / 12.92$

else

$\gamma^{1}(x) = [(x + 0.0556) / 1.0556]^{2.4}$

To convert from lRGB to sRGB, the gamma mapping is used:

$$\begin{aligned} R'_{sRGB} &= \gamma(R) \\ G'_{sRGB} &= \gamma(G) \\ B'_{sRGB} &= \gamma(B) \end{aligned} \qquad (1)$$

To convert from sRGB to lRGB, the inverse gamma mapping is used:

$$\begin{aligned} R &= \gamma^{1}(R'_{sRGB}) \\ G &= \gamma^{1}(G'_{sRGB}) \\ B &= \gamma^{1}(B'_{sRGB}) \end{aligned} \qquad (2)$$

Because the gamma function involves offset and scaling factors, it behaves similarly to a pure power function with an exponent of 1/2.2 (or approximately 0.45) rather than the “advertised” exponent of 1/2.4, (or approximately 0.42).

The linear grayscale (luminance) color space (which we denote as lL) is related to the linear lRGB color space by the equations:

$$L = 0.2126\ R + 0.7152\ G + 0.0722\ B \qquad (3)$$

$$R = G = B = L \qquad (4)$$

The perceptually-uniform grayscale color space (which we denote as sL) is related to the linear grayscale (luminance) color space by the gamma mapping:

$$L' = \gamma(L) \qquad (5)$$

$$L = \gamma^{1}(L') \qquad (6)$$

Conversion from perceptually-uniform grayscale to sRGB is performed by replication:

$$R' = G' = B' = L' \qquad (7)$$

The remaining conversions take place in multiple steps, as shown in Table 2 below. The source format is indicated by the left column, and the destination format is indicated by the top row. The numbers indicate the equations from this section that are to be applied, in left-to-right order.

Source/Dest	***lRGB***	***sRGB***	***lL***	***sL***
lRGB	–	1	3	3,5
sRGB	2	–	2,3	2,3,5
lL	4	4,1	–	5
sL	7,2	7	6	–

Table 2: Pixel Format Conversions

3.4.3 Premultiplied Alpha

In *premultiplied alpha* (or simply *premultiplied*) formats, a pixel (R, G, B, α) is represented as ($\alpha{*}R, \alpha{*}G, \alpha{*}B, \alpha$). Alpha is always coded linearly, regardless of the color space. The terms *associated* and *premultiplied* are synonymous.

In OpenVG, color interpolation takes place in premultiplied format in order to obtain correct results for translucent pixels.

3.4.4 Color Format Conversion

Color values are converted between different formats and bit depths as follows. First, premultiplied color values are clamped to their corresponding alpha values and non-zero alpha values are divided out to obtain a non-premultiplied representation for the color.

If the source and destination color formats are of differing color spaces (*i.e.*, linear RGB, sRGB, linear grayscale, perceptually-uniform grayscale), each source channel is divided by the maximum channel value to produce a number between 0 and 1. The color space conversion is performed as described above. The resulting values are then scaled by the maximum value for each destination channel.

If the source and destination formats have the same color format, but differ in the number of bits per color channel, the source value is multiplied by the quotient $(2^d - 1)/(2^s - 1)$ (where d is the number of bits in the destination and s is the number of bits in the source) and rounded to the nearest integer.

The following approximation may be used in place of exact multiplication: If the source channel has a greater number of bits than the destination, the most significant bits are preserved and the least significant bits are discarded. If the source channel has a lesser number of bits than the destination, the value is shifted left and the most significant bits are replicated in the less significant bit positions. For example, a 5-bit source value *b4b3b2b1b0* will be converted to an 8-bit destination value *b4b3b2b1b0b4b3b2*. This rule approximates the correct result since if $d = k*s$ for some integer $k > 1$ the quotient $(2^d - 1)/(2^s - 1)$ will be an integer of the form $2^{(k-1)*s} + 2^{(k-2)*s} + \ldots + 2^s + 1$, and multiplication of an s-bit value by this value will be exactly equivalent to bit replication. When the destination bit depth is not an integer multiple of the source bit depth, this rule still provides greater accuracy than other possible approximations such as padding the source with zeros or with copies of the rightmost bit.

Note that converting from a lesser to a greater number of bits and back again using either exact scaling or the approximation will result in an unchanged value.

If the destination format has stored alpha, the previously saved alpha value is stored into the destination. If the destination format has premultiplied alpha, each color channel value is multiplied by the corresponding alpha value and the resulting values are clamped between 0 and the corresponding alpha value.

3.5 Enumerated Data Types

A number of data types are defined using the C `enum` keyword. In all cases, this specification assigns each enumerated constant a particular integer value. Extensions to the specification wishing to add new enumerated values must register with the Khronos Group to receive a unique value (see Section 14).

Applications making use of extensions should cast the extension-defined integer value to the proper enumerated type.

The enumerated types (apart from `VGboolean`) defined by OpenVG are:

- `VGBlendMode`
- `VGCapStyle`
- `VGColorRampSpreadMode`
- `VGErrorCode`
- `VGFillRule`
- `VGHardwareQueryResult`
- `VGHardwareQueryType`
- `VGImageChannel`
- `VGImageFormat`
- `VGImageMode`
- `VGImageParamType`
- `VGImageQuality`
- `VGJoinStyle`
- `VGMaskOperation`
- `VGMatrixMode`
- `VGPaintMode`
- `VGPaintParamType`
- `VGPaintType`
- `VGParamType`
- `VGPathAbsRel`
- `VGPathCapabilities`
- `VGPathCommand`
- `VGPathDatatype`
- `VGPathParamType`
- `VGPathSegment`
- `VGPixelLayout`
- `VGRenderingQuality`
- `VGStringID`
- `VGTilingMode`

The VGU utility library defines the enumerated types:

- `VGUArcType`
- `VGUErrorCode`

3.6 Handle-based Data Types

Images, paint objects, and paths are accessed using opaque *handles*. The use of handles allows these potentially large and complex objects to be stored under API control. For example, they may be stored in special memory and/or formatted in a way that is suitable for use by a hardware implementation. Handles are created relative to the current context, and may only be used as OpenVG function parameters when that context or one of its shared contexts is current.

VGHandle

Handles make use of the `VGHandle` data type. For reasons of binary compatibility between different OpenVG implementations on a given platform, a `VGHandle` is defined as a `VGuint`.

```
typedef VGuint VGHandle;
```

Handles to distinct objects must compare as unequal using the C == (double equals) operator.

The `VGHandle` subtypes defined in the API are:

- `VGImage` – a reference to image data (see Section 10)
- `VGPaint` – a reference to a paint specification (see Section 9)
- `VGPath` – a reference to path data (see Section 8)

VG_INVALID_HANDLE

The symbol `VG_INVALID_HANDLE` represents an invalid `VGHandle` that is used as an error return value from functions that return a `VGHandle`. `VG_INVALID_HANDLE` is defined as `(VGHandle)0`.

```
#define VG_INVALID_HANDLE ((VGHandle)0)
```

4 The Drawing Context

OpenVG functions that perform drawing, or that modify or query drawing state make use of an implicit *drawing context* (or simply a *context*). A context is created, attached to a drawing surface, and bound to a running application thread outside the scope of the OpenVG API, for example by the Khronos EGL API. OpenVG API calls are executed with respect to the context currently bound to the thread in which they are executed. A call to any OpenVG API function when no drawing context is bound to the current thread has no effect. The drawing context currently bound to a running thread is referred to as the *current context*.

When an image, paint, or path handle is defined, it is permanently attached to the context that is current at that time. It is an error to use the handle as an argument to any OpenVG function when a different context is active, unless that context has been designated as a *shared context* of the original context by the API responsible for context creation (usually EGL).

Images created by OpenVG may be used as the rendering target of a drawing context. All drawing performed by any API that makes use of that context, such as OpenVG or OpenGL ES, will use that image as the drawing surface.

Passing an image that is currently the rendering target of a drawing context to any OpenVG function will result in a `VG_IMAGE_IN_USE_ERROR`. The image may once again be used by OpenVG when it is no longer in use as a rendering target.

An image that shares storage with any other image (via use of the **vgChildImage** function), or that is set as a paint pattern image on a paint object, may not be used as a rendering target. The image may once again be used as a rendering target when all other images that share storage with it have been destroyed and it is no longer set as a paint pattern image on any paint object.

It is possible to provide OpenVG on a platform without supporting EGL. In this case, the host operating system must provide some alternative means of creating a context and binding it to a drawing surface and a rendering thread.

The context is responsible for maintaining the API state, as shown in Table 3.

State Element	*Description*
Drawing Surface	Surface for drawing
Matrix Mode	Transformation to be manipulated
Path user-to-surface Transformation	Affine transformation for filled and stroked geometry
Image user-to-surface Transformation	Affine or projective transformation for images

State Element	***Description***
Paint-to-user Transformation	Affine transformation for paint applied to geometry
Fill Rule	Rule for filling paths
Quality Settings	Image and rendering quality, pixel layout
Blend Mode	Pixel blend function
Image Mode	Image/paint combination function
Scissoring	Current scissoring rectangles and enable/disable
Stroke	Stroke parameters
Tile fill color	Color for FILL tiling mode
Clear color	Color for fast clear
Filter Parameters	Image filtering parameters
Paint	Paint definitions
Mask	Alpha stencil mask and enable/disable
Error	Oldest unreported error code

Table 3: State Elements of a Context

4.1 Errors

Some OpenVG functions may encounter errors. Unless otherwise specified, any value returned from a function following an error is undefined.

All OpenVG functions may signal `VG_OUT_OF_MEMORY_ERROR`. This allows implementations to defer memory allocation until it is needed, rather than requiring them to proactively allocate memory only in certain functions that are allowed to generate an error. Such an error may occur midway through the execution of an OpenVG function, in which case the function may have caused changes to the state of OpenVG or to drawing surface pixels prior to failure.

When an OpenVG function encounters an error other than a `VG_OUT_OF_MEMORY_ERROR`, the context state is not modified and no drawing takes place.

An error condition within an OpenVG function must never result in process termination, with the exception of illegal memory accesses taking place within functions that accept an application-provided pointer. Applications should take care to check return values where provided. Functions that do not provide return values may still flag errors that may be retrieved using the **vgGetError** function described below. Errors are stored in the context in which the function was called.

All pointer arguments must be aligned according to their datatype, *e.g.*, a `VGfloat *` argument must be a multiple of 4 bytes.

VGErrorCode

The error codes and their numerical values are defined by the `VGErrorCode` enumeration:

```
typedef enum {
  VG_NO_ERROR                        = 0,
  VG_BAD_HANDLE_ERROR                = 0x1000,
  VG_ILLEGAL_ARGUMENT_ERROR          = 0x1001,
  VG_OUT_OF_MEMORY_ERROR             = 0x1002,
  VG_PATH_CAPABILITY_ERROR           = 0x1003,
  VG_UNSUPPORTED_IMAGE_FORMAT_ERROR = 0x1004,
  VG_UNSUPPORTED_PATH_FORMAT_ERROR  = 0x1005,
  VG_IMAGE_IN_USE_ERROR              = 0x1006,
  VG_NO_CONTEXT_ERROR                = 0x1007
} VGErrorCode;
```

vgGetError

vgGetError returns the oldest error code provided by an API call on the current context since the previous call to **vgGetError** on that context (or since the creation of the context). No error is indicated by a return value of 0 (`VG_NO_ERROR`). After the call, the error code is cleared to 0. The possible errors that may be generated by each OpenVG function (apart from `VG_OUT_OF_MEMORY_ERROR`) are shown below the definition of the function.

If no context is current at the time **vgGetError** is called, the error code `VG_NO_CONTEXT_ERROR` is returned. Pending error codes on existing contexts are not affected by the call.

```
VGErrorCode vgGetError(void)
```

4.2 Manipulating the Context Using EGL

Most OpenVG implementations are expected to make use of version 1.2 or later of the EGL API to obtain drawing contexts. This section provides only a partial, non-normative description of some aspects of the use of EGL that are specific to OpenVG. Refer to the EGL 1.2 specification for more details.

4.2.1 EGLConfig Attributes

An `EGLConfig` describes the capabilities of a configuration. Each `EGLConfig` encapsulates a set of *attributes* and their values.

EGL_OPENVG_BIT

EGLConfigs that may be used with OpenVG will have the bit EGL_OPENVG_BIT set in their attribute EGL_RENDERABLE_TYPE attribute.

EGL_ALPHA_MASK_SIZE

The EGL_ALPHA_MASK_SIZE attribute contains the bit depth of the alpha mask associated with a configuration. Alpha masking will take place in the OpenVG pipeline only if the alpha mask bit depth for the drawing surface is greater than zero.

4.2.2 EGL Functions

eglBindAPI

EGL has a notion of the *current rendering API*. This setting acts as an implied parameter to some EGL functions. To set OpenVG as the current rendering API in EGL, it is necessary to call **eglBindAPI** with an api argument of EGL_OPENVG_API:

```
EGLBoolean eglBindAPI(EGLenum api)
```

eglCreateContext

Once **eglBindAPI** has been called to set OpenVG as the current rendering API, an EGL context that is suitable for use with OpenVG may be obtained by calling **eglCreateContext**. An existing OpenVG context may be passed in as the share_context parameter; any VGPath and VGImage objects defined in share_context will be accessible from the new context, and vice versa. If no sharing is desired, the value EGL_NO_CONTEXT should be used.

```
EGLContext eglCreateContext(EGLDisplay dpy,
                            EGLConfig config,
                            EGLContext share_context,
                            const EGLint * attrib_list)
```

eglCreateWindowSurface

Drawing takes place onto an EGLSurface. An EGLSurface may be created from a platform native window using **eglCreateWindowSurface**. It is possible to request *single-buffered* rendering, in which drawing takes place directly to the visible window, using the attrib_list parameter to set the EGL_RENDER_BUFFER attribute to a value of EGL_SINGLE_BUFFER. Implementations that do not support single-buffered rendering may ignore this setting. Applications should query the returned surface to determine if it is single- or double-buffered.

```
EGLSurface eglCreateWindowSurface(EGLDisplay dpy,
                                  EGLConfig config,
                                  NativeWindowType win,
                                  const EGLint *attrib list);
```

eglCreatePbufferFromClientBuffer

An EGLSurface that allows rendering into a `VGImage` (see Section 10) may be created by binding the `VGImage` to a *Pbuffer* (off-screen buffer). EGL defines the function **eglCreatePbufferFromClientBuffer**, which may be used with a `buftype` argument of `EGL_OPENVG_IMAGE`. The `VGImage` to be targeted is cast to the `EGLClientBuffer` type and passed as the `buffer` parameter.

If EGL is used with OpenVG, the version of EGL used must support the creation of a Pbuffer from a `VGImage` either as part of its core functionality or by means of an extension.

```
EGLSurface eglCreatePbufferFromClientBuffer(EGLDisplay dpy,
                                            EGLenum buftype,
                                            EGLClientBuffer buffer,
                                            EGLConfig config,
                                            const EGLint *attrib_list)
```

eglMakeCurrent

The **eglMakeCurrent** function causes a given context to become current on the running thread. Any context that is current on the thread prior to the call is flushed and marked as no longer current.

```
EGLBoolean eglMakeCurrent(EGLDisplay dpy,
                          EGLSurface draw,
                          EGLSurface read,
                          EGLContext ctx)
```

eglGetCurrentContext

The OpenVG context for the current rendering API that is bound to the current thread may be retrieved by calling **eglGetCurrentContext**:

```
EGLContext eglGetCurrentContext()
```

eglDestroyContext

An EGL context is destroyed by calling **eglDestroyContext**.

```
EGLBoolean eglDestroyContext(EGLDisplay display, EGLContext context)
```

eglSwapBuffers

When drawing occurs in *double-buffered* mode, all drawing takes place into an invisible back buffer, and it is necessary to call **eglSwapBuffers** to force the buffer contents to be copied to the visible window. If the visible buffer has a lesser color bit depth than the back buffer, dithering may be performed as part of the buffer copy operation.

```
EGLBoolean eglSwapBuffers(EGLDisplay dpy,
                          EGLSurface surface);
```

4.3 Forcing Drawing to Complete

OpenVG provides functions to force the completion of rendering, in order to allow applications to synchronize between multiple rendering APIs.

vgFlush

The **vgFlush** function ensures that all outstanding requests on the current context will complete in finite time. **vgFlush** may return prior to the actual completion of all requests.

```
void vgFlush(void)
```

vgFinish

The **vgFinish** function forces all outstanding requests on the current context to complete, returning only when the last request has completed.

```
void vgFinish(void)
```

5 Setting API Parameters

API parameters may be set and retrieved using generic *get* and *set* functions. The use of generic functions allows for extensibility of the API without the addition of additional functions. Extensions may receive unique identifier values for new parameter types by registering with the Khronos group.

Parameters take two forms: some are set relative to a rendering context, and others are set on a particular `VGHandle`-based object. The former make use of the **vgSet** and **vgGet** functions and the latter make use of the **vgSetParameter** and **vgGetParameter** functions.

5.1 Context Parameter Types

Parameter types that are set on a rendering context are defined in the `VGParamType` enumeration. The datatype and default value associated with each parameter is shown in Table 4.

VGParamType

The `VGParamType` enumeration defines the parameter type of the value to be set or retrieved using **vgSet** and **vgGet**:

```
typedef enum {
  /* Mode settings */
  VG_MATRIX_MODE                 = 0x1100,
  VG_FILL_RULE                   = 0x1101,
  VG_IMAGE_QUALITY               = 0x1102,
  VG_RENDERING_QUALITY           = 0x1103,
  VG_BLEND_MODE                  = 0x1104,
  VG_IMAGE_MODE                  = 0x1105,

  /* Scissoring rectangles */
  VG_SCISSOR_RECTS               = 0x1106,

  /* Stroke parameters */
  VG_STROKE_LINE_WIDTH           = 0x1110,
  VG_STROKE_CAP_STYLE            = 0x1111,
  VG_STROKE_JOIN_STYLE           = 0x1112,
  VG_STROKE_MITER_LIMIT          = 0x1113,
  VG_STROKE_DASH_PATTERN         = 0x1114,
  VG_STROKE_DASH_PHASE           = 0x1115,
  VG_STROKE_DASH_PHASE_RESET     = 0x1116,

  /* Edge fill color for VG_TILE_FILL tiling mode */
  VG_TILE_FILL_COLOR             = 0x1120,

  /* Color for vgClear */
  VG_CLEAR_COLOR                 = 0x1121,
  /* Enable/disable alpha masking and scissoring */
  VG_MASKING                      = 0x1130,
  VG_SCISSORING                   = 0x1131,

  /* Pixel layout information */
  VG_PIXEL_LAYOUT                 = 0x1140,
  VG_SCREEN_LAYOUT                = 0x1141,

  /* Source format selection for image filters */
  VG_FILTER_FORMAT_LINEAR         = 0x1150,
  VG_FILTER_FORMAT_PREMULTIPLIED  = 0x1151,

  /* Destination write enable mask for image filters */
  VG_FILTER_CHANNEL_MASK          = 0x1152,

  /* Implementation limits (read-only) */
  VG_MAX_SCISSOR_RECTS            = 0x1160,
  VG_MAX_DASH_COUNT               = 0x1161,
  VG_MAX_KERNEL_SIZE              = 0x1162,
  VG_MAX_SEPARABLE_KERNEL_SIZE    = 0x1163,
  VG_MAX_COLOR_RAMP_STOPS         = 0x1164,
  VG_MAX_IMAGE_WIDTH              = 0x1165,
  VG_MAX_IMAGE_HEIGHT             = 0x1166,
  VG_MAX_IMAGE_PIXELS             = 0x1167,
  VG_MAX_IMAGE_BYTES              = 0x1168,
  VG_MAX_FLOAT                    = 0x1169
} VGParamType;
```

5.2 Setting and Querying Context Parameter Values

Each **vgGet/vgGetParameter** or **vgSet/vgSetParameter** function has four variants, depending on the data type of the value being set, differentiated by a suffix: **i** for scalar integral values, **f** for scalar floating-point values, and **iv** and **fv** for vectors of integers and floating-point values, respectively. The vector variants may also be used to set scalar values using a `count` of 1. When setting a value of integral type using a floating-point **vgSet** variant (ending with **f** or **fv**), or retrieving a floating-point value using an integer **vgGet** function (ending with **i** or **iv**), the value is converted to an integer using a mathematical *floor* operation. If the resulting value is outside the range of integer values, the closest valid integer value is substituted.

The `count` parameter used by the array variants (ending with **iv** or **fv**) limits the number of values that are read from the `values` array parameter. For parameters that require a fixed number of values (*e.g.*, color values of type `VGfloat[4]`), `count` must have the appropriate value. For parameters that place restrictions on the number of values that may be accepted (*e.g.*, that it be a multiple of a specific number, as for scissor rectangles which are specified as a set of 4-tuples), `count` must obey the restriction. For parameters that accept an arbitrary number of values up to some maximum number (*e.g.*, dash patterns) , all `count` specified values up to the maximum are used and values beyond the maximum are ignored. If the count parameter is 0, the pointer argument is not dereferenced. For example, the call `vgSet(VG_STROKE_DASH_PATTERN, 0, (void *) 0)` sets the dash pattern to a zero-length array (which has the effect of disabling dashing) without dereferencing the third parameter. If an error occurs due to an inappropriate value of `count`, the call has no effect on the parameter value.

Certain parameter values are read-only. Calling **vgSet** or **vgSetParameter** on these values has no effect.

vgSet

The **vgSet** functions set the value of a parameter on the current context.

```
void vgSetf (VGParamType paramType, VGfloat value)
void vgSeti (VGParamType paramType, VGint value)
void vgSetfv(VGParamType paramType, VGint count,
             const VGfloat * values)
void vgSetiv(VGParamType paramType, VGint count,
             const VGint * values)
```

ERRORS

`VG_ILLEGAL_ARGUMENT_ERROR`

- if `paramType` is not a valid value from the `VGParamType` enumeration
- if `paramType` refers to a vector parameter in **vgSetf** or **vgSeti**
- if `paramType` refers to a scalar parameter in **vgSetfv** or **vgSetiv** and `count` is not equal to 1
- if `value` is not a legal enumerated value for the given parameter in **vgSetf** or **vgSeti**, or if `values[i]` is not a legal enumerated value for the given parameter in **vgSetfv** or **vgSetiv** for 0 ≤ i < `count`
- if `values` is `NULL` in **vgSetfv** or **vgSetiv** and `count` is greater than 0
- if `values` is not properly aligned in **vgSetfv** or **vgSetiv**
- if `count` is less than 0 in **vgSetfv** or **vgSetiv**
- if `count` is not a valid value for the given parameter

For example, to set the blending mode to the integral value `VG_BLEND_SRC_OVER` (see Section 12.5), the application would call:

```
vgSeti(VG_BLEND_MODE, VG_BLEND_SRC_OVER);
```

vgGet and vgGetVectorSize

The **vgGet** functions return the value of a parameter on the current context.

The **vgGetVectorSize** function returns the maximum number of elements in the vector that will be retrieved by the **vgGetiv** or **vgGetfv** functions if called with the given `paramType` argument. For scalar values, 1 is returned. If **vgGetiv** or **vgGetfv** is called with a smaller value for `count` than that returned by **vgGetVectorSize**, only the first `count` elements of the vector are retrieved. Use of a greater value for `count` will result in an error.

The original value passed to **vgSet** (except as specifically noted, and provided the call to **vgSet** completed without error) is returned by **vgGet**, even if the implementation makes use of a truncated or quantized value internally. This rule ensures that OpenVG state may be saved and restored without degradation.

If an error occurs during a call to **vgGetf**, **vgGeti**, or **vgGetVectorSize**, the return value is undefined. If an error occurs during a call to **vgGetfv** or **vgGetiv**, nothing is written to `values`.

```
VGfloat vgGetf (VGParamType paramType)
VGint   vgGeti (VGParamType paramType)

VGint   vgGetVectorSize(VGParamType paramType)

void    vgGetfv(VGParamType paramType, VGint count, VGfloat * values)
void    vgGetiv(VGParamType paramType, VGint count, VGint * values)
```

ERRORS

VG_ILLEGAL_ARGUMENT_ERROR

- if `paramType` is not a valid value from the `VGParamType` enumeration
- if `paramType` refers to a vector parameter in **vgGetf** or **vgGeti**
- if `values` is `NULL` in **vgGetfv** or **vgGetiv**
- if `values` is not properly aligned in **vgGetfv** or **vgGetiv**
- if `count` is less than or equal to 0 in **vgGetfv** or **vgGetiv**
- if `count` is greater than the value returned by **vgGetVectorSize** for the given parameter in **vgGetfv** or **vgGetiv**

5.2.1 Default Context Parameter Values

When a new OpenVG context is created, it contains default values as shown in Table 4. Note that some tokens have been split across lines for reasons of space.

Parameter	***Datatype***	***Default Value***
`VG_MATRIX_MODE`	`VGMatrixMode`	`VG_MATRIX_PATH_USER_` `TO_SURFACE`
`VG_FILL_RULE`	`VGFillRule`	`VG_EVEN_ODD`
`VG_IMAGE_QUALITY`	`VGImageQuality`	`VG_IMAGE_QUALITY_FASTER`
`VG_RENDERING_QUALITY`	`VGRendering` `Quality`	`VG_RENDERING_QUALITY_BETTER`
`VG_BLEND_MODE`	`VGBlendMode`	`VG_BLEND_SRC_OVER`
`VG_IMAGE_MODE`	`VGImageMode`	`VG_DRAW_IMAGE_NORMAL`
`VG_SCISSOR_RECTS`	`VGint *`	{ } (array of length 0)
`VG_STROKE_LINE_WIDTH`	`VGfloat`	1.0f
`VG_STROKE_CAP_STYLE`	`VGCapStyle`	`VG_CAP_BUTT`
`VG_STROKE_JOIN_STYLE`	`VGJoinStyle`	`VG_JOIN_MITER`
`VG_STROKE_MITER_LIMIT`	`VGfloat`	4.0f
`VG_STROKE_DASH_PATTERN`	`VGfloat *`	{ } (array of length 0) (disabled)
`VG_STROKE_DASH_PHASE`	`VGfloat`	0.0f
`VG_STROKE_DASH_PHASE_` `RESET`	`VGboolean`	`VG_FALSE` (disabled)
`VG_TILE_FILL_COLOR`	`VGfloat[4]`	{ 0.0f, 0.0f, 0.0f, 0.0f }
`VG_CLEAR_COLOR`	`VGfloat[4]`	{ 0.0f, 0.0f, 0.0f, 0.0f }
`VG_MASKING`	`VGboolean`	`VG_FALSE` (disabled)
`VG_SCISSORING`	`VGboolean`	`VG_FALSE` (disabled)
`VG_PIXEL_LAYOUT`	`VGPixelLayout`	`VG_PIXEL_LAYOUT_UNKNOWN`
`VG_SCREEN_LAYOUT`	`VGPixelLayout`	Layout of the drawing surface
`VG_FILTER_FORMAT_LINEAR`	`VGboolean`	`VG_FALSE` (disabled)
`VG_FILTER_FORMAT_` `PREMULTIPLIED`	`VGboolean`	`VG_FALSE` (disabled)
`VG_FILTER_CHANNEL_MASK`	`VGbitfield`	`(VG_RED | VG_GREEN |` `VG_BLUE | VG_ALPHA)`

Table 4: Default Parameter Values for a Context

The read-only parameter values VG_MAX_SCISSOR_RECTS, VG_MAX_DASH_COUNT, VG_MAX_KERNEL_SIZE, VG_MAX_SEPARABLE_KERNEL_SIZE, VG_MAX_GAUSSIAN_STD_DEVIATION, VG_MAX_COLOR_RAMP_STOPS, VG_MAX_IMAGE_WIDTH, VG_MAX_IMAGE_HEIGHT, VG_MAX_IMAGE_PIXELS, VG_MAX_IMAGE_BYTES, and VG_MAX_FLOAT are initialized to implementation-defined values.

The VG_SCREEN_LAYOUT parameter is initialized to the current layout of the display device containing the current drawing surface, if applicable.

The matrices for matrix modes VG_MATRIX_PATH_USER_TO_SURFACE, VG_MATRIX_IMAGE_USER_TO_SURFACE, VG_MATRIX_FILL_PAINT_TO_USER, and VG_MATRIX_STROKE_PAINT_TO_USER are initialized to the identity matrix (see Section 6.5):

$$\begin{bmatrix} sx & shx & tx \\ shy & sy & ty \\ w_0 & w_1 & w_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Default paint parameter values are set for the filling and stroking paint modes.

5.3 Setting and Querying Object Parameter Values

Objects that are referenced using a VGHandle (*i.e.*, VGImage, VGPaint, and VGPath objects) may have their parameters set and queried using a number of **vgSetParameter** and **vgGetParameter** functions. The semantics of these functions (including the handling of invalid count values) are similar to those of the **vgGet** and **vgSet** functions.

vgSetParameter

The **vgSetParameter** functions set the value of a parameter on a given VGHandle-based object.

```
void vgSetParameterf (VGHandle object, VGint paramType,
                      VGfloat value)
void vgSetParameteri (VGHandle object, VGint paramType,
                      VGint value)
void vgSetParameterfv(VGHandle object, VGint paramType,
                      VGint count, const VGfloat * values)
void vgSetParameteriv(VGHandle object, VGint paramType,
                      VGint count, const VGint * values)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

- if `object` is not a valid handle, or is not shared with the current context

`VG_ILLEGAL_ARGUMENT_ERROR`

- if `paramType` is not a valid value from the appropriate enumeration
- if `paramType` refers to a vector parameter in **vgSetParameterf** or **vgSetParameteri**
- if `paramType` refers to a scalar parameter in **vgSetParameterfv** or **vgSetParameteriv** and `count` is not equal to 1
- if `value` is not a legal enumerated value for the given parameter in **vgSetParameterf** or **vgSetParameteri**, or if `values[i]` is not a legal enumerated value for the given parameter in **vgSetParameterfv** or **vgSetParameteriv** for 0 ≤ i < `count`
- if `values` is `NULL` in **vgSetParameterfv** or **vgSetParameteriv** and count is greater than 0
- if `values` is not properly aligned in **vgSetParameterfv** or **vgSetParameteriv**
- if `count` is less than 0 in **vgSetParameterfv** or **vgSetParameteriv**
- if `count` is not a valid value for the given parameter

vgGetParameter and vgGetParameterVectorSize

The **vgGetParameter** functions return the value of a parameter on a given `VGHandle`-based object.

The **vgGetParameterVectorSize** function returns the number of elements in the vector that will be returned by the **vgGetParameteriv** or **vgGetParameterfv** functions if called with the given `paramType` argument. For scalar values, 1 is returned. If **vgGetParameteriv** or **vgGetParameterfv** is called with a smaller value for `count` than that returned by **vgGetParameterVectorSize**, only the first `count` elements of the vector are retrieved. Use of a greater value for `count` will result in an error.

The original value passed to **vgSetParameter** (provided the call to **vgSetParameter** completed without error) should be returned by **vgGetParameter** (except where specifically noted), even if the implementation makes use of a truncated or quantized value internally.

If an error occurs during a call to **vgGetParameterf**, **vgGetParameteri**, or **vgGetParameterVectorSize**, the return value is undefined. If an error occurs during a call to **vgGetParameterfv** or **vgGetParameteriv**, nothing is written to `values`.

```
VGfloat vgGetParameterf (VGHandle object,
                         VGint paramType)
VGint   vgGetParameteri (VGHandle object,
                         VGint paramType)

VGint   vgGetParameterVectorSize (VGHandle object,
                                  VGint paramType)

void    vgGetParameterfv(VGHandle object,
                         VGint paramType,
                         VGint count, VGfloat * values)
void    vgGetParameteriv(VGHandle object,
                         VGint paramType,
                         VGint count, VGint * values)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `object` is not a valid handle, or is not shared with the current context

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `paramType` is not a valid value from the appropriate enumeration
– if `paramType` refers to a vector parameter in **vgGetParameterf** or **vgGetParameteri**
– if `values` is `NULL` in **vgGetParameterfv** or **vgGetParameteriv**
– if `values` is not properly aligned in **vgGetParameterfv** or **vgGetParameteriv**
– if `count` is less than or equal to 0 in **vgGetParameterfv** or **vgGetParameteriv**
– if `count` is greater than the value returned by **vgGetParameterVectorSize** for the given parameter in **vgGetParameterfv** or **vgGetParameteriv**

6 Rendering Quality and Antialiasing

Rendering quality settings are available to control implementation-specific trade-offs between quality and performance. For example, an application might wish to use the highest quality setting for still images, and the fastest setting during UI operations or animation. The implementation must satisfy conformance requirements regardless of the quality setting.

A non-antialiased mode is provided in which pixel coverage is always assigned to be 0 or 1, based on the inclusion of the pixel center in the geometry being rendered. When antialiasing is disabled, a coverage value of 1 will be assigned to each pixel whose center lies within the estimated path geometry, and a coverage value of 0 will be assigned otherwise. A consistent tie-breaking rule must be used for paths that pass through pixel centers.

For purposes of estimating whether a pixel center is included within a path, implementations may make use of approximations to the exact path geometry, providing that the following constraints are met. Conceptually, draw a disc *D* around each pixel center with a radius of just under ½ a pixel (in topological terms, an open disc of radius ½) and consider its intersection with the exact path geometry:

1. If *D* is entirely inside the path, the coverage at the pixel center must be estimated as 1;
2. If *D* is entirely outside the path, the coverage at the pixel center must be estimated as 0;
3. If *D* lies partially inside and partially outside the path, the coverage may be estimated as either 0 or 1 subject to the additional constraints that:
   a. The estimation is deterministic and invariant with respect to state variables apart from the current user-to-surface transformation and path coordinate geometry; and
   b. For two disjoint paths that share a common segment, if *D* is partially covered by each path and completely covered by the union of the paths, the coverage must be estimated as 1 for exactly one of the paths. A segment is considered common to two paths if and only if both paths have the same path format, path datatype, scale, and bias, and the segments have bit-for-bit identical segment types and coordinate values. If the segment is specified using relative coordinates, any preceding segments that may influence the segment must also have identical segment types and coordinate values.

Non-antialiased rendering may be useful for previewing results or for techniques such as *picking* (selecting the geometric primitive that appears at a given screen location) that require a single geometric entity to be associated with each pixel after rendering has completed.

Applications may indicate the sub-pixel color layout of the display in order to optimize rendering quality.

6.1 Rendering Quality

The overall rendering quality may be set to one of three settings: non-antialiased, faster, or better.

VGRenderingQuality

The `VGRenderingQuality` enumeration defines the values for setting the rendering quality:

```
typedef enum {
  VG_RENDERING_QUALITY_NONANTIALIASED = 0x1200,
  VG_RENDERING_QUALITY_FASTER         = 0x1201,
  VG_RENDERING_QUALITY_BETTER         = 0x1202  /* Default */
} VGRenderingQuality;
```

The `VG_RENDERING_QUALITY_NONANTIALIASED` setting disables antialiasing. The `VG_RENDERING_QUALITY_FASTER` setting causes rendering to be done at the highest available speed, while still satisfying all API conformance criteria. The `VG_RENDERING_QUALITY_BETTER` setting, which is the default, causes rendering to be done with the highest available quality.

The **vgSet** function is used to control the quality setting to one of `VG_RENDERING_QUALITY_NONANTIALIASED`, `VG_RENDERING_QUALITY_FASTER`, or `VG_RENDERING_QUALITY_BETTER`:

```
vgSeti(VG_RENDERING_QUALITY, VG_RENDERING_QUALITY_NONANTIALIASED);
vgSeti(VG_RENDERING_QUALITY, VG_RENDERING_QUALITY_FASTER);
vgSeti(VG_RENDERING_QUALITY, VG_RENDERING_QUALITY_BETTER);
```

6.2 Additional Quality Settings

VGPixelLayout

The `VGPixelLayout` enumeration describes a number of possible geometric layouts of the red, green, and blue emissive or reflective elements within a pixel. This information may be used as a hint to the rendering engine to improve rendering quality. The supported pixel layouts are illustrated in Figure 2.

```
typedef enum {
  VG_PIXEL_LAYOUT_UNKNOWN        = 0x1300,
  VG_PIXEL_LAYOUT_RGB_VERTICAL   = 0x1301,
  VG_PIXEL_LAYOUT_BGR_VERTICAL   = 0x1302,
  VG_PIXEL_LAYOUT_RGB_HORIZONTAL = 0x1303,
  VG_PIXEL_LAYOUT_BGR_HORIZONTAL = 0x1304
} VGPixelLayout;
```

The pixel layout of the display device associated with the current drawing surface may be queried using **vgGeti** with a `paramType` value of `VG_SCREEN_LAYOUT`. The value `VG_PIXEL_LAYOUT_UNKNOWN` may indicate that the color elements of a pixel are geometrically coincident, or that no layout information is available to the implementation.

To provide the renderer with a pixel layout hint, use **vgSeti** with a `paramType` value of `VG_PIXEL_LAYOUT` and a value from the `VGPixelLayout` enumeration. The value `VG_PIXEL_LAYOUT_UNKNOWN` disables any optimizations based on pixel layout, treating the color elements of a pixel as geometrically coincident. Reading back the value of `VG_PIXEL_LAYOUT` with **vgGet** simply returns the value set by the application or the default value and does not reflect the properties of the drawing surface.

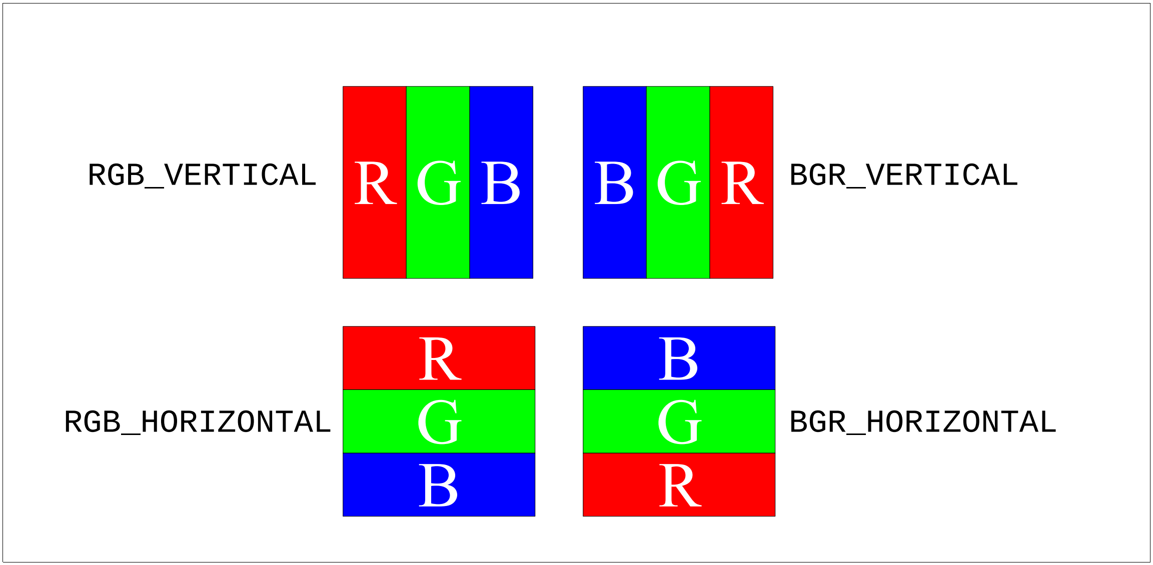


Figure 2: `VGPixelLayout` Values

6.3 Coordinate Systems and Transformations

Geometry is defined in a two-dimensional coordinate system that may or may not correspond to pixel coordinates. Drawing may be performed independently of the details of screen size, resolution, and drawing area by establishing suitable transformations between coordinate systems.

6.4 Coordinate Systems

Geometric coordinates are specified in the *user coordinate system*. The *path-user-to-surface* and *image-user-to-surface* transformations map between the user coordinate system and pixel coordinates on the destination drawing surface. This pixel-based coordinate system is known as the *surface coordinate system*. The relationship between the user and surface coordinate systems and the transformations that map between them is shown in Figure 3 below.

The user coordinate system is oriented such that values along the X axis increase from left to right and values along the Y axis increase from bottom to top, as in OpenGL. When the user-to-surface transformation is the identity transformation, a change of 1 unit along the X axis corresponds to moving by one pixel.

In the surface coordinate system, pixel (0, 0) is located at the lower-left corner of the drawing surface. The pixel (x, y) has its center at the point $(x + ½, y + ½)$. Antialiasing filters used to evaluate the color or coverage of a pixel are centered at the pixel center. If antialiasing is disabled, the evaluation of each pixel occurs at its center.

6.5 Transformations

Geometry is defined in the user coordinate system, and is ultimately transformed into surface coordinates and assigned colors by means of a set of user-specified transformations that apply to geometric path data and to paint.

6.5.1 Homogeneous Coordinates

Homogeneous coordinates are used in order to allow translation factors to be included in the affine matrix formulation, as well as to allow perspective effects for images. In homogeneous coordinates, a two-dimensional point (x, y) is represented by the three-dimensional column vector $[x, y, 1]^T$. The same point may be equivalently represented by the vector $[s*x, s*y, s]^T$ for any non-zero scale factor s. More detailed explanations of the use of homogeneous coordinates may be found in most standard computer graphics textbooks, for example [FvDFH95].

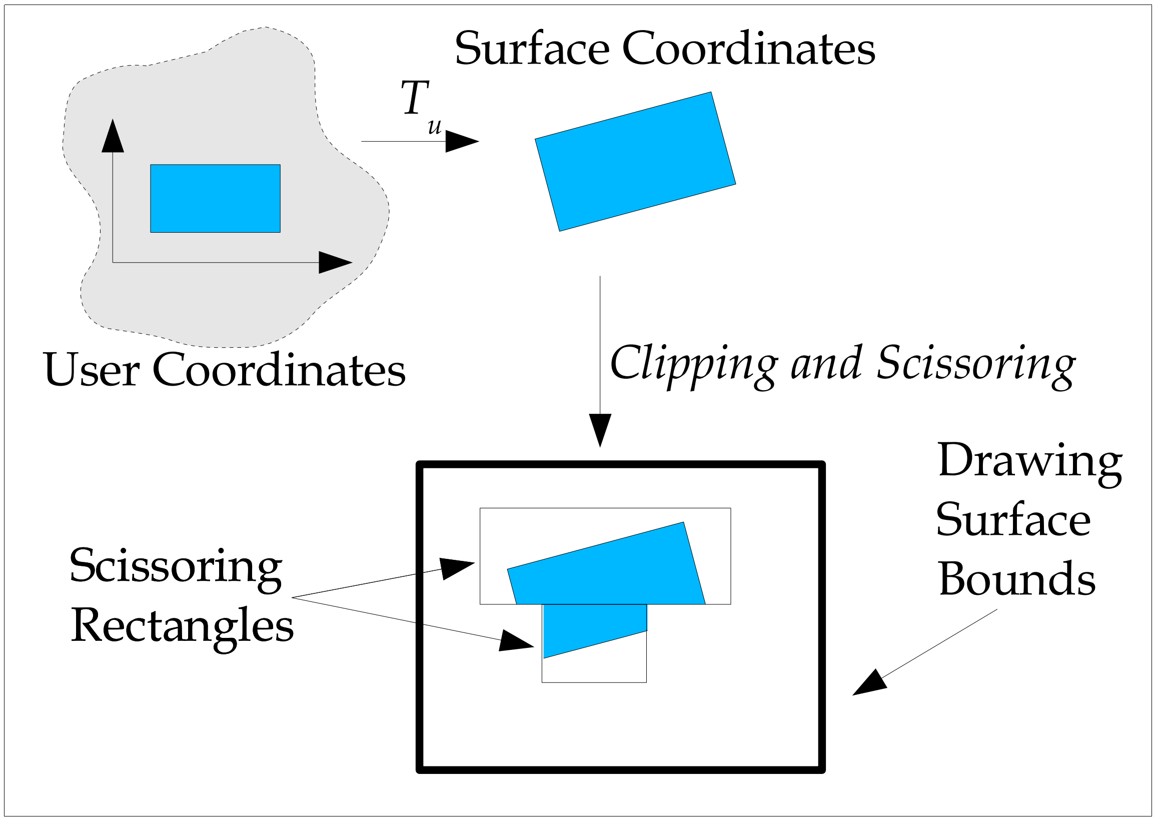


Figure 3: Coordinates, Transformation, Clipping, and Scissoring

6.5.2 Affine Transformations

Geometric objects to be drawn are transformed from user coordinates to surface coordinates as they are drawn by means of a 3x3 *affine* transformation matrix with the following entries:

$$\begin{bmatrix} sx & shx & tx \\ shy & sy & ty \\ 0 & 0 & 1 \end{bmatrix}$$

The entries may be divided by their function:

- *sx* and *sy* define *scaling* in the *x* and *y* directions, respectively;
- *shx* and *shy* define *shearing* in the *x* and *y* directions, respectively;
- *tx* and *ty* define *translation* in the *x* and *y* directions, respectively.

An affine transformation maps a point (*x, y*) (represented using homogeneous coordinates as the column vector $[x, y, 1]^T$) into the point (*x*sx + y*shx + tx, x*shy + y*sy + ty*) using matrix multiplication:

$$\begin{bmatrix} sx & shx & tx \\ shy & sy & ty \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x*sx & + y*shx & + tx \\ x*shy & + y*sy & + ty \\ & 1 & \end{bmatrix}$$

Affine transformations allow any combination of scaling, rotation, shearing, and translation. The concatenation of two affine transformations is an affine transformation, whose matrix form is the product of the matrices of the original transformations.

Gradients and patterns are subject to an additional affine transformation mapping the coordinate system used to specify the gradient parameters into user coordinates. The path-user-to-surface transformation is then applied to yield surface coordinates.

OpenVG does not provide the notion of a hierarchy of transformations; applications must maintain their own matrix stacks if desired.

6.5.3 Projective (Perspective) Transformations

The **vgDrawImage** function uses a 3x3 *projective* (or *perspective*) transformation matrix (representing the image-user-to-surface transformation) with the following entries to transform from user coordinates to surface coordinates:

$$\begin{bmatrix} sx & shx & tx \\ shy & sy & ty \\ w_0 & w_1 & w_2 \end{bmatrix}$$

A projective transformation maps a point (*x, y*) into the point:

$$\left(\frac{x*sx + y*shx + tx}{x*w_0 + y*w_1 + w_2}, \frac{x*shy + y*sy + ty}{x*w_0 + y*w_1 + w_2} \right)$$

using matrix multiplication and division by the third homogeneous coordinate:

$$\begin{bmatrix} sx & shx & tx \\ shy & sy & ty \\ w_0 & w_1 & w_2 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} x*sx + y*shx + tx \\ x*shy + y*sy + ty \\ x*w_0 + y*w_1 + w_2 \end{bmatrix} \equiv \begin{bmatrix} \dfrac{x*sx + y*shx + tx}{x*w_0 + y*w_1 + w_2} \\ \dfrac{x*shy + y*sy + ty}{x*w_0 + y*w_1 + w_2} \\ 1 \end{bmatrix}$$

The concatenation of two projective transformations is a projective transformation, whose matrix form is the product of the matrices of the original transformations.

Both affine and projective transformations map straight lines to straight lines. However, affine transformations map evenly spaced points along a source line to evenly spaced points in the destination, whereas projective transformations allow the distance between points to vary due to the effect of division by the denominator $d = (x*w_0 + y*w_1 + w_2)$. Although OpenVG does not provide support for three-dimensional coordinates, proper setting of the w matrix entries can simulate the effect of placement of images in three dimensions, as well as other warping effects.

6.6 Matrix Manipulation

Transformation matrices are manipulated using the **vgLoadIdentity**, **vgLoadMatrix**, and **vgMultMatrix** functions. For convenience, the **vgTranslate**, **vgScale**, **vgShear**, and **vgRotate** functions may be used to concatenate common types of transformations.

The matrix conventions used by OpenVG are similar to those of OpenGL. A point to be transformed is given by a homogeneous column vector $[x, y, 1]^T$. Transformation of a point p by a matrix M is defined as the product $M \cdot p$. Concatenation of transformations is performed using right-multiplication of matrices.

In the following sections, the matrix being updated by each call will be represented by the symbol M.

VGMatrixMode

The current matrix to be manipulated is specified by setting the matrix mode. Separate matrices are maintained for transforming paths, images, and paint (gradients and patterns). The matrix modes are defined in the `VGMatrixMode` enumeration:

```
typedef enum {
  VG_MATRIX_PATH_USER_TO_SURFACE  = 0x1400,
  VG_MATRIX_IMAGE_USER_TO_SURFACE = 0x1401,
  VG_MATRIX_FILL_PAINT_TO_USER    = 0x1402,
  VG_MATRIX_STROKE_PAINT_TO_USER  = 0x1403
} VGMatrixMode;
```

To set the matrix mode, call **vgSeti** with a type of `VG_MATRIX_MODE` and a value of `VG_MATRIX_*`. For example, to set the matrix mode to allow manipulation of the path-user-to-surface transformation, call:

```
vgSeti(VG_MATRIX_MODE, VG_MATRIX_PATH_USER_TO_SURFACE);
```

vgLoadIdentity

The **vgLoadIdentity** function sets the current matrix M to the identity matrix:

$$M = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
void vgLoadIdentity(void)
```

vgLoadMatrix

The **vgLoadMatrix** function loads an arbitrary set of matrix values into the current matrix. Nine matrix values are read from `m`, in the order:

$$\{ sx, shy, w_0, shx, sy, w_1, tx, ty, w_2 \}$$

defining the matrix:

$$M = \begin{bmatrix} sx & shx & tx \\ shy & sy & ty \\ w_0 & w_1 & w_2 \end{bmatrix}$$

However, if the targeted matrix is affine (*i.e.*, the matrix mode is not `VG_MATRIX_IMAGE_USER_TO_SURFACE`), the values $\{ w_0, w_1, w_2 \}$ are ignored and replaced by the values $\{ 0, 0, 1 \}$, resulting in the affine matrix:

$$M = \begin{bmatrix} sx & shx & tx \\ shy & sy & ty \\ 0 & 0 & 1 \end{bmatrix}$$

```
void vgLoadMatrix(const VGfloat * m)
```

ERRORS

`VG_ILLEGAL_ARGUMENT_ERROR`

- if `m` is `NULL`
- if `m` is not properly aligned

vgGetMatrix

It is possible to retrieve the value of the current transformation by calling **vgGetMatrix**. Nine values are written to `m` in the order:

$$\{ sx, shy, w_0, shx, sy, w_1, tx, ty, w_2 \}$$

For an affine matrix, w_0 and w_1 will always be 0 and w_2 will always be 1.

```
void vgGetMatrix(VGfloat * m)
```

ERRORS

`VG_ILLEGAL_ARGUMENT_ERROR`

- if `m` is `NULL`
- if `m` is not properly aligned

vgMultMatrix

The **vgMultMatrix** function right-multiplies the current matrix M by a given matrix*:*

$$M = M \cdot \begin{bmatrix} sx & shx & tx \\ shy & sy & ty \\ w_0 & w_1 & w_2 \end{bmatrix}$$

Nine matrix values are read from `m` in the order:

$$\{ sx, shy, w_0, shx, sy, w_1, tx, ty, w_2 \}$$

and the current matrix is multiplied by the resulting matrix. However, if the targeted matrix is affine (*i.e.*, the matrix mode is not `VG_MATRIX_IMAGE_USER_TO_SURFACE`), the values $\{ w_0, w_1, w_2 \}$ are ignored and replaced by the values *{ 0, 0, 1 }* prior to multiplication.

```
void vgMultMatrix(const VGfloat * m)
```

ERRORS

`VG_ILLEGAL_ARGUMENT_ERROR`

- if `m` is `NULL`
- if `m` is not properly aligned

vgTranslate

The **vgTranslate** function modifies the current transformation by appending a translation. This is equivalent to right-multiplying the current matrix *M* by a translation matrix:

$$M = M \cdot \begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$$

```
void vgTranslate(VGfloat tx, VGfloat ty)
```

vgScale

The **vgScale** function modifies the current transformation by appending a scale. This is equivalent to right-multiplying the current matrix *M* by a scale matrix:

$$M = M \cdot \begin{bmatrix} sx & 0 & 0 \\ 0 & sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
void vgScale(VGfloat sx, VGfloat sy)
```

vgShear

The **vgShear** function modifies the current transformation by appending a shear. This is equivalent to right-multiplying the current matrix *M* by a shear matrix:

$$M = M \cdot \begin{bmatrix} 1 & shx & 0 \\ shy & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

```
void vgShear(VGfloat shx, VGfloat shy)
```

vgRotate

The **vgRotate** function modifies the current transformation by appending a counter-clockwise rotation by a given angle (expressed in degrees) about the origin. This is equivalent to right-multiplying the current matrix M by the following matrix (using the symbol a to represent the value of the `angle` parameter):

$$M = M \cdot \begin{bmatrix} \cos(a) & -\sin(a) & 0 \\ \sin(a) & \cos(a) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

To rotate about a center point (*cx, cy*) other than the origin, the application may perform a translation by (*-cx, -cy*), followed by the rotation, followed by a translation by (*cx, cy*).

```
void vgRotate(VGfloat angle)
```

7 Scissoring, Masking, and Clearing

All drawing is *clipped* (restricted) to the bounds of the drawing surface, and may be further clipped to the interior of a set of *scissoring* rectangles. If available, an alpha *mask* is applied for further clipping and to create soft edge and partial transparency effects.

Pixels outside the drawing surface bounds, or (when scissoring is enabled) not in any scissoring rectangle are not affected by any drawing operation. For any drawing operation, each pixel will receive the same value for any setting of the scissoring rectangles that contains the pixel. That is, the placement of the scissoring rectangles, and whether scissoring is enabled, affects only whether a given pixel will be written, without affecting what value it will receive.

7.1 Scissoring

Drawing may be restricted to the union of a set of scissoring rectangles. Scissoring is enabled when the parameter `VG_SCISSORING` has the value `VG_TRUE`. Scissoring may be disabled by calling **vgSeti** with a `paramType` argument of `VG_SCISSORING` and a value of `VG_FALSE`.

VG_MAX_SCISSOR_RECTS

The `VG_MAX_SCISSOR_RECTS` parameter contains the maximum number of scissoring rectangles that may be supplied for the `VG_SCISSOR_RECTS` parameter. All implementations must support at least 32 scissor rectangles. If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. The value may be retrieved by calling **vgGeti** with a `paramType` argument of `VG_MAX_SCISSOR_RECTS`:

```
VGint maxScissorRects = vgGeti(VG_MAX_SCISSOR_RECTS);
```

Specifying Scissoring Rectangles

Each scissoring rectangle is specified as an integer 4-tuple of the form (*minX, minY, width, height*), where *minX* and *minY* are inclusive. A rectangle with *width* ≤ *0* or *height* ≤ *0* is ignored. The scissoring region is defined as the union of all the specified rectangles. The rectangles as specified need not be disjoint. If scissoring is enabled and no valid scissoring rectangles are present, no drawing occurs. If more than `VG_MAX_SCISSOR_RECTS` rectangles are specified, those beyond the first `VG_MAX_SCISSOR_RECTS` are discarded immediately (and will not be returned by **vgGet**).

```
#define NUM_RECTS 2
/* { Min X, Min Y, Width, Height } 4-Tuples */
VGint coords[4*NUM_RECTS] = { 20, 30, 100, 200,
                              50, 70,  80,  80 };
vgSetiv(VG_SCISSOR_RECTS, 4*NUM_RECTS, coords)
```

7.2 Alpha Masking

All drawing operations may be modified by an alpha mask, defining an additional alpha value at each pixel of the drawing surface that is multiplied by the coverage value computed by the rasterization stage of the pipeline. Alpha masking is enabled when an alpha mask is present for the drawing surface (*e.g.,* by specifying an EGLConfig with an EGL_ALPHA_MASK_SIZE attribute having a value greater than zero) and the VG_MASKING parameter has the value VG_TRUE. Alpha masking may be disabled by calling **vgSeti** with a parameter of VG_MASKING and a value of VG_FALSE. If an alpha mask is present, it may be manipulated by the **vgMask** function regardless of the value of VG_MASKING at the time of the call. If an alpha mask is not present, the behavior is the same as though there were an alpha mask having a value of 1 at every pixel; functions that manipulate the mask values have no effect.

In this section, we will describe alpha values as ranging from 0 to 1. The actual bit depth used for computation is implementation-dependent. It must be possible to obtain configurations supporting a mask with at least 1 bit for 1-bit black and white drawing surfaces, a mask with at least 4 bits for 16-bit color drawing surfaces, and a mask with at least 8 bits for 8-bit grayscale and 24-bit color drawing surfaces.

An alpha mask may be thought of as an alpha-only image with the same size as the current drawing surface. Initially, an alpha mask has the value of 1 at every pixel. Changes to the alpha mask outside of the current drawing surface bounds are ignored. If the drawing surface size changes, the alpha mask is resized accordingly, with new pixels being initialized to an alpha value of 1. If the context acquires a new drawing surface, the alpha mask is reset.

An alpha mask defines a stencil area through which primitives are placed before being drawn. The union, intersection, and subtraction operations on masks are defined by analogy with the corresponding operations on the stencil areas.

The mask alpha values are multiplied by the corresponding coverage values of each primitive being drawn in the clipping and masking stage (stage 5) of the rendering pipeline (see Section 2.5). The masking step is equivalent to replacing the source image with the result of the Porter-Duff operation "Src **in** Mask" (see Section 12.2).

VGMaskOperation

The VGMaskOperation enumeration defines the set of possible operations that may be used to modify the drawing surface alpha mask, possibly making use of a new mask image. Each operation occurs within a rectangular region of interest.

The VG_CLEAR_MASK operation sets all mask alpha values in the region of interest to 0, ignoring the new mask image.

The `VG_FILL_MASK` operation sets all mask alpha values in the region of interest to 1, ignoring the new mask image.

The `VG_SET_MASK` operation copies alpha values in the region of interest from the new mask image, overwriting the previous alpha mask values.

The `VG_UNION_MASK` operation replaces the previous alpha mask in the region of interest by its union with the new mask image. The resulting alpha values are always greater than or equal to their previous value.

The `VG_INTERSECT_MASK` operation replaces the previous alpha mask in the region of interest by its intersection with the new mask image. The resulting mask values are always less than or equal to their previous value.

The `VG_SUBTRACT_MASK` operation subtracts the new alpha mask from the previous alpha mask and replaces the previous alpha mask in the region of interest by the resulting mask. The resulting alpha values are always less than or equal to their previous value.

Table 5 gives the equations defining the new mask alpha value for each mask operation in terms of the previous alpha value α_{prev} and the newly supplied mask alpha value α_{mask}.

Operation	***Alpha Equation***
`VG_CLEAR_MASK`	$\alpha_{new} = 0$
`VG_FILL_MASK`	$\alpha_{new} = 1$
`VG_SET_MASK`	$\alpha_{new} = \alpha_{mask}$
`VG_UNION_MASK`	$\alpha_{new} = 1 - (1 - \alpha_{mask})*(1 - \alpha_{prev})$
`VG_INTERSECT_MASK`	$\alpha_{new} = \alpha_{mask} * \alpha_{prev}$
`VG_SUBTRACT_MASK`	$\alpha_{new} = \alpha_{prev}*(1 - \alpha_{mask})$

Table 5: VGMaskOperation Equations

```
typedef enum {
  VG_CLEAR_MASK     = 0x1500,
  VG_FILL_MASK      = 0x1501,
  VG_SET_MASK       = 0x1502,
  VG_UNION_MASK     = 0x1503,
  VG_INTERSECT_MASK = 0x1504,
  VG_SUBTRACT_MASK  = 0x1505
} VGMaskOperation;
```

vgMask

The **vgMask** function modifies the alpha mask values according to a given `operation`, possibly using alpha values taken from a `mask` image. If no alpha mask is configured, **vgMask** has no effect.

The affected region is the intersection of the drawing surface bounds with the rectangle extending from pixel (*x*, *y*) of the drawing surface and having the given `width` and `height` in pixels. For operations that make use of the `mask` image parameter (*i.e.*, operations other than `VG_CLEAR_MASK` and `VG_FILL_MASK`), mask image pixels starting at (0, 0) are used, and the region is further limited to the width and height of `mask`. For the `VG_CLEAR_MASK` and `VG_FILL_MASK` operations, the `mask` parameter is ignored and does not affect the region being modified. The value `VG_INVALID_HANDLE` may be supplied in place of an actual image handle.

The `mask` image defines alpha values at each of its pixels as follows. If the image pixel format includes an alpha channel, the alpha channel is used. Otherwise, values from the red (for color image formats) or grayscale (for grayscale formats) channel are used. The value is divided by the maximum value for the channel to obtain an alpha value between 0 and 1. If the image is bi-level (black and white), black pixels receive an alpha value of 0 and white pixels receive an alpha value of 1.

```
void vgMask(VGImage mask, VGMaskOperation operation,
            VGint x, VGint y, VGint width, VGint height)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if operation is not `VG_CLEAR_MASK` or `VG_FILL_MASK`, and `mask` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if `mask` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `operation` is not a valid value from the `VGMaskOperation` enumeration

– if `width` or `height` is less than or equal to 0

7.3 Fast Clearing

The **vgClear** function allows a region of pixels to be set to a single color with a single call.

vgClear

The **vgClear** function fills the portion of the drawing surface intersecting the rectangle extending from pixel (*x*, *y*) and having the given `width` and `height` with a constant color value, taken from the `VG_CLEAR_COLOR` parameter. The color value is expressed in non-premultiplied sRGBA (sRGB color plus alpha)format. Values outside the [0, 1] range are interpreted as the nearest endpoint of the range. The color is converted to the destination color space in the same manner as if a rectangular path were being filled. Clipping and scissoring take place in the usual fashion, but antialiasing, masking, and blending do not occur.

```
void vgClear(VGint x, VGint y, VGint width, VGint height)
```

ERRORS

`VG_ILLEGAL_ARGUMENT_ERROR`

– `if` `width` or `height` is less than or equal to 0

For example, to set the entire drawing surface with dimensions `WIDTH` and `HEIGHT` to an opaque yellow color, the following code could be used:

```
VGfloat color[4] = { 1.0f, 1.0f, 0.0f, 1.0f }; /* Opaque yellow */

vgSeti(VG_SCISSORING, VG_FALSE);
vgSetfv(VG_CLEAR_COLOR, 4, color);
vgClear(0, 0, WIDTH, HEIGHT);
```

8 Paths

Paths are the heart of the OpenVG API. All geometry to be drawn must be defined in terms of one or more paths. Paths are defined by a sequence of *segment commands* (or *segments*). Each segment command in the standard format may specify a move, a straight line segment, a quadratic or cubic Bézier segment, or an elliptical arc. Extensions may define other segment types.

8.1 Moves

A path segment may consist of a "move to" segment command that causes the path to jump directly to a given point, starting a new subpath without drawing.

8.2 Straight Line Segments

Paths may contain horizontal, vertical, or arbitrary line segment commands. A special "close path" segment command may be used to generate a straight line segment joining the current vertex of a path to the vertex that began the current portion of the path.

8.3 Bézier Curves

Bézier curves are polynomial curves defined using a parametric representation. That is, they are defined as the set of points of the form ($x(t)$, $y(t)$), where $x(t)$ and $y(t)$ are polynomials of t and t varies continuously from 0 to 1. Paths may contain quadratic or cubic Bézier segment commands.

8.3.1 Quadratic Bézier Curves

A quadratic Bézier segment is defined by three *control points*, (x_0, y_0), (x_1, y_1), and (x_2, y_2). The curve starts at (x_0, y_0) and ends at (x_2, y_2). The shape of the curve is influenced by the placement of the internal control point (x_1, y_1), but the curve does not usually pass through that point. Assuming non-coincident control points, the tangent of the curve at the initial point x_0 is aligned with and has the same direction as the vector $x_1 - x_0$ and the tangent at the final point x_2 is aligned with and has the same direction as the vector $x_2 - x_1$. The curve is defined by the set of points ($x(t)$, $y(t)$) as t varies from 0 to 1, where:

$$x(t)=x_0*(1-t)^2+2*x_1*(1-t)*t+x_2*t^2$$
$$y(t)=y_0*(1-t)^2+2*y_1*(1-t)*t+y_2*t^2$$

8.3.2 Cubic Bézier Curves

Cubic Bézier segments are defined by four control points (x_0, y_0), (x_1, y_1), (x_2, y_2), and (x_3, y_3). The curve starts at (x_0, y_0) and ends at (x_3, y_3). The shape of the curve is influenced by the placement of the internal control points (x_1, y_1) and (x_2, y_2), but the curve does not usually pass through those points. Assuming non-coincident control points, the tangent of the curve at the initial point x_0 is aligned with and has the same direction as the vector $x_1 - x_0$ and the tangent at the final point x_3 is aligned with and has the same direction as the vector $x_3 - x_2$. The curve is defined by the set of points ($x(t)$, $y(t)$) as t varies from 0 to 1, where:

$$x(t) = x_0 * (1-t)^3 + 3 * x_1 * (1-t)^2 * t + 3 * x_2 * (1-t) * t^2 + x_3 * t^3$$
$$y(t) = y_0 * (1-t)^3 + 3 * y_1 * (1-t)^2 * t + 3 * y_2 * (1-t) * t^2 + y_3 * t^3$$

8.3.3 G¹ Smooth Segments

G¹ Smooth quadratic or cubic segments implicitly define their first internal control point in such a manner as to guarantee a continuous tangent direction at the join point when they are joined to a preceding quadratic or cubic segment. Geometrically, this ensures that the two segments meet without a sharp corner. However, the length of the unnormalized tangent vector may experience a discontinuity at the join point.

G^1 smoothness at the initial point of a quadratic or cubic segment may be guaranteed by suitable placement of the first internal control point (x_1, y_1) of the following segment. Given a previous quadratic or cubic segment with an internal control point (*px*, *py*) and final endpoint (*ox*, *oy*), we compute (x_1, y_1) as (2**ox* – *px*, 2**oy* – *py*) (*i.e.*, the reflection of the point (*px*, *py*) about the point (*ox*, *oy*)). For segments of the same type, this will provide C^1 smoothness (see the next section).

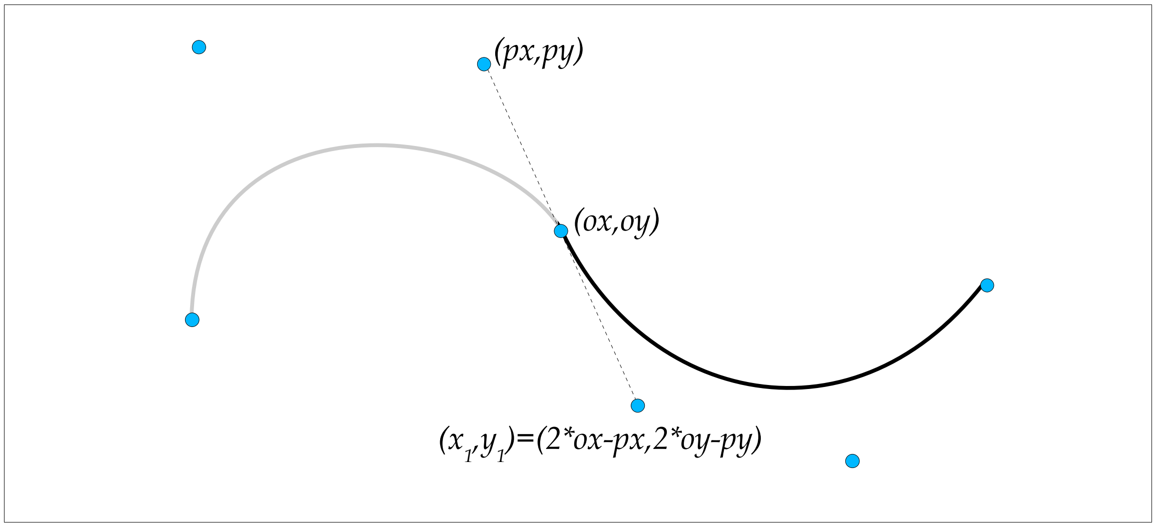


Figure 4: Smooth Curve Construction

8.3.4 C^1 Smooth Segments

[Note: this section is informative only.]

C^1 smooth quadratic or cubic segments define their first internal control point (x_1, y_1) in such a manner as to guarantee a continuous first derivative at the join point when they are joined to a preceding quadratic or cubic segment. Geometrically, this ensures that the two segments meet with continuous parametric velocity at the join point. This is a stronger condition than G^1 continuity.

Note that joining a C^1 smooth segment to a preceding line segment will not produce a smooth join. To guarantee a smooth join, convert line segments to equivalent quadratic or cubic curves whose internal control points all lie along the line segment.

Given a previous quadratic or cubic segment with an internal control point (*px, py*) and final endpoint (*ox, oy*), (x_1, y_1) is computed as follows:

- When joining a previous quadratic or cubic segment to a following segment of the same type (quadratic or cubic):

$$(x_1, y_1) = (2*ox - px, 2*oy - py)$$

- When joining a previous quadratic segment to a following cubic segment:

$$(x_1, y_1) = (5*ox - 2*px, 5*oy - 2*py)/3$$

- When joining a previous cubic segment to a following quadratic segment:

$$(x_1, y_1) = (5*ox - 3*px, 5*oy - 3*py)/2$$

8.3.5 C^2 Smooth Segments

[Note: this section is informative only.]

C^2 smooth cubic segments implicitly define both of their internal control points (x_1, y_1) and (x_2, y_2) in such a manner as to guarantee continuous first and second derivatives at the join point when they are joined to a preceding quadratic or cubic segment. Geometrically, this ensures that the two segments meet with continuous velocity and acceleration at the join point.

Note that joining a C^2 smooth segment to a preceding line segment will not produce a smooth join. To guarantee a smooth join, convert line segments to equivalent quadratic or cubic curves whose internal control points all lie along the line segment.

Given three previous control points (*qx, qy*), (*px, py*), and (*ox, oy*) (for a quadratic segment, (*qx, qy*) is the initial endpoint, (*px, py*) is the internal control point and (*ox, oy*) is the final endpoint; for a cubic segment, (*qx, qy*), and (*px, py*) are the first and second internal control points, respectively, and (*ox, oy*) is the final endpoint), (x_1, y_1) is computed as described in the preceding section, and (x_2, y_2) is computed as follows.

- When joining a previous quadratic segment to a following cubic segment:

$$(x_2, y_2) = (8*ox - 6*px + qx, 8*oy - 6*py + qy)/3$$

- When joining a previous cubic segment to a following cubic segment:

$$(x_2, y_2) = (4*(ox - px) + qx, 4*(oy - py) + qy)$$

8.3.6 Converting Segments From Quadratic to Cubic Form

[Note: This section is informative only.]

Given a quadratic Bézier curve with control points (x_0, y_0), (x_1, y_1), and (x_2, y_2), an identical cubic Bézier curve may be formed using the control points (x_0, y_0), $(x_0 + 2*x_1, y_0 + 2*y_1)/3$, $(x_2 + 2*x_1, y_2 + 2*y_1)/3$, (x_2, y_2).

8.4 Elliptical Arcs

Elliptical arc segments join a pair of points with a section of an ellipse with given horizontal and vertical axes and a rotation angle (in degrees). Given these parameters, there are four possible arcs distinguished by their direction around the ellipse (clockwise or counter-clockwise) and whether they take the smaller or larger path around the ellipse.

Figure 5 below shows the two possible ellipses with horizontal axis *rh*, vertical axis *rv*, and counter-clockwise rotation angle *rot* (shown as the angle between the vertical line labeled *rot* and the line labeled *rv*) passing through the points (x_0, y_0) and (x_1, y_1). The four arcs connecting the points are labeled L and S for large and small, and CW and CCW for clockwise and counter-clockwise.

Negative values of *rh* and *rv* are replaced with their absolute values. If exactly one of *rh* and *rv* is 0, and the arc endpoints are not coincident, the arc is drawn as if it were projected onto the line containing the endpoints. If both *rh* and *rv* are 0, or if the arc endpoints are coincident, the arc is drawn as a line segment between its endpoints. The *rot* parameter is taken modulo 360 degrees.

If no elliptical arc exists with the given parameters because the endpoints are too far apart (as detailed in the next section), the arc is drawn as if the radii were scaled up uniformly by the smallest factor that permits a solution.

Some notes on the mathematics of ellipses are provided in Appendix A (Section 17).

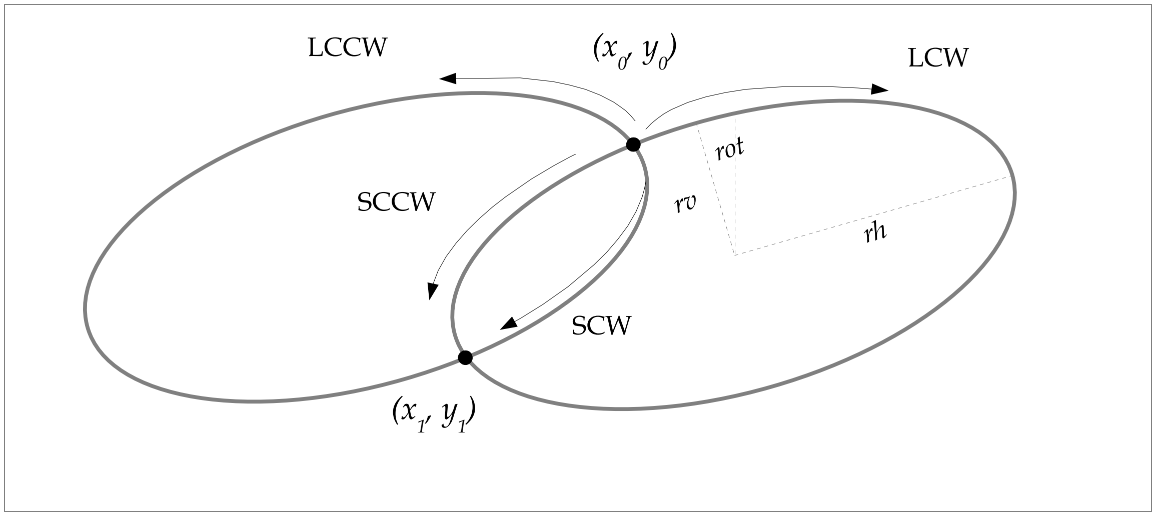


Figure 5: Elliptical Arcs

8.5 The Standard Path Format

Complex paths may be constructed in application (client-side) memory and passed into OpenVG to define a `VGPath` object. Such path data is defined by a sequence of segment commands referencing a separate sequence of geometric coordinates and parameters.

In this section, we define the standard data format for paths that may be used to define sequences of various types of path segments. Extensions may define other path data formats.

VG_PATH_FORMAT_STANDARD

The `VG_PATH_FORMAT_STANDARD` macro defines a constant to be used as an argument to **vgCreatePath** to indicate that path data are stored using the standard format. As this API is revised, the lower 16 bits of version number may increase. Each version of OpenVG will accept formats defined in all prior specification versions with which it is backwards-compatible.

Extensions wishing to define additional path formats may register for format identifiers that will differ in their upper 16 bits; the lower 16 bits may be used by the extension vendor for versioning purposes.

```
#define VG_PATH_FORMAT_STANDARD 0
```

8.5.1 Path Segment Command Side Effects

In order to define the semantics of each segment command type, we define three reference points (all are initially (0, 0)):

- (*sx, sy*): the beginning of the current subpath, *i.e.*, the position of the last `MOVE_TO` segment.
- (*ox, oy*): the last point of the previous segment.
- (*px, py*): the last internal control point of the previous segment, if the segment was a (regular or smooth) quadratic or cubic Bézier, or else the last point of the previous segment.

Figure 6 illustrates the locations of these points at the end of a sequence of segment commands { MOVE_TO, LINE_TO, CUBIC_TO }.

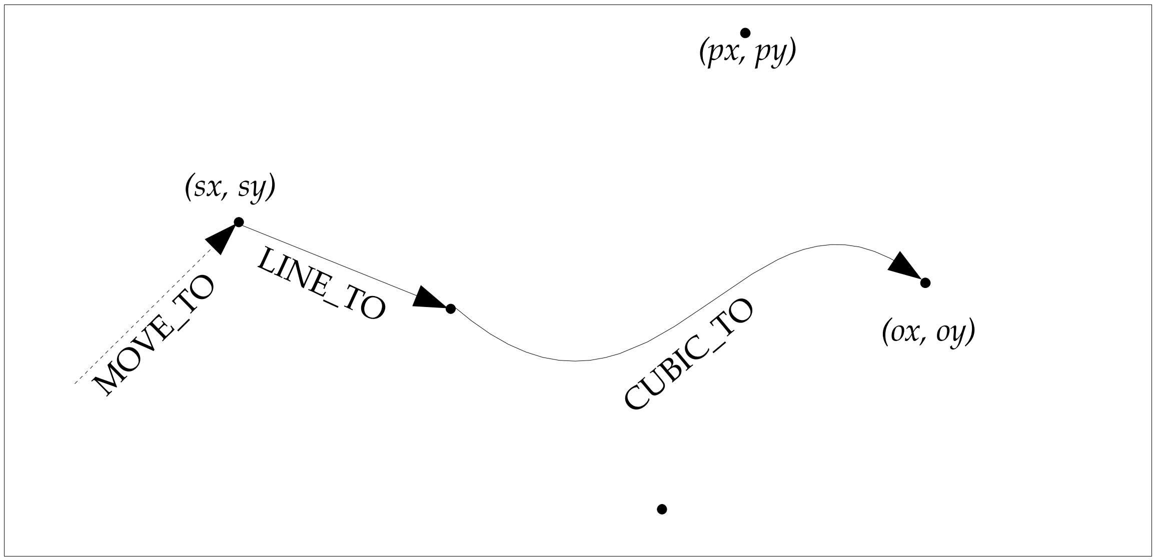


Figure 6: Segment Reference Points

We define points (*x0, y0*), (*x1, y1*), and (*x2, y2*) in the discussion below as absolute coordinates. For segments defined using relative coordinates, (*x0, y0*), etc., are defined as the incoming coordinate values added to (*ox, oy*). Ellipse *rh*, *rv*, and *rot* parameters are unaffected by the use of relative coordinates.

Each segment (except for `MOVE_TO` segments) begins at the point (*ox, oy*) defined by the previous segment.

A path consists of a sequence of subpaths. As path segment commands are encountered, each segment is appended to the *current subpath*. The current subpath is ended by a `MOVE_TO` or `CLOSE_PATH` segment, and a new current subpath is begun. The end of the path data also ends the current subpath.

8.5.2 Segment Commands

The following table describes each segment command type along with its prefix, the number of specified coordinates and parameters it requires, the numerical value of the segment command, the formulas for any implicit coordinates, and the side effects of the segment command on the points (*ox, oy*), (*sx, sy*), and (*px, py*) and on the termination of the current subpath.

Type	***Command***	***Coordinates***	***Value***	***Implicit Points***	***Side Effects***
Close Path	CLOSE_PATH	*none*	0		*(px,py)=(ox,oy)=(sx,sy)* End current subpath
Move	MOVE_TO	*x0,y0*	1		*(sx,sy)=(px,py)=(ox,oy) =(x0,y0)* End current subpath
Line	LINE_TO	*x0,y0*	2		*(px,py)=(ox,oy)=(x0,y0)*
Horizontal Line	HLINE_TO	*x0*	3	*y0=oy*	*(px,py)=(x0,oy)* *ox=x0*
Vertical Line	VLINE_TO	*y0*	4	*x0=ox*	*(px,py)=(ox,y0)* *oy=y0*
Quadratic	QUAD_TO	*x0,y0,x1,y1*	5		*(px,py)=(x0,y0)* *(ox,oy)=(x1,y1)*
Cubic	CUBIC_TO	*x0,y0,x1,y1, x2,y2*	6		*(px,py)=(x1,y1)* *(ox,oy)=(x2,y2)*
G^1 Smooth Quad	SQUAD_TO	*x1,y1*	7	*(x0,y0)= (2*ox-px, 2*oy-py)*	*(px,py)= (2*ox-px, 2*oy-py)* *(ox,oy)=(x1,y1)*
G^1 Smooth Cubic	SCUBIC_TO	*x1,y1,x2,y2*	8	*(x0,y0)= (2*ox-px, 2*oy-py)*	*(px,py)=(x1,y1)* *(ox,oy)=(x2,y2)*
Small CCW Arc	SCCWARC_TO	*rh,rv,rot,x0,y0*	9		*(px,py)=(ox,oy)=(x0,y0)*
Small CW Arc	SCWARC_TO	*rh,rv,rot,x0,y0*	10		*(px,py)=(ox,oy)=(x0,y0)*
Large CCW Arc	LCCWARC_TO	*rh,rv,rot,x0,y0*	11		*(px,py)=(ox,oy)=(x0,y0)*
Large CW Arc	LCWARC_TO	*rh,rv,rot,x0,y0*	12		*(px,py)=(ox,oy)=(x0,y0)*
Reserved	Reserved		13-15		

Table 6: Client-Side Path Segment Commands

Each segment type may be defined using either absolute or relative coordinates. A relative coordinate (*x, y*) is added to (*ox, oy*) to obtain the corresponding absolute coordinate (*ox* + *x, oy* + *y*). Relative coordinates are converted to absolute coordinates immediately as each segment is encountered during rendering.

The `HLINE_TO` and `VLINE_TO` segment types are provided in order to avoid the need for an SVG viewing application (for example) to perform its own relative to absolute conversions when parsing path data.

In SVG, the behavior of smooth quadratic and cubic segments differs slightly from the behavior defined above. If a smooth quadratic segment does not follow a quadratic segment, or a smooth cubic segment does not follow a cubic segment, the initial control point (*x0, y0*) is placed at (*ox, oy*) instead of being computed as the reflection of (*px, py*). This behavior may be emulated by converting an SVG smooth segment into a regular segment with all of its control points specified when the preceding segment is of a different degree.

Note that the coordinates of a path are defined even if the path begins with a segment type other than `MOVE_TO` (including `HLINE_TO`, `VLINE_TO`, or relative segment types) since the coordinates are based on the initial values of (*ox, oy*), (*sx, sy*), and (*px, py*) which are each defined as (0, 0).

8.5.3 Coordinate Data Formats

Coordinate and parameter data (henceforth called simply coordinate data) may be expressed in the set of formats shown in Table 7 below. Multi-byte coordinate data (*i.e.*, `S_16`, `S_32` and `F` datatypes) are stored within a client-side representation using the native byte order (endianness) of the platform. Implementations may quantize incoming data in the `S_32` and `F` formats to a lesser number of bits, provided at least 16 bits of precision are maintained.

Judicious use of smooth curve segments and 8- and 16-bit datatypes can result in substantial memory savings for common path data, such as font glyphs. Using smaller datatypes also conserves bus bandwidth when transferring paths from application memory to OpenVG.

Datatype	`VG_PATH_DATATYPE` *Suffix*	*Bytes*	*Value*
8-bit signed integer	`S_8`	1	0
16-bit signed integer	`S_16`	2	1
32-bit signed integer	`S_32`	4	2
IEEE 754 floating-point	`F`	4	3

Table 7: Client-Side Path Coordinate Datatypes

VGPathDatatype

The `VGPathDatatype` enumeration defines values describing the possible numerical datatypes for path coordinate data.

```
typedef enum {
  VG_PATH_DATATYPE_S_8          =  0,
  VG_PATH_DATATYPE_S_16         =  1,
  VG_PATH_DATATYPE_S_32         =  2,
  VG_PATH_DATATYPE_F            =  3
} VGPathDatatype;
```

8.5.4 Segment Type Marker Definitions

Segment type markers are defined as 8-bit integers, with the leading 3 bits reserved for future use, the next 4 bits containing the segment command type, and the least significant bit indicating absolute vs. relative coordinates (0 for absolute, 1 for relative). The reserved bits must be set to 0.

For the `CLOSE_PATH` segment command, the value of the Abs/Rel bit is ignored.

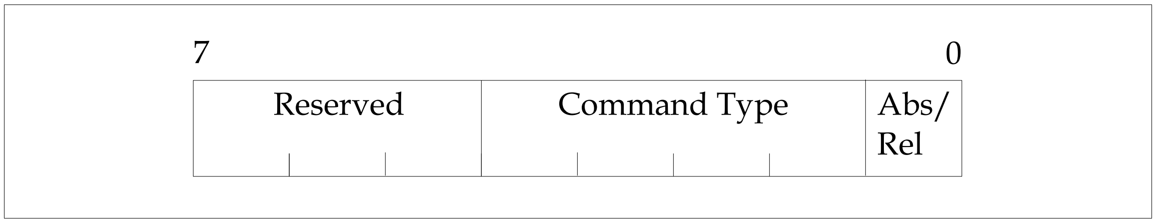


Figure 7: Segment Type Marker Layout

VGPathAbsRel

The `VGPathAbsRel` enumeration defines values indicating absolute (`VG_ABSOLUTE`) and relative (`VG_RELATIVE`) values.

```
typedef enum {
  VG_ABSOLUTE = 0,
  VG_RELATIVE = 1
} VGPathAbsRel;
```

VGPathSegment

The `VGPathSegment` enumeration defines values for each segment command type. The values are pre-shifted by 1 bit to allow them to be combined easily with values from `VGPathAbsRel`.

```
typedef enum {
  VG_CLOSE_PATH                    = ( 0 << 1),
  VG_MOVE_TO                       = ( 1 << 1),
  VG_LINE_TO                       = ( 2 << 1),
  VG_HLINE_TO                      = ( 3 << 1),
  VG_VLINE_TO                      = ( 4 << 1),
  VG_QUAD_TO                       = ( 5 << 1),
  VG_CUBIC_TO                      = ( 6 << 1),
  VG_SQUAD_TO                      = ( 7 << 1),
  VG_SCUBIC_TO                     = ( 8 << 1),
  VG_SCCWARC_TO                    = ( 9 << 1),
  VG_SCWARC_TO                     = (10 << 1),
  VG_LCCWARC_TO                    = (11 << 1),
  VG_LCWARC_TO                     = (12 << 1)
} VGPathSegment;
```

VGPathCommand

The `VGPathCommand` enumeration defines combined values for each segment command type and absolute/relative value. The values are shifted left by one bit and ORed bitwise (*i.e.,* using the C `|` operator) with the appropriate value from `VGPathAbsRel` to obtain a complete segment command value.

```
typedef enum {
  VG_MOVE_TO_ABS                     = VG_MOVE_TO    | VG_ABSOLUTE,
  VG_MOVE_TO_REL                     = VG_MOVE_TO    | VG_RELATIVE,
  VG_LINE_TO_ABS                     = VG_LINE_TO    | VG_ABSOLUTE,
  VG_LINE_TO_REL                     = VG_LINE_TO    | VG_RELATIVE,
  VG_HLINE_TO_ABS                    = VG_HLINE_TO   | VG_ABSOLUTE,
  VG_HLINE_TO_REL                    = VG_HLINE_TO   | VG_RELATIVE,
  VG_VLINE_TO_ABS                    = VG_VLINE_TO   | VG_ABSOLUTE,
  VG_VLINE_TO_REL                    = VG_VLINE_TO   | VG_RELATIVE,
  VG_QUAD_TO_ABS                     = VG_QUAD_TO    | VG_ABSOLUTE,
  VG_QUAD_TO_REL                     = VG_QUAD_TO    | VG_RELATIVE,
  VG_CUBIC_TO_ABS                    = VG_CUBIC_TO   | VG_ABSOLUTE,
  VG_CUBIC_TO_REL                    = VG_CUBIC_TO   | VG_RELATIVE,
  VG_SQUAD_TO_ABS                    = VG_SQUAD_TO   | VG_ABSOLUTE,
  VG_SQUAD_TO_REL                    = VG_SQUAD_TO   | VG_RELATIVE,
  VG_SCUBIC_TO_ABS                   = VG_SCUBIC_TO  | VG_ABSOLUTE,
  VG_SCUBIC_TO_REL                   = VG_SCUBIC_TO  | VG_RELATIVE,
  VG_SCCWARC_TO_ABS                  = VG_SCCWARC_TO | VG_ABSOLUTE,
  VG_SCCWARC_TO_REL                  = VG_SCCWARC_TO | VG_RELATIVE,
  VG_SCWARC_TO_ABS                   = VG_SCWARC_TO  | VG_ABSOLUTE,
  VG_SCWARC_TO_REL                   = VG_SCWARC_TO  | VG_RELATIVE,
  VG_LCCWARC_TO_ABS                  = VG_LCCWARC_TO | VG_ABSOLUTE,
  VG_LCCWARC_TO_REL                  = VG_LCCWARC_TO | VG_RELATIVE,
  VG_LCWARC_TO_ABS                   = VG_LCWARC_TO  | VG_ABSOLUTE,
  VG_LCWARC_TO_REL                   = VG_LCWARC_TO  | VG_RELATIVE
} VGPathCommand;
```

8.5.5 Client-Side Path Example

The following code example shows how to traverse a client-side path stored using the standard representation. A byte is read containing a segment command, and the segment command type and relative/absolute flag are extracted by application-defined `SEGMENT_COMMAND` and `SEGMENT_ABS_REL` macros. The number of coordinates and number of bytes per coordinate (for the given data format) are also determined using lookup tables. Finally, the relevant portion of the path data stream representing the current segment is copied into a temporary buffer and used as an argument to a user-defined **processSegment** function that may perform further processing.

```
#define PATH_MAX_COORDS  6 /* Maximum number of coordinates/command */
#define PATH_MAX_BYTES   4 /* Bytes in largest data type */
#define SEGMENT_COMMAND(command) /* Extract segment type */ \
    (((command) & 0x1e) >> 1)
#define SEGMENT_ABS_REL(command) /* Extract absolute/relative bit */ \
    ((command) & 0x1)

/* Number of coordinates for each command */
static const VGint numCoords[] = {0,2,2,1,1,4,6,2,4,5,5,5,5};
/* Number of bytes for each datatype */
static const VGint numBytes[] = {1,2,4,4};

/* User-defined function to process a single segment */
extern void
processSegment(VGPathSegment command, VGPathAbsRel absRel,
               VGPathDatatype datatype,
               void * segmentData);

/* Process a path in the standard format, one segment at a time. */
void
processPath(const VGubyte * pathSegments, const void * pathData,
            int numSegments, VGPathDatatype datatype)
{
  VGubyte segmentType, segmentData[PATH_MAX_COORDS*PATH_MAX_BYTES];
  VGint segIdx = 0, dataIdx = 0;
  VGint command, absRel, numBytes;

  while (segIdx < numSegments) {
    segmentType = pathSegments[segIdx++];
    command = SEGMENT_COMMAND(segmentType);
    absRel = SEGMENT_ABS_REL(segmentType);
    numBytes = numCoords[command]*numBytes[datatype];

    /* Copy segment data for further processing */
    memcpy(segmentData, &pathData[dataIdx], numBytes);

    /* Process command */
    processSegment(command, absRel, datatype, (void *) segmentData);
    dataIdx += numBytes;
  }
}
```

8.6 Path Operations

In addition to filling or stroking a path, the API allows the following basic operations on paths:

- Create a path with a given set of capabilities (**vgCreatePath**)
- Remove data from a path (**vgClearPath**)
- Deallocate a path (**vgDestroyPath**)

- Query path information (using **vgGetParameter**)
- Query the set of capabilities for a path (**vgGetPathCapabilities**)
- Reduce the set of capabilities for a path (**vgRemovePathCapabilities**)
- Append data from one path onto another (**vgAppendPath**)
- Append client-side data onto a path (**vgAppendPathData**)
- Modify coordinates stored in a path (**vgModifyPathCoords**)
- Transform a path (**vgTransformPath**)
- Interpolate between two paths (**vgInterpolatePath**)
- Determine the geometrical length of a path (**vgPathLength**)
- Get position and tangent information for a point at a given geometric distance along path (**vgPointAlongPath**)
- Get an axis-aligned bounding box for a path (**vgPathBounds**, **vgTransformedPathBounds**)

Higher-level geometric primitives are defined in the optional `VGU` utility library (see Section 16):

- Append a line to a path (**vguLine**)
- Append a polyline or polygon to a path (**vguPolygon**)
- Append a rectangle to a path (**vguRect**)
- Append a round-cornered rectangle to a path (**vguRoundRect**)
- Append an ellipse to a path (**vguEllipse**)
- Append a circular arc to a path (**vguArc**)

8.6.1 Storage of Paths

OpenVG stores path data internally to the implementation. Paths are referenced via opaque `VGPath` handles. Applications may initialize paths using the client-side memory representation defined above or other representations defined by extensions.

It is possible for an implementation to store path data in hardware-accelerated memory. Implementations may also make use of their own internal representation of path segments. The intent is for applications to be able to define a set of paths, for example one for each glyph in the current typeface, and to be able to re-render each previously defined path with maximum efficiency.

VGPath

`VGPath` represents an opaque handle to a path.

```
typedef VGHandle VGPath;
```

8.6.2 Creating and Destroying Paths

Paths are created and destroyed using the **vgCreatePath** and **vgDestroyPath** functions. During the lifetime of a path, an application may indicate which path operations it plans to perform using path capability flags defined by the `VGPathCapabilities` enumeration.

`VGPathCapabilities`

The `VGPathCapabilities` enumeration defines a set of constants specifying which operations may be performed on a given path object. At the time a path is defined, the application specifies which operations it wishes to be able to perform on the path. Over time, the application may disable previously enabled capabilities, but it may not re-enable capabilities once they have been disabled. This feature allows OpenVG implementations to make use of internal path representations that may not support all path operations, possibly resulting in higher performance on paths where those operations will not be performed.

The capability bits and the functionality they allow are described below:

- `VG_PATH_CAPABILITY_APPEND_FROM` – use path as the `srcPath` argument to **vgAppendPath**
- `VG_PATH_CAPABILITY_APPEND_TO` - use path as the `dstPath` argument to **vgAppendPath** and **vgAppendPathData**
- `VG_PATH_CAPABILITY_MODIFY` – use path as the `dstPath` argument to **vgModifyPathCoords**
- `VG_PATH_CAPABILITY_TRANSFORM_FROM` – use path as the `srcPath` argument to **vgTransformPath**
- `VG_PATH_CAPABILITY_TRANSFORM_TO` – use path as the `dstPath` argument to **vgTransformPath**
- `VG_PATH_CAPABILITY_INTERPOLATE_FROM` – use path as the `startPath` or `endPath` argument to **vgInterpolatePath**
- `VG_PATH_CAPABILITY_INTERPOLATE_TO` – use path as the `dstPath` argument to **vgInterpolatePath**
- `VG_PATH_CAPABILITY_PATH_LENGTH` – use path as the `path` argument to **vgPathLength**
- `VG_PATH_CAPABILITY_POINT_ALONG_PATH` – use path as the `path` argument to **vgPointAlongPath**
- `VG_PATH_CAPABILITY_TANGENT_ALONG_PATH` – use path as the `path` argument to **vgPointAlongPath** with non-`NULL` `tangentX` and `tangentY` arguments
- `VG_PATH_CAPABILITY_PATH_BOUNDS` – use path as the `path` argument to **vgPathBounds**
- `VG_PATH_CAPABILITY_PATH_TRANSFORMED_BOUNDS` – use path as the `path` argument to **vgPathTransformedBounds**
- `VG_PATH_CAPABILITY_ALL` – a bitwise OR of all the defined path capabilities

```
typedef enum {
  VG_PATH_CAPABILITY_APPEND_FROM              = (1 <<  0),
  VG_PATH_CAPABILITY_APPEND_TO                = (1 <<  1),
  VG_PATH_CAPABILITY_MODIFY                   = (1 <<  2),
  VG_PATH_CAPABILITY_TRANSFORM_FROM           = (1 <<  3),
  VG_PATH_CAPABILITY_TRANSFORM_TO             = (1 <<  4),
  VG_PATH_CAPABILITY_INTERPOLATE_FROM         = (1 <<  5),
  VG_PATH_CAPABILITY_INTERPOLATE_TO           = (1 <<  6),
  VG_PATH_CAPABILITY_PATH_LENGTH              = (1 <<  7),
  VG_PATH_CAPABILITY_POINT_ALONG_PATH         = (1 <<  8),
  VG_PATH_CAPABILITY_TANGENT_ALONG_PATH       = (1 <<  9),
  VG_PATH_CAPABILITY_PATH_BOUNDS              = (1 << 10),
  VG_PATH_CAPABILITY_PATH_TRANSFORMED_BOUNDS  = (1 << 11),
  VG_PATH_CAPABILITY_ALL                      = (1 << 12) - 1
} VGPathCapabilities;
```

It is legal to call **vgCreatePath**, **vgClearPath**, and **vgDestroyPath** regardless of the current setting of the path's capability bits, as these functions discard the existing path definition.

vgCreatePath

vgCreatePath creates a new path that is ready to accept segment data and returns a `VGPath` handle to it. The path data will be formatted in the format given by `pathFormat`, typically `VG_PATH_FORMAT_STANDARD`. The `datatype` parameter contains a value from the `VGPathDatatype` enumeration indicating the datatype that will be used for coordinate data. The `capabilities` argument is a bitwise OR of the desired `VGPathCapabilities` values. Bits of `capabilities` that do not correspond to values from `VGPathCapabilities` have no effect. If an error occurs, `VG_INVALID_HANDLE` is returned.

The `scale` and `bias` parameters are used to interpret each coordinate of the path data; an incoming coordinate value *v* will be interpreted as the value (*scale*v* + *bia*s). `scale` must not equal 0. The datatype, scale, and bias together define a valid coordinate data range for the path; segment commands that attempt to place a coordinate in the path that is outside this range will overflow silently, resulting in an undefined coordinate value. Functions that query a path containing such values, such as **vgPathLength** and **vgPointAlongPath,** also return undefined results.

The `segmentCapacityHint` parameter provides a hint as to the total number of segments that will eventually be stored in the path. The `coordCapacityHint` parameter provides a hint as to the total number of specified coordinates (as defined in the "Coordinates" column of Table 6) that will eventually be stored in the path. A value less than or equal to 0 for either hint indicates that the capacity is unknown. The path storage space will in any case grow as needed, regardless of the hint values. However, supplying hints may improve performance by reducing the need to allocate additional space as the path grows. Implementations should allow applications to append segments and coordinates up to the stated capacity in small batches without degrading performance due to excessive memory reallocation.

```
VGPath vgCreatePath(VGint pathFormat,
                    VGPathDatatype datatype,
                    VGfloat scale, VGfloat bias,
                    VGint segmentCapacityHint,
                    VGint coordCapacityHint,
                    VGbitfield capabilities)
```

ERRORS

`VG_UNSUPPORTED_PATH_FORMAT_ERROR`

– if `pathFormat` is not a supported format

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `datatype` is not a valid value from the `VGPathDatatype` enumeration

– if `scale` is equal to 0

vgClearPath

vgClearPath removes all segment command and coordinate data associated with a `path`. The handle continues to be valid for use in the future, and the path format and datatype retain their existing values. The `capabilities` argument is a bitwise OR of the desired `VGPathCapabilities` values. Bits of `capabilities` that do not correspond to values from `VGPathCapabilities` have no effect. Using **vgClearPath** may be more efficient than destroying and re-creating a path for short-lived paths.

```
void vgClearPath(VGPath path, VGbitfield capabilities)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

vgDestroyPath

vgDestroyPath releases any resources associated with `path`, and makes the handle invalid in all contexts that shared it.

```
void vgDestroyPath(VGPath path)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

8.6.3 Path Queries

VGPathParamType

Values from the `VGPathParamType` enumeration may be used as the `paramType` argument to **vgGetParameter** to query various features of a path. All of the parameters defined by `VGPathParamType` are read-only. Table 8 shows the datatypes for each parameter type.

```
typedef enum {
  VG_PATH_FORMAT       = 0x1600,
  VG_PATH_DATATYPE     = 0x1601,
  VG_PATH_SCALE        = 0x1602,
  VG_PATH_BIAS         = 0x1603,
  VG_PATH_NUM_SEGMENTS = 0x1604,
  VG_PATH_NUM_COORDS   = 0x1605
} VGPathParamType;
```

Parameter	***Datatype***
`VG_PATH_FORMAT`	`VGint`
`VG_PATH_DATATYPE`	`VGint`
`VG_PATH_SCALE`	`VGfloat`
`VG_PATH_BIAS`	`VGfloat`
`VG_PATH_NUM_SEGMENTS`	`VGint`
`VG_PATH_NUM_COORDS`	`VGint`

Table 8: VGPathParamType Datatypes

Path Format

The command format of a path is queried as an integer value using the `VG_PATH_FORMAT` parameter:

```
VGPath path;
VGint pathFormat = vgGetParameteri(path, VG_PATH_FORMAT);
```

Path Datatype

The coordinate datatype of a path is queried as an integer value using the `VG_PATH_DATATYPE` parameter. The returned integral value should be cast to the `VGPathDatatype` enumeration:

```
VGPath path;
VGPathDatatype pathDatatype =
  (VGPathDatatype)vgGetParameteri(path, VG_PATH_DATATYPE);
```

Path Scale

The scale factor of the `path` is queried as a floating-point value using the `VG_PATH_SCALE` parameter:

```
VGPath path;
VGfloat pathScale = vgGetParameterf(path, VG_PATH_SCALE);
```

Path Bias

The bias of the `path` is queried as a floating-point value using the `VG_PATH_BIAS` parameter:

```
VGPath path;
VGfloat pathBias = vgGetParameterf(path, VG_PATH_BIAS);
```

Number of Segments

The number of segments stored in the path is queried as an integer value using the `VG_PATH_NUM_SEGMENTS` parameter:

```
VGPath path;
VGint pathNumSegments = vgGetParameteri(path, VG_PATH_NUM_SEGMENTS);
```

Number of Coordinates

The total number of specified coordinates (*i.e.*, those defined in the "Coordinates" column of Table 6) stored in the path is queried as an integer value using the `VG_PATH_NUM_COORDS` parameter:

```
VGPath path;
VGint pathNumCoords = vgGetParameteri(path, VG_PATH_NUM_COORDS);
```

8.6.4 Querying and Modifying Path Capabilities

vgGetPathCapabilities

The **vgGetPathCapabilities** function returns the current capabilities of the `path`, as a bitwise OR of `VGPathCapabilities` constants. If an error occurs, 0 is returned.

```
VGbitfield vgGetPathCapabilities(VGPath path)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

vgRemovePathCapabilities

The **vgRemovePathCapabilities** function requests the set of capabilities specified in the `capabilities` argument to be disabled for the given `path`. The `capabilities` argument is a bitwise OR of the `VGPathCapabilities` values whose removal is requested. Attempting to remove a capability that is already disabled has no effect. Bits of `capabilities` that do not correspond to values from `VGPathCapabilities` have no effect.

An implementation may choose to ignore the request to remove a particular capability if no significant performance improvement would result. In this case, **vgGetPathCapabilities** will continue to report the capability as enabled.

```
void vgRemovePathCapabilities(VGPath path, VGbitfield capabilities)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

8.6.5 Copying Data Between Paths

vgAppendPath

vgAppendPath appends a copy of all path segments from `srcPath` onto the end of the existing data in `dstPath`. It is legal for `srcPath` and `dstPath` to be handles to the same path object, in which case the contents of the path are duplicated. If `srcPath` and `dstPath` are handles to distinct path objects, the contents of `srcPath` will not be affected by the call.

The `VG_PATH_CAPABILITY_APPEND_FROM` capability must be enabled for `srcPath`, and the `VG_PATH_CAPABILITY_APPEND_TO` capability must be enabled for `dstPath`.

If the scale and bias of `dstPath` define a narrower range than that of `srcPath`, overflow may occur silently.

```
void vgAppendPath(VGPath dstPath, VGPath srcPath)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if either `dstPath` or `srcPath` is not a valid path handle, or is not shared with the current context

`VG_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_APPEND_FROM` is not enabled for `srcPath`

– if `VG_PATH_CAPABILITY_APPEND_TO` is not enabled for `dstPath`

8.6.6 Appending Client-Side Data to a Path

vgAppendPathData

vgAppendPathData appends data taken from a client-side representation stored in `pathData` to the given path `dstPath`. The data are formatted using the path format of `dstPath` (as returned by querying the path's `VG_PATH_FORMAT` parameter using **vgGetParameteri**). The `numSegments` parameter gives the total number of entries in the `pathSegments` array, and must be greater than 0. Legal values for the **pathSegments** array are the values from the `VGPathCommand` enumeration as well as `VG_CLOSE_PATH` and `(VG_CLOSE_PATH | VG_RELATIVE)` (which are synonymous).

The `pathData` pointer must be aligned on a 1-, 2-, or 4-byte boundary (as defined in the "Bytes" column of Table 7) depending on the size of the coordinate datatype (as returned by querying the path's `VG_PATH_DATATYPE` parameter using **vgGetParameteri**). The `VG_PATH_CAPABILITY_APPEND_TO` capability must be enabled for `path`.

Each incoming coordinate value, regardless of datatype, is transformed by the scale factor and bias of the path.

```
void vgAppendPathData(VGPath dstPath,
                      VGint numSegments,
                      const VGubyte * pathSegments,
                      const void * pathData)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `dstPath` is not a valid path handle, or is not shared with the current context

`VG_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_APPEND_TO` is not enabled for `dstPath`

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `pathSegments` or `pathData` is `NULL`
– if `pathData` is not properly aligned
– if `numSegments` is less than or equal to 0
– if `pathSegments` contains an illegal command

8.6.7 Modifying Path Data

Coordinate data in an existing path may be modified, for example to create animation effects. Implementations should choose an internal representation for paths that have the `VG_PATH_CAPABILITY_MODIFY` capability enabled that allows for efficient modification of the coordinate data.

vgModifyPathCoords

vgModifyPathCoords modifies the coordinate data for a contiguous range of segments of `dstPath`, starting at `startIndex` (where 0 is the index of the first path segment) and having length `numSegments`. The data in `pathData` must be formatted in exactly the same manner as the original coordinate data for the given segment range, unless the path has been transformed using **vgTransformPath** or interpolated using **vgInterpolatePath**. In these cases, the path will have been subject to the segment promotion rules as specified in those functions.

The `pathData` pointer must be aligned on a 1-, 2-, or 4-byte boundary depending on the size of the coordinate datatype (as returned by querying the path's `VG_PATH_DATATYPE` parameter using **vgGetParameteri**). The `VG_PATH_CAPABILITY_MODIFY` capability must be enabled for `path`.

Each incoming coordinate value, regardless of datatype, is transformed by the scale factor and bias of the path.

```
void vgModifyPathCoords(VGPath dstPath,
                        VGint startIndex, VGint numSegments,
                        const void * pathData)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `dstPath` is not a valid path handle, or is not shared with the current context

`VG_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_MODIFY` is not enabled for `dstPath`

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `pathData` is `NULL`

– if `pathData` is not properly aligned

– if `startIndex` is less than 0

– if `numSegments` is less than or equal to 0

– if `startIndex + numSegments` is greater than the number of segments in the path

8.6.8 Transforming a Path

vgTransformPath

vgTransformPath appends a transformed copy of `srcPath` to the current contents of `dstPath`. The appended path is equivalent to the results of applying the current path-user-to-surface transformation (`VG_MATRIX_PATH_USER_TO_SURFACE`) to `srcPath`.

It is legal for `srcPath` and `dstPath` to be handles to the same path object, in which case the the transformed path will be appended to the existing path. If `srcPath` and `dstPath` are handles to distinct path objects, the contents of `srcPath` will not be affected by the call.

All `HLINE_TO_*` and `VLINE_TO_*` segments present in `srcPath` are implicitly converted to `LINE_TO_*` segments prior to applying the transformation. The original copies of these segments in `srcPath` remain unchanged.

Any `*ARC_TO` segments are transformed, but the endpoint parametrization of the resulting arc segments are implementation-dependent. The results of calling **vgInterpolatePath** on a transformed path that contains such segments are undefined.

The `VG_PATH_CAPABILITY_TRANSFORM_FROM` capability must be enabled for `srcPath`, and the `VG_PATH_CAPABILITY_TRANSFORM_TO` capability must be enabled for `dstPath`.

Overflow may occur silently if coordinates are transformed outside the datatype range of `dstPath`.

```
void vgTransformPath(VGPath dstPath, VGPath srcPath)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if either `dstPath` or `srcPath` is not a valid path handle, or is not shared with the current context

`VG_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_TRANSFORM_FROM` is not enabled for `srcPath`

– if `VG_PATH_CAPABILITY_TRANSFORM_TO` is not enabled for `dstPath`

8.6.9 Interpolating Between Paths

Interpolation takes two compatible paths, in a sense described below, and defines a new path that interpolates between them by a parameter `amount`. When `amount` is equal to 0, the result is equivalent to the first path; when `amount` is equal to 1, the result is equivalent to the second path. Values between 0 and 1 produce paths that smoothly interpolate between the two extremes. Values outside the [0, 1] range produce extrapolated paths. Conceptually, interpolation occurs as follows. First, the two path parameters are copied and the copies are normalized by:

- Converting all coordinates to floating-point format, applying the path scale and bias parameters
- Converting all relative segments to absolute form
- Converting {H,V}LINE_TO_* segments to LINE_TO form
- Converting (S)QUAD_TO_*/SCUBIC_TO_* segments to CUBIC_TO form
- Retaining all *ARC_TO_* and CLOSE_PATH segments

If, following normalization, both paths have the same sequence of segment types (treating all forms of arc as the same), interpolation proceeds by linearly interpolating between each corresponding pair of segment parameters in the normalized paths. If the starting arc type differs from the final arc type, the starting arc type is used for values of `amount` less than 0.5, and the final arc type is used for values greater than or equal to 0.5. Finally, the coordinates are converted to the data type of the destination.

vgInterpolatePath

The **vgInterpolatePath** function appends a path, defined by interpolation (or extrapolation) between the paths `startPath` and `endPath` by the given `amount`, to the path `dstPath`. It returns `VG_TRUE` if interpolation was successful (*i.e.*, the paths had compatible segment types after normalization), and `VG_FALSE` otherwise. If interpolation is unsuccessful, `dstPath` is left unchanged.

It is legal for dst`Path` to be a handle to the same path object as either `startPath` or `endPath` or both, in which case the contents of the source path or paths referenced by `dstPath` will have the interpolated path appended. If `dstPath` is not the a handle to the same path object as either `startPath` or `endPath`, the contents of `startPath` and `endPath` will not be affected by the call.

Overflow may occur silently if the datatype of `dstPath` has insufficient range to store an interpolated coordinate value.

The `VG_PATH_CAPABILITY_INTERPOLATE_FROM` capability must be enabled for both of `startPath` and `endPath`, and the `VG_PATH_CAPABILITY_INTERPOLATE_TO` capability must be enabled for `dstPath`.

```
VGboolean vgInterpolatePath(VGPath dstPath,
                            VGPath startPath,
                            VGPath endPath,
                            VGfloat amount)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if any of `dstPath`, `startPath`, or `endPath` is not a valid path handle, or is not shared with the current context

`VG_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_PATH_INTERPOLATE_TO` is not enabled for `dstPath`

– if `VG_PATH_CAPABILITY_PATH_INTERPOLATE_FROM` is not enabled for `startPath` or `endPath`

8.6.10 Length of a Path

An approximation to the geometric length of a portion of a path may be obtained by calling the **vgPathLength** function. `MOVE_TO` segments and implicit path closures (see Section 8.7.1) do not contribute to the path length. `CLOSE_PATH` segments have the same length as a `LINE_TO` segment with the same endpoints.

vgPathLength

The **vgPathLength** function returns the length of a given portion of a path in the user coordinate system (that is, in the path's own coordinate system, disregarding any matrix settings). Only the subpath consisting of the `numSegments` path segments beginning with `startSegment` (where the initial path segment has index 0) is used. If an error occurs, -1.0f is returned.

The `VG_PATH_CAPABILITY_PATH_LENGTH` capability must be enabled for `path`.

```
VGfloat vgPathLength(VGPath path,
                     VGint startSegment, VGint numSegments);
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

`VG_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_PATH_LENGTH` is not enabled for `path`

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `startSegment` is less than 0 or greater than the index of the final path segment

– if `numSegments` is less than or equal to 0

– if (`startSegment` + `numSegments` – 1) is less than 0 or greater than the index of the final path segment

8.6.11 Position and Tangent Along a Path

Some path operations, such as the placement and orientation of text along a path, require the computation of a set of points along a path as well as a normal (perpendicular) vector at each point. The **vgPointAlongPath** function provides points along the path as well as normalized tangent vectors (from which normals may easily be derived).

The Tangents of a Path Segment

The tangent at a given point along a path is defined as a vector pointing in the same direction as the path at that point. The tangent at any point of a line segment is parallel to the line segment; the tangent at any point along a Bézier curve or elliptical arc segment may be defined using the derivatives of the parametric equations $x(t)$ and $y(t)$ that define the curve. The incoming tangent at a point is defined using the direction in which the curve is "traveling" prior to arriving at the point; the outgoing tangent is defined using the direction the curve is traveling as it leaves the point. The incoming and outgoing tangents may differ at a vertex joining different curve segments, or at a sharp "cusp" in a curve.

If a point along a path segment has no tangent defined, for example where a path segment has collapsed to a single point, the following algorithm is used to define incoming and outgoing tangents at the point. Search backwards until a segment is found with a tangent defined at its end point, or the start of the current path is reached; if a tangent is found, use it as the incoming tangent. Search forwards until a segment is found with a tangent defined at its starting point, or the end of the current path is reached; if a tangent is found, use it as the outgoing tangent. If these searches produce exactly one defined tangent, that tangent is used as both the incoming and outgoing tangent. If the searches produced no defined tangent, the incoming and outgoing tangents are both assigned the value (1, 0). Tangent vectors are normalized to have unit length.

vgPointAlongPath

The **vgPointAlongPath** function returns the point lying a given distance along a given portion of a path and the unit-length tangent vector at that point. Only the subpath consisting of the `numSegments` path segments beginning with `startSegment` (where the initial path segment has index 0) is used. For the remainder of this section we refer only to this subpath when discussing paths.

If `distance` is less than or equal to 0, the starting point of the path is used. If `distance` is greater than or equal to the path length (*i.e.*, the value returned by **vgPathLength** when called with the same `startSegment` and `numSegments` parameters), the visual ending point of the path is used.

Intermediate values return the (*x, y*) coordinates and tangent vector of the point at the given distance along the path. Because it is not possible in general to compute exact distances along a path, an implementation is not required to use exact computation even for segments where such computation would be possible. For example, the path:

```
MOVE_TO  0,  0; LINE_TO 10,  0 // draw a line of length 10
MOVE_TO 10, 10 // create a discontinuity
LINE_TO 10, 20 // draw a line of length 10
```

may return either (10, 0) or (10, 10) (or points nearby) as the point at distance 10.0. Implementations are not required to compute distances exactly, as long as they satisfy the constraint that as `distance` increases monotonically the returned point and tangent move forward monotonically along the path.

Where the implementation is able to determine that the point being queried lies exactly at a discontinuity or cusp, the incoming point and tangent should be returned. In the example above, returning the pre-discontinuity point (10, 0) and incoming tangent (1, 0) is preferred to returning the post-discontinuity point (10, 10) and outgoing tangent (0, 1).

The `VG_PATH_CAPABILITY_POINT_ALONG_PATH` capability must be enabled for `path`.

If the reference arguments `x` and `y` are both non-`NULL`, and the `VG_PATH_CAPABILITY_POINT_ALONG_PATH` capability is enabled for `path`, the point (*x, y*) is returned in `x` and `y`. Otherwise the variables referenced by `x` and `y` are not written.

If the reference arguments `tangentX` and `tangentY` are both non-`NULL`, and the `VG_PATH_CAPABILITY_TANGENT_ALONG_PATH` capability is enabled for `path`, the geometric tangent vector at the point (*x, y*) is returned in `tangentX` and `tangentY`. Otherwise the variables referenced by `tangentX` and `tangentY` are not written.

Where the incoming tangent is defined, **vgPointAlongPath** returns it. Where only the outgoing tangent is defined, the outgoing tangent is returned.

The points returned by **vgPointAlongPath** are not guaranteed to match the path as rendered; some deviation is to be expected.

```
void vgPointAlongPath(VGPath path,
                      VGint startSegment, VGint numSegments,
                      VGfloat distance,
                      VGfloat * x, VGfloat * y,
                      VGfloat * tangentX, VGfloat * tangentY)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

`VG_PATH_CAPABILITY_ERROR`

– If `x` and `y` are both non-`NULL`, and the `VG_PATH_CAPABILITY_POINT_ALONG_PATH` is not enabled for `path`

– If `tangentX` and `tangentY` are both non-`NULL`, and the `VG_PATH_CAPABILITY_TANGENT_ALONG_PATH` capability is not enabled for `path`

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `startSegment` is less than 0 or greater than the index of the final path segment

– if `numSegments` is less than or equal to 0

– if (`startSegment` + `numSegments` – 1) is less than 0 or greater than the index of the final path segment

– if `x`, `y`, `tangentX` or `tangentY` is not properly aligned

8.6.12 Querying the Bounding Box of a Path

To draw complex scenes efficiently, it is important to avoid drawing objects that do not appear in the region being drawn. A simple way to determine whether an object may be visible is to determine whether its *bounding box* – an axis-aligned rectangle that is guaranteed to contain the entire object – intersects the drawn region. The **vgPathBounds** and **vgPathTransformedBounds** functions provide bounding box information.

Two types of bounding boxes may be obtained for a path. The first, obtained by calling **vgPathBounds**, returns a tight axis-aligned bounding box for the area contained within the path in its own coordinate system. The second, obtained by calling **vgPathTransformedBounds**, returns an axis-aligned bounding box for the path as it will appear when drawn on the drawing surface (*i.e.*, following application of the current path-user-to-surface transform). The latter function does not guarantee to bound the shape tightly, but still may provide tighter bounds than those obtained by transforming the result of **vgPathBounds**, at a lower cost.

The bounding box of a path is defined to contain the area within the path, *i.e.*, the area that would be drawn if the path were to be filled. If the path is to be stroked, the application must adjust the bounding box to take the stroking parameters into account. Note that Miter joins in particular may extend far outside the bounding box.

vgPathBounds

The **vgPathBounds** function returns an axis-aligned bounding box that tightly bounds the interior of the given `path`. Stroking parameters are ignored. If `path` is empty, `minX` and `minY` are set to 0 and `width` and `height` are set to -1. If `path` contains a single point, `minX` and `minY` are set to the coordinates of the point and `width` and `height` are set to 0.

The `VG_PATH_CAPABILITY_PATH_BOUNDS` capability must be enabled for `path`.

```
void vgPathBounds(VGPath path,
                  VGfloat * minX, VGfloat * minY,
                  VGfloat * width, VGfloat * height)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `minX`, `minY`, `width`, or `height` is `NULL`

– if `minX`, `minY`, `width`, or `height` is not properly aligned

`VG_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_PATH_BOUNDS` is not enabled for `path`

vgPathTransformedBounds

The **vgPathTransformedBounds** function returns an axis-aligned bounding box that is guaranteed to enclose the geometry of the given `path` following transformation by the current path-user-to-surface transform. The returned bounding box is not guaranteed to fit tightly around the path geometry. If `path` is empty, minX and minY are set to 0 and width and height are set to -1. If path contains a single point, minX and minY are set to the transformed coordinates of the point and width and height are set to 0.

The VG_PATH_CAPABILITY_PATH_TRANSFORMED_BOUNDS capability must be enabled for path.

```
void vgPathTransformedBounds(VGPath path,
                             VGfloat * minX, VGfloat * minY,
                             VGfloat * width, VGfloat * height)
```

ERRORS

VG_BAD_HANDLE_ERROR

– if path is not a valid path handle, or is not shared with the current context

VG_ILLEGAL_ARGUMENT_ERROR

– if minX, minY, width, or height is NULL

– if minX, minY, width, or height is not properly aligned

VG_PATH_CAPABILITY_ERROR

– if VG_PATH_CAPABILITY_PATH_TRANSFORMED_BOUNDS is not enabled for path

8.7 Interpretation of Paths

The interpretation of a path, composed of a sequence of one or more subpaths, depends on whether it is to be stroked or filled. For stroked paths, each subpath has stroking parameters applied to it separately, with the dash phase at the end of each subpath used at the beginning of the next subpath. This process results in a set of stroked shapes. The union of these shapes then defines the outline path to be filled. For filled paths, the interior of the path (as defined below) is filled.

8.7.1 Filling Paths

A simple, non-self-intersecting closed path divides the plane into two regions, a bounded *inside* region and an unbounded *outside* region. Note that knowing the orientation of the outermost path (*i.e.*, clockwise or counter-clockwise) is not necessary to differentiate between the inside and outside regions.

A path that self-intersects, or that has multiple overlapping subpaths, requires additional information in order to define the inside region. Two rules that provide different definitions for the area enclosed by such paths, known as the non-zero and even/odd fill rules, are supported by OpenVG. To determine whether any point in the plane is contained in the inside region, imagine drawing a line from that point out to infinity in any direction such that the line does not cross any vertex of the path. For each edge that is crossed by the line, add 1 to the counter if the edge crosses from left to right, as seen by an observer walking along the line towards infinity, and subtract 1 if the edge crosses from right to left. In this way, each region of the plane will receive an integer value.

The non-zero fill rule says that the point is inside the shape if the resulting sum is not equal to 0. The even/odd rule says that the point is inside the shape if the resulting sum is odd, regardless of sign (*e.g.*, -7 is odd, 0 is even). Consider the star-shaped path shown in Figure 8 below, indicated with solid lines. The orientation of the lines making up the path is indicated with arrows. An imaginary line to infinity starting in the central region of the star is shown as a dashed line pointing to the right. Two edges of the star cross the line to infinity going left to right, indicated by the downward-pointing arrows. The central region therefore has a count of +2. According to the even/odd rule, it is outside the path, whereas according to the non-zero rule it is inside. Implementations must be able to deal with paths having up to 255 crossings along any line. The behavior of more complex paths is undefined.

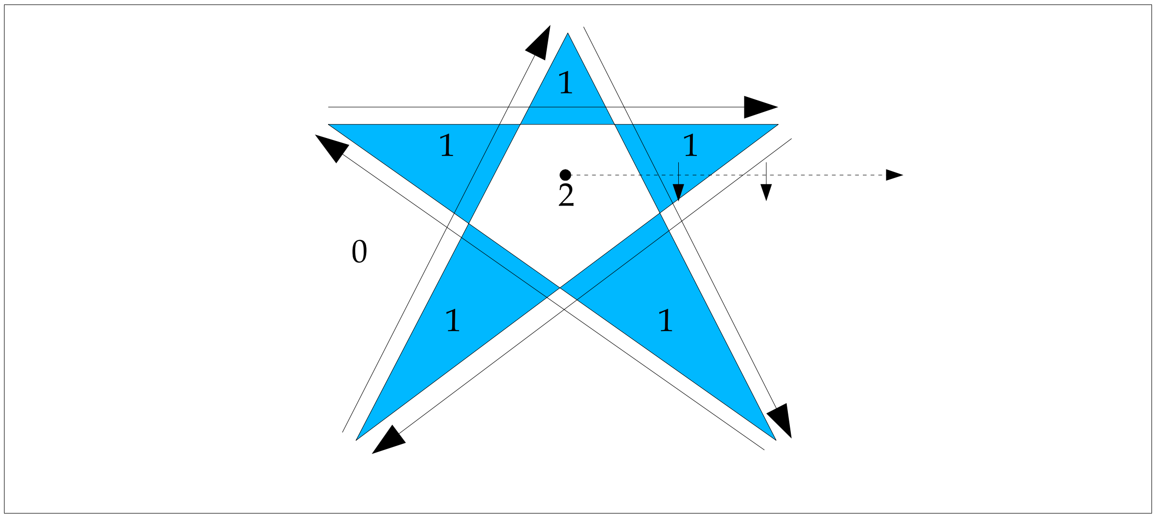


Figure 8: Even/Odd Fill Rule

Creating Holes in Paths

The fill rule is applied with respect to all subpaths simultaneously during filling. Thus, one subpath may be used to create a hole inside an enclosing subpath by defining the two subpaths with opposing orientations (clockwise versus counter-clockwise). Note that the orientation of extremely small paths may depend on the numerical precision of the internal representation of points. Care should be taken to avoid the use of paths that have nearly collapsed to a line or a point.

The relative orientation of subpaths, along with the fill rule, determines whether overlapping subpaths will result in holes, as shown in Figure 9 below.

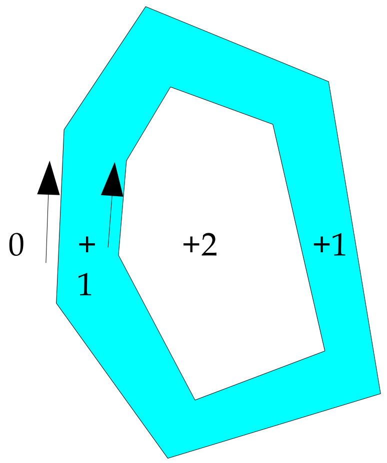
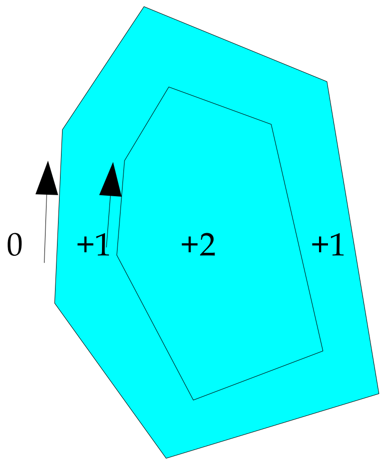
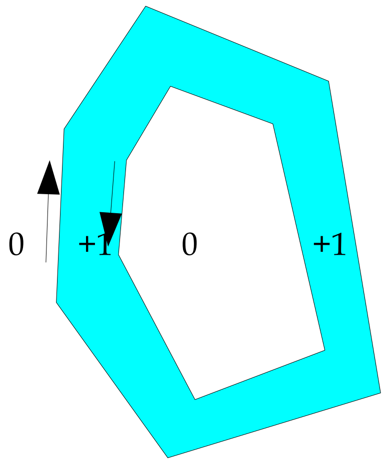
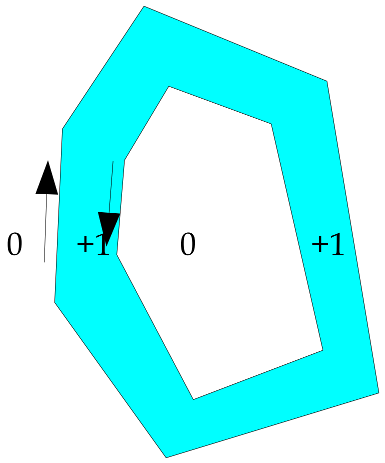
	Even/Odd Fill Rule	***Non-Zero Fill Rule***
Same Orientation		
Opposing Orientation		

Figure 9: Creating Holes with Subpaths

Implicit Closure of Filled Subpaths

When filling a path, any subpaths that do not end with a `CLOSE_PATH` segment command (*i.e.*, they are terminated with a `MOVE_TO_ABS` or `MOVE_TO_REL` segment command, or they contain the final segment of the path) are implicitly closed, without affecting the position of any other vertices of the path or the (*sx, sy*), (*px, py*) or (*ox, oy*) variables. For example, consider the sequence of segment commands:

```
MOVE_TO_ABS    0,  0; LINE_TO_ABS   10, 10; LINE_TO_ABS   10,  0
MOVE_TO_REL   10,  2; LINE_TO_ABS   30, 12; LINE_TO_ABS   30,  2
```

If filled, this sequence will result in one filled triangle with vertices (*0, 0*), (*10, 10*), and (*10, 0*) and another filled triangle with vertices (*20, 2*), (*30, 12*), and (*30, 2*). Note that the implicit closure of the initial subpath prior to the `MOVE_TO_REL` segment command has no effect on the starting coordinate of the second triangle; it is computed by adding the relative offset (*10, 2*) to the final coordinate of the previous segment (*10, 0*) to obtain (*20, 2*) and is not altered by the (virtual) insertion of the line connecting the first subpath's final vertex (*10, 0*) to its initial vertex (*0, 0*)). Figure 10 illustrates this process, with the resulting filled areas highlighted. When stroking a path, no implicit closure takes place, as shown in Figure 11. Implicit closure affects only the output when filling a path, and does not alter the path data in any way.

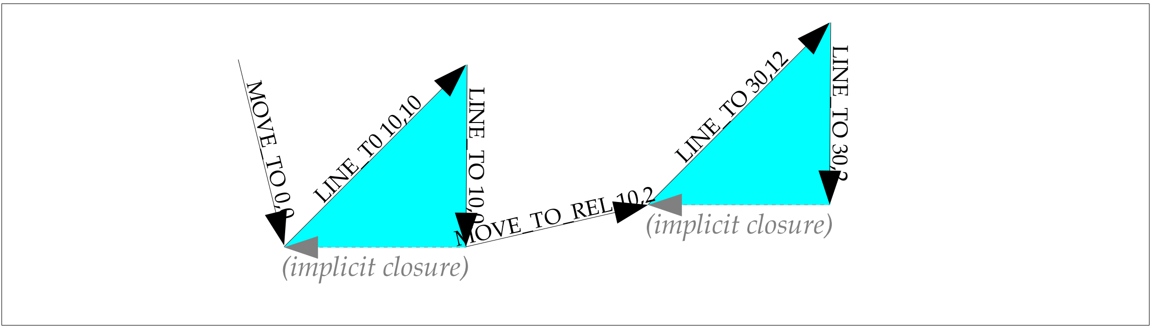


Figure 10: Implicit Closure of Filled Paths

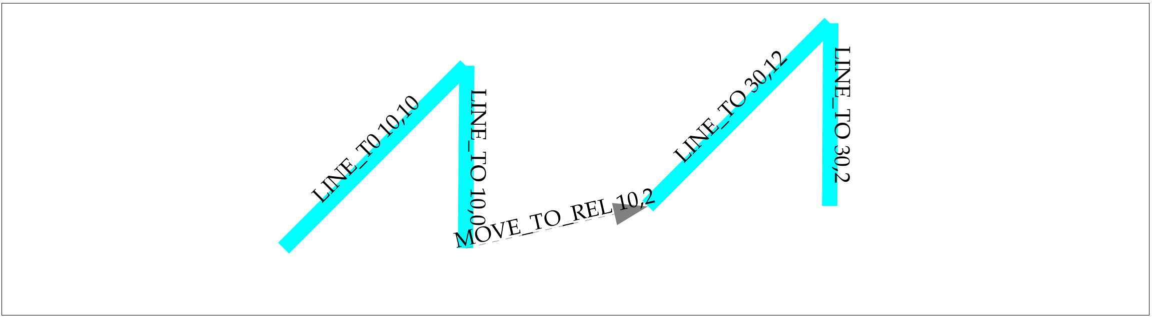


Figure 11: Stroked Paths Have No Implicit Closure

8.7.2 Stroking Paths

Stroking a path consists of "widening" the edges of the path using a straight-line pen held perpendicularly to the path. At the start and end vertices of the path, an additional end-cap style is applied. At interior vertices of the path, a line join style is applied.

Conceptually, stroking of a path is performed in two steps. First, the stroke parameters are applied in the user coordinate system to form a new shape representing the end result of dashing, widening the path, and applying the end cap and line join styles. Second, a path is created that defines the outline of this stroked shape. This path is transformed using the path-user-to-surface transformation (possibly involving shape distortions due to non-uniform scaling or shearing). Finally, the resulting path is filled with paint in exactly the same manner as when filling a user-defined path using the non-zero fill rule.

Stroking a path applies a single "layer" of paint, regardless of any intersections between portions of the thickened path. Figure 12 illustrates this principle. A single stroke (above) is drawn with a black color and an alpha value of 50%, compared with two separate strokes (below) drawn with the same color and alpha values. The single stroke produces a shape with a uniform color of 50% gray, as if a single layer of translucent paint has been applied, even where portions of the path overlap one another. By contrast, the separate strokes produce two applications of the translucent paint in the area of overlap, resulting in a darkened area.

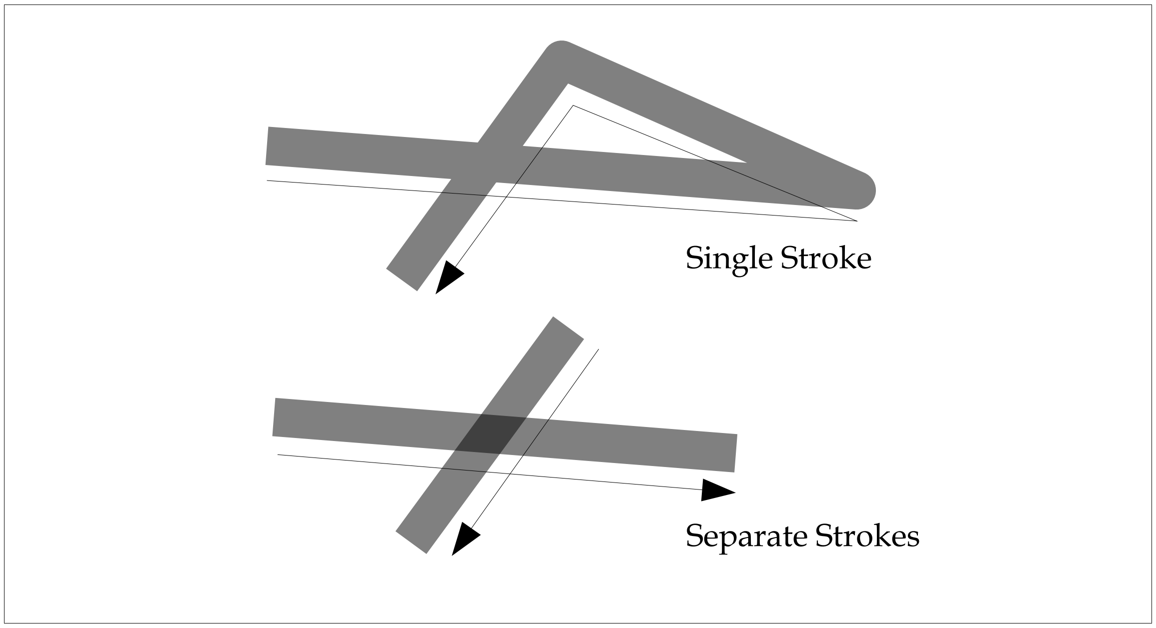


Figure 12: Each Stroke Applies a Single Layer of Paint

8.7.3 Stroke Parameters

Stroking a path involves the following parameters, set on a context:

- Line width in user coordinate system units
- End cap style – one of Butt, Round, or Square
- Line join style – one of Miter, Round, or Bevel
- Miter limit – if using Miter join style
- Dash pattern – array of dash on/off lengths in user units
- Dash phase – initial offset into the dash pattern

These parameters are set on the current context using the variants of the **vgSet** function. The values most recently set prior to calling **vgDrawPath** (see Section 8.8) are applied to generate the stroke.

End Cap Styles

Figure 13 illustrates the Butt (top), Round (center), and Square (bottom) end cap styles applied to a path consisting of a single line segment. Figure 14 highlights the additional geometry created by the end caps. The Butt end cap style terminates each segment with a line perpendicular to the tangent at each endpoint. The Round end cap style appends a semicircle with a diameter equal to the line width centered around each endpoint. The Square end cap style appends a rectangle with two sides of length equal to the line width perpendicular to the tangent, and two sides of length equal to half the line width parallel to the tangent, at each endpoint. The outgoing tangent is used at the left endpoint and the incoming tangent is used at the right endpoint.

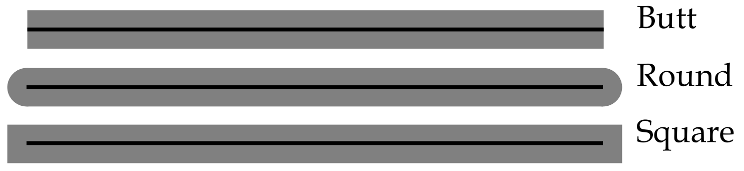


Figure 13: End Cap Styles

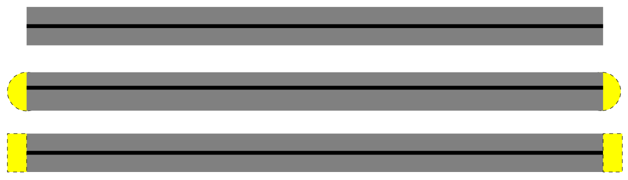


Figure 14: End Cap Styles with Additional Geometry Highlighted

Line Join Styles

Figure 15 illustrates the Bevel (left), Round (center), and Miter (right) line join styles applied to a pair of line segments. Figure 16 highlights the additional geometry created by the line joins. The Bevel join style appends a triangle with two vertices at the outer endpoints of the two "fattened" lines and a third vertex at the intersection point of the two original lines. The Round join style appends a wedge-shaped portion of a circle, centered at the intersection point of the two original lines, having a radius equal to half the line width. The Miter join style appends a trapezoid with one vertex at the intersection point of the two original lines, two adjacent vertices at the outer endpoints of the two "fattened" lines and a fourth vertex at the extrapolated intersection point of the outer perimeters of the two "fattened" lines. A Round join is used at a cusp of a cubic Bézier segment.

When stroking using the Miter join style, the *miter length* (*i.e.*, the length between the intersection points of the inner and outer perimeters of the two "fattened" lines) is compared to the product of the user-set miter limit and the line width. If the miter length exceeds this product, the Miter join is not drawn and a Bevel join is substituted.

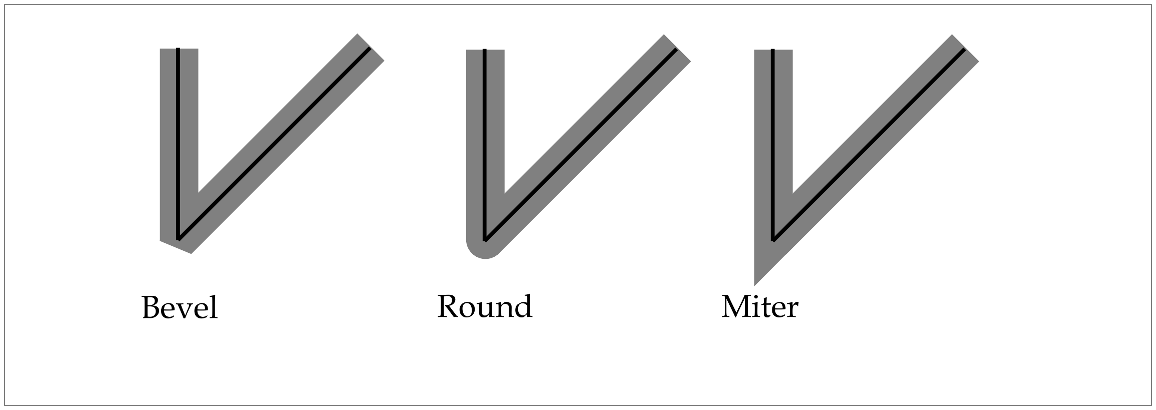


Figure 15: Line Join Styles

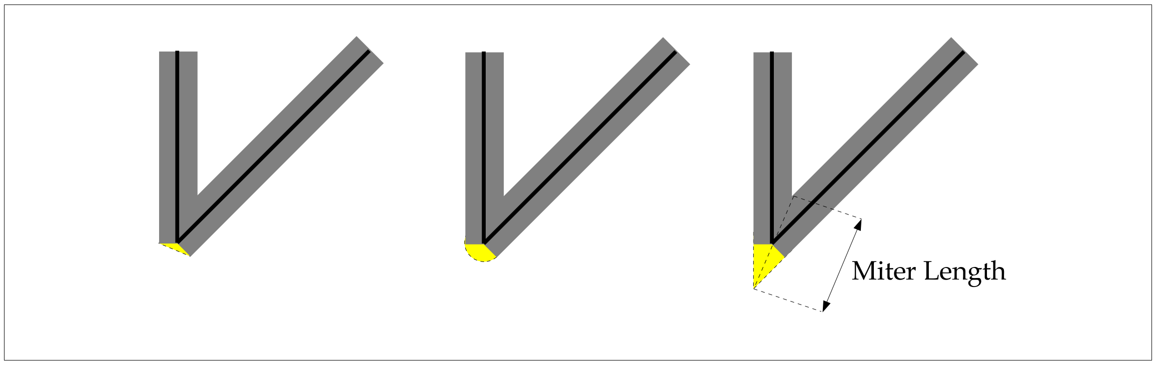


Figure 16: Line Join Styles with Additional Geometry Highlighted

Miter Length

The ratio of miter length to line width may be computed directly from the angle θ between the two line segments being joined as *1/sin*(*θ/2*). A number of angles with their corresponding miter limits for a line width of 1 are shown in Table 9.

Angle (degrees)	*Miter Limit*	*Angle (degrees)*	*Miter Limit*
10	11.47	45	2.61
11.47	10	60	2
23	5	90	1.41
28.95	4	120	1.15
30	3.86	150	1.03
38.94	3	180	1

Table 9: Corresponding Angles and Miter Limits

Dashing

The dash pattern consists of a sequence of lengths of alternating "on" and "off" dash segments. The first value of the dash array defines the length, in user coordinates, of the first "on" dash segment. The second value defines the length of the following "off" segment. Each subsequent pair of values defines one "on" and one "off" segment.

The dash phase defines the starting point in the dash pattern that is associated with the start of the first segment of the path. For example, if the dash pattern is [10 20 30 40] and the dash phase is 35, the path will be stroked with an "on" segment of length 25 (skipping the first "on" segment of length 10, the following "off" segment of length 20, and the first 5 units of the next "on" segment), followed by an "off" segment of length 40. The pattern will then repeat from the beginning, with an "on" segment of length 10, an "off" segment of length 20, an "on" segment of length 30, etc. Figure 17 illustrates this dash pattern.

Conceptually, dashing is performed by breaking the path into a set of subpaths according to the dash pattern. Each subpath is then drawn independently using the end cap, line join style, and miter limit that were set for the path as a whole.

Dashes of length 0 are drawn only if the end cap style is `VG_CAP_ROUND` or `VG_CAP_SQUARE`. The incoming and outgoing tangents (which may differ if the dash falls at a vertex of the path) are evaluated at the point, using the **vgPointAlongPath** algorithm. The end caps are drawn using the orientation of each tangent, and a join is drawn between them if the tangent directions differ. If the end cap style is `VG_CAP_BUTT`, nothing will be drawn.

A dash, or space between dashes, with length less than 0 is treated as having a length of 0.

A negative dash phase is equivalent to the positive phase obtained by adding a suitable multiple of the dash pattern length.

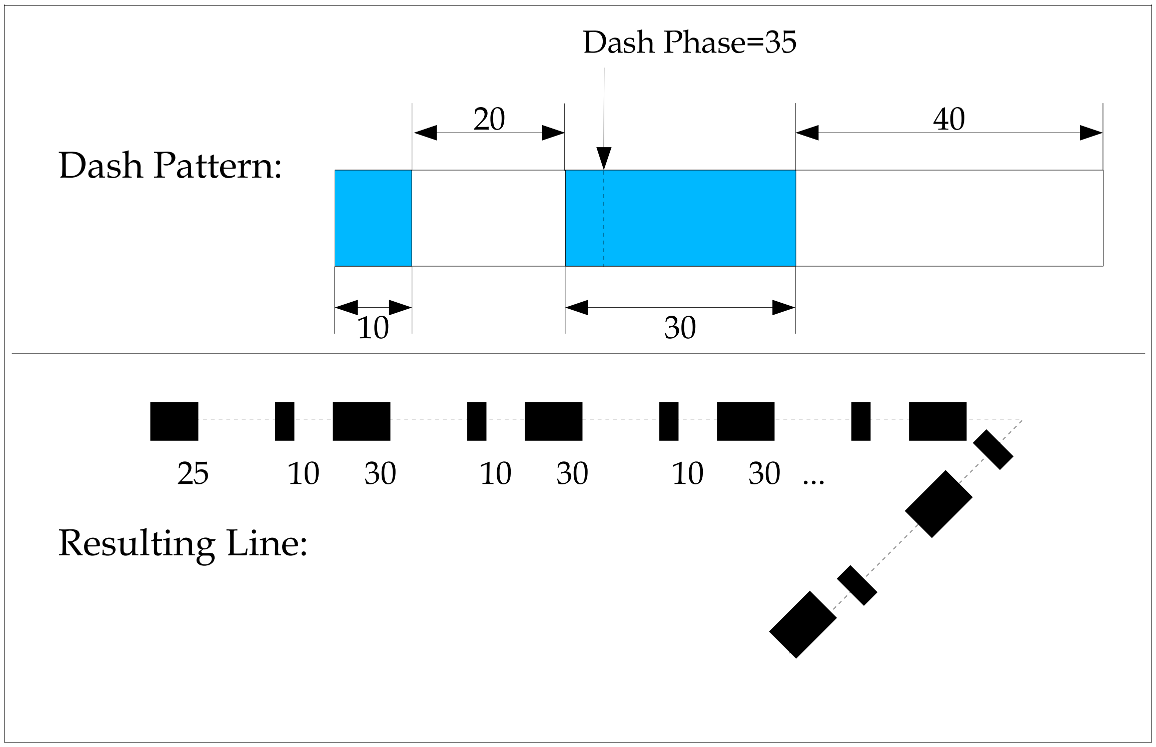


Figure 17: Dash Pattern and Phase Example

8.7.4 Stroke Generation

The algorithm for generating a stroke is as follows. The steps described in this section conceptually take place in user coordinates, on a copy of the path being stroked in which all relative and implicit coordinates have been converted to absolute coordinates. An initial `MOVE_TO 0,0` segment is added if the path does not begin with a `MOVE_TO`.

The path to be stroked is divided into subpaths, each ending with a `MOVE_TO` or `CLOSE_PATH` segment command or with the final path segment. Subpaths consisting of only a single `MOVE_TO` segment are discarded.

A subpath consisting of a single point (*i.e.*, a `MOVE_TO` segment followed by a sequence of `LINE_TO`, `QUAD_TO`, `CUBIC_TO`, and/or `ARC_TO` segments with all control points equal to the current point, possibly followed by a `CLOSE_PATH` segment) is collapsed to a lone vertex, which is marked as an END vertex (for later generation of end caps). A tangent vector of (1, 0) is used for Square end caps.

Subpaths that do not consist only of a single point have any zero-length segments removed.

If a subpath does not end with a `CLOSE_PATH` segment command, its first and last vertices are marked as END vertices. All the internal vertices that begin or end path segments within the subpath, as well as the initial/final vertex if the subpath ends with a `CLOSE_PATH` segment, are marked as JOIN vertices (for later generation of line joins).

Each subpath is processed in turn as described below until all subpaths have been stroked.

If dashing is enabled, the dash pattern and phase are used to break the subpath into a series of smaller subpaths representing the "on" portions of the dash pattern. New vertices are created at the endpoints of each dash subpath and marked as END vertices. The old subpath is discarded and replaced with the dash subpaths for the remainder of the stroke processing. The dash phase is advanced for each subsequent segment by the length of the previous segment (where `CLOSE_PATH` segments are treated as `LINE_TO` segments). If `VG_DASH_PHASE_RESET` is disabled (set to `VG_FALSE`), the final dash phase at the end of the subpath is used as the initial dash phase for the next subpath. Otherwise, the original dash phase is used for all subpaths.

For each END vertex, an end cap is created (if Square or Round end caps have been requested) using the orientation given by the tangent vector. The tangent vector is defined in the same manner as for the **vgPointAlongPath** function (see p. 71).

For each JOIN vertex, a line join is created using the orientations given by the tangent vectors of the two adjacent path segments. If Miter joins are being used, the length of the miter is computed and compared to the product of the line width and miter limit; if the miter would be too long, a Bevel join is substituted.

8.7.5 Setting Stroke Parameters

Setting the line width of a stroke is performed using **vgSetf** with a `paramType` argument of `VG_STROKE_LINE_WIDTH`**.** A line width less than or equal to 0 prevents stroking from taking place.

```
VGfloat lineWidth;
vgSetf(VG_STROKE_LINE_WIDTH, lineWidth);
```

VGCapStyle

The `VGCapStyle` enumeration defines constants for the Butt, Round, and Square end cap styles:

```
typedef enum {
  VG_CAP_BUTT   = 0x1700,
  VG_CAP_ROUND  = 0x1701,
  VG_CAP_SQUARE = 0x1702
} VGCapStyle;
```

Setting the end cap style is performed using **vgSeti** with a `paramType` argument of `VG_STROKE_CAP_STYLE` and a value from the `VGCapStyle` enumeration.

```
VGCapStyle capStyle;
vgSeti(VG_STROKE_CAP_STYLE, capStyle);
```

VGJoinStyle

The `VGJoinStyle` enumeration defines constants for the Miter, Round, and Bevel line join styles:

```
typedef enum {
  VG_JOIN_MITER = 0x1800,
  VG_JOIN_ROUND = 0x1801,
  VG_JOIN_BEVEL = 0x1802
} VGJoinStyle;
```

Setting the line join style is performed using **vgSeti** with a `paramType` argument of `VG_STROKE_JOIN_STYLE` and a value from the `VGJoinStyle` enum.

```
VGJoinStyle joinStyle;
vgSeti(VG_STROKE_JOIN_STYLE, joinStyle);
```

Setting the miter limit is performed using **vgSetf** with a `paramType` argument of `VG_STROKE_MITER_LIMIT`:

```
VGfloat miterLimit;
vgSetf(VG_STROKE_MITER_LIMIT, miterLimit);
```

Miter limit values less than 1 are silently clamped to 1.

VG_MAX_DASH_COUNT

The `VG_MAX_DASH_COUNT` parameter contains the maximum number of dash segments that may be supplied for the `VG_STROKE_DASH_PATTERN` parameter. All implementations must must support at least 16 dash segments (8 on/off pairs). If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. The value may be retrieved by calling **vgGeti**:

```
VGint maxDashCount = vgGeti(VG_MAX_DASH_COUNT);
```

Setting the Dash Pattern

The dash pattern is set using **vgSetfv** with a `paramType` argument of `VG_STROKE_DASH_PATTERN`:

```
VGfloat dashPattern[DASH_COUNT];
VGint count = DASH_COUNT;
vgSetfv(VG_STROKE_DASH_PATTERN, count, dashPattern);
```

Dashing may be disabled by calling **vgSetfv** with a `count` of 0:

```
vgSetfv(VG_STROKE_DASH_PATTERN, 0, NULL);
```

The dash phase is set using **vgSetf** with a `paramType` argument of `VG_STROKE_DASH_PHASE`. The resetting behavior of the dash phase when advancing to a new subpath is set using **vgSeti** with a `paramType` argument of `VG_STROKE_DASH_PHASE_RESET`:

```
VGfloat dashPhase;
VGboolean dashPhaseReset;
vgSetf(VG_STROKE_DASH_PHASE, dashPhase);
vgSeti(VG_STROKE_DASH_PHASE_RESET, dashPhaseReset);
```

If the dash pattern has length 0, dashing is not performed. If the dash pattern has an odd number of elements, the final element is ignored. Note that this behavior is different from that defined by SVG; the SVG behavior may be implemented by duplicating the odd-length dash pattern to obtain one with even length.

If more than `VG_MAX_DASH_COUNT` dashes are specified, those beyond the first `VG_MAX_DASH_COUNT` are discarded immediately (and will not be returned by **vgGet)**.

8.7.6 Non-Scaling Strokes

In some cases, applications may wish stroked geometry to appear with a particular stroke width in the surface coordinate system, independent of the current user-to-surface transformation. For example, a stroke representing a road on a map might stay the same width as the user zooms in and out of the map, since the stroke width is intended to indicate the type of road (*e.g.*, one-way street, divided road, interstate highway or Autobahn) rather than its true width on the ground.

OpenVG does not provide direct support for this “non-scaling stroke” behavior. However, the behavior may be obtained relatively simply using a combination of features.

If the current user-to-surface transformation consists only of uniform scaling, rotation, and translation (*i.e.*, no shearing or non-uniform scaling), then the stroke width may be set to the desired stroke width in drawing surface coordinates, divided by the scaling factor introduced by the transformation. This scaling factor may be known to the application *a priori*, or else it may be computed as the square root of the absolute value of the determinant (*sx*sy – shx*shy*) of the user-to-surface transformation.

If the user-to-surface transformation includes shearing or non-uniform scaling, the geometry to be stroked must be transformed into surface coordinates prior to stroking. The paint transformation must also be set to the concatenation of the paint-to-user and user-to-surface transformations in order to allow correct painting of the stroked geometry. The following code illustrates this technique:

```
VGPath srcPath; /* Path to be drawn with non-scaling stroke */
VGPath dstPath; /* Path in drawing surface coordinates */
VGfloat strokePaintToUser[9]; /* Paint-to-user transformation */
VGfloat pathUserToSurface[9]; /* User-to-surface transformation */

/* Transform the geometry into surface coordinates. */
vgMatrixMode(VG_MATRIX_PATH_USER_TO_SURFACE);
vgLoadMatrix(pathUserToSurface);
vgTransformPath(dstPath, srcPath);

/* Use the identity matrix for drawing the stroked path. */
vgLoadIdentity();

/* Set the paint transformation to the concatenation of the
 * paint-to-user and user-to-surface transformations.
 */
vgMatrixMode(VG_MATRIX_FILL_PAINT_TO_USER);
vgLoadMatrix(pathUserToSurface);
vgMultMatrix(strokePaintToUser);

/* Stroke the transformed path. */
vgDrawPath(dstPath, VG_STROKE_PATH);
```

8.8 Filling or Stroking a Path

VGFillRule

The `VGFillRule` enumeration defines constants for the even/odd and non-zero fill rules.

```
typedef enum {
  VG_EVEN_ODD = 0x1900,
  VG_NON_ZERO = 0x1901
} VGFillRule;
```

To set the rule for filling, call **vgSeti** with a `type` parameter value of `VG_FILL_RULE` and a `value` parameter defined using a value from the `VGFillRule` enumeration. When the path is filled, the most recent setting of the fill rule on the current context is used. The fill rule setting has no effect on stroking.

```
VGFillRule fillRule;
vgSeti(VG_FILL_RULE, fillRule);
```

VGPaintMode

The `VGPaintMode` enumeration defines constants for stroking and filling paths, to be used by the **vgDrawPath**, **vgSetPaint**, and **vgGetPaint** functions.

```
typedef enum {
  VG_STROKE_PATH = (1 << 0),
  VG_FILL_PATH   = (1 << 1)
} VGPaintMode;
```

vgDrawPath

Filling and stroking are performed by the **vgDrawPath** function. The `paintModes` argument is a bitwise OR of values from the `VGPaintMode` enumeration, determining whether the path is to be filled (`VG_FILL_PATH`), stroked (`VG_STROKE_PATH`), or both (`VG_FILL_PATH | VG_STROKE_PATH`). If both filling and stroking are to be performed, the path is first filled, then stroked.

```
void vgDrawPath(VGPath path, VGbitfield paintModes)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `paintModes` is not a valid bitwise OR of values from the `VGPaintMode` enumeration

Filling a Path

Calling **vgDrawPath** with a `paintModes` argument of `VG_FILL_PATH` causes the given path to be filled, using the paint defined for the `VG_FILL_PATH` paint mode and the current fill rule.

The matrix currently set for the `VG_MATRIX_FILL_PAINT_TO_USER` matrix mode is applied to the paint used to fill the path outline. The matrix currently set for the `VG_MATRIX_PATH_USER_TO_SURFACE` matrix mode is used to transform the outline of the path and the paint into surface coordinates.

```
vgDrawPath(VGPath path, VG_FILL_PATH);
```

Stroking a Path

Calling **vgDrawPath** with a `paintModes` argument of `VG_STROKE_PATH` causes the given path to be stroked, using the paint defined for the `VG_STROKE_PATH` paint mode and the current set of stroke parameters.

The matrix currently set for the `VG_MATRIX_STROKE_PAINT_TO_USER` matrix mode is applied to the paint used to fill the stroked path outline. The matrix currently set for the `VG_MATRIX_PATH_USER_TO_SURFACE` matrix mode is used to transform the outline of the stroked path and the paint into surface coordinates.

```
vgDrawPath(VGPath path, VG_STROKE_PATH);
```

The following code sample shows how an application might set stroke parameters using variants of **vgSet**, and stroke a path object (defined elsewhere):

```
VGPath path;

/* Set the line width to 2.5 */
vgSetf(VG_STROKE_LINE_WIDTH, 2.5f);
/* Set the miter limit to 10.5 */
vgSetf(VG_STROKE_MITER_LIMIT, 10.5f);
/* Set the cap style to CAP_SQUARE */
vgSeti(VG_STROKE_CAP_STYLE, VG_CAP_SQUARE);
/* Set the join style to JOIN_MITER */
vgSeti(VG_STROKE_JOIN_STYLE, VG_JOIN_MITER);

/* Set the dash pattern */
VGfloat dashes[] = { 1.0f, 2.0f, 2.0f, 2.0f };
vgSetfv(VG_STROKE_DASH_PATTERN, 4, dashes);

/* Set the dash phase to 0.5 and reset it for every subpath */
vgSetf(VG_STROKE_DASH_PHASE, 0.5f);
vgSeti(VG_STROKE_DASH_PHASE_RESET, VG_TRUE);

/* Stroke the path */
vgDrawPath(path, VG_STROKE_PATH);
```

Filling and Stroking a Path

Calling **vgDrawPath** with a `paintModes` argument of `(VG_FILL_PATH | VG_STROKE_PATH)` causes the given path to be first filled, then stroked, exactly as if **vgDrawPath** were called twice in succession, first with a `paintModes` argument of `VG_FILL_PATH` and second with a `paintModes` argument of `VG_STROKE_PATH`.

```
vgDrawPath(VGPath path, VG_FILL_PATH | VG_STROKE_PATH);
```

9 Paint

Paint defines a color and an alpha value for each pixel being drawn. *Color paint* defines a constant color for all pixels; *gradient paint* defines a linear or radial pattern of smoothly varying colors; and *pattern paint* defines a possibly repeating rectangular pattern of colors based on a source image. It is possible to define new types of paint as extensions.

Paint is defined in its own coordinate system, which is transformed into user coordinates by means of the fill-paint-to-user and stroke-paint-to-user transformations (set using the `VG_MATRIX_FILL_PAINT_TO_USER` and `VG_MATRIX_STROKE_PAINT_TO_USER` matrix modes) depending on whether the current geometry is being filled or stroked.

Given a (fill or stroke) paint-to-user transformation Tp and user-to-surface transformation Tu, the paint color and alpha of a pixel to be drawn with surface coordinates (x, y) is defined by mapping its center point $(x + ½, y + ½)$ through the inverse transformation $(Tu \circ Tp)^{-1}$, resulting in a sample point in the paint coordinate space. This transformation must be evaluated with sufficient accuracy to ensure a deviation from the ideal of no more than `1/8` of a pixel along either axis. The paint value nearest that point may be used (point sampling), or paint values from multiple points surrounding the central sample point may be combined to produce an interpolated paint value. Paint color values are processed in premultiplied alpha format during interpolation. The user-to-surface transformation Tu is taken from the path-user-to-surface transformation when fulfilling a **vgDrawPath** call, or from the image-user-to-surface transformation when fulfilling a **vgDrawImage** call.

If the inverse transformation cannot be computed due to a (near-)singularity, no drawing occurs.

9.1 Paint Definitions

The OpenVG context stores two paint definitions at a time, one to be applied to stroked shapes and one for filled shapes. This allows the interior of a path to be filled using one type of paint and its outline to be stroked with another kind of paint in a single **vgDrawPath** operation. Initially, default values are used.

VGPaint

`VGPaint` represents an opaque handle to a paint object. A `VGPaint` object is *live*; changes to a `VGPaint` object (using `vgSetParameter`, or by altering an attached pattern image) attached to a context will immediately affect drawing calls on that context. If a `VGPaint` object is accessed from multiple threads, the application must ensure (using **vgFinish** along with application-level synchronization primitives) that the paint definition is not altered from one context while another context may still be using it for drawing.

```
typedef VGHandle VGPaint;
```

9.1.1 Creating and Destroying Paint Objects

vgCreatePaint

vgCreatePaint creates a new paint object that is initialized to a set of default values and returns a `VGPaint` handle to it. If insufficient memory is available to allocate a new object, `VG_INVALID_HANDLE` is returned.

```
VGPaint vgCreatePaint(void)
```

vgDestroyPaint

The resources associated with a paint object may be deallocated by calling **vgDestroyPaint**. Following the call, the `paint` handle is no longer valid in any of the contexts that shared it. If the paint object is currently active in a drawing context, the context continues to access it until it is replaced or the context is destroyed.

```
void vgDestroyPaint(VGPaint paint)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `paint` is not a valid paint handle, or is not shared with the current context

9.1.2 Setting the Current Paint

vgSetPaint

Paint definitions are set on the current context using the **vgSetPaint** function. The `paintModes` argument is a bitwise OR of values from the `VGPaintMode` enumeration, determining whether the paint object is to be used for filling (`VG_FILL_PATH`), stroking (`VG_STROKE_PATH`), or both (`VG_FILL_PATH | VG_STROKE_PATH`). The current `paint` replaces the previously set paint object, if any, for the given paint mode or modes. If `paint` is equal to `VG_INVALID_HANDLE`, the previously set paint object for the given mode (if present) is removed and the paint settings are restored to their default values.

```
void vgSetPaint(VGPaint paint, VGbitfield paintModes)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `paint` is neither a valid paint handle nor equal to `VG_INVALID_HANDLE`, or is not shared with the current context

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `paintModes` is not a valid bitwise OR of values from the `VGPaintMode` enumeration

vgGetPaint

The **vgGetPaint** function returns the paint object currently set for the given `paintMode`, or `VG_INVALID_HANDLE` if an error occurs or if no paint object is set (*i.e.*, the default paint is present) on the given context with the given `paintMode`.

```
VGPaint vgGetPaint(VGPaintMode paintMode)
```

ERRORS

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `paintMode` is not a valid value from the `VGPaintMode` enumeration

9.1.3 Setting Paint Parameters

Paint functionality is controlled by a number of paint parameters that are stored in each paint object.

VGPaintParamType

Values from the `VGPaintParamType` enumeration may be used as the `paramType` argument to **vgSetParameter** and **vgGetParameter** to set and query various features of a paint object:

```
typedef enum {
  /* Color paint parameters */
  VG_PAINT_TYPE                        = 0x1A00,
  VG_PAINT_COLOR                       = 0x1A01,
  VG_PAINT_COLOR_RAMP_SPREAD_MODE      = 0x1A02,
  VG_PAINT_COLOR_RAMP_STOPS            = 0x1A03,
  VG_PAINT_COLOR_RAMP_PREMULTIPLIED    = 0x1A07,

  /* Linear gradient paint parameters */
  VG_PAINT_LINEAR_GRADIENT             = 0x1A04,

  /* Radial gradient paint parameters */
  VG_PAINT_RADIAL_GRADIENT             = 0x1A05,

  /* Pattern paint parameters */
  VG_PAINT_PATTERN_TILING_MODE         = 0x1A06
} VGPaintParamType;
```

The default values that are used when no paint object is present (*i.e.,* in a newly-created context or following a call to **vgSetPaint** with a `paint` value of `VG_INVALID_HANDLE`) are shown in Table 10. These values are also used as the initial parameter value for a newly created paint object.

Parameter	***Datatype***	***Default Value***
`VG_PAINT_TYPE`	`VGPaintType`	`VG_PAINT_TYPE_COLOR`
`VG_PAINT_COLOR`	`VGfloat[4]`	{ 0.0f, 0.0f, 0.0f, 1.0f }
`VG_PAINT_COLOR_RAMP_ SPREAD_MODE`	`VGColorRampSpreadMode`	`VG_COLOR_RAMP_SPREAD_PAD`
`VG_PAINT_COLOR_RAMP_ STOPS`	`VGfloat *`	Array of Length 0
`VG_PAINT_COLOR_RAMP_ PREMULTIPLIED`	`VGboolean`	`VG_TRUE`
`VG_PAINT_LINEAR_ GRADIENT`	`VGfloat[4]`	{ 0.0f, 0.0f, 1.0f, 0.0f }
`VG_PAINT_RADIAL_ GRADIENT`	`VGfloat[5]`	{ 0.0f, 0.0f, 0.0f, 0.0f, 1.0f }
`VG_PAINT_PATTERN_ TILING_MODE`	`VGTilingMode`	`VG_TILE_FILL`

Table 10: VGPaintParamType Defaults

VGPaintType

The `VGPaintType` enumeration is used to supply values for the `VG_PAINT_TYPE` paint parameter to determine the type of paint to be applied.

```
typedef enum {
  VG_PAINT_TYPE_COLOR           = 0x1B00,
  VG_PAINT_TYPE_LINEAR_GRADIENT = 0x1B01,
  VG_PAINT_TYPE_RADIAL_GRADIENT = 0x1B02,
  VG_PAINT_TYPE_PATTERN         = 0x1B03
} VGPaintType;
```

9.2 Color Paint

Color paint uses a fixed color and alpha for all pixels. An alpha value of 1 produces a fully opaque color. Colors are specified in non-premultiplied sRGBA format.

Setting Color Paint Parameters

To enable color paint, use **vgSetParameteri** to set the paint type to `VG_PAINT_TYPE_COLOR`.

The **vgSetParameterfv** function allows the color and alpha values to be set using the `VG_PAINT_COLOR` paint parameter to values between 0 and 1. Values outside this range are interpreted as the nearest endpoint of the range.

```
VGfloat fill_red, fill_green, fill_blue, fill_alpha;
VGfloat stroke_red, stroke_green, stroke_blue, stroke_alpha;
VGPaint myFillPaint, myStrokePaint;

VGfloat * fill_RGBA = {
  fill_red, fill_green, fill_blue, fill_alpha
};

VGfloat * stroke_RGBA = {
  stroke_red, stroke_green, stroke_blue, stroke_alpha
};

/* Fill with color paint */
vgSetParameteri(myFillPaint, VG_PAINT_TYPE, VG_PAINT_TYPE_COLOR);
vgSetParameterfv(myFillPaint, VG_PAINT_COLOR, 4, fill_RGBA);

/* Stroke with color paint */
vgSetParameteri(myStrokePaint, VG_PAINT_TYPE, VG_PAINT_TYPE_COLOR);
vgSetParameterfv(myStrokePaint, VG_PAINT_COLOR, 4, stroke_RGBA);
```

vgSetColor

As a shorthand, the **vgSetColor** function allows the `VG_PAINT_COLOR` parameter of a given `paint` object to be set using a 32-bit non-premultiplied sRGBA_8888 representation (see Section 10.210.2). The rgba parameter is a VGuint with 8 bits of red starting at the most significant bit, followed by 8 bits each of green, blue, and alpha. Each color or alpha channel value is conceptually divided by 255.0f to obtain a value between 0 and 1.

```
void vgSetColor(VGPaint paint, VGuint rgba)
```

ERRORS

VG_BAD_HANDLE_ERROR

– if paint is not a valid paint handle, or is not shared with the current context

The code:

```
VGPaint paint;
VGuint rgba;
vgSetColor(paint, rgba)
```

is equivalent to the code:

```
VGfloat rgba_f[4];
rgba_f[0] = ((rgba >> 24) & 0xff)/255.0f;
rgba_f[1] = ((rgba >> 16) & 0xff)/255.0f;
rgba_f[2] = ((rgba >>  8) & 0xff)/255.0f;
rgba_f[3] = ( rgba        & 0xff)/255.0f;
vgSetParameterfv(paint, VG_PAINT_COLOR, 4, rgba_f);
```

vgGetColor

The current setting of the VG_PAINT_COLOR parameter on a given paint object may be queried as a 32-bit non-premultiplied sRGBA_8888 value. Each color channel or alpha value is clamped to the [0, 1] range, multiplied by 255, and rounded to obtain an 8-bit integer; the resulting values are packed into a 32-bit value in the same format as for **vgSetColor**.

```
VGuint vgGetColor(VGPaint paint)
```

ERRORS

VG_BAD_HANDLE_ERROR

– if paint is not a valid paint handle, or is not shared with the current context

The code:

```
VGPaint paint;
VGuint rgba;
rgba = vgGetColor(paint);
```

is equivalent to the code:

```
#define CLAMP(x) ((x) < 0.0f ? 0.0f : ((x) > 1.0f ? 1.0f : (x)))

VGfloat rgba_f[4];
int red, green, blue, alpha;

vgGetParameterfv(paint, VG_PAINT_COLOR, 4, rgba_f);

/*
 * Clamp color and alpha values from vgGetParameterfv to the
 * [0, 1] range, scale to 8 bits, and round to integer.
 */
red   = (int)(CLAMP(rgba_f[0])*255.0f + 0.5f);
green = (int)(CLAMP(rgba_f[1])*255.0f + 0.5f);
blue  = (int)(CLAMP(rgba_f[2])*255.0f + 0.5f);
alpha = (int)(CLAMP(rgba_f[3])*255.0f + 0.5f);
rgba = (red << 24) | (green << 16) | (blue << 8) | alpha;
```

9.3 Gradient Paint

Gradients are patterns used for filling or stroking. They are defined mathematically in two parts; a scalar-valued *gradient function* defined at every point in the two-dimensional plane (in paint coordinates), followed by a *color ramp* mapping.

9.3.1 Linear Gradients

Linear gradients define a scalar-valued gradient function based on two points (*x0, y0*) and (*x1, y1*) (in the paint coordinate system) with the following properties:

- It is equal to 0 at (*x0, y0*)
- It is equal to 1 at (*x1, y1*)
- It increases linearly along the line from (*x0, y0*) to (*x1, y1*)
- It is constant along lines perpendicular to the line from (*x0, y0*) to (*x1, y1*)

An expression for the gradient function is:

$$g(x, y) = \frac{\Delta x(x - x0) + \Delta y(y - y0)}{\Delta x^2 + \Delta y^2}$$

where $\Delta x = x1 - x0$ and $\Delta y = y1 - y0$. If the points (*x0, y0*) and (*x1, y1*) are coincident (and thus $\Delta x^2 + \Delta y^2 = 0$), the function is given the value 1 everywhere.

Setting Linear Gradient Parameters

To enable linear gradient paint, use **vgSetParameteri** to set the paint type to `VG_PAINT_TYPE_LINEAR_GRADIENT`.

The linear gradient parameters are set using **vgSetParameterfv** with a `paramType` argument of `VG_PAINT_LINEAR_GRADIENT`. The gradient values are supplied as a vector of 4 floats in the order *{ x0, y0, x1, y1 }*.

```
VGfloat fill_x0, fill_y0, fill_x1, fill_y1;
VGfloat stroke_x0, stroke_y0, stroke_x1, stroke_y1;
VGPaint myFillPaint, myStrokePaint;
VGfloat * fill_linear_gradient = {
  fill_x0, fill_y0, fill_x1, fill_y1
};
VGfloat * stroke_linear_gradient = {
  stroke_x0, stroke_y0, stroke_x1, stroke_y1
};

/* Fill with linear gradient paint */
vgSetParameteri(myFillPaint, VG_PAINT_TYPE,
                VG_PAINT_TYPE_LINEAR_GRADIENT);
vgSetParameterfv(myFillPaint, VG_PAINT_LINEAR_GRADIENT,
                 4, fill_linear_gradient);
/* Stroke with linear gradient paint */
vgSetParameteri(myStrokePaint, VG_PAINT_TYPE,
                VG_PAINT_TYPE_LINEAR_GRADIENT);
vgSetParameterfv(myStrokePaint, VG_PAINT_LINEAR_GRADIENT,
                 4, stroke_linear_gradient);
```

9.3.2 Radial Gradients

Radial gradients define a scalar-valued gradient function based on a *gradient circle* defined by a *center point* (*cx, cy*), a radius *r*, and a *focal point* (*fx, fy*) that is forced to lie within the circle. All parameters are given in the paint coordinate system.

The computation of the radial gradient function is illustrated in Figure 18. The function is equal to 0 at the focal point and 1 along the circumference of the gradient circle. Elsewhere, it is equal to the distance between (*x, y*) and (*fx, fy*) (shown as d_1) divided by the length of the line segment starting at (*fx, fy*), passing through (*x, y*), and ending on the circumference of the gradient circle (shown as d_2). If the radius is less than or equal to 0, the function is given the value 1 everywhere.

An expression for the gradient function may be derived by defining the line between (*fx, fy*) and (*x, y*) by the parametric expression (*fx, fy*) + *t**(*x – fx, y – fy*) and determining the positive value of *t* at which the line intersects the circle $(x - cx)^2 + (y - cy)^2 = r^2$. Figure 18 illustrates the construction. The gradient value *g*(*x, y*) is then given by *1/t*. The resulting expression is:

$$g(x,y)=\frac{dx^2+dy^2}{\sqrt{r^2(dx^2+dy^2)-(dx\,fy'-dy\,fx')^2}-(dx\,fx'+dy\,fy')}$$

where *fx*' = *fx* – *cx*, *fy*' = *fy* – *cy*, *dx* = *x* – *fx* and *dy* = *y* – *fy*.

This may be rearranged and simplified to obtain a formula that does not require per-pixel division:

$$g(x,y)=\frac{(dx\,fx'+dy\,fy')+\sqrt{r^2(dx^2+dy^2)-(dx\,fy'-dy\,fx')^2}}{r^2-(fx'^2+fy'^2)}$$

One way to evaluate the gradient function efficiently is to rewrite it in the form:

$$g_y(x)=(A\,x+B)+\sqrt{C\,x^2+D\,x+E}$$

and to use forward differencing of $Ax + B$ and $Cx^2 + Dx + E$ to evaluate it incrementally along a scanline with several additions and a single square root per pixel.

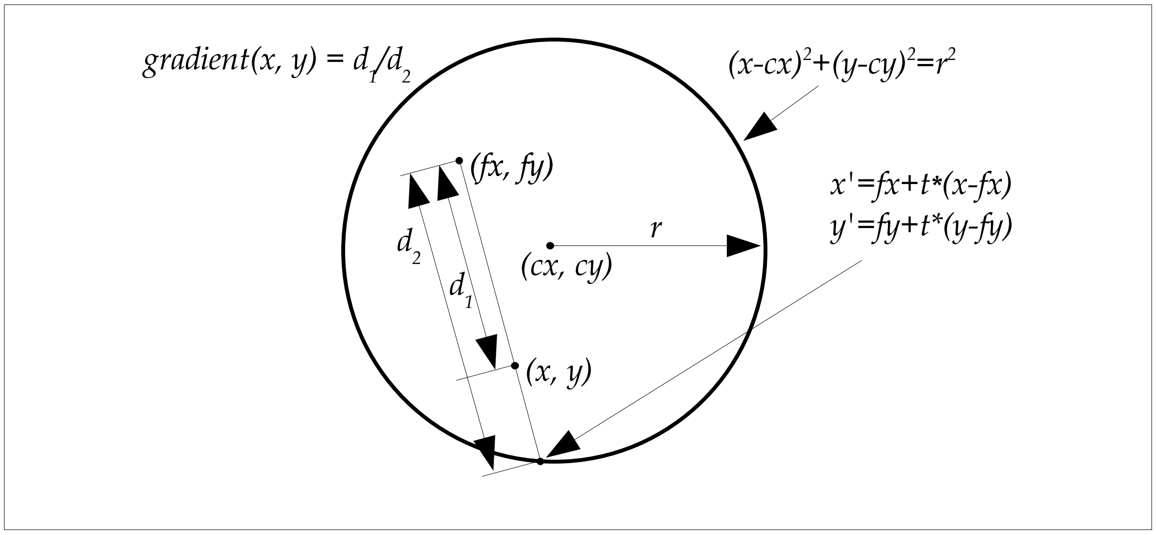

gradient(x, y) = d_1/d_2
$(x\text{-}cx)^2+(y\text{-}cy)^2=r^2$
(fx, fy)
x'=fx+t*(x-fx)
y'=fy+t*(y-fy)
r
d_2
(cx, cy)
d_1
(x, y)

Figure 18: Radial Gradient Function

Setting Radial Gradient Parameters

To enable radial gradient paint, use **vgSetParameteri** to set the paint type to `VG_PAINT_TYPE_RADIAL_GRADIENT`. The radial gradient parameters are set using **vgSetParameterfv** with a `paramType` argument of `VG_PAINT_RADIAL_GRADIENT`. The gradient values are supplied as a vector of 5 floats in the order *{ cx, cy, fx, fy, r }*.

If (*fx, fy*) lies outside the circumference of the circle, the intersection of the line from the center to the focal point with the circumference of the circle is used as the focal point in place of the specified point. To avoid a division by 0, the implementation may move the focal point along the line towards the center of the circle by an amount sufficient to avoid numerical instability, provided the new location lies at a distance of at least *.99r* from the circle center. The following code illustrates the setting of radial gradient parameters:

```
VGPaint myFillPaint, myStrokePaint;
VGfloat fill_cx, fill_cy, fill_fx, fill_fy, fill_r;
VGfloat stroke_cx, stroke_cy, stroke_fx, stroke_fy, stroke_r;
VGfloat * fill_radial_gradient = { fill_cx, fill_cy,
  fill_fx, fill_fy, fill_r };
VGfloat * stroke_radial_gradient = { stroke_cx, stroke_cy,
  stroke_fx, stroke_fy, stroke_r };
vgSetParameteri(myFillPaint, VG_PAINT_TYPE,     /* Fill */
                VG_PAINT_TYPE_RADIAL_GRADIENT);
vgSetParameterfv(myFillPaint, VG_PAINT_RADIAL_GRADIENT,
                 5, fill_radial_gradient);
vgSetParameteri(myStrokePaint, VG_PAINT_TYPE,   /* Stroke */
                VG_PAINT_TYPE_RADIAL_GRADIENT);
vgSetParameterfv(myStrokePaint, VG_PAINT_RADIAL_GRADIENT,
                 5, stroke_radial_gradient);
```

9.3.3 Color Ramps

Color ramps map the scalar values produced by gradient functions to colors. The application defines the non-premultiplied sRGBA color and alpha value associated with each of a number of values, called *stops*. A stop is defined by an *offset* between 0 and 1, inclusive, and a color value. Stops must be specified in increasing order; if they are not, the entire sequence is ignored. It is legal to have multiple stops with the same offset value, which will result in a discontinuity in the color ramp, with the first stop with a given offset value defining the right endpoint of one interval and the last stop with the same offset value defining the left endpoint of the next interval. At an offset value equal to that of a stop, the color value is that of the last stop with the given offset. Intermediate stops with the same offset value have no effect. Stops with offsets less than 0 or greater than 1 are ignored.

If no valid stops have been specified (*e.g.*, due to an empty input array, out-of-range, or out-of-order stops), a stop at 0 with (*R, G, B, α*) color (*0.0, 0.0, 0.0, 1.0*) (opaque black) and a stop at 1 with color (*1.0, 1.0, 1.0, 1.0*) (opaque white) are implicitly defined. If at least one valid stop has been specified, but none has been defined with an offset of 0, an implicit stop is added with an offset of 0 and the same color as the first user-defined stop. If at least one valid stop has been specified, but none has been defined with an offset of 1, an implicit stop is added with an offset of 1 and the same color as the last user-defined stop.

If a color or alpha value of a given stop falls outside of the range [0, 1], the closest endpoint of the range is used instead.

If the paint's `VG_PAINT_COLOR_RAMP_PREMULTIPLIED` flag is set to `VG_TRUE`, color and alpha values at each gradient stop are multiplied together to form premultiplied sRGBA values prior to interpolation. Otherwise, color and alpha values are processed independently.

Color and alpha values at offset values between the stops are defined by means of linear interpolation between the premultiplied or non-premultiplied color values defined at the nearest stops above and below the given offset value.

VG_MAX_COLOR_RAMP_STOPS

The `VG_MAX_COLOR_RAMP_STOPS` parameter contains the maximum number of gradient stops supported by the OpenVG implementation. All implementations must support at least 32 stops. If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. Implicitly defined stops at offsets 0 and 1 are not counted against this maximum. The value may be retrieved by calling **vgGeti**:

```
VGint maxStops = vgGeti(VG_MAX_COLOR_RAMP_STOPS);
```

VGColorRampSpreadMode

The application may only define stops with offsets between 0 and 1. Spread modes define how the given set of stops are repeated or extended in order to define interpolated color values for arbitrary input values outside the [0,1] range. The `VGColorRampSpreadMode` enumeration defines three modes:

- `VG_COLOR_RAMP_SPREAD_PAD` – extend stops
- `VG_COLOR_RAMP_SPREAD_REPEAT` – repeat stops
- `VG_COLOR_RAMP_SPREAD_REFLECT` – repeat stops in reflected order

```
typedef enum {
  VG_COLOR_RAMP_SPREAD_PAD     = 0x1C00,
  VG_COLOR_RAMP_SPREAD_REPEAT  = 0x1C01,
  VG_COLOR_RAMP_SPREAD_REFLECT = 0x1C02
} VGColorRampSpreadMode;
```

In pad mode, the colors defined at 0 and 1 are used for all stop values less than 0 or greater than 1, respectively.

In repeat mode, the color values defined between 0 and 1 are repeated indefinitely in both directions. Gradient values outside the [0, 1] range are shifted by an integer amount to place them into that range. For example, a gradient value of 5.6 will receive the same color as a gradient value of 0.6. A gradient value of -5.6 will receive the same color as a gradient value of 0.4 (since 0.4 = -5.6 + 6).

In reflect mode, the color values defined between 0 and 1 are repeated indefinitely in both directions, but with alternate copies of the range reversed. A gradient value of 1.2 will receive the same color as a gradient value of 0.8, since 0.8 = 1.0 – 0.2 and 1.2 = 1.0 + 0.2. A gradient value of 2.4 will receive the same color as a gradient value of 0.4.

The color ramp pad modes are illustrated schematically in Figure 19.

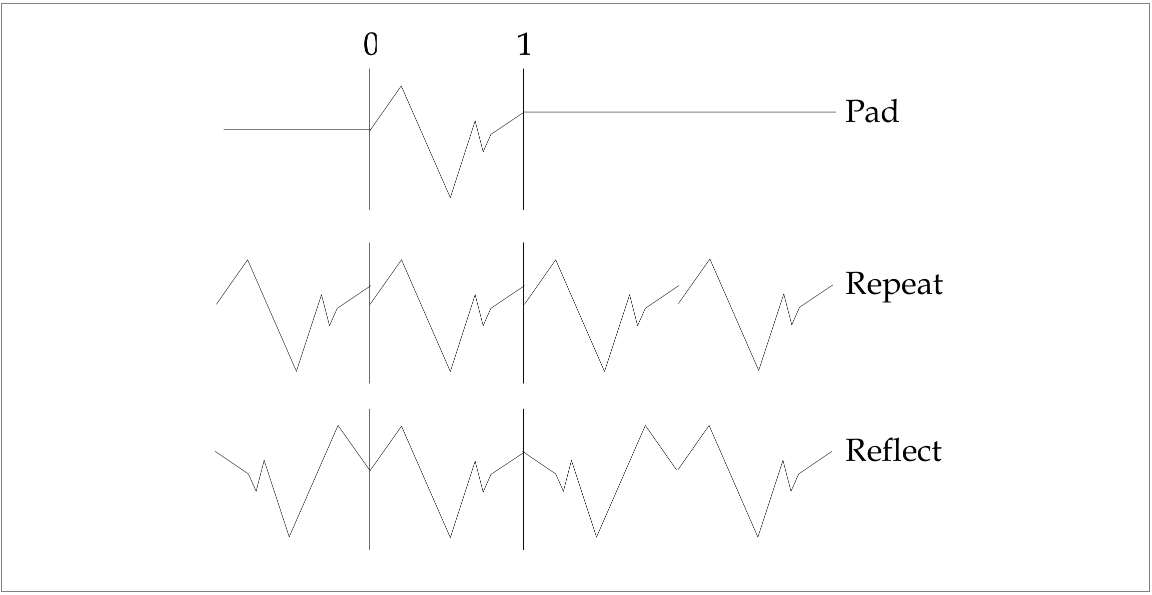


Figure 19: Color Ramp Pad Modes

Setting Color Ramp Parameters

Color ramp parameters are set using **vgSetParameter**. The `VG_PAINT_COLOR_RAMP_SPREAD_MODE` parameter controls the spread mode using a value from the `VGColorRampSpreadMode` enumeration. The `VG_PAINT_COLOR_RAMP_PREMULTIPLIED` parameter takes a `VGboolean` value and controls whether color and alpha values are interpolated in premultiplied or non-premultiplied form. The `VG_PAINT_COLOR_RAMP_STOPS` parameter takes an array of floating-point values giving the offsets and colors of the stops, in order. Each stop is defined by a floating-point offset value and four floating-point values containing the sRGBA color and alpha value associated with each stop, in the form of a non-premultiplied (R, G, B, α) quad. The **vgSetParameter** function will generate an error if the number of values submitted is not a multiple of 5 (zero is acceptable). Up to `VG_MAX_COLOR_RAMP_STOPS` 5-tuples may be set. If more than `VG_MAX_COLOR_RAMP_STOPS` 5-tuples are specified, those beyond the first `VG_MAX_COLOR_RAMP_STOPS` are discarded immediately (and will not be returned by **vgGetParameter)**.

```
VGPaint myFillPaint, myStrokePaint;

VGColorRampSpreadMode fill_spreadMode;
VGboolean fill_premultiplied;
VGfloat fill_stops[5*FILL_NUM_STOPS];

VGColorRampSpreadMode stroke_spreadMode;
VGboolean stroke_premultiplied;
VGfloat stroke_stops[5*STROKE_NUM_STOPS];

vgSetParameteri(myFillPaint, VG_PAINT_COLOR_RAMP_SPREAD_MODE,
                fill_spreadMode);
vgSetParameteri(myFillPaint, VG_PAINT_COLOR_RAMP_PREMULTIPLIED,
                fill_premultiplied);
vgSetParameterfv(myFillPaint, VG_PAINT_COLOR_RAMP_STOPS,
                 5*FILL_NUM_STOPS, fill_stops);

vgSetParameteri(myStrokePaint, VG_PAINT_COLOR_RAMP_SPREAD_MODE,
                stroke_spreadMode);
vgSetParameteri(myStrokePaint, VG_PAINT_COLOR_RAMP_PREMULTIPLIED,
                stroke_premultiplied);
vgSetParameterfv(myStrokePaint, VG_PAINT_COLOR_RAMP_STOPS,
                 5*STROKE_NUM_STOPS, stroke_stops);
```

A common set of color ramp settings are used for both linear and radial gradients defined on a given paint object.

Formal Definition of Spread Modes

This section provides a formal definition of the color ramp spread modes.

In the following, assume that a sequence of stops $\{S_0, S_1, ..., S_{N-1}\}$ have been defined by the application, and/or by default or implicit values. The stop S_i is defined to have offset x_i and color c_i. The stops are assumed to be ordered by offset but may have duplicate offsets; that is, for all $i < j$, $x_i \le x_j$. To determine the interpolated color value at a given offset value v, determine the smallest i such that $x_{i+1} > v$. If $x_i = v$, use the color c_i, otherwise perform linear interpolation between the stops S_i and S_{i+1} to produce the color $c_i + (c_{i+1} - c_i)(v - x_i)/(x_{i+1} - x_i)$.

In pad mode, values smaller than 0 are assigned the color c_0 and values greater than or equal to 1 are assigned the color c_{N-1}.

In repeat mode, the offset value v is mapped to a new value *v´* that is guaranteed to lie between 0 and 1. Following this mapping, the color is defined as for pad mode:

$$v'_{repeat} = v - \lfloor v \rfloor$$

In reflect mode, the offset value *v* is mapped to a new value *v´* that is guaranteed to lie between 0 and 1. Following this mapping, the color is defined as for pad mode:

$$v'_{reflect} = \begin{cases} v - \lfloor v \rfloor, & \text{if } \lfloor v \rfloor \text{is even} \\ 1 - (v - \lfloor v \rfloor), & \text{if } \lfloor v \rfloor \text{is odd} \end{cases}$$

9.3.4 Gradient Examples

Figure 20 shows a square from (0, 0) to (400, 400) painted with a set of linear gradients with (*x0, y0*) = (50, 50), (*x1, y1*) = (350, 350).

Figure 21 shows the same square painted with radial gradients with centered and non-centered focal points. The centered gradient, shown in the top row, has its center (*cx, cy*) and focal point (*fx, fy*) both at (200, 200). The non-centered gradient, shown in the bottom row, has its center (*cx, cy*) at (200, 200) and its focal point (*fx, fy*) at (250, 250). The radius *r* for both gradients is equal to 100.

All the gradients shown in this section utilize a color ramp with stops at offsets 0.0, 0.33, 0.66, and 1.0 colored white, red, green, and blue, respectively, as shown in Figure 22.

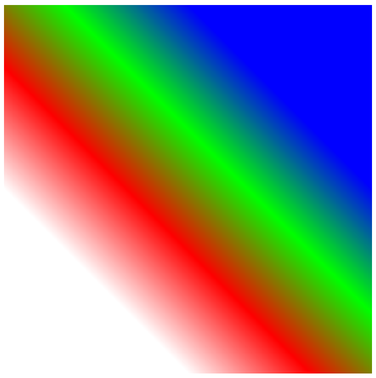
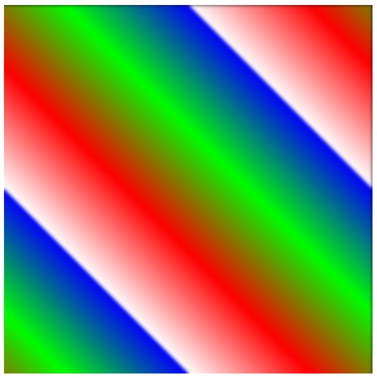
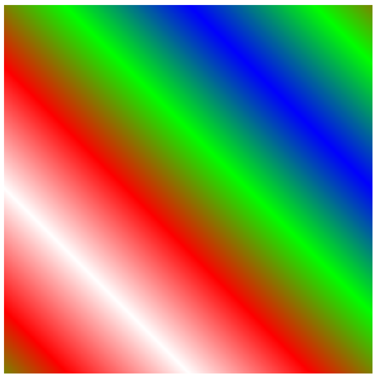
Pad	***Repeat***	***Reflect***
		

Figure 20: Linear Gradients

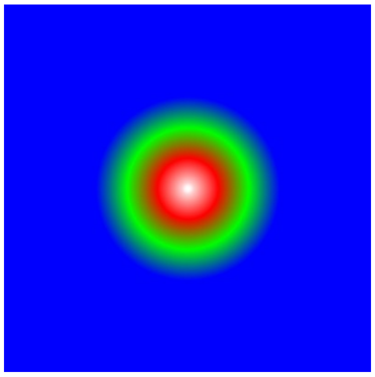
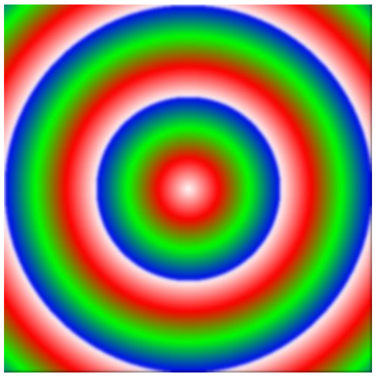
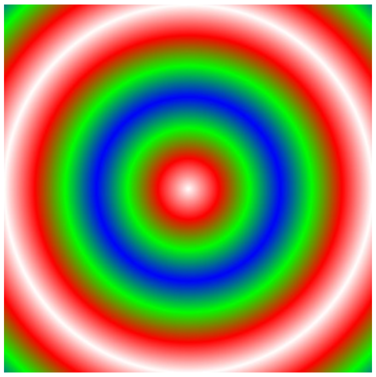
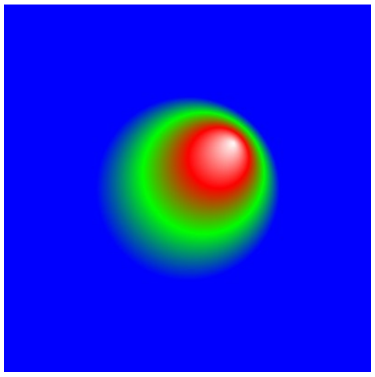
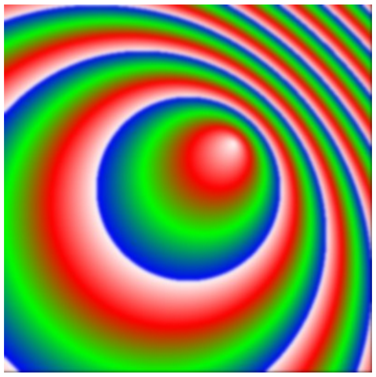
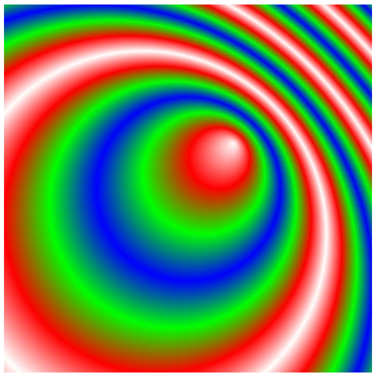
Pad	***Repeat***	***Reflect***
		
		

Figure 21: Centered and Non-Centered Radial Gradients

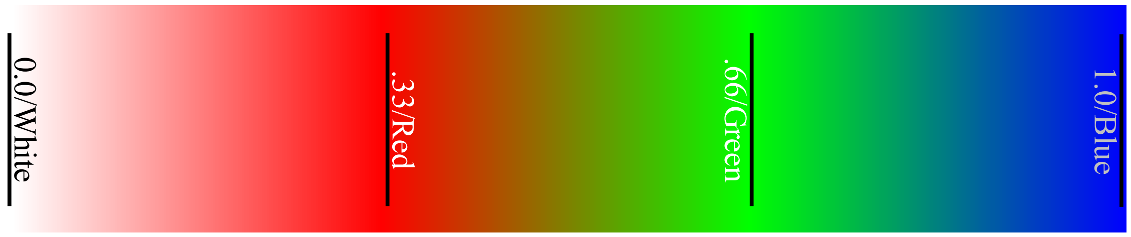


Figure 22: Color Ramp used for Gradient Examples

9.4 Pattern Paint

Pattern paint defines a rectangular pattern of colors based on the pixel values of an image. Images are described below in Section 10. Each pixel (x, y) of the pattern image defines a point of color at the pixel center ($x + ½$, $y + ½$).

Filtering may be used to construct an interpolated pattern value at the sample point, based on the pattern image pixel values. The pattern tiling mode is used to define values for pixel centers in the pattern space that lie outside of the bounds of the pattern.

Interpolation may be performed between multiple pixels of the pattern image to produce an antialiased pattern value. The image quality setting at the time of drawing (determined by the `VG_IMAGE_QUALITY` parameter) is used to control the quality of pattern interpolation. If the image quality is set to `VG_IMAGE_QUALITY_NONANTIALIASED`, nearest-neighbor interpolation (point sampling) is used. If the image quality is set to `VG_IMAGE_QUALITY_FASTER` or `VG_IMAGE_QUALITY_BETTER`, higher-quality interpolation will be used if available. Interpolation is done in the color space of the image using a premultiplied representation.

vgPaintPattern

The **vgPaintPattern** function replaces any previous pattern image defined on the given `paint` object for the given set of paint modes with a new pattern image. A value of `VG_INVALID_HANDLE` for the `pattern` parameter removes the current pattern image from the paint object.

If the current paint object has its `VG_PAINT_TYPE` parameter set to `VG_PAINT_TYPE_PATTERN`, but no pattern image is set, the paint object behaves as if `VG_PAINT_TYPE` were set to `VG_PAINT_TYPE_COLOR`.

While an image is set as the paint pattern for any paint object, it may not be used as a rendering target. Conversely, an image that is currently a rendering target may not be set as a paint pattern.

```
void vgPaintPattern(VGPaint paint, VGImage pattern)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

- if `paint` is not a valid paint handle, or is not shared with the current context
- if `pattern` is neither a valid image handle nor equal to `VG_INVALID_HANDLE`, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

- if `pattern` is currently a rendering target

9.4.1 Pattern Tiling

Patterns may be extended (tiled) using one of four possible tiling modes, defined by the `VGTilingMode` enumeration.

VGTilingMode

The `VGTilingMode` enumeration defines possible methods for defining colors for source pixels that lie outside the bounds of the source image.

The `VG_TILE_FILL` condition specifies that pixels outside the bounds of the source image should be taken as the color `VG_TILE_FILL_COLOR`. The color is expressed as a non-premultiplied sRGBA color and alpha value. Values outside the [0, 1] range are interpreted as the nearest endpoint of the range.

The `VG_TILE_PAD` condition specifies that pixels outside the bounds of the source image should be taken as having the same color as the closest edge pixel of the source image. That is, a pixel (x, y) has the same value as the image pixel $(max(0, min(x, width - 1)), max(0, min(y, height - 1)))$.

The `VG_TILE_REPEAT` condition specifies that the source image should be repeated indefinitely in all directions. That is, a pixel (x, y) has the same value as the image pixel $(x \ mod \ width, y \ mod \ height)$ where the operator '*a mod b*' returns a value between 0 and $(b - 1)$ such that $a = k{*}b + (a \ mod \ b)$ for some integer k.

The `VG_TILE_REFLECT` condition specifies that the source image should be reflected indefinitely in all directions. That is, a pixel (x, y) has the same value as the image pixel (x', y') where:

$$x' = \begin{cases} x \ mod \ width & \text{if } floor(x/width) \text{ is even,} \\ width - 1 - (x \ mod \ width) & \text{otherwise.} \end{cases}$$

$$y' = \begin{cases} y \ mod \ height & \text{if } floor(y/height) \text{ is even,} \\ height - 1 - (y \ mod \ height) & \text{otherwise.} \end{cases}$$

```
typedef enum {
  VG_TILE_FILL    = 0x1D00,
  VG_TILE_PAD     = 0x1D01,
  VG_TILE_REPEAT  = 0x1D02,
  VG_TILE_REFLECT = 0x1D03,
} VGTilingMode;
```

Setting the Pattern Tiling Mode

The pattern tiling mode is set using **vgSetParameteri** with a `paramType` argument of `VG_PAINT_PATTERN_TILING_MODE`.

```
VGPaint myFillPaint, myStrokePaint;
VGImage myFillPaintPatternImage, myStrokePaintPatternImage;

VGTilingMode fill_tilingMode, stroke_tilingMode;

vgSetParameteri(myFillPaint, VG_PAINT_TYPE,
                VG_PAINT_TYPE_PATTERN);
vgSetParameteri(myFillPaint, VG_PAINT_PATTERN_TILING_MODE,
                fill_tilingMode);
vgPaintPattern(myFillPaint, myFillPaintPatternImage);

vgSetParameteri(myStrokePaint, VG_PAINT_TYPE,
                VG_PAINT_TYPE_PATTERN);
vgSetParameteri(myStrokePaint, VG_PAINT_PATTERN_TILING_MODE,
                stroke_tilingMode);
vgPaintPattern(myStrokePaint, myStrokePaintPatternImage);
```

10 Images

Images are rectangular collections of pixels. Image data may be inserted or extracted in a variety of formats with varying bit depths, color spaces, and alpha channel types. The actual storage format of an image is implementation-dependent, and may be optimized for a given device. Images may be drawn to a drawing surface, used to define paint patterns, or operated on directly by image filter operations.

10.1 Image Coordinate Systems

An image defines a coordinate system in which pixels are indexed using integer coordinates, with each integer corresponding to a distinct pixel. The lower-left pixel has a coordinate of $(0, 0)$, the x coordinate increases horizontally from left to right, and the y coordinate increases vertically from bottom to top. Note that this orientation is consistent with the other coordinate systems used in the OpenVG API, but differs from the top-to-bottom orientation used by many other imaging systems.

The “energy” of a pixel is located at the pixel center; that is, the pixel with coordinate (x, y) has its energy at the point $(x + ½, y + ½)$. The color at a point not located at a pixel center may be defined by applying a suitable filter to the colors defined at a set of nearby pixel centers.

10.2 Image Formats

`VGImageFormat`

The `VGImageFormat` enumeration defines the set of supported pixel formats and color spaces for images:

```
typedef enum {
  /* RGB{A,X} channel ordering */
  VG_sRGBX_8888            =  0,
  VG_sRGBA_8888            =  1,
  VG_sRGBA_8888_PRE        =  2,
  VG_sRGB_565              =  3,
  VG_sRGBA_5551            =  4,
  VG_sRGBA_4444            =  5,
  VG_sL_8                  =  6,
  VG_lRGBX_8888            =  7,
  VG_lRGBA_8888            =  8,
  VG_lRGBA_8888_PRE        =  9,
  VG_lL_8                  = 10,
  VG_A_8                   = 11,
  VG_BW_1                  = 12,
  /* {A,X}RGB channel ordering */
  VG_sXRGB_8888           =  0 | (1 << 6),
  VG_sARGB_8888           =  1 | (1 << 6),
  VG_sARGB_8888_PRE       =  2 | (1 << 6),
  VG_sARGB_1555           =  4 | (1 << 6),
  VG_sARGB_4444           =  5 | (1 << 6),
  VG_lXRGB_8888           =  7 | (1 << 6),
  VG_lARGB_8888           =  8 | (1 << 6),
  VG_lARGB_8888_PRE       =  9 | (1 << 6),

  /* BGR{A,X} channel ordering */
  VG_sBGRX_8888           =  0 | (1 << 7),
  VG_sBGRA_8888           =  1 | (1 << 7),
  VG_sBGRA_8888_PRE       =  2 | (1 << 7),
  VG_sBGR_565             =  3 | (1 << 7),
  VG_sBGRA_5551           =  4 | (1 << 7),
  VG_sBGRA_4444           =  5 | (1 << 7),
  VG_lBGRX_8888           =  7 | (1 << 7),
  VG_lBGRA_8888           =  8 | (1 << 7),
  VG_lBGRA_8888_PRE       =  9 | (1 << 7),

  /* {A,X}BGR channel ordering */
  VG_sXBGR_8888           =  0 | (1 << 6) | (1 << 7),
  VG_sABGR_8888           =  1 | (1 << 6) | (1 << 7),
  VG_sABGR_8888_PRE       =  2 | (1 << 6) | (1 << 7),
  VG_sABGR_1555           =  4 | (1 << 6) | (1 << 7),
  VG_sABGR_4444           =  5 | (1 << 6) | (1 << 7),
  VG_lXBGR_8888           =  7 | (1 << 6) | (1 << 7),
  VG_lABGR_8888           =  8 | (1 << 6) | (1 << 7),
  VG_lABGR_8888_PRE       =  9 | (1 << 6) | (1 << 7)
} VGImageFormat;
```

The letter `A` denotes an alpha (α) channel , `R` denotes red, `G` denotes green, and `B` denotes blue. `X` denotes a padding byte that is ignored. `L` denotes grayscale, and `BW` denotes (linear) bi-level grayscale (black-and-white), with 0 representing black and 1 representing white in either case. A lower-case letter `s` represents a non-linear, perceptually-uniform color space, as in sRGB and sL; a lower-case letter `l` represents a linear color space using the sRGB primaries. Formats with a suffix of `_PRE` store pixel values in premultiplied format.

Bit 6 of the numeric values of the enumeration indicates the position of the alpha channel (or unused byte for formats that do not include alpha). If bit 6 is disabled, the alpha or unused channel appears as the last channel, otherwise it appears as the first channel. Bit 7 indicates the ordering of the RGB color channels. If bit 7 is disabled, the color channels appear in RGB order, otherwise they appear in BGR order.

The `VG_A_8` format is treated as though it were `VG_lRGBA_8888`, with R=G=B=1. Color information is discarded when placing an RGBA value into a `VG_A_8` pixel.

Abbreviated names such as `lL` or `sRGBA_PRE` are used in this document where the exact number of bits per channel is not relevant, such as when pixel values are considered to have been remapped to a [0, 1] range. Such abbreviated names are not an official part of the API.

The bits for each color channel are stored within a machine word representing a single pixel from left to right (MSB to LSB) in the order indicated by the pixel format name. For example, in a pixel with a format of `VG_sRGB_565`, the bits representing the red channel may be obtained by shifting right by 11 bits (to remove 6 bits of green and 5 bits of blue) and masking with the 5-bit wide mask value 0x1f. Note that this definition is independent of the endianness of the underlying platform as sub-word memory addresses are not involved.

Table 11 summarizes the symbols used in image format names.

Table 12 lists the size of a single pixel for each image format, in terms of bytes and bits. Note that all formats other than VG_BW_1 use a whole number of bytes per pixel.

Formats having linear-light coding (`VG_lRGBX_8888`, `VG_lRGBA_8888`, `VG_lRGBA_8888_PRE`, and `VG_lL8`) are liable to exhibit banding (or contouring) artifacts when viewed with a contrast ratio greater than about 10:1 [POYN03] and are intended mainly for inputting existing linearly-coded imagery. For high-quality imaging, consider using one of the non-linear, perceptually uniform image formats such as `VG_sRGBX_8888`, `VG_sRGBA_8888`, `VG_sRGBA_8888_PRE`, and `VG_sL_8`.

Symbol	***Interpretation***
`A`	Alpha channel
`R`	Red color channel
`G`	Green color channel
`B`	Blue color channel
`X`	Uninterpreted padding byte
`L`	Grayscale
`BW`	1-bit Black and White
`l`	Linear color space
`s`	Non-linear (sRGB) color space
`PRE`	Alpha values are premultiplied

Table 11: Symbols Used in Image Format Names

Format	*Bytes Per Pixel*	*Bits Per Pixel*
`VG_sRGBX_8888`	4	32
`VG_sRGBA_8888`	4	32
`VG_sRGBA_8888_PRE`	4	32
`VG_sRGB_565`	2	16
`VG_sRGBA_5551`	2	16
`VG_sRGBA_4444`	2	16
`VG_sL_8`	1	8
`VG_lRGBX_8888`	4	32
`VG_lRGBA_8888`	4	32
`VG_lRGBA_8888_PRE`	4	32
`VG_lL_8`	1	8
`VG_A_8`	1	8
`VG_BW_1`	n/a	1

Table 12: Image Format Pixel Sizes

10.3 Creating and Destroying Images

VGImage

Images are accessed using opaque handles of type `VGImage`.

```
typedef VGHandle VGImage;
```

VGImageQuality

The `VGImageQuality` enumeration defines varying levels of resampling quality to be used when drawing images.

The `VG_IMAGE_QUALITY_NONANTIALIASED` setting disables resampling; images are drawn using point sampling (also known as nearest-neighbor interpolation) only. `VG_IMAGE_QUALITY_FASTER` enables low-to-medium quality resampling that does not require extensive additional resource allocation. `VG_IMAGE_QUALITY_BETTER` enables high-quality resampling that may allocate additional memory for pre-filtering, tables, and the like. Implementations are not required to provide three distinct resampling algorithms, but the non-antialiased (point sampling) mode must be supported.

```
typedef enum {
  VG_IMAGE_QUALITY_NONANTIALIASED  = (1 << 0),
  VG_IMAGE_QUALITY_FASTER          = (1 << 1),
  VG_IMAGE_QUALITY_BETTER          = (1 << 2)
} VGImageQuality;
```

Use **vgSeti** with a parameter type of `VG_IMAGE_QUALITY` to set the filter type to be used for image drawing:

```
VGImageQuality quality;
vgSeti(VG_IMAGE_QUALITY, quality);
```

VG_MAX_IMAGE_WIDTH

The `VG_MAX_IMAGE_WIDTH` read-only parameter contains the largest legal value of the `width` parameter to the **vgCreateImage** function. All implementations must define `VG_MAX_IMAGE_WIDTH` to be an integer no smaller than 256. If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. The value may be retrieved by calling **vgGeti**:

```
VGint imageMaxWidth = vgGeti(VG_MAX_IMAGE_WIDTH);
```

VG_MAX_IMAGE_HEIGHT

The `VG_MAX_IMAGE_HEIGHT` read-only parameter contains the largest legal value of the `height` parameter to the **vgCreateImage** function. All implementations must define `VG_MAX_IMAGE_HEIGHT` to be an integer no smaller than 256. If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. The value may be retrieved by calling **vgGeti**:

```
VGint imageMaxHeight = vgGeti(VG_MAX_IMAGE_HEIGHT);
```

VG_MAX_IMAGE_PIXELS

The `VG_MAX_IMAGE_PIXELS` read-only parameter contains the largest legal value of the product of the `width` and `height` parameters to the **vgCreateImage** function. All implementations must define `VG_MAX_IMAGE_PIXELS` to be an integer no smaller than 65536. If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. The value may be retrieved by calling **vgGeti**:

```
VGint imageMaxPixels = vgGeti(VG_MAX_IMAGE_PIXELS);
```

VG_MAX_IMAGE_BYTES

The `VG_MAX_IMAGE_BYTES` read-only parameter contains the largest number of bytes that may make up the image data passed to the **vgCreateImage** function. All implementations must define `VG_MAX_IMAGE_BYTES` to be an integer no smaller than 65536. If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. The value may be retrieved by calling **vgGeti**:

```
VGint imageMaxBytes = vgGeti(VG_MAX_IMAGE_BYTES);
```

vgCreateImage

vgCreateImage creates an image with the given `width`, `height`, and pixel `format` and returns a `VGImage` handle to it. If an error occurs, `VG_INVALID_HANDLE` is returned. All color and alpha channel values are initially set to zero. The `format` parameter must contain a value from the `VGImageFormat` enumeration.

The `allowedQuality` parameter is a bitwise OR of values from the `VGImageQuality` enumeration, indicating which levels of resampling quality may be used to draw the image. It is always possible to draw an image using the `VG_IMAGE_QUALITY_NONANTIALIASED` quality setting even if it is not explicitly specified.

```
VGImage vgCreateImage(VGImageFormat format,
                      VGint width, VGint height,
                      VGbitfield allowedQuality)
```

ERRORS

`VG_UNSUPPORTED_IMAGE_FORMAT_ERROR`

– if format is not a valid value from the `VGImageFormat` enumeration

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` are less than or equal to 0
– if `width` is greater than `VG_MAX_IMAGE_WIDTH`
– if `height` is greater than `VG_MAX_IMAGE_HEIGHT`
– if `width*height` is greater than `VG_MAX_IMAGE_PIXELS`
– if `width*height*(`pixel size of `format)` is greater than `VG_MAX_IMAGE_BYTES`
– if `allowedQuality` is not a bitwise OR of values from the `VGImageQuality` enumeration

vgDestroyImage

The resources associated with an image may be deallocated by calling **vgDestroyImage.** Following the call, the `image` handle is no longer valid in any context that shared it. If the image is currently in use as a rendering target, is the ancestor of another image (see **vgChildImage**), or is set as a paint pattern image on a VGPaint object, its definition remains available to those consumers as long as they remain valid, but the handle may no longer be used. When those uses cease, the image's resources will automatically be deallocated.

```
void vgDestroyImage(VGImage image);
```

ERRORS

VG_BAD_HANDLE_ERROR

– if image is not a valid image handle, or is not shared with the current context

10.4 Querying Images

VGImageParamType

Values from the VGImageParamType enumeration may be used as the paramType argument to **vgGetParameter** to query various features of an image. All of the parameters defined by VGImageParamType have integer values and are read-only.

```
typedef enum {
  VG_IMAGE_FORMAT = 0x1E00,
  VG_IMAGE_WIDTH  = 0x1E01,
  VG_IMAGE_HEIGHT = 0x1E02
} VGImageParamType;
```

Image Format

The value of the format parameter that was used to define the image may be queried using the VG_IMAGE_FORMAT parameter. The returned integral value should be cast to the VGImageFormat enumeration:

```
VGImage image;
VGImageFormat imageFormat =
  (VGImageFormat)vgGetParameteri(image, VG_IMAGE_FORMAT);
```

Image Width

The value of the width parameter that was used to define the image may be queried using the VG_IMAGE_WIDTH parameter:

```
VGImage image;
VGint imageWidth = vgGetParameteri(image, VG_IMAGE_WIDTH);
```

Image Height

The value of the `height` parameter that was used to define the image may be queried using the `VG_IMAGE_HEIGHT` parameter:

```
VGImage image;
VGint imageHeight = vgGetParameteri(image, VG_IMAGE_HEIGHT);
```

10.5 Reading and Writing Image Pixels

vgClearImage

The **vgClearImage** function fills a given rectangle of an image with the color specified by the `VG_CLEAR_COLOR` parameter. The rectangle to be cleared is given by `x`, `y`, `width`, and `height`, which must define a positive region. The rectangle is clipped to the bounds of the image.

```
void vgClearImage(VGImage image,
                  VGint x, VGint y, VGint width, VGint height)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `image` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if `image` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` is less than or equal to 0

vgImageSubData

The **vgImageSubData** function reads pixel values from memory, performs format conversion if necessary, and stores the resulting pixels into a rectangular portion of an `image`.

Pixel values are read starting at the address given by the pointer `data`; adjacent scanlines are separated by `dataStride` bytes. Negative or zero values of `dataStride` are allowed. The region to be written is given by `x`, `y`, `width`, and `height`, which must define a positive region. Pixels that fall outside the bounds of the image are ignored.

Pixel values in memory are formatted according to the `dataFormat` parameter, which must contain a value from the `VGImageFormat` enumeration. The `data` pointer must be aligned according to the number of bytes of the pixel format specified by `dataFormat`, unless `dataFormat` is equal to `VG_BW_1`, in which case 1 byte alignment is sufficient. Each pixel is converted into the format of the destination image as it is written.

If `dataFormat` is not equal to `VG_BW_1`, the destination image pixel (x + *i*, y + *j*) for $0 \le i <$ `width` and $0 \le j <$ `height` is taken from the *N* bytes of memory starting at *data* + *j*dataStride* + *i*N*, where *N* is the number of bytes per pixel given in Table 12. For multi-byte pixels, the bits are arranged in the same order used to store native multi-byte primitive datatypes. For example, a 16-bit pixel would be written to memory in the same format as when writing through a pointer with a native 16-bit integral datatype.

If `dataFormat` is equal to `VG_BW_1`, pixel (x + *i*, y + *j*) of the destination image is taken from the bit at position (*i % 8*) within the byte at *data* + *j*dataStride* + *floor*(*i/8*) where the least significant bit (LSB) of a byte is considered to be at position 0 and the most significant bit (MSB) is at position 7. Each scanline must be padded to a multiple of 8 bits. Note that `dataStride` is always given in terms of bytes, not bits.

If `dataFormat` specifies a premultiplied format (`VG_sRGBA_8888_PRE` or `VG_lRGBA_8888_PRE`), color channel values of a pixel greater than their corresponding alpha value are clamped to the alpha value.

```
void vgImageSubData(VGImage image,
                    const void * data, VGint dataStride,
                    VGImageFormat dataFormat,
                    VGint x, VGint y, VGint width, VGint height)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `image` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if `image` is currently a rendering target

`VG_UNSUPPORTED_IMAGE_FORMAT_ERROR`

– if `dataFormat` is not a valid value from the `VGImageFormat` enumeration

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` is less than or equal to 0

– if `data` is `NULL`

– if `data` is not properly aligned

vgGetImageSubData

The **vgGetImageSubData** function reads pixel values from a rectangular portion of an `image`, performs format conversion if necessary, and stores the resulting pixels into memory.

Pixel values are written starting at the address given by the pointer `data`; adjacent scanlines are separated by `dataStride` bytes. Negative or zero values of `dataStride` are allowed. The region to be read is given by `x`, `y`, `width`, and `height`, which must define a positive region. Pixels that fall outside the bounds of the image are ignored.

Pixel values in memory are formatted according to the `dataFormat` parameter, which must contain a value from the `VGImageFormat` enumeration. The `data` pointer must be aligned according to the number of bytes of the pixel format specified by `dataFormat`, unless `dataFormat` is equal to `VG_BW_1`, in which case 1 byte alignment is sufficient. Each pixel is converted from the format of the source image as it is read.

The pixel layout in memory is identical to that of **vgImageSubData**.

```
void vgGetImageSubData(VGImage image,
                       void * data, VGint dataStride,
                       VGImageFormat dataFormat,
                       VGint x, VGint y, VGint width, VGint height)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `image` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if `image` is currently a rendering target

`VG_UNSUPPORTED_IMAGE_FORMAT_ERROR`

– if `dataFormat` is not a valid value from the `VGImageFormat` enumeration

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` is less than or equal to 0

– if `data` is `NULL`

– if `data` is not properly aligned

10.6 Child Images

A *child image* is an image that shares physical storage with a portion of an existing image, known as its *parent*. An image may have any number of children, but each image has only one parent (that may be itself). An *ancestor* of an image is defined as the image itself, its parent, its parent's parent, etc. Thus a pair of images share storage if and only if they have a common ancestor. Changes to an image are immediately reflected in all other images that share storage with it.

A child image remains valid even following a call to **vgDestroyImage** on one of its ancestors (other than itself). When the last image of a set of images that share pixel storage is destroyed, the storage will be reclaimed. Implementations may use a reference count to determine when image storage may be reclaimed.

An image that shares storage with any other image may not be used as a rendering target until all the images with which it shares storage have been destroyed.

vgChildImage

The **vgChildImage** function returns a new `VGImage` handle that refers to a portion of the `parent` image. The region is given by the intersection of the bounds of the parent image with the rectangle beginning at pixel (`x`, `y`) with dimensions `width` and `height`, which must define a positive region contained entirely within `parent`.

```
VGImage vgChildImage(VGImage parent,
                     VGint x, VGint y, VGint width, VGint height)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `parent` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if `parent` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `x` is less than 0 or greater than or equal to the parent width

– if `y` is less than 0 or greater than or equal to the parent height

– if `width` or `height` is less than or equal to 0

– if `x + width` is greater than the parent width

– if `y + height` is greater than the parent height

vgGetParent

The **vgGetParent** function returns the parent of the given `image`. If `image` has no parent, `image` is returned.

```
VGImage vgGetParent(VGImage image)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `image` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if `image` is currently a rendering target

10.7 Copying Pixels Between Images

vgCopyImage

Pixels may be copied between images using the **vgCopyImage** function. The source image pixel (`sx` + *i*, `sy` + *j*) is copied to the destination image pixel (`dx` + *i*, `dy` + *j*), for $0 \le i <$ `width` and $0 \le j <$ `height`. Pixels whose source or destination lie outside of the bounds of the respective image are ignored. Pixel format conversion is applied as needed.

If the `dither` flag is equal to `VG_TRUE`, an implementation-dependent dithering algorithm may be applied. This may be useful when copying into a destination image with a smaller color bit depth than that of the source image. Implementations should choose an algorithm that will provide good results when the output images are displayed as successive frames in an animation.

If `src` and `dst` are the same image, or have a common ancestor and thus share storage, the copy will occur in a consistent fashion as though the source pixels were first copied into a temporary buffer and then copied from the temporary buffer to the destination.

```
void vgCopyImage(VGImage dst, VGint dx, VGint dy,
                 VGImage src, VGint sx, VGint sy,
                 VGint width, VGint height,
                 VGboolean dither)
```

ERRORS

VG_BAD_HANDLE_ERROR

– if either dst or src is not a valid image handle, or is not shared with the current context

VG_IMAGE_IN_USE_ERROR

– if either dst or src is currently a rendering target

VG_ILLEGAL_ARGUMENT_ERROR

– if width or height is less than or equal to 0

10.8 Drawing Images to the Drawing Surface

Images may be drawn onto a drawing surface. An affine or projective transformation may be applied while drawing. The current image and blending modes are used to control how image pixels are combined with the current paint and blended into the destination. Conversion between the image and destination pixel formats is applied automatically.

VGImageMode

The VGImageMode enumeration is used to select between several styles of image drawing, described in the **vgDrawImage** section below.

```
typedef enum {
  VG_DRAW_IMAGE_NORMAL   = 0x1F00,
  VG_DRAW_IMAGE_MULTIPLY = 0x1F01,
  VG_DRAW_IMAGE_STENCIL  = 0x1F02
} VGImageMode;
```

To set the image drawing mode, use **vgSeti** with a paramType value of VG_IMAGE_MODE:

```
VGImageMode drawImageMode;
vgSeti(VG_IMAGE_MODE, drawImageMode);
```

vgDrawImage

An image may be drawn to the current drawing surface using the **vgDrawImage** function. The current image-user-to-surface transformation *Ti* is applied to the image, so that the image pixel centered at (*px* + ½, *py* + ½) is mapped to the point (*Ti*)(*px* + ½, *py* + ½). In practice, backwards mapping may be used. That is, a sample located at (*x*, *y*) in the surface coordinate system is colored according to an interpolated image pixel value at the point $(Ti)^{-1}$(*x*, *y*) in the image coordinate system. If *Ti* is non-invertible (or nearly so, within the limits of numerical accuracy), no drawing occurs.

Interpolation is done in the color space of the image. Image color values are processed in premultiplied alpha format during interpolation.

When a projective transformation is used (*i.e.*, the bottom row of the image-user-to-surface transformation contains values [w_0 w_1 w_2] different from [0 0 1]), each corner point (x, y) of the image must result in a positive value of $d = (x*w_0 + y*w_1 + w_2)$, or else nothing is drawn. This rule prevents degeneracies due to transformed image points passing through infinity, which occurs when d passes through 0. By requiring d to be positive at the corners, it is guaranteed to be positive at all interior points as well.

When a projective transformation is used, the value of the `VG_IMAGE_MODE` parameter is ignored and the behavior of `VG_DRAW_IMAGE_NORMAL` is substituted. This avoids the need to generate paint pixels in perspective.

The set of pixels affected consists of the quadrilateral with vertices (*Ti*)(0, 0), (*Ti*)(*w*, 0), (*Ti*)(*w*, *h*), and (*Ti*)(0, *h*) (where *w* and *h* are respectively the width and height of the image), plus a boundary of up to 1½ pixels for filtering purposes.

Clipping, masking, and scissoring are applied in the same manner as with **vgDrawPath**. To limit drawing to a subregion of the image, create a child image using **vgChildImage**.

The image quality will be the maximum quality allowed by the image (as determined by the `allowedQuality` parameter to **vgCreateImage**) that is not higher than the current setting of `VG_IMAGE_QUALITY`.

```
void vgDrawImage(VGImage image)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `image` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if `image` is currently a rendering target

The effects of **vgDrawImage** depend on the current setting of the `VG_IMAGE_MODE` parameter:

VG_DRAW_IMAGE_NORMAL

When the `VG_IMAGE_MODE` parameter is set to `VG_DRAW_IMAGE_NORMAL`, the image is drawn. If the image contains an alpha channel, the alpha values associated with each pixel are used as the source alpha values. Otherwise, the source alpha is taken to be 1 at each pixel. No paint generation takes place. When a projective transformation is used, this mode is used regardless of the setting of the `VG_IMAGE_MODE` parameter.

VG_DRAW_IMAGE_MULTIPLY

When the `VG_IMAGE_MODE` parameter is set to `VG_DRAW_IMAGE_MULTIPLY`, the image being drawn is multiplied by the paint color and alpha values. This allows the image to be drawn translucently (by setting the paint color to R=G=B=1 and A=opacity), or to be modulated in other ways. For example, a gradient paint could be used to create a fading effect, or a pattern paint could be used to vary the opacity on a pixel-by-pixel basis. If the paint color is opaque white (R=G=B=A=1) everywhere, the results are equivalent to those of `VG_DRAW_IMAGE_NORMAL`.

Paint generation (using the `VGPaint` object defined for the `VG_FILL_PATH` paint mode) occurs at each pixel, and the interpolated image and paint color and alpha values are multiplied channel-by-channel. The result (considered to be in the same color space as the image) is used as the input to the current blend function and normal blending takes place.

Note that the use of a source image having a linear pixel format (e.g., `lRGB_888`) will result in a brightened output due to the fact that the paint values are not converted from sRGB to linear, yet the results are treated as linear. Therefore the use of a linear source image in this mode is recommended only for special effects.

VG_DRAW_IMAGE_STENCIL

When the `VG_IMAGE_MODE` parameter is set to `VG_DRAW_IMAGE_STENCIL`, the image being drawn acts as a stencil through which the current paint is applied. This allows an image to take the place of a geometric path definition in some cases, such as drawing text glyphs. A special set of blending equations allows the red, green, and blue channels to be blended using distinct alpha values taken from the image. This feature allows stencils to take advantage of sub-pixel effects on LCD displays.

Paint generation (using the `VGPaint` object defined for the `VG_FILL_PATH` paint mode) occurs at each pixel. The interpolated image and paint color and alpha values are combined at each pixel as follows. Each image color channel value is multiplied by its corresponding alpha value (if the image has an alpha channel) and by the paint alpha value to produce an alpha value associated with that color channel. The result is considered to be in the same color space as the paint (*i.e.*, sRGB for all forms of paint except pattern paint with a linear pattern image). The current blending equation (see Section 12) is applied separately for each destination color channel, using the alpha value computed above as the source alpha value for the blend, and the paint color value as the source color value.

In terms of the blending functions $\alpha(\alpha_{src}, \alpha_{dst})$ and $c(c_{src}, c_{dst}, \alpha_{src}, \alpha_{dst})$ defined in Section 12.1, the stenciled output color and alpha values are:

$$\begin{aligned}
\alpha_{tmp} &= \alpha(\alpha_{image}*\alpha_{paint}, \alpha_{dst}) \\
R_{dst} &\leftarrow c(R_{paint}, R_{dst}, R_{image}*\alpha_{image}*\alpha_{paint}, \alpha_{dst}) \;/\; \alpha_{tmp} \\
G_{dst} &\leftarrow c(G_{paint}, G_{dst}, G_{image}*\alpha_{image}*\alpha_{paint}, \alpha_{dst}) \;/\; \alpha_{tmp} \\
B_{dst} &\leftarrow c(B_{paint}, B_{dst}, B_{image}*\alpha_{image}*\alpha_{paint}, \alpha_{dst}) \;/\; \alpha_{tmp} \\
\alpha_{dst} &\leftarrow \alpha_{tmp}
\end{aligned}$$

For example, if Porter-Duff "Src **over** Dst" blending is enabled (see Section 12.2), the destination alpha and color values are computed as:

$$\begin{aligned}
\alpha_{tmp} &= (\alpha_{image}*\alpha_{paint} + \alpha_{dst}*(1 - \alpha_{image}*\alpha_{paint})) \\
R_{dst} &\leftarrow (\alpha_{image}*\alpha_{paint}*R_{image}*R_{paint} \quad + \alpha_{dst}*R_{dst}*(1 - \alpha_{image}*\alpha_{paint}*R_{image})) \;/\; \alpha_{tmp} \\
G_{dst} &\leftarrow (\alpha_{image}*\alpha_{paint}*G_{image}*G_{paint} \quad + \alpha_{dst}*G_{dst}*(1 - \alpha_{image}*\alpha_{paint}*G_{image})) \;/\; \alpha_{tmp} \\
B_{dst} &\leftarrow (\alpha_{image}*\alpha_{paint}*B_{image}*B_{paint} \quad + \alpha_{dst}*B_{dst}*(1 - \alpha_{image}*\alpha_{paint}*B_{image})) \;/\; \alpha_{tmp} \\
\alpha_{dst} &\leftarrow \alpha_{tmp}
\end{aligned}$$

10.9 Reading and Writing Drawing Surface Pixels

Several functions are provided to read and write pixels on the drawing surface directly, without applying transformations, masking, or blending.

10.9.1 Writing Drawing Surface Pixels

vgSetPixels

The **vgSetPixels** function copies pixel data from the image `src` onto the drawing surface. The image pixel (`sx` + *i*, `sy` + *j*) is copied to the drawing surface pixel (`dx` + *i*, `dy` + *j*), for $0 \le i <$ `width` and $0 \le j <$ `height`. Pixels whose source lies outside of the bounds of `src` or whose destination lies outside the bounds of the drawing surface are ignored. Pixel format conversion is applied as needed. Scissoring takes place normally. Transformations, masking, and blending are not applied.

```
void vgSetPixels(VGint dx, VGint dy,
                 VGImage src, VGint sx, VGint sy,
                 VGint width, VGint height)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `src` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if `src` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` is less than or equal to 0

vgWritePixels

The **vgWritePixels** function allows pixel data to be copied to the drawing surface without the creation of a `VGImage` object. The pixel values to be drawn are taken from the data pointer at the time of the **vgWritePixels** call, so future changes to the data have no effect. The effects of changes to the data by another thread at the time of the call to **vgWritePixels** are undefined.

The `dataFormat` parameter must contain a value from the `VGImageFormat`. If `dataFormat` is not equal to `VG_BW_1`, `data` must be aligned according to the number of bytes of `dataFormat`, and the pixel at memory location *data* + *j***dataStride* + *i**(*bytes per pixel of dataFormat*) is written to the drawing surface pixel (`dx` + *i*, `dy` + *j*), for $0 \le i <$ `width` and $0 \le j <$ `height`. If `dataFormat` is equal to `VG_BW_1`, data must only be 1-byte aligned, and pixel (`dx` + *i*, `dy` + *j*) of the destination image is taken from the bit at position (*i % 8*) within the byte at *data* + *j***dataStride* + *floor*(*i/8*) where the least significant bit (LSB) of a byte is considered to be at position 0 and the most significant bit (MSB) is at position 7.

If `dataFormat` specifies a premultiplied format (`VG_sRGBA_8888_PRE` or `VG_lRGBA_8888_PRE`), color channel values of a pixel greater than their corresponding alpha value are clamped to the alpha value.

Pixels whose destination coordinate lies outside the bounds of the drawing surface are ignored. Pixel format conversion is applied as needed. Scissoring takes place normally. Transformations, masking, and blending are not applied.

```
void vgWritePixels(const void * data, VGint dataStride,
                   VGImageFormat dataFormat,
                   VGint dx, VGint dy,
                   VGint width, VGint height)
```

ERRORS

`VG_UNSUPPORTED_IMAGE_FORMAT_ERROR`

– if `dataFormat` is not a valid value from the `VGImageFormat` enumeration

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` is less than or equal to 0
– if `data` is `NULL`
– if `data` is not properly aligned

The code:

```
void * data;
VGImageFormat dataFormat;
VGint dataStride;
VGint dx, dy, width, height;

vgWritePixels(data, dataStride, dataFormat, dx, dy, width, height);
```

is equivalent to the code:

```
VGImage image;
void * data;
VGImageFormat dataFormat;
VGint dataStride;
VGint dx, dy, width, height;

image = vgCreateImage(dataFormat, width, height, 0);
vgImageSubData(image, data, dataStride, dataFormat,
                   0, 0, width, height);
vgSetPixels(dx, dy, image, width, height);
vgDestroyImage(image);
```

10.9.2 Reading Drawing Surface Pixels

vgGetPixels

The **vgGetPixels** function retrieves pixel data from the drawing surface into the image `dst`. The drawing surface pixel (`sx` + *i*, `sy` + *j*) is copied to pixel (`dx` + *i*, `dy` + *j*) of the image `dst`, for $0 \le i <$ `width` and $0 \le j <$ `height`. Pixels whose source lies outside of the bounds of the drawing surface or whose destination lies outside the bounds of `dst` are ignored. Pixel format conversion is applied as needed. The scissoring region does not affect the reading of pixels.

```
void vgGetPixels(VGImage dst, VGint dx, VGint dy,
                 VGint sx, VGint sy,
                 VGint width, VGint height)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if `dst` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if `dst` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` is less than or equal to 0

vgReadPixels

The **vgReadPixels** function allows pixel data to be copied from the drawing surface without the creation of a `VGImage` object.

The `dataFormat` parameter must contain a value from the `VGImageFormat` enumeration. If `dataFormat` is not equal to `VG_BW_1`, `data` must be aligned according to the number of bytes of `dataFormat`, and the drawing surface pixel (`sx` + *i*, `sy` + *j*) is written to the memory location *data* + *j*dataStride* + *i**(*bytes per pixel of dataFormat*), for *0* ≤ *i* < `width` *and 0* ≤ *j* < `height`. If `dataFormat` is equal to `VG_BW_1`, `data` must only be 1-byte aligned, and the drawing surface pixel (`sx` + *i*, `sy` + *j*) is written to the bit at position (*i % 8*) within the byte at *data* + *j*dataStride* + *floor*(*i/8*) where the least significant bit (LSB) of a byte is considered to be at position 0 and the most significant bit (MSB) is at position 7.

Pixels whose source lies outside of the bounds of the drawing surface are ignored. Pixel format conversion is applied as needed. The scissoring region does not affect the reading of pixels.

```
void vgReadPixels(void * data, VGint dataStride,
                  VGImageFormat dataFormat,
                  VGint sx, VGint sy,
                  VGint width, VGint height)
```

ERRORS

`VG_UNSUPPORTED_IMAGE_FORMAT_ERROR`

– if `dataFormat` is not a valid value from the `VGImageFormat` enumeration

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` is less than or equal to 0

– if `data` is `NULL`

– if `data` is not properly aligned

The code:

```
void * data;
VGImageFormat dataFormat;
VGint dataStride;
VGint sx, sy, width, height;

vgReadPixels(data, dataStride, dataFormat, sx, sy, width, height);
```

is equivalent to the code:

```
VGImage image;
void * data;
VGint dataStride;
VGImageFormat dataFormat;
VGint sx, sy, width, height;

image = vgCreateImage(dataFormat, width, height, 0);
vgGetPixels(image, 0, 0, sx, sy, width, height);
vgGetImageSubData(image, data, dataStride, dataFormat, width, height);
vgDestroyImage(image);
```

10.10 Copying Portions of the Drawing Surface

vgCopyPixels

The **vgCopyPixels** function copies pixels from one region of the drawing surface to another. Copies between overlapping regions are allowed and always produce consistent results identical to copying the entire source region to a scratch buffer followed by copying the scratch buffer into the destination region.

The drawing surface pixel (`sx` + *i*, `sy` + *j*) is copied to pixel (`dx` + *i*, `dy` + *j*) for $0 \le i <$ `width` and $0 \le j <$ `height`. Pixels whose source or destination lies outside of the bounds of the drawing surface are ignored. Transformations, masking, and blending are not applied. Scissoring is applied to the destination, but does not affect the reading of pixels.

```
void vgCopyPixels(VGint dx, VGint dy,
                  VGint sx, VGint sy,
                  VGint width, VGint height)
```

ERRORS

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` is less than or equal to 0

11 Image Filters

Image filters allow images to be modified and/or combined using a variety of imaging operations. Operations are carried out using a bit depth greater than or equal to the largest bit depth of the supplied images. The lower-left corners of all source and destination images are aligned. The destination area to be written is the intersection of the source and destination image areas. The source and destination images involved in the filter operation must not overlap (*i.e.*, have any pixels in common within any common ancestor image). Source and destination images may have a common ancestor as long as they occupy disjoint areas within that area.

11.1 Format Normalization

A series of steps are carried out on application-supplied source images in order to produce normalized source images for filtering. In practice, these normalizations may be combined with the filter operations themselves for efficiency.

The source pixels are converted to one of `sRGBA`, `sRGBA_PRE`, `lRGBA`, or `lRGBA_PRE` formats, as determined by the current values of the `VG_FILTER_FORMAT_PREMULTIPLIED` and `VG_FILTER_FORMAT_LINEAR` parameters. The conversions take place in the following order (equivalent to the conversion rules defined in Section 3.4):

1) Source color and alpha values are scaled linearly to lie in a [0, 1] range. The exact precision of the internal representation is implementation-dependent.
2) If the source image has premultiplied alpha, the alpha values are divided out of each source color channel, and stored for later use. If the source image has no alpha channel, an alpha value of 1 is added to each pixel.
3) If the source pixel is in a grayscale format (`lL` or `sL`), it is converted to an RGB format (`lRGB` or `sRGB,` respectively) by replication.
4) If the `VG_FILTER_FORMAT_LINEAR` parameter is set to `VG_TRUE`, and the source pixel is in non-linear format, it is converted into the corresponding linear format (`sRGBA`→`lRGBA`). If the `VG_FILTER_FORMAT_LINEAR` parameter is set to `VG_FALSE`, and the source pixel is in linear format, it is converted into the corresponding non-linear format (`lRGBA`→`sRGBA`).
5) If the `VG_FILTER_FORMAT_PREMULTIPLIED` parameter is equal to `VG_TRUE`, each source color channel is multiplied by the corresponding alpha value. Otherwise, the color channels are left undisturbed.

An implementation may collapse steps algebraically; for example, if no conversion is to take place in step 4, the division and multiplication by alpha in steps 2 and 5 may be implemented as a no-op.

The resulting pixel will be in sRGBA, sRGBA_PRE, lRGBA, or lRGBA_PRE format. The image filter then processes each of the four source channels in an identical manner, resulting in a set of filtered pixels in the same pixel format as the incoming pixels.

Finally, the filtered pixels are converted into the destination format using the normal pixel format conversion rules, as described in section 3.4. Premultiplied alpha values are divided out prior to color-space conversion, and restored afterwards if necessary. The destination channels specified by the VG_FILTER_CHANNEL_MASK parameter (see below) are written into the destination image.

11.2 Channel Masks

VGImageChannel

All image filter functions make use of the VG_FILTER_CHANNEL_MASK parameter that specifies which destination channels are to be written. The parameter is supplied as a bitwise OR of values from the VGImageChannel enumeration. If the destination pixel format is one of VG_sL_8, VG_lL_8 or VG_BW_1 pixel format, the parameter is ignored. If the destination pixel format does not contain an alpha channel, the VG_ALPHA bit is ignored. Bits other than those defined by the VGImageChannel enumeration are ignored.

VG_FILTER_CHANNEL_MASK controls which color channels of the filtered image are written into the destination image. In the case where the destination image is premultiplied, and VG_FILTER_CHANNEL_MASK does not specify that all channels are to be written, the following steps are taken to ensure consistency:

1. If VG_FILTER_FORMAT_PREMULTIPLIED is enabled, the filtered color channels are clamped between 0 and their corresponding alpha value, and converted into non-premultiplied form (as described in Section 3.4)
2. The resulting color is converted into the destination color space
3. The destination is read and converted into non-premultiplied form
4. The destination channels specified by VG_FILTER_CHANNEL_MASK are replaced by the corresponding filtered values
5. The results are converted into premultiplied form and written to the destination image

```
typedef enum {
  VG_RED   = (1 << 3),
  VG_GREEN = (1 << 2),
  VG_BLUE  = (1 << 1),
  VG_ALPHA = (1 << 0)
} VGImageChannel;
```

11.3 Color Combination

Color channel values may be combined using the **vgColorMatrix** function, which computes output colors as linear combinations of input colors.

vgColorMatrix

The **vgColorMatrix** function computes a linear combination of color and alpha values (R_{src}, G_{src}, B_{src}, α_{src}) from the normalized source image `src` at each pixel:

$$\begin{bmatrix} R_{dst} \\ G_{dst} \\ B_{dst} \\ \alpha_{dst} \end{bmatrix} = \begin{bmatrix} m_{00} & m_{01} & m_{02} & m_{03} \\ m_{10} & m_{11} & m_{12} & m_{13} \\ m_{20} & m_{21} & m_{22} & m_{23} \\ m_{30} & m_{31} & m_{32} & m_{33} \end{bmatrix} \cdot \begin{bmatrix} R_{src} \\ G_{src} \\ B_{src} \\ \alpha_{src} \end{bmatrix} + \begin{bmatrix} m_{04} \\ m_{14} \\ m_{24} \\ m_{34} \end{bmatrix}$$

or:

$$\begin{aligned} R_{dst} &= m_{00} R_{src} + m_{01} G_{src} + m_{02} B_{src} + m_{03} \alpha_{src} + m_{04} \\ G_{dst} &= m_{10} R_{src} + m_{11} G_{src} + m_{12} B_{src} + m_{13} \alpha_{src} + m_{14} \\ B_{dst} &= m_{20} R_{src} + m_{21} G_{src} + m_{22} B_{src} + m_{23} \alpha_{src} + m_{24} \\ \alpha_{dst} &= m_{30} R_{src} + m_{31} G_{src} + m_{32} B_{src} + m_{33} \alpha_{src} + m_{34} \end{aligned}$$

The matrix entries are supplied in the `matrix` argument in the order *{ m_{00}, m_{10}, m_{20}, m_{30}, m_{01}, m_{11}, m_{21}, m_{31}, m_{02}, m_{12}, m_{22}, m_{32}, m_{03}, m_{13}, m_{23}, m_{33}, m_{04}, m_{14}, m_{24}, m_{34} }*.

```
void vgColorMatrix(VGImage dst, VGImage src,
                   const VGfloat * matrix)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if either `dst` or `src` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if either `dst` or `src` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `src` and `dst` overlap

– if `matrix` is `NULL`

– if `matrix` is not properly aligned

11.4 Convolution

The **vgConvolve, vgSeparableConvolve**, and **vgGaussianBlur** functions define destination pixels based on a weighted average of neighboring source pixels, a process known as *convolution*. The set of weights, along with their relative locations, is known as the *convolution kernel*. In the discussion below, *width* and *height* refer to the dimensions of the source image.

VG_MAX_KERNEL_SIZE

The `VG_MAX_KERNEL_SIZE` parameter contains the largest legal value of the `width` and `height` parameters to the **vgConvolve** function. All implementations must define `VG_MAX_KERNEL_SIZE` to be an integer no smaller than 7. If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. The value may be retrieved by calling **vgGeti**:

```
VGint maxKernelSize = vgGeti(VG_MAX_KERNEL_SIZE);
```

VG_MAX_SEPARABLE_KERNEL_SIZE

The `VG_MAX_SEPARABLE_KERNEL_SIZE` parameter contains the largest legal value of the `size` parameter to the **vgSeparableConvolve** function. All implementations must define `VG_MAX_SEPARABLE_KERNEL_SIZE` to be an integer no smaller than 15. If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. The value may be retrieved by calling **vgGeti**:

```
VGint maxSeparableKernelSize = vgGeti(VG_MAX_SEPARABLE_KERNEL_SIZE);
```

VG_MAX_GAUSSIAN_STD_DEVIATION

The `VG_MAX_GAUSSIAN_STD_DEVIATION` parameter contains the largest legal value of the `stdDeviationX` and `stdDeviationY` parameters to the

vgGaussianBlur function. All implementations must define `VG_MAX_GAUSSIAN_STD_DEVIATION` to be an integer no smaller than 128. If there is no implementation-defined limit, a value of `VG_MAXINT` may be returned. The value may be retrieved by calling **vgGeti**:

```
VGint maxGaussianStdDeviation = vgGeti(VG_MAX_GAUSSIAN_STD_DEVIATION);
```

vgConvolve

The **vgConvolve** function applies a user-supplied convolution kernel to a normalized source image `src`. The dimensions of the kernel are given by `kernelWidth` and `kernelHeight`; the kernel values are specified as `kernelWidth*kernelHeight VGshorts` in column-major order. That is, the kernel entry (*i, j*) is located at position *i*kernelHeight* + *j* in the input sequence. The `shiftX` and `shiftY` parameters specify a translation between the source and destination images. The result of the convolution is multiplied by a `scale` factor, and a `bias` is added.

The output pixel (*x, y*) is defined as:

$$s\left(\sum_{0 \le i < w} \sum_{0 \le j < h} k_{(w-i-1),(h-j-1)}\, p(x+i-shiftX,\, y+j-shiftY)\right) + b\,,$$

where *w* = `kernelWidth`, *h* = `kernelHeight`, $k_{i,j}$ is the kernel element at position (*i, j*), *s* is the `scale`, *b* is the bias, and *p*(*x, y*) is the source pixel at (*x, y*), or the result of source edge extension defined by `tilingMode`, which takes a value from the `VGTilingMode` enumeration (see Section 9.4.1). Note that the use of the kernel index (*w–i–1, h–j–1*) implies that the kernel is rotated 180 degrees relative to the source image in order to conform to the mathematical definition of convolution. Figure 23 depicts the flipping of the kernel relative to the image pixels for a 3x3 kernel.

The operation is applied to all channels (color and alpha) independently.

```
void vgConvolve(VGImage dst, VGImage src,
                VGint kernelWidth, VGint kernelHeight,
                VGint shiftX, VGint shiftY,
                const VGshort * kernel,
                VGfloat scale,
                VGfloat bias,
                VGTilingMode tilingMode)
```

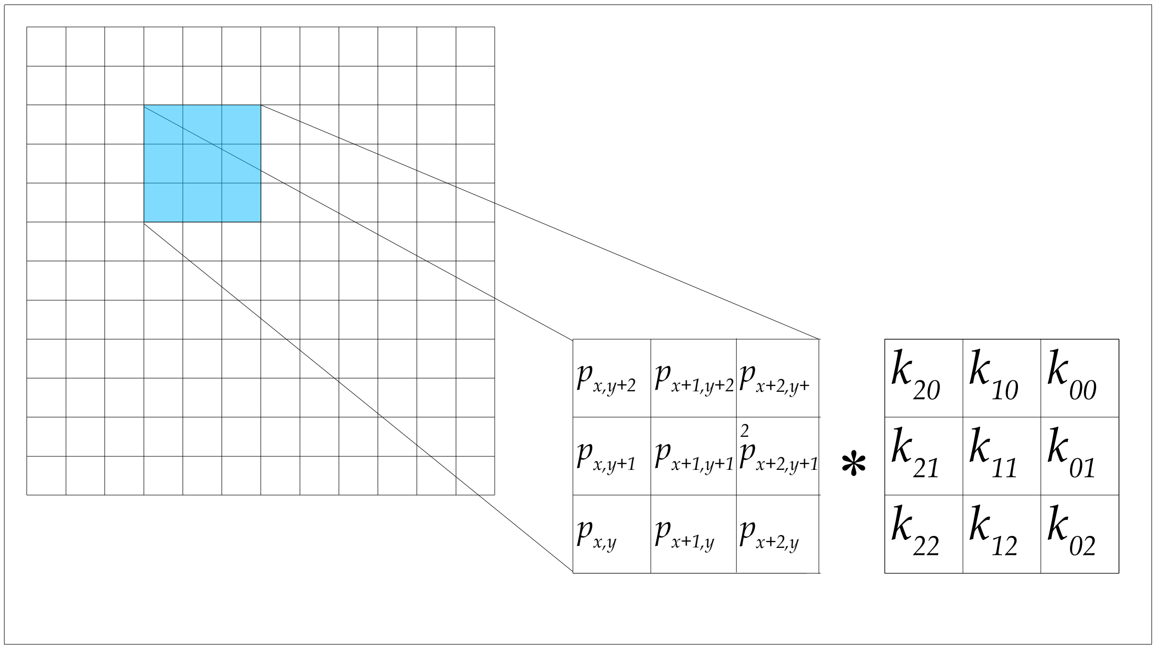


Figure 23: Convolution With a Flipped Kernel

ERRORS

`VG_BAD_HANDLE_ERROR`

– if either `dst` or `src` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if either `dst` or `src` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `src` and `dst` overlap
– if `kernelWidth` or `kernelHeight` is less than or equal to 0 or greater than `VG_MAX_KERNEL_SIZE`
– if `kernel` is `NULL`
– if `kernel` is not properly aligned
– if `tilingMode` is not one of the values from the `VGTilingMode` enumeration

vgSeparableConvolve

The **vgSeparableConvolve** function applies a user-supplied separable convolution kernel to a normalized source image `src`. A separable kernel is a two-dimensional kernel in which each entry k_{ij} is equal to a product $kx_i * ky_j$ of elements from two one-dimensional kernels, one horizontal and one vertical.

The lengths of the one-dimensional arrays `kernelX` and `kernelY` are given by `kernelWidth` and `kernelHeight`, respectively; the kernel values are specified as arrays of `VGshorts`. The `shiftX` and `shiftY` parameters specify a translation between the source and destination images. The result of the convolution is multiplied by a `scale` factor, and a `bias` is added.

The output pixel (x, y) is defined as:

$$s\Big(\sum_{0\le i<w}\sum_{0\le j<h} kx_{(w-i-1)}ky_{(h-j-i)}\,p\,(x+i-shiftX\,,\,y+j-shiftY)\Big)+b\,,$$

where w = `kernelWidth`, h = `kernelHeight`, kx_i is the one-dimensional horizontal kernel element at position i, ky_j is the one-dimensional vertical kernel element at position j, s is the `scale`, b is the bias, and $p(x, y)$ is the source pixel at (x, y), or the result of source edge extension defined by `tilingMode`, which takes a value from the `VGTilingMode` enumeration (see Section 9.4.1). Note that the use of the kernel indices $(w–i–1)$ and $(h–j–1)$ implies that the kernel is rotated 180 degrees relative to the source image in order to conform to the mathematical definition of convolution.

The operation is applied to all channels (color and alpha) independently.

```
void vgSeparableConvolve(VGImage dst, VGImage src,
                         VGint kernelWidth, VGint kernelHeight,
                         VGint shiftX, VGint shiftY,
                         const VGshort * kernelX,
                         const VGshort * kernelY,
                         VGfloat scale,
                         VGfloat bias,
                         VGTilingMode tilingMode)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

- if either `dst` or `src` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

- if either `dst` or `src` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

- if `src` and `dst` overlap
- if `kernelWidth` or `kernelHeight` is less than or equal to 0 or greater than `VG_MAX_SEPARABLE_KERNEL_SIZE`
- if `kernelX` or `kernelY` is `NULL`
- if `kernelX` or `kernelY` is not properly aligned
- if `tilingMode` is not one of the values from the `VGTilingMode` enumeration

vgGaussianBlur

The **vgGaussianBlur** function computes the convolution of a normalized source image `src` with a separable kernel defined in each dimension by the Gaussian function *G*(*x, s*):

$$G(x,s)=\frac{1}{\sqrt{2\pi s^2}}e^{-\frac{x^2}{2s^2}}$$

where *s* is the *standard deviation*.

The two-dimensional kernel is defined by multiplying together two one-dimensional kernels, one for each axis:

$$k(x,y)=G(x,s_x)*G(y,s_y)=\frac{1}{2\pi s_x s_y}e^{-\left(\frac{x^2}{2s_x^{\,2}}+\frac{y^2}{2s_y^{\,2}}\right)}$$

where s_x and s_y are the (positive) standard deviations in the horizontal and vertical directions, given by the `stdDeviationX` and `stdDeviationY` parameters respectively. This kernel has special properties that allow for very efficient implementation; for example, the implementation may use multiple passes with simple kernels to obtain the same overall result with higher performance than direct convolution. If `stdDeviationX` and `stdDeviationY` are equal, the kernel is rotationally symmetric.

Source pixels outside the source image bounds are defined by `tilingMode`, which takes a value from the `VGTilingMode` enumeration (see Section 9.4.1)

The operation is applied to all channels (color and alpha) independently.

```
void vgGaussianBlur(VGImage dst, VGImage src,
                    VGfloat stdDeviationX,
                    VGfloat stdDeviationY,
                    VGTilingMode tilingMode)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if either `dst` or `src` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if either `dst` or `src` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `src` and `dst` overlap

– if `stdDeviationX` or `stdDeviationY` is less than or equal to 0 or greater than `VG_MAX_GAUSSIAN_STD_DEVIATION`

– if `tilingMode` is not one of the values from the `VGTilingMode` enumeration

11.5 Lookup Tables

vgLookup

The **vgLookup** function passes each image channel of the normalized source image `src` through a separate lookup table.

Each channel of the normalized source pixel is used as an index into the lookup table for that channel by multiplying the normalized value by 255 and rounding to obtain an 8-bit integral value. Each `LUT` parameter should contain 256 `VGubyte` entries. The outputs of the lookup tables are concatenated to form an `RGBA_8888` pixel value, which is interpreted as `lRGBA_8888`, `lRGBA_8888_PRE`, `sRGBA_8888`, or `sRGBA_8888_PRE`, depending on the values of `outputLinear` and `outputPremultiplied`.

The resulting pixels are converted into the destination format using the normal pixel format conversion rules.

```
void vgLookup(VGImage dst, VGImage src,
              const VGubyte * redLUT,
              const VGubyte * greenLUT,
              const VGubyte * blueLUT,
              const VGubyte * alphaLUT,
              VGboolean outputLinear,
              VGboolean outputPremultiplied)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if either `dst` or `src` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if either `dst` or `src` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `src` and `dst` overlap

– if any pointer parameter is `NULL`

vgLookupSingle

The **vgLookupSingle** function passes a single image channel of the normalized source image `src`, selected by the `sourceChannel` parameter, through a combined lookup table that produces whole pixel values. Each normalized source channel value is multiplied by 255 and rounded to obtain an 8 bit integral value.

The specified `sourceChannel` of the normalized source pixel is used as an index into the lookup table. If the source image is in a single-channel grayscale (`VG_lL_8`, `VG_sL_8`, or `VG_BW_1`) or alpha-only (`VG_A_8`) format, the `sourceChannel` parameter is ignored and the single channel is used. The `lookupTable` parameter should contain 256 4-byte aligned entries in an `RGBA_8888` pixel value, which is interpreted as `lRGBA_8888`, `lRGBA_8888_PRE`, `sRGBA_8888`, or `sRGBA_8888_PRE`, depending on the values of `outputLinear` and `outputPremultiplied`.

The resulting pixels are converted into the destination format using the normal pixel format conversion rules.

```
void vgLookupSingle(VGImage dst, VGImage src,
                    const VGuint * lookupTable,
                    VGImageChannel sourceChannel,
                    VGboolean outputLinear,
                    VGboolean outputPremultiplied)
```

ERRORS

`VG_BAD_HANDLE_ERROR`

– if either `dst` or `src` is not a valid image handle, or is not shared with the current context

`VG_IMAGE_IN_USE_ERROR`

– if either `dst` or `src` is currently a rendering target

`VG_ILLEGAL_ARGUMENT_ERROR`

– if `src` and `dst` overlap

– if `src` is in an RGB pixel format and `sourceChannel` is not one of `VG_RED`, `VG_GREEN`, `VG_BLUE` or `VG_ALPHA` from the `VGImageChannel` enumeration

– if `lookupTable` is `NULL`

– if `lookupTable` is not properly aligned

12 Blending

As drawing takes place, the painted pixels that result from the paint generation (stage 6) or image interpolation (stage 7) stages of the rendering pipeline are blended into the existing pixels of the drawing surface. Blending is performed using a subset of the standard Porter-Duff blending rules [PORT84] along with several additional rules.

The source pixels are converted into the destination color space prior to blending.

12.1 Blending Equations

A blending mode defines an alpha blending function $\alpha(\alpha_{src}, \alpha_{dst})$ and a color blending function $c(c_{src}, c_{dst}, \alpha_{src}, \alpha_{dst})$. Given a non-premultiplied source alpha and color tuple $(R_{src}, G_{src}, B_{src}, \alpha_{src})$ and a non-premultiplied destination alpha and color tuple $(R_{dst}, G_{dst}, B_{dst}, \alpha_{dst})$, blending replaces the destination with the blended tuple $(c(R_{src}, R_{dst}, \alpha_{src}, \alpha_{dst}), c(G_{src},G_{dst}, \alpha_{src}, \alpha_{dst}), c(B_{src}, B_{dst}, \alpha_{src}, \alpha_{dst}), \alpha(\alpha_{src}, \alpha_{dst}))$.

If either the source or destination is stored in a premultiplied format (*i.e.*, pixels are stored as tuples of the form $(\alpha{*}R, \alpha{*}G, \alpha{*}B, \alpha)$), the alpha value is conceptually divided out prior to applying the blending equations described above. If the destination is premultiplied, the destination alpha value is multiplied into each color channel prior to storage. If the destination format does not store alpha values, an alpha value of 1 is used in place of α_{dst}.

12.2 Porter-Duff Blending

Porter-Duff blending defines an alpha value $\alpha(\alpha_{src}, \alpha_{dst}) = \alpha_{src}{*}F_{src} + \alpha_{dst}{*}F_{dst}$ and color $c'(c'_{src}, c'_{dst}, \alpha_{src}, \alpha_{dst}) = c'_{src}{*}F_{src} + c'_{dst}{*}F_{dst}$, where F_{src} and F_{dst} are defined by the blend mode and the source and destination alpha values according to Table 13 below and $c' = \alpha{*}c$ is a premultiplied color value. For non-premultiplied colors, we define the equivalent formula $c(c_{src}, c_{dst}, \alpha_{src}, \alpha_{dst}) = (\alpha_{src}{*}c_{src}{*}F_{src} + \alpha_{dst}{*}c_{dst}{*}F_{dst})/\alpha(\alpha_{src}, \alpha_{dst})$ (taking the value to be 0 where division by 0 would occur).

Porter-Duff blending modes are derived from the assumption that each additional primitive being drawn is uncorrelated with previous ones. That is, if a previously drawn primitive p occupies a fraction f_p of a pixel, and a new primitive q occupies a fraction f_q, Porter-Duff blending assumes that a fraction $f_p{*}f_q$ of the pixel will be occupied by both primitives, a fraction $f_p - f_p{*}f_q = f_p\ (1 - f_q)$ will be occupied by p only, and a fraction $f_q - f_p{*}f_q = f_q\ (1 - f_p)$ will be occupied by q only. A total fraction of $f_p + f_q - f_p{*}f_q$ of the pixel is occupied by the union of the primitives.

Blend Mode	F_{src}	F_{dst}
Src	1	0
Src **over** Dst	1	$1 - \alpha_{src}$
Dst **over** Src	$1 - \alpha_{dst}$	1
Src **in** Dst	α_{dst}	0
Dst **in** Src	0	α_{src}

Table 13: Porter-Duff Blending Modes

12.3 Additional Blending Modes

A number of additional blending modes are available. These modes are a subset of the SVG image blending modes. Note that the SVG "Normal" blending mode is equivalent to the Porter-Duff "Src **over** Dst" mode described above. The additional blend modes have the following effects:

- `VG_BLEND_MULTIPLY` – Multiply the source and destination colors together, producing the effect of placing a transparent filter over a background. A black source pixel forces the destination to black, while a white source pixel leaves the destination unchanged. If all alpha values are 1, this reduces to multiplying the source and destination color values.
- `VG_BLEND_SCREEN` – The opposite of multiplication, producing the effect of projecting a slide over a background. A black source pixel leaves the destination unchanged, while a white source pixel forces the destination to white. If all alpha values are 1, this reduces to adding the source and destination color values, and subtracting their product.
- `VG_BLEND_DARKEN` – Compute (Src **over** Dst) and (Dst **over** Src) and take the smaller (darker) value for each channel. If all alpha values are 1, this reduces to choosing the smaller value for each color channel.
- `VG_BLEND_LIGHTEN` – Compute (Src **over** Dst) and (Dst **over** Src) and take the larger (lighter) value for each channel. If all alpha values are 1, this reduces to choosing the larger value for each color channel.

The new destination alpha value for the blending modes defined in this section is always equal to $\alpha(\alpha_{src}, \alpha_{dst}) = \alpha_{src} + \alpha_{dst}*(1 - \alpha_{src})$, as for Porter-Duff "Src **over** Dst" blending. The formulas for each additional blending mode are shown in Table 14. The right-hand column contains the pre-multiplied output values, that is, the products of the new color value $c(c_{src}, c_{dst}, \alpha_{src}, \alpha_{dst})$ and alpha value $\alpha(\alpha_{src}, \alpha_{dst})$. The source and destination color values c_{src} and c_{dst} are given in non-premultiplied form.

Blend Type	$c'(c_{src}, c_{dst}, \alpha_{src}, \alpha_{dst})$
VG_BLEND_MULTIPLY	$\alpha_{src}*c_{src}*(1-\alpha_{dst}) + \alpha_{dst}*c_{dst}*(1-\alpha_{src}) + \alpha_{src}*c_{src}*\alpha_{dst}*c_{dst}$
VG_BLEND_SCREEN	$\alpha_{src}*c_{src} + \alpha_{dst}*c_{dst} - \alpha_{src}*c_{src}*\alpha_{dst}*c_{dst}$
VG_BLEND_DARKEN	$min(\alpha_{src}*c_{src} + \alpha_{dst}*c_{dst}*(1-\alpha_{src}), \alpha_{dst}*c_{dst} + \alpha_{src}*c_{src}*(1-\alpha_{dst}))$
VG_BLEND_LIGHTEN	$max(\alpha_{src}*c_{src} + \alpha_{dst}*c_{dst}*(1-\alpha_{src}), \alpha_{dst}*c_{dst} + \alpha_{src}*c_{src}*(1-\alpha_{dst}))$

Table 14: Additional Blending Equations

12.4 Additive Blending

The Porter-Duff assumption of uncorrelated alpha described above does not hold for primitives that are known to be disjoint (for example, a set of triangles with shared vertices and edges forming a mesh, or a series of text glyphs that have been spaced according to known metrics). In these cases, we expect no portion of the pixel to be occupied by both primitives and a total fraction of $f_p + f_q$ to be occupied by the union of the primitives. The *additive* blending rule may be used in this case. It sets the final alpha value of the blended pixel to the clamped sum $\alpha(\alpha_{src}, \alpha_{dst}) = min(\alpha_{src}+\alpha_{dst}, 1)$ and the color to $c(c_{src}, c_{dst}) = min((\alpha_{src}*c_{src} + \alpha_{dst}*c_{dst})/min(\alpha_{src} + \alpha_{dst}, 1), 1)$. If all alpha values are 1, this reduces to adding the values of each source color channel and clamping the result.

12.5 Setting the Blend Mode

VGBlendMode

The `VGBlendMode` enumeration defines the possible blending modes:

```
typedef enum {
  VG_BLEND_SRC        = 0x2000,
  VG_BLEND_SRC_OVER   = 0x2001,
  VG_BLEND_DST_OVER   = 0x2002,
  VG_BLEND_SRC_IN     = 0x2003,
  VG_BLEND_DST_IN     = 0x2004,
  VG_BLEND_MULTIPLY   = 0x2005,
  VG_BLEND_SCREEN     = 0x2006,
  VG_BLEND_DARKEN     = 0x2007,
  VG_BLEND_LIGHTEN    = 0x2008,
  VG_BLEND_ADDITIVE   = 0x2009
} VGBlendMode;
```

Use **vgSeti** with a parameter type of `VG_BLEND_MODE` to set the blend mode:

```
VGBlendMode mode;
vgSeti(VG_BLEND_MODE, mode);
```

13 Querying Hardware Capabilities

OpenVG implementations may vary considerably in their performance characteristics. A simple hardware query mechanism is provided to allow applications to make informed choices regarding data representations, in order to maximize their chances of obtaining hardware-accelerated performance. Currently, OpenVG provides hardware queries for image formats and path datatypes.

VGHardwareQueryType

The `VGHardwareQueryType` enumeration defines the set of possible hardware queries. Currently these are restricted to queries regarding image formats and path datatypes.

```
typedef enum {
  VG_IMAGE_FORMAT_QUERY  = 0x2100,
  VG_PATH_DATATYPE_QUERY = 0x2101
} VGHardwareQueryType;
```

VGHardwareQueryResult

The `VGHardwareQueryResult` enumeration defines the return values from a hardware query, indicating whether or not the item being queried is hardware accelerated.

```
typedef enum {
  VG_HARDWARE_ACCELERATED   = 0x2200,
  VG_HARDWARE_UNACCELERATED = 0x2201
} VGHardwareQueryResult;
```

vgHardwareQuery

The **vgHardwareQuery** function returns a value indicating whether a given `setting` of a property of a type given by `key` is generally accelerated in hardware on the currently running OpenVG implementation.

The return value will be one of the values `VG_HARDWARE_ACCELERATED` or `VG_HARDWARE_UNACCELERATED`, taken from the `VGHardwareQueryResult` enumeration. The legal values for the setting parameter depend on the value of the `key` parameter, as indicated by Table 15.

Value of `key`	*Allowable values for* `setting`
`VG_IMAGE_FORMAT_QUERY`	`VGImageFormat` (p. 109)
`VG_PATH_DATATYPE_QUERY`	`VGPathDatatype` (p. 54)

Table 15: Query Key Enumeration Types

```
VGHardwareQueryResult vgHardwareQuery(VGHardwareQueryType key,
                                      VGint setting)
```

ERRORS

`VG_ILLEGAL_ARGUMENT_ERROR`

- if `key` is not one of the values from the `VGHardwareQueryType` enumeration
- if `setting` is not one of the values from the enumeration associated with `key`

14 Extending the API

OpenVG is designed to be extended using an extension mechanism modeled after that of OpenGL and OpenGL ES. An extension may define new state elements, new datatypes, new values for existing parameter types, and new functions. Use of these features may alter the operation of the rendering pipeline. However, an extension must have no effect on programs that do not enable any of its features.

14.1 Extension Naming Conventions

An OpenVG extension is named by a string of the form `OVG_`*type_name*, where *type* is either the string `EXT` or a vendor-specific string and *name* is a name assigned by the extension author. A letter `X` added to the end of *type* indicates that the extension is experimental.

Values (*e.g.*, enumerated values or preprocessor `#define`s) defined by an extension carry the suffix *_type*. Functions and datatypes carry the suffix *type* without a separating underscore.

The `openvg.h` header file will define a preprocessor macro with the name `OVG_`*type_name* and a value of 1 for each supported extension.

14.2 The Extension Registry

Khronos, or its designee, will maintain a publicly-accessible registry of extensions. This registry will contain, for each extension, at least the following information:

- The name of the extension in the form `OVG_`*type_name*
- An email address of a contact person
- A list of dependencies on other extensions
- A statement on the IP status of the extension
- An overview of the scope and semantics of the extension
- New functions defined by the extension
- New datatypes defined by the extension
- New values to be added to existing enumerated datatypes
- Additions and changes to the OpenVG specification
- New errors generated by functions affected by the extension
- New state defined by the extension
- Authorship information and revision history

14.3 Using Extensions

Extensions may be detected statically, by means of preprocessor symbols, or dynamically, by means of the **vgGetString** function. Extension functions may be included in application code statically by placing appropriate "#ifdef" directives around functions that require the presence of a particular extension, and may also be accessed dynamically through function pointers returned by **eglGetProcAddress** or by other platform-specific means.

14.3.1 Accessing Extensions Statically

The extensions defined by a given platform are defined in the `openvg.h` header file, or in header files automatically included by `openvg.h`. In order to write applications that run on platforms with and without a given extension, conditional compilation based on the presence of the extension's preprocessor macro may be used:

```
#ifdef OVG_EXT_my_extension
  vgMyExtensionFuncEXT(...);
#endif
```

14.3.2 Accessing Extensions Dynamically

OpenVG contains a mechanism for applications to access information about the runtime platform, and to access extensions that may not have been present when the application was compiled.

VGStringID

```
typedef enum {
  VG_VENDOR     = 0x2300,
  VG_RENDERER   = 0x2301,
  VG_VERSION    = 0x2302,
  VG_EXTENSIONS = 0x2303
} VGStringID;
```

vgGetString

The **vgGetString** function returns information about the OpenVG implementation, including extension information. The values returned may vary according to the display (*e.g.*, the `EGLDisplay` when using EGL) associated with the current context. If no context is current, **vgGetString** returns `NULL`.

The combination of `VG_VENDOR` and `VG_RENDERER` may be used together as a platform identifier by applications that wish to recognize a particular platform and adjust their algorithms based on prior knowledge of platform bugs and performance characteristics .

If `name` is `VG_VENDOR`, the name of company responsible for this OpenVG implementation is returned. This name does not change from release to release.

If `name` is `VG_RENDERER`, the name of the renderer is returned. This name is typically specific to a particular configuration of a hardware platform, and does not change from release to release.

If `name` is `VG_VERSION`, the version number of the specification implemented by the renderer is returned as a string in the form *major_number.minor_number*. For this specification, "1.0" is returned.

If `name` is `VG_EXTENSIONS`, a space-separated list of supported extensions to OpenVG is returned.

For other values of `name`, `NULL` is returned.

```
const VGubyte * vgGetString(VGStringID name)
```

eglGetProcAddress

Functions defined by an extension may be accessed by means of a function pointer obtained from the EGL function `eglGetProcAddress`. If EGL is not present, the platform may define an alternate method of obtaining extension function pointers.

14.4 Creating Extensions

Any vendor may define a vendor-specific extension. Each vendor should apply to Khronos to obtain a vendor string and any numerical token values required by the extension.

An OpenVG extension may be deemed a shared extension if two or more vendors agree in good faith to ship an extension, or the Khronos OpenVG working group determines that it is in the best interest of its members that the extension be shared. A shared extension may be adopted (with appropriate naming changes) into a subsequent release of the OpenVG specification.

15 API Conformance

All OpenVG implementations are required to pass a conformance test suite. The exact details of the conformance testing process are available in a separate document. This chapter outlines the OpenVG conformance test philosophy and provides information that may be useful in order to ensure conformant implementations.

15.1 Conformance Test Principles

The OpenVG specification attempts to strike a balance between the needs of implementers and application developers. While application developers desire a stable platform that delivers predictable results, they also wish to avoid reduced performance due to an excessively strict API definition. By allowing some flexibility in how the API is implemented, implementations may be optimized for a wide variety of platforms with varying price, performance, and power characteristics. The purpose of conformance testing is to ensure that implementations with different internal approaches produce similar results.

15.1.1 Window System Independence

Because OpenVG does not mandate a specific window system or display management API, the conformance test suite will isolate all display dependencies in a module that may be customized for each platform. An EGL-based implementation of this module will be provided, but implementers are free to replace this implementation with one that is specific to their platform.

15.1.2 Antialiasing Algorithm Independence

It is anticipated that a wide variety of antialiasing approaches will be used in the marketplace. Low-cost antialiasing remains a research topic, and new algorithms continue to emerge. The conformance suite must allow for this variation, while not allowing differences in antialiasing to cover up inadequacies in other portions of the implementation such as matrix transformation or curve subdivision.

15.1.3 On-Device and Off-Device Testing

Certain conformance tests require only a small memory footprint, and may be run directly on the target device. Other tests operate by generating an image, which must be copied off-device. A desktop tool is used to compare the generated images against a set of reference images.

15.2 Types of Conformance Tests

Conformance tests fall into several classes, outlined below.

15.2.1 Pipeline Tests

A set of tests will be provided that attempt to isolate each pipeline stage by means of suitable parameter settings. These tests will provide assurance that each stage is functioning correctly.

15.2.2 Self-Consistency Tests

Certain portions of the API are required to produce exact results. For example, setting and retrieving API state, image, paint, and path parameters, setting and retrieving matrix values; error generation; and pixel copies are defined to have exact results. The conformance suite will provide strict checking for these behaviors.

15.2.3 Matrix Tests

The conformance suite will exercise various matrix operations and compare the results against double-precision values. The comparison threshold will be set to exclude implementations with insufficient internal precision.

15.2.4 Interior/Exterior Tests

Although antialiasing may have varying effects on shape boundaries, the portions of the interior and exterior of shapes that are more than 1 ½ pixels from a geometric boundary should not be affected by that boundary. If a shape is drawn using color paint, a set of known interior and exterior pixels may be tested for equality with the paint color.

15.2.5 Positional Invariance

Drawing should not depend on absolute screen coordinates, except for minor differences due to spatially-variant sampling and dither patterns when copying to the screen. The conformance suite will include tests that verify the positional independence of drawing.

15.2.6 Image Comparison Tests

To allow for controlled variation, the conformance suite will provide a set of rendering code fragments, along with reference images that have been generated using a high-quality implementation. Implementation-generated images will be compared to these reference images using a fuzzy comparison system. This approach is intended to allow for small differences in the accuracy of geometry and color processing and antialiasing, while rejecting larger differences that are considered visually unacceptable. The comparison threshold will be determined by generating images with a variety of acceptable and unacceptable differences and comparing them against the reference image.

16 The VGU Utility Library

For convenience, OpenVG provides an optional utility library known as VGU. Applications may choose whether to link to VGU at compile time; the library is not guaranteed to be present on the run-time platform. VGU is designed so it may be implemented in a portable manner using only the public functionality provided by the OpenVG library. VGU functions may alter the error state of the OpenVG context in which they run (*i.e.*, the value returned by **vgGetError**), but do not otherwise alter the OpenVG state when they complete without generating a `VGU_OUT_OF_MEMORY_ERROR`. VGU functions are defined in a `vgu.h` header file.

VGU_VERSION_1_0

Each version of the VGU library will define constants indicating the set of supported library versions. For the current version, the constant `VGU_VERSION_1_0` is defined. Future versions will continue to define the constants for all previous versions with which they are backward compatible.

```
#define VGU_VERSION_1_0 1
```

VGUErrorCode

The `VGUErrorCode` enumeration contains constants specifying possible errors generated by VGU functions. Any VGU function may return `VGU_OUT_OF_MEMORY_ERROR,` in which case the function may have caused changes to the state of OpenVG or to drawing surface pixels prior to failure.

```
typedef enum {
  VGU_NO_ERROR                        = 0,
  VGU_BAD_HANDLE_ERROR                = 0xF000,
  VGU_ILLEGAL_ARGUMENT_ERROR          = 0xF001,
  VGU_OUT_OF_MEMORY_ERROR             = 0xF002,
  VGU_PATH_CAPABILITY_ERROR           = 0xF003,
  VGU_BAD_WARP_ERROR                  = 0xF004
} VGUErrorCode;
```

16.1 Higher-level Geometric Primitives

The `VGU` library contains functions that allow applications to specify a number of higher-level geometric primitives to be appended to a path. Each primitive is immediately reduced to a series of line segments, Bézier curves, and arcs. Coordinates may overflow silently if they fall outside the range defined by the path datatype, scale, and bias.

16.1.1 Lines

vguLine

vguLine appends a line segment to a path. This is equivalent to the following pseudo-code:

```
LINE(x0, y0, x1, y1):

MOVE_TO_ABS x0, y0
LINE_TO_ABS x1, y1
```

```
VGUErrorCode vguLine(VGPath path,
                     VGfloat x0, VGfloat y0,
                     VGfloat x1, VGfloat y1)
```

ERRORS

VGU_BAD_HANDLE_ERROR

– if path is not a valid path handle, or is not shared with the current context

VGU_PATH_CAPABILITY_ERROR

– if VG_PATH_CAPABILITY_APPEND_TO is not enabled for path

16.1.2 Polylines and Polygons

vguPolygon

vguPolygon appends a polyline or polygon to a path. This is equivalent to the following pseudo-code:

```
POLYGON(points, count):

MOVE_TO_ABS points[0], points[1]
for (i = 1; i < count; i++) {
     LINE_TO_ABS points[2*i], points[2*i + 1]
}
if (closed) CLOSE_PATH
```

There are 2*count coordinates in points.

```
VGUErrorCode vguPolygon(VGPath path,
                        const VGfloat * points, VGint count,
                        VGboolean closed)
```

ERRORS

`VGU_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

`VGU_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_APPEND_TO` is not enabled for `path`

`VGU_ILLEGAL_ARGUMENT_ERROR`

– if `points` is `NULL`

– if `points` is not properly aligned

– if `count` is less than or equal to 0

16.1.3 Rectangles

vguRect

The **vguRect** function appends an axis-aligned rectangle with its lower-left corner at (`x`, `y`) and a given `width` and `height` to a path. This is equivalent to the following pseudo-code:

```
RECT(x, y, width, height):

MOVE_TO_ABS    x, y
HLINE_TO_REL   width
VLINE_TO_REL   height
HLINE_TO_REL  -width
CLOSE_PATH
```

```
VGUErrorCode vguRect(VGPath path,
                     VGfloat x, VGfloat y,
                     VGfloat width, VGfloat height)
```

ERRORS

`VGU_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

`VGU_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_APPEND_TO` is not enabled for `path`

`VGU_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` are less than or equal to 0

16.1.4 Round-Cornered Rectangles

vguRoundRect

The **vguRoundRect** function appends an axis-aligned round-cornered rectangle with the lower-left corner of its rectangular bounding box at (x, y) and a given width, height, arcWidth, and arcHeight to a path. This is equivalent to the following pseudo-code:

```
ROUNDRECT(x, y, w, h, arcWidth, arcHeight):

MOVE_TO_ABS     (x + arcWidth/2), y
HLINE_TO_REL    width – arcWidth
SCCWARC_TO_REL  arcWidth/2, arcHeight/2, 0,  arcWidth/2,  arcHeight/2
VLINE_TO_REL    height – arcHeight
SCCWARC_TO_REL  arcWidth/2, arcHeight/2, 0, -arcWidth/2,  arcHeight/2
HLINE_TO_REL    -(width – arcWidth)
SCCWARC_TO_REL  arcWidth/2, arcHeight/2, 0, -arcWidth/2, -arcHeight/2
VLINE_TO_REL    -(height – arcHeight)
SCCWARC_TO_REL  arcWidth/2, arcHeight/2, 0,  arcWidth/2, -arcHeight/2
CLOSE_PATH
```

If arcWidth is less than 0, it is clamped to 0. If arcWidth is greater than width, its value is clamped to that of width. Similarly, arcHeight is clamped to a value between 0 and height. The arcs are included even when arcWidth and/or arcHeight is 0.

```
VGUErrorCode vguRoundRect(VGPath path,
                          VGfloat x, VGfloat y,
                          VGfloat width, VGfloat height,
                          VGfloat arcWidth, VGfloat arcHeight)
```

ERRORS

VGU_BAD_HANDLE_ERROR

– if path is not a valid path handle, or is not shared with the current context

VGU_PATH_CAPABILITY_ERROR

– if VG_PATH_CAPABILITY_APPEND_TO is not enabled for path

VGU_ILLEGAL_ARGUMENT_ERROR

– if width or height is less than or equal to 0

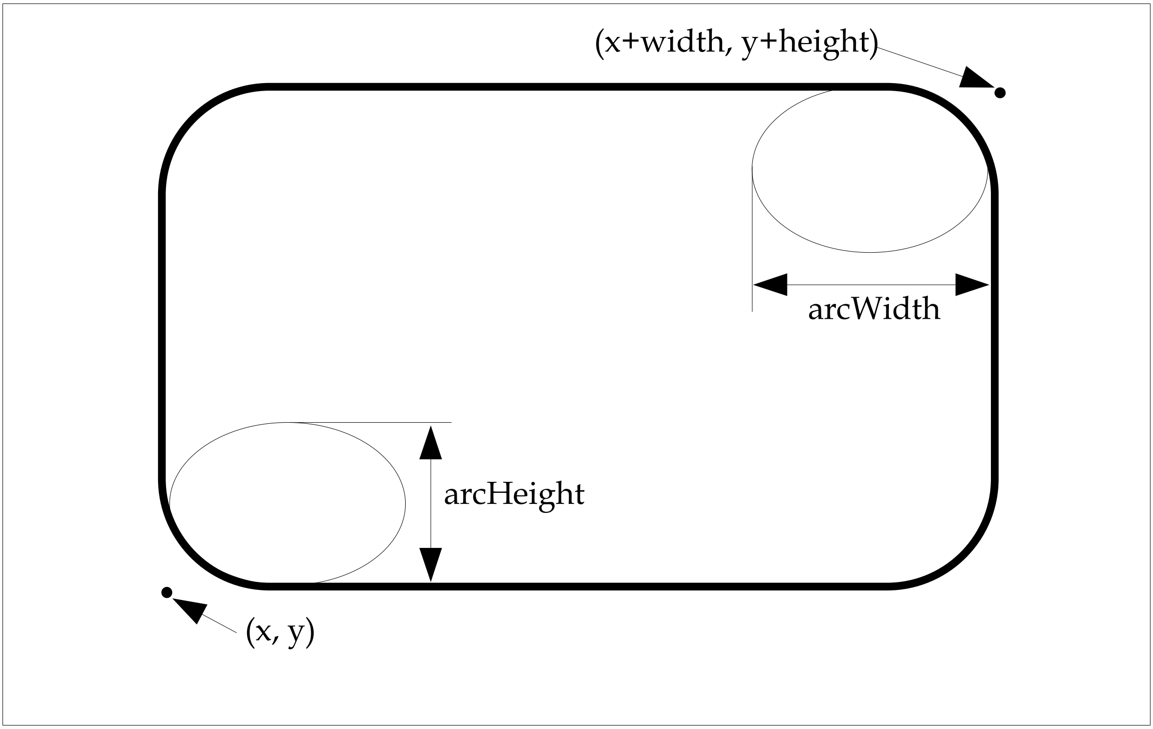


Figure 24: Round Rectangle Parameters

16.1.5 Ellipses

vguEllipse

vguEllipse appends an axis-aligned ellipse to a path. The center of the ellipse is given by (`cx`, `cy`) and the dimensions of the axis-aligned rectangle enclosing the ellipse are given by `width` and `height`. The ellipse begins at (*cx* + *width/2, cy*) and is stroked as two equal counter-clockwise arcs. This is equivalent to the following pseudo-code:

```
ELLIPSE(cx, cy, width, height):

MOVE_TO_ABS     cx + width/2, cy
SCCWARC_TO_REL  width/2, height/2, 0, -width, 0
SCCWARC_TO_REL  width/2, height/2, 0,  width, 0
CLOSE_PATH
```

```
VGUErrorCode vguEllipse(VGPath path,
                        VGfloat cx, VGfloat cy,
                        VGfloat width, VGfloat height)
```

ERRORS

VGU_BAD_HANDLE_ERROR

– if path is not a valid path handle, or is not shared with the current context

VGU_PATH_CAPABILITY_ERROR

– if VG_PATH_CAPABILITY_APPEND_TO is not enabled for path

VGU_ILLEGAL_ARGUMENT_ERROR

– if width or height is less than or equal to 0

16.1.6 Arcs

VGUArcType

The VGUArcType enumeration defines three values to control the style of arcs drawn by the **vguArc** function:

VGU_ARC_OPEN – arc segment only
VGU_ARC_CHORD – arc, plus line between arc endpoints
VGU_ARC_PIE – arc, plus lines from each endpoint to the ellipse center.

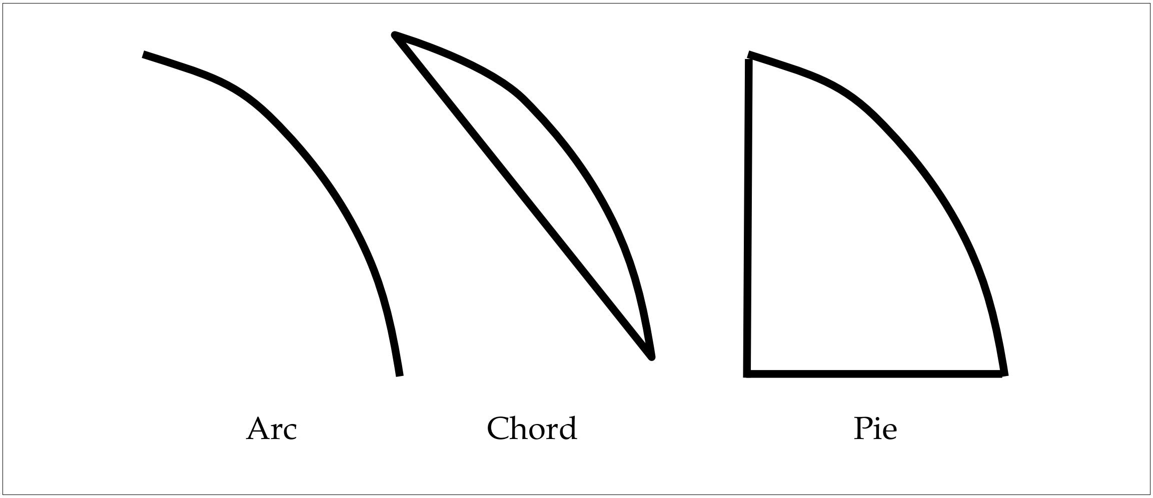


Figure 25: VGUArcType Values

vguArc

vguArc appends an elliptical arc to a path, possibly along with one or two line segments, according to the arcType parameter. The startAngle and angleExtent parameters are given in degrees, proceeding counter-clockwise from the positive X axis. The arc is defined on the unit circle, then scaled by the width and height of the ellipse; thus, the starting point of the arc has coordinates (*x* + *cos*(*startAngle*)**w*/2, *y* + *sin*(*startAngle*)**h*/2) and the ending point has coordinates (*x* + *cos*(*startAngle* + *angleExtent*)**w*/2, *y* + *sin*(*startAngle* + *angleExtent*)**h*/2).

If `angleExtent` is negative, the arc will proceed clockwise; if it is larger than 360 or smaller than -360, the arc will wrap around itself. The following pseudo-code illustrates the arc path generation:

```
ARC(x, y, w, h, startAngle, angleExtent, arcType):

last = startAngle + angleExtent
MOVE_TO_ABS  x+cos(startAngle)*w/2, y+sin(startAngle)*h/2
if (angleExtent > 0) {
  angle = startAngle + 180
  while (angle < last) {
    SCCWARC_TO_ABS w/2, h/2, 0, x+cos(angle)*w/2, y+sin(angle)*h/2
    angle += 180
  }
  SCCWARC_TO_ABS w/2, h/2, 0, x+cos(last)*w/2, y+sin(last)*h/2
} else {
  angle = startAngle – 180
  while (angle > last) {
    SCWARC_TO_ABS w/2, h/2, 0, x+cos(angle)*w/2, y+sin(angle)*h/2
    angle -= 180
  }
  SCWARC_TO_ABS w/2, h/2, 0, x+cos(last)*w/2, y+sin(last)*h/2
}

if arcType == VGU_ARC_PIE
  LINE_TO_ABS x, y
if arcType == VGU_ARC_PIE || arcType == VGU_ARC_CHORD
  CLOSE_PATH
```

```
VGUErrorCode vguArc(VGPath path,
                    VGfloat x, VGfloat y,
                    VGfloat width, VGfloat height,
                    VGfloat startAngle, VGfloat angleExtent,
                    VGUArcType arcType)
```

ERRORS

`VGU_BAD_HANDLE_ERROR`

– if `path` is not a valid path handle, or is not shared with the current context

`VGU_PATH_CAPABILITY_ERROR`

– if `VG_PATH_CAPABILITY_APPEND_TO` is not enabled for `path`

`VGU_ILLEGAL_ARGUMENT_ERROR`

– if `width` or `height` is less than or equal to 0

– if `arcType` is not one of the values from the `VGUArcType` enumeration

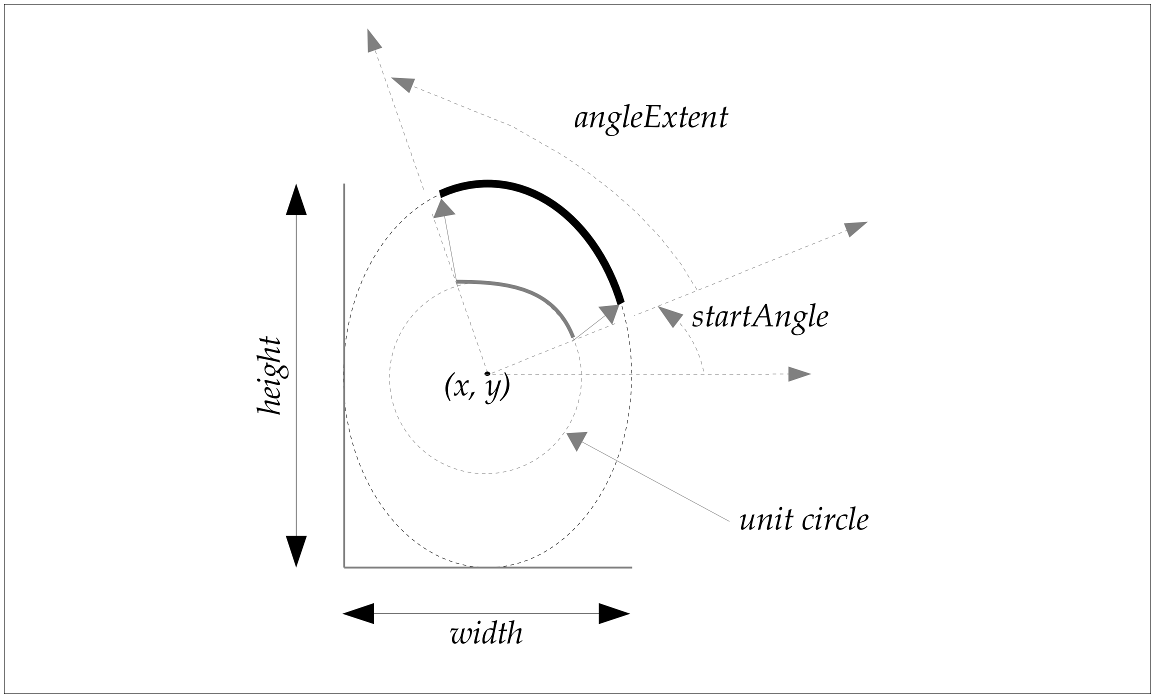


Figure 26: **vguArc** *Parameters*

16.2 Image Warping

VGU provides three utility functions to compute 3x3 projective transform matrices. The first two compute the transformation from an arbitrary quadrilateral onto the unit square, and vice versa. The third computes the transformation from an arbitrary quadrilateral to an arbitrary quadrilateral. The output transformation is stored into `matrix` as 9 elements in the order *{ sx, shy, w_0, shx, sy, w_1, tx, ty, w_2 }* (using the nomenclature of Section 6.3).

In all cases, if there is no projective mapping that satisfies the given constraints, or the mapping would be degenerate (*i.e.*, non-invertible), `VGU_BAD_WARP_ERROR` is returned and `matrix` is unchanged.

Formulas for computing projective warps may be found in [HECK89] and [WOLB90].

vguComputeWarpQuadToSquare

The **vguComputeWarpQuadToSquare** function sets the entries of matrix to a projective transformation that maps the point (*sx0, sy0*) to (*0, 0*); (*sx1, sy1*) to (*1, 0*); (*sx2, sy2*) to (*0, 1*); and (*sx3, sy3*) to (*1, 1*). If no non-degenerate matrix satisfies the constraints, `VGU_BAD_WARP_ERROR` is returned and `matrix` is unchanged.

```
VGUErrorCode vguComputeWarpQuadToSquare(VGfloat sx0, VGfloat sy0,
                                        VGfloat sx1, VGfloat sy1,
                                        VGfloat sx2, VGfloat sy2,
                                        VGfloat sx3, VGfloat sy3,
                                        VGfloat * matrix)
```

ERRORS

`VGU_ILLEGAL_ARGUMENT_ERROR`

- if `matrix` is `NULL`
- if `matrix` is not properly aligned

`VGU_BAD_WARP_ERROR`

- if no non-degenerate transformation satisfies the constraints

vguComputeWarpSquareToQuad

The **vguComputeWarpSquareToQuad** function sets the entries of matrix to a projective transformation that maps the point (*0, 0*) to (*dx0, dy0*); (*1, 0*) to (*dx1, dy1*); (*0, 1*) to (*dx2, dy2*); and (*1, 1*) to (*dx3, dy3*). If no non-degenerate matrix satisfies the constraints, `VGU_BAD_WARP_ERROR` is returned and `matrix` is unchanged.

```
VGUErrorCode vguComputeWarpSquareToQuad(VGfloat dx0, VGfloat dy0,
                                        VGfloat dx1, VGfloat dy1,
                                        VGfloat dx2, VGfloat dy2,
                                        VGfloat dx3, VGfloat dy3,
                                        VGfloat * matrix)
```

ERRORS

VGU_ILLEGAL_ARGUMENT_ERROR

– if matrix is NULL

– if matrix is not properly aligned

VGU_BAD_WARP_ERROR

– if no non-degenerate transformation satisfies the constraints

vguComputeWarpQuadToQuad

The **vguComputeWarpQuadToQuad** function sets the entries of matrix to a projective transformation that maps the point (*sx0, sy0*) to (*dx0, dy0*); (*sx1, sy1*) to (*dx1, dy1*); (*sx2, sy2*) to (*dx2, dy2*); and (*sx3, sy3*) to (*dx3, dy3*). If no non-degenerate matrix satisfies the constraints, VGU_BAD_WARP_ERROR is returned and matrix is unchanged.

```
VGUErrorCode vguComputeWarpQuadToQuad(VGfloat dx0, VGfloat dy0,
                                      VGfloat dx1, VGfloat dy1,
                                      VGfloat dx2, VGfloat dy2,
                                      VGfloat dx3, VGfloat dy3,
                                      VGfloat sx0, VGfloat sy0,
                                      VGfloat sx1, VGfloat sy1,
                                      VGfloat sx2, VGfloat sy2,
                                      VGfloat sx3, VGfloat sy3,
                                      VGfloat * matrix)
```

ERRORS

VGU_ILLEGAL_ARGUMENT_ERROR

– if matrix is NULL

– if matrix is not properly aligned

VGU_BAD_WARP_ERROR

– if no non-degenerate transformation satisfies the constraints

17 Appendix A: Mathematics of Ellipses

The following sections are informative only. It contains mathematics pertaining to the representation of ellipses that may be of use to implementers. Some of the material is adapted from [SVGF04].

17.1 The Center Parameterization

A common parameterization of an ellipse is in terms of the ellipse center point (*cx, cy*), horizontal and vertical radii *rh* and *rv*, rotation angle , and starting and ending angles θ_1 and θ_2 between 0 and 360 degrees. The parameters are listed in Table 16.

The elliptical arc may be evaluated in terms of an angular parameter θ that ranges from θ_1 to θ_2:

$$f(cx, cy, rh, rv, \phi, \theta) = \begin{bmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{bmatrix} \cdot \begin{bmatrix} rh\cos\theta \\ rv\sin\theta \end{bmatrix} + \begin{bmatrix} cx \\ cy \end{bmatrix}$$

An ellipse in the center parameterization may be viewed as a unit circle, parameterized as (*x, y*) = (*cos*(θ)*, sin*(θ)) that has been placed through an affine transformation consisting of a rotation and a non-uniform scale:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} rh\cos(\phi) & -rv\sin(\phi) & cx \\ rh\sin(\phi) & rv\cos(\phi) & cy \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ 1 \end{bmatrix}$$

(*cx, cy*)	The center point of the ellipse
rh, rv	The radii of the unrotated ellipse
	The counter-clockwise angle of the ellipse relative to the x axis, measured prior to scaling by (*rh, rv*)
θ_1	Angle of initial point (as measured on the unscaled circle)
θ_2	Angle of final point (as measured on the unscaled circle)

Table 16: Center Ellipse Parameters

17.2 The Endpoint Parameterization

OpenVG paths use the endpoint parameterization of elliptical arcs as defined in SVG. An elliptical arc segment is defined in terms of its endpoints (*x0, y0*), (*x1, y1*), radii *rh* and *rv*, rotation angle , large arc flag *fA*, and sweep flag *fS*. These parameters are listed in Table 17.

(*x0, y0*)	The initial endpoint of the arc
(*x1, y1*)	The final endpoint of the arc
rh, *rv*	The radii of the unrotated ellipse
rot	The counter-clockwise angle of the ellipse relative to the x axis, measured prior to scaling by (*rh, rv*)
fA	Large arc flag: 1 if more than 180 degrees of the arc is to be traversed (as measured on the unscaled circle), 0 otherwise
fS	Sweep flag: 1 if the arc is to be traversed in the counter-clockwise direction, 0 otherwise

Table 17: Endpoint Ellipse Parameters

17.3 Converting from Center to Endpoint Parameterization

Conversion from a center parameterization to an endpoint parameterization simply requires evaluation the initial and final endpoints of the arc, and determining the values of the large arc and sweep flags:

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = f(cx, cy, rh, rv, \phi, \theta_1)$$

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = f(cx, cy, rh, rv, \phi, \theta_2)$$

$$f_A = \begin{cases} 1 & if\, |\theta_2 - \theta_1| > 180\, degrees \\ 0 & otherwise \end{cases}$$

$$f_S = \begin{cases} 1 & if\, \theta_2 - \theta_1 > 0 \\ 0 & otherwise \end{cases}$$

17.4 Converting from Endpoint to Center Parameterization

Given an endpoint representation of an ellipse as the set of parameters $(x0, y0)$, $(x1, y1)$, rh, rv, , fS, and fA, we wish to determine the center point (cx, cy) and the initial and final angles $\theta 1$ and $\theta 2$.

An ellipse with center point (cx, cy), radii rh and rv, and rotation angle rot satisfies the implicit equation $(x')^2 + (y')^2 = 1$, where $x' = ((x - cx)*cos(rot) + (y - cy)*sin(rot))/rh$ and $y' = (-(x - cx)*sin(rot) + (y - cy)*cos(rot))/rv$. The transformation from (x, y) to (x', y') simply maps the desired ellipse into a unit circle centered at the origin.

To determine the center points of the pair of ellipses with common radii and rotation angle that pass through the two given points (x_0, y_0) and (x_1, y_1), the plane is first transformed into a suitably scaled and rotated coordinate system such that the equation of each ellipse becomes $(x' - cx')^2 + (y' - cy')^2 = 1$. Then the problem is reduced to finding the centers (cx_0', cy_0') and (cx_1', cy_1') of the two unit circles whose circumferences pass through two given points. Finally, the center points are placed through an inverse transformation to obtain solutions in the original coordinate system.

The center points of the two unit circles that pass through points (x_0, y_0) and (x_1, y_1) are given by $(x_m \pm \Delta y*d, y_m \mp \Delta x*d)$, where $x_m = (x_0 + x_1)/2$, $y_m = (y_0 + y_1)/2$, $\Delta x = (x_0 - x_1)$, $\Delta y = (y_0 - y_1)$, and $d = \sqrt{}(1/(\Delta x^2 + \Delta y^2) - 1/4)$. If d is infinite or imaginary, no solution exists due to the input points being coincident or too far apart, respectively.

The angles $\theta 1$ and $\theta 2$ may be found by finding the slope of the endpoints on the circle and computing arctangents.

The following code illustrates the process of computing the ellipse centers. The findUnitCircles function is called by findEllipses following inverse transformation of the original ellipse parameters.

```
#include <math.h>

#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

/*  Given: Points (x0, y0) and (x1, y1)
 * Return: TRUE if a solution exists, FALSE otherwise
 *         Circle centers are written to (cx0, cy0) and (cx1, cy1)
 */
static VGboolean
findUnitCircles(double x0, double y0, double x1, double y1,
                double *cx0, double *cy0,
                double *cx1, double *cy1)
{
  /* Compute differences and averages */
  double dx = x0 – x1;
  double dy = y0 – y1;
  double xm = (x0 + x1)/2;
  double ym = (y0 + y1)/2;
  double dsq, disc, s, sdx, sdy;

  /* Solve for intersecting unit circles */
  dsq = dx*dx + dy*dy;
  if (dsq == 0.0) return VG_FALSE; /* Points are coincident */
  disc = 1.0/dsq – 1.0/4.0;
  if (disc < 0.0) return VG_FALSE; /* Points are too far apart */
  s = sqrt(disc);
  sdx = s*dx;
  sdy = s*dy;

  *cx0 = xm + sdy;
  *cy0 = ym – sdx;
  *cx1 = xm – sdy;
  *cy1 = ym + sdx;
  return VG_TRUE;
}
```

```
/*  Given: Ellipse parameters rh, rv, rot (in degrees),
 *         endpoints (x0, y0) and (x1, y1)
 * Return: TRUE if a solution exists, FALSE otherwise
 *         Ellipse centers are written to (cx0, cy0) and (cx1, cy1)
 */

VGboolean
findEllipses(double rh, double rv, double rot,
             double x0, double y0, double x1, double y1,
             double *cx0, double *cy0, double *cx1, double *cy1)
{
  double COS, SIN, x0p, y0p, x1p, y1p, pcx0, pcy0, pcx1, pcy1;

  /* Convert rotation angle from degrees to radians */
  rot *= M_PI/180.0;

  /* Pre-compute rotation matrix entries */
  COS = cos(rot); SIN = sin(rot);

  /* Transform (x0, y0) and (x1, y1) into unit space */
  /* using (inverse) rotate, followed by (inverse) scale   */

  x0p = (x0*COS + y0*SIN)/rh;
  y0p = (-x0*SIN + y0*COS)/rv;
  x1p = (x1*COS + y1*SIN)/rh;
  y1p = (-x1*SIN + y1*COS)/rv;

  if (!findUnitCircles(x0p, y0p, x1p, y1p,
                       &pcx0, &pcy0, &pcx1, &pcy1)) {
    return VG_FALSE;
  }

  /* Transform back to original coordinate space  */
  /* using (forward) scale followed by (forward) rotate */

  pcx0 *= rh; pcy0 *= rv;
  pcx1 *= rh; pcy1 *= rv;

  *cx0 = pcx0*COS – pcy0*SIN;
  *cy0 = pcx0*SIN + pcy0*COS;
  *cx1 = pcx1*COS – pcy1*SIN;
  *cy1 = pcx1*SIN + pcy1*COS;

  return VG_TRUE;
}
```

17.5 Implicit Representation of an Ellipse

An ellipse (or any conic section) may be written in the implicit form:

$$A x^2 + B x y + C y^2 + D x + E y + F = 0$$

This equation describes an ellipse (or circle) if $B^2 - 4\,A\,C < 0$ (and certain other degeneracies do not occur). The center of the ellipse is located at:

$$(cx, cy) = \frac{1}{B^2 - 4AC}(2CD - BE, 2AE - BD)$$

The ellipse may be re-centered about $(0, 0)$ by substituting $x \leftarrow x + cx$, $y \leftarrow y + cy$ to obtain an implicit equation with $D = E = 0$:

$$A x^2 + B x y + C y^2 + \left(\frac{A E^2 + C D^2 - B D E}{B^2 - 4AC} + F \right) = 0$$

For a centered ellipse, the constant term must be equal to -1 since the entire formula has the form of $(x')^2 + (y')^2 - 1$ where x' and y' contain no constant terms. Thus in order to determine the radius and axes of a centered ellipse we only need to be concerned with equations of the form:

$$A x^2 + B x y + C y^2 - 1 = 0$$

The angle of rotation is given by:

$$\theta = \begin{cases} 0, & \text{if } B = 0 \\ \dfrac{\pi}{4}, & \text{if } B \neq 0 \text{ and } A = C \\ \dfrac{1}{2}\tan^{-1}\left(\dfrac{B}{A - C}\right), & \text{otherwise} \end{cases}$$

Applying an inverse rotation by substituting $x \leftarrow x\,cos(-\theta) + y\,sin(-\theta)$ and $y \leftarrow y\,cos(-\theta) - x\,sin(-\theta)$, we obtain a further simplification to an unrotated form:

where:

$$A' = \begin{cases} A' x^2 + C' y^2 - 1 = 0 \quad A, & \text{if } B = 0 \\ A + \dfrac{B}{2}, & \text{if } B \neq 0 \text{ and } A = C \\ \dfrac{1}{2}\left(A + C + K(A - C) \right), & \text{otherwise} \end{cases}$$

$$C' = \begin{cases} C, & \text{if } B = 0 \\ A - \dfrac{B}{2}, & \text{if } B \neq 0 \text{ and } A = C \\ \dfrac{1}{2}\left(A + C - K(A - C) \right), & \text{otherwise} \end{cases}$$

$$\text{where } K = \sqrt{1 + \frac{B^2}{(A - C)^2}}$$

The radii of the centered, unrotated ellipse are given by:

$$rh = \frac{1}{\sqrt{A'}} \qquad rv = \frac{1}{\sqrt{C'}}$$

17.6 Transformation of Ellipses

As previously noted, an ellipse may be viewed as the result of a scale, rotation, and translation applied to the unit circle:

$$\begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} rh\cos(\phi) & -rv\sin(\phi) & cx \\ rh\sin(\phi) & rv\cos(\phi) & cy \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} \cos(\theta) \\ \sin(\theta) \\ 1 \end{bmatrix}$$

The resulting ellipse satisfies an implicit equation generated by placing each point on the ellipse through an affine transformation *M* that is the inverse of the transformation above. The resulting points lie on the unit circle, and therefore satisfy the implicit equation $x^2 + y^2 = 1$.

If M is defined as:

$$M = \begin{bmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} rh\cos(\phi) & -rv\sin(\phi) & cx \\ rh\sin(\phi) & rv\cos(\phi) & cy \\ 0 & 0 & 1 \end{bmatrix}^{-1}$$

then the implicit equation for the ellipse is:

$$(m_{00}x + m_{01}y + m_{02})^2 + (m_{10}x + m_{11}y + m_{12})^2 - 1 = 0$$

which may be written in standard form as:

$$A x^2 + B x y + C y^2 + D x + E y + F = 0$$

where:

$$\begin{aligned} &A = m_{00}^2 + m_{10}^2 \\ &B = 2(m_{00}m_{01} + m_{10}m_{11}) \\ &C = m_{01}^2 + m_{11}^2 \\ &D = 2(m_{00}m_{02} + m_{10}m_{12}) \\ &E = 2(m_{01}m_{02} + m_{11}m_{12}) \\ &F = m_{02}^2 + m_{12}^2 - 1 \end{aligned}$$

The center, rotation angle, and radii of the ellipse may be determined using the formulas from the previous section.

In practice, it may be simpler to represent a transformed ellipse as the affine transformation mapping an arc of the unit circle into it. The ellipse may be rendered by concatenating its transform with the current transform and rendering the circular arc. It may be transformed by simply concatenating the transforms.

18 Appendix B: Header Files

This section defines minimal C language header files for the type definitions and functions of OpenVG and the VGU utility library. The actual header files provided by a platform vendor may differ from those shown here.

openvg.h

```
/**********************************************************************
 *                                                                    *
 * Sample implementation of openvg.h, version 1.0.1                   *
 *                                                                    *
 * Copyright © 2005-2007 The Khronos Group                            *
 *                                                                    *
 **********************************************************************/

#ifndef _OPENVG_H
#define _OPENVG_H

#ifdef __cplusplus
extern "C" {
#endif

#include <khronos_types.h>

#define OPENVG_VERSION_1_0 1

typedef khronos_float    VGfloat;
typedef khronos_int8_t   VGbyte;
typedef khronos_uint8_t  VGubyte;
typedef khronos_int16_t  VGshort;
typedef khronos_int32_t  VGint;
typedef khronos_uint32_t VGuint;
typedef khronos_uint32_t VGbitfield;

typedef enum {
  VG_FALSE = 0,
  VG_TRUE  = 1
} VGboolean;

#define VG_MAXSHORT ((VGshort)((~((unsigned)0)) >> 1))
#define VG_MAXINT ((VGint)((~((unsigned)0)) >> 1))

typedef VGuint VGHandle;

#define VG_INVALID_HANDLE ((VGHandle)0)

typedef enum {
  VG_NO_ERROR                                 = 0,
  VG_BAD_HANDLE_ERROR                         = 0x1000,
  VG_ILLEGAL_ARGUMENT_ERROR                   = 0x1001,
  VG_OUT_OF_MEMORY_ERROR                      = 0x1002,
  VG_PATH_CAPABILITY_ERROR                    = 0x1003,
  VG_UNSUPPORTED_IMAGE_FORMAT_ERROR           = 0x1004,
  VG_UNSUPPORTED_PATH_FORMAT_ERROR            = 0x1005,
  VG_IMAGE_IN_USE_ERROR                        = 0x1006,
  VG_NO_CONTEXT_ERROR                          = 0x1007
} VGErrorCode;

typedef enum {
  /* Mode settings */
  VG_MATRIX_MODE                               = 0x1100,
  VG_FILL_RULE                                 = 0x1101,
  VG_IMAGE_QUALITY                             = 0x1102,
  VG_RENDERING_QUALITY                         = 0x1103,
  VG_BLEND_MODE                                = 0x1104,
  VG_IMAGE_MODE                                = 0x1105,

  /* Scissoring rectangles */
  VG_SCISSOR_RECTS                             = 0x1106,

  /* Stroke parameters */
  VG_STROKE_LINE_WIDTH                         = 0x1110,
  VG_STROKE_CAP_STYLE                          = 0x1111,
  VG_STROKE_JOIN_STYLE                         = 0x1112,
  VG_STROKE_MITER_LIMIT                        = 0x1113,
  VG_STROKE_DASH_PATTERN                       = 0x1114,
  VG_STROKE_DASH_PHASE                         = 0x1115,
  VG_STROKE_DASH_PHASE_RESET                   = 0x1116,

  /* Edge fill color for VG_TILE_FILL tiling mode */
  VG_TILE_FILL_COLOR                           = 0x1120,

  /* Color for vgClear */
  VG_CLEAR_COLOR                               = 0x1121,

  /* Enable/disable alpha masking and scissoring */
  VG_MASKING                                   = 0x1130,
  VG_SCISSORING                                = 0x1131,

  /* Pixel layout information */
  VG_PIXEL_LAYOUT                              = 0x1140,
  VG_SCREEN_LAYOUT                             = 0x1141,

  /* Source format selection for image filters */
  VG_FILTER_FORMAT_LINEAR                      = 0x1150,
  VG_FILTER_FORMAT_PREMULTIPLIED               = 0x1151,

  /* Destination write enable mask for image filters */
  VG_FILTER_CHANNEL_MASK                       = 0x1152,

  /* Implementation limits (read-only) */
  VG_MAX_SCISSOR_RECTS                         = 0x1160,
  VG_MAX_DASH_COUNT                            = 0x1161,
  VG_MAX_KERNEL_SIZE                           = 0x1162,
  VG_MAX_SEPARABLE_KERNEL_SIZE                 = 0x1163,
  VG_MAX_COLOR_RAMP_STOPS                      = 0x1164,
  VG_MAX_IMAGE_WIDTH                           = 0x1165,
  VG_MAX_IMAGE_HEIGHT                          = 0x1166,
  VG_MAX_IMAGE_PIXELS                          = 0x1167,
  VG_MAX_IMAGE_BYTES                           = 0x1168,
  VG_MAX_FLOAT                              = 0x1169,
  VG_MAX_GAUSSIAN_STD_DEVIATION             = 0x116A
} VGParamType;

typedef enum {
  VG_RENDERING_QUALITY_NONANTIALIASED       = 0x1200,
  VG_RENDERING_QUALITY_FASTER               = 0x1201,
  VG_RENDERING_QUALITY_BETTER               = 0x1202  /* Default */
} VGRenderingQuality;

typedef enum {
  VG_PIXEL_LAYOUT_UNKNOWN                   = 0x1300,
  VG_PIXEL_LAYOUT_RGB_VERTICAL              = 0x1301,
  VG_PIXEL_LAYOUT_BGR_VERTICAL              = 0x1302,
  VG_PIXEL_LAYOUT_RGB_HORIZONTAL            = 0x1303,
  VG_PIXEL_LAYOUT_BGR_HORIZONTAL            = 0x1304
} VGPixelLayout;

typedef enum {
  VG_MATRIX_PATH_USER_TO_SURFACE            = 0x1400,
  VG_MATRIX_IMAGE_USER_TO_SURFACE           = 0x1401,
  VG_MATRIX_FILL_PAINT_TO_USER              = 0x1402,
  VG_MATRIX_STROKE_PAINT_TO_USER            = 0x1403
} VGMatrixMode;

typedef enum {
  VG_CLEAR_MASK                             = 0x1500,
  VG_FILL_MASK                              = 0x1501,
  VG_SET_MASK                               = 0x1502,
  VG_UNION_MASK                             = 0x1503,
  VG_INTERSECT_MASK                         = 0x1504,
  VG_SUBTRACT_MASK                          = 0x1505
} VGMaskOperation;

#define VG_PATH_FORMAT_STANDARD 0

typedef enum {
  VG_PATH_DATATYPE_S_8                      =  0,
  VG_PATH_DATATYPE_S_16                     =  1,
  VG_PATH_DATATYPE_S_32                     =  2,
  VG_PATH_DATATYPE_F                        =  3
} VGPathDatatype;

typedef enum {
  VG_ABSOLUTE                               = 0,
  VG_RELATIVE                               = 1
} VGPathAbsRel;

typedef enum {
  VG_CLOSE_PATH                             = ( 0 << 1),
  VG_MOVE_TO                                = ( 1 << 1),
  VG_LINE_TO                                = ( 2 << 1),
  VG_HLINE_TO                               = ( 3 << 1),
  VG_VLINE_TO                               = ( 4 << 1),
  VG_QUAD_TO                                = ( 5 << 1),
  VG_CUBIC_TO                               = ( 6 << 1),
  VG_SQUAD_TO                                 = ( 7 << 1),
  VG_SCUBIC_TO                                = ( 8 << 1),
  VG_SCCWARC_TO                               = ( 9 << 1),
  VG_SCWARC_TO                                = (10 << 1),
  VG_LCCWARC_TO                               = (11 << 1),
  VG_LCWARC_TO                                = (12 << 1)
} VGPathSegment;

typedef enum {
  VG_MOVE_TO_ABS                              = VG_MOVE_TO    | VG_ABSOLUTE,
  VG_MOVE_TO_REL                              = VG_MOVE_TO    | VG_RELATIVE,
  VG_LINE_TO_ABS                              = VG_LINE_TO    | VG_ABSOLUTE,
  VG_LINE_TO_REL                              = VG_LINE_TO    | VG_RELATIVE,
  VG_HLINE_TO_ABS                             = VG_HLINE_TO   | VG_ABSOLUTE,
  VG_HLINE_TO_REL                             = VG_HLINE_TO   | VG_RELATIVE,
  VG_VLINE_TO_ABS                             = VG_VLINE_TO   | VG_ABSOLUTE,
  VG_VLINE_TO_REL                             = VG_VLINE_TO   | VG_RELATIVE,
  VG_QUAD_TO_ABS                              = VG_QUAD_TO    | VG_ABSOLUTE,
  VG_QUAD_TO_REL                              = VG_QUAD_TO    | VG_RELATIVE,
  VG_CUBIC_TO_ABS                             = VG_CUBIC_TO   | VG_ABSOLUTE,
  VG_CUBIC_TO_REL                             = VG_CUBIC_TO   | VG_RELATIVE,
  VG_SQUAD_TO_ABS                             = VG_SQUAD_TO   | VG_ABSOLUTE,
  VG_SQUAD_TO_REL                             = VG_SQUAD_TO   | VG_RELATIVE,
  VG_SCUBIC_TO_ABS                            = VG_SCUBIC_TO  | VG_ABSOLUTE,
  VG_SCUBIC_TO_REL                            = VG_SCUBIC_TO  | VG_RELATIVE,
  VG_SCCWARC_TO_ABS                           = VG_SCCWARC_TO | VG_ABSOLUTE,
  VG_SCCWARC_TO_REL                           = VG_SCCWARC_TO | VG_RELATIVE,
  VG_SCWARC_TO_ABS                            = VG_SCWARC_TO  | VG_ABSOLUTE,
  VG_SCWARC_TO_REL                            = VG_SCWARC_TO  | VG_RELATIVE,
  VG_LCCWARC_TO_ABS                           = VG_LCCWARC_TO | VG_ABSOLUTE,
  VG_LCCWARC_TO_REL                           = VG_LCCWARC_TO | VG_RELATIVE,
  VG_LCWARC_TO_ABS                            = VG_LCWARC_TO  | VG_ABSOLUTE,
  VG_LCWARC_TO_REL                            = VG_LCWARC_TO  | VG_RELATIVE
} VGPathCommand;

typedef VGHandle VGPath;

typedef enum {
  VG_PATH_CAPABILITY_APPEND_FROM              = (1 <<  0),
  VG_PATH_CAPABILITY_APPEND_TO                = (1 <<  1),
  VG_PATH_CAPABILITY_MODIFY                   = (1 <<  2),
  VG_PATH_CAPABILITY_TRANSFORM_FROM           = (1 <<  3),
  VG_PATH_CAPABILITY_TRANSFORM_TO             = (1 <<  4),
  VG_PATH_CAPABILITY_INTERPOLATE_FROM         = (1 <<  5),
  VG_PATH_CAPABILITY_INTERPOLATE_TO           = (1 <<  6),
  VG_PATH_CAPABILITY_PATH_LENGTH              = (1 <<  7),
  VG_PATH_CAPABILITY_POINT_ALONG_PATH         = (1 <<  8),
  VG_PATH_CAPABILITY_TANGENT_ALONG_PATH       = (1 <<  9),
  VG_PATH_CAPABILITY_PATH_BOUNDS              = (1 << 10),
  VG_PATH_CAPABILITY_PATH_TRANSFORMED_BOUNDS  = (1 << 11),
  VG_PATH_CAPABILITY_ALL                      = (1 << 12) - 1
} VGPathCapabilities;

typedef enum {
  VG_PATH_FORMAT                              = 0x1600,
  VG_PATH_DATATYPE                            = 0x1601,
  VG_PATH_SCALE                               = 0x1602,
  VG_PATH_BIAS                                = 0x1603,
  VG_PATH_NUM_SEGMENTS                        = 0x1604,
  VG_PATH_NUM_COORDS                          = 0x1605
} VGPathParamType;

typedef enum {
  VG_CAP_BUTT                                 = 0x1700,
  VG_CAP_ROUND                                = 0x1701,
  VG_CAP_SQUARE                               = 0x1702
} VGCapStyle;

typedef enum {
  VG_JOIN_MITER                               = 0x1800,
  VG_JOIN_ROUND                               = 0x1801,
  VG_JOIN_BEVEL                               = 0x1802
} VGJoinStyle;

typedef enum {
  VG_EVEN_ODD                                 = 0x1900,
  VG_NON_ZERO                                 = 0x1901
} VGFillRule;

typedef enum {
  VG_STROKE_PATH                              = (1 << 0),
  VG_FILL_PATH                                = (1 << 1)
} VGPaintMode;

typedef VGHandle VGPaint;

typedef enum {
  /* Color paint parameters */
  VG_PAINT_TYPE                               = 0x1A00,
  VG_PAINT_COLOR                              = 0x1A01,
  VG_PAINT_COLOR_RAMP_SPREAD_MODE             = 0x1A02,
  VG_PAINT_COLOR_RAMP_PREMULTIPLIED           = 0x1A07,
  VG_PAINT_COLOR_RAMP_STOPS                   = 0x1A03,

  /* Linear gradient paint parameters */
  VG_PAINT_LINEAR_GRADIENT                    = 0x1A04,

  /* Radial gradient paint parameters */
  VG_PAINT_RADIAL_GRADIENT                    = 0x1A05,

  /* Pattern paint parameters */
  VG_PAINT_PATTERN_TILING_MODE                = 0x1A06
} VGPaintParamType;

typedef enum {
  VG_PAINT_TYPE_COLOR                         = 0x1B00,
  VG_PAINT_TYPE_LINEAR_GRADIENT               = 0x1B01,
  VG_PAINT_TYPE_RADIAL_GRADIENT               = 0x1B02,
  VG_PAINT_TYPE_PATTERN                       = 0x1B03
} VGPaintType;

typedef enum {
  VG_COLOR_RAMP_SPREAD_PAD                   = 0x1C00,
  VG_COLOR_RAMP_SPREAD_REPEAT                = 0x1C01,
  VG_COLOR_RAMP_SPREAD_REFLECT               = 0x1C02
} VGColorRampSpreadMode;

typedef enum {
  VG_TILE_FILL                               = 0x1D00,
  VG_TILE_PAD                                = 0x1D01,
  VG_TILE_REPEAT                             = 0x1D02,
  VG_TILE_REFLECT                            = 0x1D03
} VGTilingMode;

typedef enum {
  /* RGB{A,X} channel ordering */
  VG_sRGBX_8888                              =  0,
  VG_sRGBA_8888                              =  1,
  VG_sRGBA_8888_PRE                          =  2,
  VG_sRGB_565                                =  3,
  VG_sRGBA_5551                              =  4,
  VG_sRGBA_4444                              =  5,
  VG_sL_8                                    =  6,
  VG_lRGBX_8888                              =  7,
  VG_lRGBA_8888                              =  8,
  VG_lRGBA_8888_PRE                          =  9,
  VG_lL_8                                    = 10,
  VG_A_8                                     = 11,
  VG_BW_1                                    = 12,

  /* {A,X}RGB channel ordering */
  VG_sXRGB_8888                              =  0 | (1 << 6),
  VG_sARGB_8888                              =  1 | (1 << 6),
  VG_sARGB_8888_PRE                          =  2 | (1 << 6),
  VG_sARGB_1555                              =  4 | (1 << 6),
  VG_sARGB_4444                              =  5 | (1 << 6),
  VG_lXRGB_8888                              =  7 | (1 << 6),
  VG_lARGB_8888                              =  8 | (1 << 6),
  VG_lARGB_8888_PRE                          =  9 | (1 << 6),

  /* BGR{A,X} channel ordering */
  VG_sBGRX_8888                              =  0 | (1 << 7),
  VG_sBGRA_8888                              =  1 | (1 << 7),
  VG_sBGRA_8888_PRE                          =  2 | (1 << 7),
  VG_sBGR_565                                =  3 | (1 << 7),
  VG_sBGRA_5551                              =  4 | (1 << 7),
  VG_sBGRA_4444                              =  5 | (1 << 7),
  VG_lBGRX_8888                              =  7 | (1 << 7),
  VG_lBGRA_8888                              =  8 | (1 << 7),
  VG_lBGRA_8888_PRE                          =  9 | (1 << 7),

  /* {A,X}BGR channel ordering */
  VG_sXBGR_8888                               =  0 | (1 << 6) | (1 << 7),
  VG_sABGR_8888                               =  1 | (1 << 6) | (1 << 7),
  VG_sABGR_8888_PRE                           =  2 | (1 << 6) | (1 << 7),
  VG_sABGR_1555                               =  4 | (1 << 6) | (1 << 7),
  VG_sABGR_4444                               =  5 | (1 << 6) | (1 << 7),
  VG_lXBGR_8888                               =  7 | (1 << 6) | (1 << 7),
  VG_lABGR_8888                               =  8 | (1 << 6) | (1 << 7),
  VG_lABGR_8888_PRE                           =  9 | (1 << 6) | (1 << 7)
} VGImageFormat;

typedef VGHandle VGImage;

typedef enum {
  VG_IMAGE_QUALITY_NONANTIALIASED             = (1 << 0),
  VG_IMAGE_QUALITY_FASTER                     = (1 << 1),
  VG_IMAGE_QUALITY_BETTER                     = (1 << 2)
} VGImageQuality;

typedef enum {
  VG_IMAGE_FORMAT                             = 0x1E00,
  VG_IMAGE_WIDTH                              = 0x1E01,
  VG_IMAGE_HEIGHT                             = 0x1E02
} VGImageParamType;

typedef enum {
  VG_DRAW_IMAGE_NORMAL                        = 0x1F00,
  VG_DRAW_IMAGE_MULTIPLY                      = 0x1F01,
  VG_DRAW_IMAGE_STENCIL                       = 0x1F02
} VGImageMode;

typedef enum {
  VG_RED                                      = (1 << 3),
  VG_GREEN                                    = (1 << 2),
  VG_BLUE                                     = (1 << 1),
  VG_ALPHA                                    = (1 << 0)
} VGImageChannel;

typedef enum {
  VG_BLEND_SRC                                = 0x2000,
  VG_BLEND_SRC_OVER                           = 0x2001,
  VG_BLEND_DST_OVER                           = 0x2002,
  VG_BLEND_SRC_IN                             = 0x2003,
  VG_BLEND_DST_IN                             = 0x2004,
  VG_BLEND_MULTIPLY                           = 0x2005,
  VG_BLEND_SCREEN                             = 0x2006,
  VG_BLEND_DARKEN                             = 0x2007,
  VG_BLEND_LIGHTEN                            = 0x2008,
  VG_BLEND_ADDITIVE                           = 0x2009
} VGBlendMode;

typedef enum {
  VG_IMAGE_FORMAT_QUERY                       = 0x2100,
  VG_PATH_DATATYPE_QUERY                      = 0x2101
} VGHardwareQueryType;

typedef enum {
  VG_HARDWARE_ACCELERATED                    = 0x2200,
  VG_HARDWARE_UNACCELERATED                  = 0x2201
} VGHardwareQueryResult;

typedef enum {
  VG_VENDOR                                  = 0x2300,
  VG_RENDERER                                = 0x2301,
  VG_VERSION                                 = 0x2302,
  VG_EXTENSIONS                              = 0x2303
} VGStringID;

/* Function Prototypes */

#ifndef VG_API_CALL
#define VG_API_CALL extern
#endif

VG_API_CALL VGErrorCode vgGetError(void);

VG_API_CALL void vgFlush(void);
VG_API_CALL void vgFinish(void);

/* Getters and Setters */
VG_API_CALL void vgSetf (VGParamType type, VGfloat value);
VG_API_CALL void vgSeti (VGParamType type, VGint value);
VG_API_CALL void vgSetfv(VGParamType type, VGint count,
                         const VGfloat * values);
VG_API_CALL void vgSetiv(VGParamType type, VGint count,
                         const VGint * values);

VG_API_CALL VGfloat vgGetf(VGParamType type);
VG_API_CALL VGint   vgGeti(VGParamType type);
VG_API_CALL VGint   vgGetVectorSize(VGParamType type);
VG_API_CALL void    vgGetfv(VGParamType type, VGint count, VGfloat * values);
VG_API_CALL void    vgGetiv(VGParamType type, VGint count, VGint * values);

VG_API_CALL void vgSetParameterf(VGHandle object,
                                 VGint paramType,
                                 VGfloat value);
VG_API_CALL void vgSetParameteri(VGHandle object,
                                 VGint paramType,
                                 VGint value);
VG_API_CALL void vgSetParameterfv(VGHandle object,
                                  VGint paramType,
                                  VGint count, const VGfloat * values);
VG_API_CALL void vgSetParameteriv(VGHandle object,
                                  VGint paramType,
                                  VGint count, const VGint * values);

VG_API_CALL VGfloat vgGetParameterf(VGHandle object,
                                    VGint paramType);
VG_API_CALL VGint vgGetParameteri(VGHandle object,
                                  VGint paramType);
VG_API_CALL VGint vgGetParameterVectorSize(VGHandle object,
                                           VGint paramType);
VG_API_CALL void vgGetParameterfv(VGHandle object,
                                  VGint paramType,
                                  VGint count, VGfloat * values);
VG_API_CALL void vgGetParameteriv(VGHandle object,
                                  VGint paramType,
                                  VGint count, VGint * values);

/* Matrix Manipulation */
VG_API_CALL void vgLoadIdentity(void);
VG_API_CALL void vgLoadMatrix(const VGfloat * m);
VG_API_CALL void vgGetMatrix(VGfloat * m);
VG_API_CALL void vgMultMatrix(const VGfloat * m);
VG_API_CALL void vgTranslate(VGfloat tx, VGfloat ty);
VG_API_CALL void vgScale(VGfloat sx, VGfloat sy);
VG_API_CALL void vgShear(VGfloat shx, VGfloat shy);
VG_API_CALL void vgRotate(VGfloat angle);

/* Masking and Clearing */
VG_API_CALL void vgMask(VGImage mask, VGMaskOperation operation,
                        VGint x, VGint y, VGint width, VGint height);
VG_API_CALL void vgClear(VGint x, VGint y, VGint width, VGint height);

/* Paths */
VG_API_CALL VGPath vgCreatePath(VGint pathFormat,
                                VGPathDatatype datatype,
                                VGfloat scale, VGfloat bias,
                                VGint segmentCapacityHint,
                                VGint coordCapacityHint,
                                VGbitfield capabilities);
VG_API_CALL void vgClearPath(VGPath path, VGbitfield capabilities);
VG_API_CALL void vgDestroyPath(VGPath path);
VG_API_CALL void vgRemovePathCapabilities(VGPath path,
                                          VGbitfield capabilities);
VG_API_CALL VGbitfield vgGetPathCapabilities(VGPath path);
VG_API_CALL void vgAppendPath(VGPath dstPath, VGPath srcPath);
VG_API_CALL void vgAppendPathData(VGPath dstPath,
                                  VGint numSegments,
                                  const VGubyte * pathSegments,
                                  const void * pathData);
VG_API_CALL void vgModifyPathCoords(VGPath dstPath, VGint startIndex,
                                    VGint numSegments,
                                    const void * pathData);
VG_API_CALL void vgTransformPath(VGPath dstPath, VGPath srcPath);
VG_API_CALL VGboolean vgInterpolatePath(VGPath dstPath,
                                        VGPath startPath,
                                        VGPath endPath,
                                        VGfloat amount);
VG_API_CALL VGfloat vgPathLength(VGPath path,
                                 VGint startSegment, VGint numSegments);
VG_API_CALL void vgPointAlongPath(VGPath path,
                                  VGint startSegment, VGint numSegments,
                                  VGfloat distance,
                                  VGfloat * x, VGfloat * y,
                                  VGfloat * tangentX, VGfloat * tangentY);
VG_API_CALL void vgPathBounds(VGPath path,
                              VGfloat * minX, VGfloat * minY,
                              VGfloat * width, VGfloat * height);
```

```
VG_API_CALL void vgPathTransformedBounds(VGPath path,
                                         VGfloat * minX, VGfloat * minY,
                                         VGfloat * width, VGfloat * height);
VG_API_CALL void vgDrawPath(VGPath path, VGbitfield paintModes);

/* Paint */
VG_API_CALL VGPaint vgCreatePaint(void);
VG_API_CALL void vgDestroyPaint(VGPaint paint);
VG_API_CALL void vgSetPaint(VGPaint paint, VGbitfield paintModes);
VG_API_CALL VGPaint vgGetPaint(VGPaintMode paintMode);
VG_API_CALL void vgSetColor(VGPaint paint, VGuint rgba);
VG_API_CALL VGuint vgGetColor(VGPaint paint);
VG_API_CALL void vgPaintPattern(VGPaint paint, VGImage pattern);

/* Images */
VG_API_CALL VGImage vgCreateImage(VGImageFormat format,
                                  VGint width, VGint height,
                                  VGbitfield allowedQuality);
VG_API_CALL void vgDestroyImage(VGImage image);
VG_API_CALL void vgClearImage(VGImage image,
                              VGint x, VGint y, VGint width, VGint height);
VG_API_CALL void vgImageSubData(VGImage image,
                                const void * data, VGint dataStride,
                                VGImageFormat dataFormat,
                                VGint x, VGint y, VGint width, VGint height);
VG_API_CALL void vgGetImageSubData(VGImage image,
                                   void * data, VGint dataStride,
                                   VGImageFormat dataFormat,
                                   VGint x, VGint y,
                                   VGint width, VGint height);
VG_API_CALL VGImage vgChildImage(VGImage parent,
                                 VGint x, VGint y, VGint width, VGint height);
VG_API_CALL VGImage vgGetParent(VGImage image);
VG_API_CALL void vgCopyImage(VGImage dst, VGint dx, VGint dy,
                             VGImage src, VGint sx, VGint sy,
                             VGint width, VGint height,
                             VGboolean dither);
VG_API_CALL void vgDrawImage(VGImage image);
VG_API_CALL void vgSetPixels(VGint dx, VGint dy,
                             VGImage src, VGint sx, VGint sy,
                             VGint width, VGint height);
VG_API_CALL void vgWritePixels(const void * data, VGint dataStride,
                               VGImageFormat dataFormat,
                               VGint dx, VGint dy,
                               VGint width, VGint height);
VG_API_CALL void vgGetPixels(VGImage dst, VGint dx, VGint dy,
                             VGint sx, VGint sy,
                             VGint width, VGint height);
VG_API_CALL void vgReadPixels(void * data, VGint dataStride,
                              VGImageFormat dataFormat,
                              VGint sx, VGint sy,
                              VGint width, VGint height);
VG_API_CALL void vgCopyPixels(VGint dx, VGint dy,
                              VGint sx, VGint sy,
                              VGint width, VGint height);
```

```
/* Image Filters */
VG_API_CALL void vgColorMatrix(VGImage dst, VGImage src,
                               const VGfloat * matrix);
VG_API_CALL void vgConvolve(VGImage dst, VGImage src,
                            VGint kernelWidth, VGint kernelHeight,
                            VGint shiftX, VGint shiftY,
                            const VGshort * kernel,
                            VGfloat scale,
                            VGfloat bias,
                            VGTilingMode tilingMode);
VG_API_CALL void vgSeparableConvolve(VGImage dst, VGImage src,
                                     VGint kernelWidth,
                                     VGint kernelHeight,
                                     VGint shiftX, VGint shiftY,
                                     const VGshort * kernelX,
                                     const VGshort * kernelY,
                                     VGfloat scale,
                                     VGfloat bias,
                                     VGTilingMode tilingMode);
VG_API_CALL void vgGaussianBlur(VGImage dst, VGImage src,
                                VGfloat stdDeviationX,
                                VGfloat stdDeviationY,
                                VGTilingMode tilingMode);
VG_API_CALL void vgLookup(VGImage dst, VGImage src,
                          const VGubyte * redLUT,
                          const VGubyte * greenLUT,
                          const VGubyte * blueLUT,
                          const VGubyte * alphaLUT,
                          VGboolean outputLinear,
                          VGboolean outputPremultiplied);
VG_API_CALL void vgLookupSingle(VGImage dst, VGImage src,
                                const VGuint * lookupTable,
                                VGImageChannel sourceChannel,
                                VGboolean outputLinear,
                                VGboolean outputPremultiplied);

/* Hardware Queries */
VG_API_CALL VGHardwareQueryResult vgHardwareQuery(VGHardwareQueryType key,
                                                  VGint setting);

/* Renderer and Extension Information */
VG_API_CALL const VGubyte * vgGetString(VGStringID name);

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif /* _OPENVG_H */
```

vgu.h

```
/*********************************************************************
 *                                                                   *
 * Sample implementation of vgu.h, version 1.0.1                     *
 *                                                                   *
 * Copyright © 2005-2007 The Khronos Group                           *
 *                                                                   *
 *********************************************************************/

#ifndef _VGU_H
#define _VGU_H

#ifdef __cplusplus
extern "C" {
#endif

#include <vg/openvg.h>

#define VGU_VERSION_1_0 1

#ifndef VGU_API_CALL
#define VGU_API_CALL extern
#endif

typedef enum {
  VGU_NO_ERROR                                = 0,
  VGU_BAD_HANDLE_ERROR                        = 0xF000,
  VGU_ILLEGAL_ARGUMENT_ERROR                  = 0xF001,
  VGU_OUT_OF_MEMORY_ERROR                     = 0xF002,
  VGU_PATH_CAPABILITY_ERROR                   = 0xF003,
  VGU_BAD_WARP_ERROR                          = 0xF004
} VGUErrorCode;

typedef enum {
  VGU_ARC_OPEN                                = 0xF100,
  VGU_ARC_CHORD                               = 0xF101,
  VGU_ARC_PIE                                 = 0xF102
} VGUArcType;

VGU_API_CALL VGUErrorCode vguLine(VGPath path,
                                  VGfloat x0, VGfloat y0,
                                  VGfloat x1, VGfloat y1);

VGU_API_CALL VGUErrorCode vguPolygon(VGPath path,
                                     const VGfloat * points, VGint count,
                                     VGboolean closed);

VGU_API_CALL VGUErrorCode vguRect(VGPath path,
                                  VGfloat x, VGfloat y,
                                  VGfloat width, VGfloat height);

VGU_API_CALL VGUErrorCode vguRoundRect(VGPath path,
                                       VGfloat x, VGfloat y,
                                       VGfloat width, VGfloat height,
                                       VGfloat arcWidth, VGfloat arcHeight);
VGU_API_CALL VGUErrorCode vguEllipse(VGPath path,
                                     VGfloat cx, VGfloat cy,
                                     VGfloat width, VGfloat height);

VGU_API_CALL VGUErrorCode vguArc(VGPath path,
                                 VGfloat x, VGfloat y,
                                 VGfloat width, VGfloat height,
                                 VGfloat startAngle, VGfloat angleExtent,
                                 VGUArcType arcType);

VGU_API_CALL VGUErrorCode vguComputeWarpQuadToSquare(VGfloat sx0, VGfloat sy0,
                                                     VGfloat sx1, VGfloat sy1,
                                                     VGfloat sx2, VGfloat sy2,
                                                     VGfloat sx3, VGfloat sy3,
                                                     VGfloat * matrix);

VGU_API_CALL VGUErrorCode vguComputeWarpSquareToQuad(VGfloat dx0, VGfloat dy0,
                                                     VGfloat dx1, VGfloat dy1,
                                                     VGfloat dx2, VGfloat dy2,
                                                     VGfloat dx3, VGfloat dy3,
                                                     VGfloat * matrix);

VGU_API_CALL VGUErrorCode vguComputeWarpQuadToQuad(VGfloat dx0, VGfloat dy0,
                                                   VGfloat dx1, VGfloat dy1,
                                                   VGfloat dx2, VGfloat dy2,
                                                   VGfloat dx3, VGfloat dy3,
                                                   VGfloat sx0, VGfloat sy0,
                                                   VGfloat sx1, VGfloat sy1,
                                                   VGfloat sx2, VGfloat sy2,
                                                   VGfloat sx3, VGfloat sy3,
                                                   VGfloat * matrix);

#ifdef __cplusplus
} /* extern "C" */
#endif

#endif /* #ifndef _VGU_H */
```

19 Bibliography

ADOB99 Adobe Systems Incorporated: *PostScript Language Reference Manual* (*third edition*), Addison-Wesley, Reading, MA, 1999.

ADOB06a Adobe Systems Incorporated: *PDF Reference* (six*th edition*): http://www.adobe.com/devnet/acrobat/pdfs/pdf_reference.pdf

ADOB06b Macromedia, Inc., Macromedia Flash Developer Center: http://www.adobe.com/devnet/flash

FvDFH95 Foley J., A. van Dam, S. Feiner and J. Hughes, *Computer Graphics: Principles and Practice* (*second edition*), Addison-Wesley, Reading, MA, 1995.

HECK89 Heckbert, Paul, *Fundamentals of Texture Mapping and Image Warping*, Master's thesis, UCB/CSD 89/516, CS Division, U.C. Berkeley, June 1989.

ITU90 Recommendation ITU-R BT.709, *Basic Parameter Values for the HDTV Standard for the Studio and for International Programme Exchange* (1990), ITU, Geneva, Switzerland.

PORT84 Porter, T. and T. Duff, "Compositing Digital Images," *Computer Graphics* 18(3):253-259 (proc. SIGGRAPH 1984), July 1984.

POYN03 Poynton, Charles, *Digital Video and HDTV Algorithms and Interfaces*, Morgan Kaufmann, San Francisco, 2003.

sRGB99 IEC 61966-2-1, *Multimedia systems and equipment – Colour measurement and management – Part 2-1: Default RGB colour space – sRGB: http://www.w3.org/Graphics/Color/sRGB.html*

SUN04 Sun Microsystems, Inc., Java 2D API Home Page: http://java.sun.com/products/java-media/2D

SVGF05 W3C Recommendation, Scalable Vector Graphics (SVG) Full 1.2 Specification: http://www.w3.org/TR/SVG12

SVGT06 W3C Recommendation, Scalable Vector Graphics (SVG) Tiny 1.2 Specification: http://www.w3.org/TR/SVGMobile12

WOLB90 Wolberg, G., *Digital Image Warping*, IEEE Computer Society Press, Washington, D.C., 1990.

WYSZ00 Wyszecki, G. and W. S. Stiles, Color Science: Concepts and Methods, Quantitative Data and Formulae, Wiley-Interscience, New York, 2000.

20 Document History

Version 1.0 ratified August 2005

Version 1.0.1 ratified January 2007

Changes from version 1.0 (by section number):

- 3.2 – clarification: `VGboolean` is an enumeration
- 3.4.1 – provide further explanation of linear pixel formats
- 5.2 – new behavior: restrict values of `count` parameter in **vgGet/Set*v, vgGet/SetParameter*v**; describe error behavior of getters
- 5.2.1 – change default value of `VG_FILTER_FORMAT_LINEAR` and `VG_RENDERING_QUALITY`; add `VG_SCREEN_LAYOUT` parameter; add `VG_STROKE_DASH_PHASE_RESET` parameter
- 6.2 – define behavior of `VG_SCREEN_LAYOUT` parameter
- 8.3.4-5 – clarify join behavior for smooth segments following line segments
- 8.4 – change behavior of elliptical arcs with one radius equal to 0
- 8.5 – typo: `VG_PATH_FORMAT_STANDARD` is passed to **vgCreatePath**, not **vgAppendPathData**
- 8.5.2 – clarification: conversion of path segments from relative to absolute form takes place during rendering
- 8.6.7-8 – new behavior: **vgTransformPath** and **vgInterpolatePath** promote `HLINE` and `VLINE` segments to general (2-coordinate) form; the parameterization of transformed elliptical arc segments is undefined
- 8.6.11 – clarification: normalization of tangents; approximate computation of path length
- 8.7.1 – clarification: implicit closure takes place during rendering
- 8.7.3 – clarification: definition and illustration of the miter length
- 8.7.4 – clarification: stroke generation takes place in user coordinates.
- 8.7.4-5 – Add new behavior controlled by `VG_STROKE_DASH_PHASE_RESET`
- 9 – paint coordinates must be evaluated within 1/8 of a pixel; clarify source of user transform *Tu*
- 9.3.3 – add `VG_PAINT_COLOR_RAMP_PREMULTIPLIED` flag to control whether gradient colors are interpolated in premultiplied form
- 9.3.3 – new behavior: `count` must be a multiple of 5 in **vgSetParameter** for color ramp stops (see 5.2); simplify description of rules for repeat and reflect pad modes
- 10.2 – add new values to `VGImageFormat` enumeration
- 10.5 – clarification: **vgImageSubData** clamps premultiplied color values to their corresponding alpha values
- 10.8 – clarify behavior of `VG_DRAW_IMAGE_NORMAL` when the source has an alpha channel; new behavior: when a projective transformation is enabled, **vgDrawImage** always uses `VG_DRAW_IMAGE_NORMAL` mode; clarify behavior when a linear source image is used in `VG_DRAW_IMAGE_MULTIPLY` mode
- 10.9.1 – clarification: **vgWritePixels** clamps premultiplied color values to their corresponding alpha values
- 12.4 – clarification: input color values are clamped at 1
- 14.3.2 – clarify display dependency of **vgGetString**; **vgGetString**(`VG_VERSION`) returns the specification version.

- 16.1.6 – typo: error in **vguArc** pseudo-code
- 18 – remove enumerated values `VG_PATH_DATATYPE_INVALID` and `VG_IMAGE_FORMAT_INVALID`

21 Acknowledgments

This specification was developed by the Khronos OpenVG working group:

- Andrzej Mamona, AMD, Chair
- Daniel Rice, Google, Specification Editor
- Koichi Mori (森 浩一), Nokia, Past Chair
- Neil Trevett, NVIDIA, Past Chair
- Gary Pallett, NVIDIA
- Mike Agar, ALT Software
- Espen Åmodt, ARM
- Frode Heggelund, ARM
- Borgar Ljosland, ARM
- Ed Plowman, ARM
- TK Chan (陳鼎鍵), AMD
- Chris Grimm, AMD
- Marko Laiho, AMD
- Petri Nordlund, AMD
- Kimihiko Sato, AMD
- Robert Simpson, AMD
- Mika Tuomi, AMD
- Keisuke Kiri (桐井 敬祐), DMP
- Eisaku Oobuchi (大渕 栄作), DMP
- Jacob Ström, Ericsson
- Claude Knaus, Esmertec
- Kimball, ETRI
- Kwang-Ho Yang (양 광호), ETRI
- Brian Murray, Freescale
- Yoshikazu Saka (坂 義和), Fujitsu
- Tero Sarkkinen, Futuremark
- Mark Callow, HI Corporation
- Naoya Yamamoto (山本 直也), HI Corporation
- Antti Hätälä, Hybrid Graphics
- Harri Holopainen, Hybrid Graphics
- Petri Kero, Hybrid Graphics
- Ville Miettinen, Hybrid Graphics
- Kalle Raita, Hybrid Graphics
- Jussi Räsänen, Hybrid Graphics
- Christofer Åkersten, Ikivo
- Ola Andersson, Ikivo
- Ben Bowman, Imagination Technologies
- Simon Fenney, Imagination Technologies
- Mohit Mehta, Imagination Technologies
- Alex Walters, Imagination Technologies
- Randy Xu, Intel
- Hang-Shin Cho (조 항신), LG Electronics
- Woo-Seon Kim (김 우섭), LG Electronics
- Yong-Moo Kim (김 용무), LG Electronics
- Masaki Hamada (濱田 雅樹), Mitsubishi Electric
- Hiroyasu Negishi (根岸 博康), Mitsubishi Electric
- Vladimir Levantovsky, Monotype Imaging
- Chris Tremblay, Motorola
- Hwanyong Lee (이 환용), HUONE
- Junyoung Lee (이 준영), HUONE
- Hyunchan Sung(성 현찬), HUONE
- Tomi Aarnio, Nokia
- Tolga Çapın, Nokia
- Suresh Chitturi, Nokia
- Sila Kayo, Nokia
- Clay Montgomery, Nokia
- Tero Pihlajakoski, Nokia
- Kari Pulli, Nokia
- Jani Vaarala, Nokia
- Rakesh Jain, NVIDIA
- Tom McReynolds, NVIDIA
- Chris Wynn, NVIDIA
- Mathias Agopian, PalmSource
- Toshiki Hijjri (樋尻 利紀), Panasonic
- Andy Methley, Panasonic
- Angus Dorbie, Qualcomm
- Keechang Lee (이 기창), Samsung
- Jitaek Lim (임 지택), Samsung
- Jon Leech, SGI
- Thomas Tannert, SGI
- Rémi Arnaud, SONY
- Axel Mamode, SONY
- Chris Campbell, Sun Microsystems
- Jerry Evans, Sun Microsystems
- Sean Ellis, Superscape
- Robert Palmer, Symbian
- Bill Pinnell, Symbian
- Lane Roberts, Symbian
- Jay Abbott, Tao Group
- Bryce Johnstone, Texas Instruments
- Tom Olson, Texas Instruments
- San-Soo Kim (김 산수), Wow4M
- Sung-Jae Kim (김 성재), Wow4M

Special thanks to Jussi Räsänen of Hybrid Graphics; Petri Nordlund, Robert Simpson, and Mika Tuomi of AMD (formerly Bitboys); and Tuomas Lukka, Jarno Paananen, and Sami Tammilehto of the former Bitboys Technology Research Group for creating the reference implementations; and to those who contributed to the conformance test suite: Hwanyong Lee, Junyoung Lee, and Hyunchan Sung of HUONE; TK Chan, Robert Simpson, Valtteri Rantala, Sami Tammilehto, Mika Tuomi, and Miikka Kangasluoma of AMD; and Jussi Räsänen and Kalle Raita of Hybrid Graphics.

Thanks are also due to the external reviewers who helped to improve the specification.

22 Indices

Index of Tables

Table 1: Naming and Typographical Conventions....................3
Table 2: Pixel Format Conversions....................12
Table 3: State Elements of a Context....................17
Table 4: Default Parameter Values for a Context....................26
Table 5: VGMaskOperation Equations....................43
Table 6: Client-Side Path Segment Commands....................53
Table 7: Client-Side Path Coordinate Datatypes....................54
Table 8: VGPathParamType Datatypes....................63
Table 9: Corresponding Angles and Miter Limits....................82
Table 10: VGPaintParamType Defaults....................93
Table 11: Symbols Used in Image Format Names....................110
Table 12: Image Format Pixel Sizes....................111
Table 13: Porter-Duff Blending Modes....................141
Table 14: Additional Blending Equations....................142
Table 15: Query Key Enumeration Types....................143
Table 16: Center Ellipse Parameters....................160
Table 17: Endpoint Ellipse Parameters....................161

Index of Figures

Figure 1: The OpenVG Pipeline....................4
Figure 2: VGPixelLayout Values....................32
Figure 3: Coordinates, Transformation, Clipping, and Scissoring....................34
Figure 4: Smooth Curve Construction....................48
Figure 5: Elliptical Arcs....................50
Figure 6: Segment Reference Points....................51
Figure 7: Segment Type Marker Layout....................54
Figure 8: Even/Odd Fill Rule....................76

Figure 9: Creating Holes with Subpaths........77
Figure 10: Implicit Closure of Filled Paths........78
Figure 11: Stroked Paths Have No Implicit Closure........78
Figure 12: Each Stroke Applies a Single Layer of Paint........79
Figure 13: End Cap Styles........80
Figure 14: End Cap Styles with Additional Geometry Highlighted........80
Figure 15: Line Join Styles........81
Figure 16: Line Join Styles with Additional Geometry Highlighted........81
Figure 17: Dash Pattern and Phase Example........83
Figure 18: Radial Gradient Function........98
Figure 19: Color Ramp Pad Modes........101
Figure 20: Linear Gradients........103
Figure 21: Centered and Non-Centered Radial Gradients........104
Figure 22: Color Ramp used for Gradient Examples........104
Figure 23: Convolution With a Flipped Kernel........134
Figure 24: Round Rectangle Parameters........154
Figure 25: VGUArcType Values........155
Figure 26: vguArc Parameters........157

Alphabetical Index

Affine transformations........34, 35, 162, 168, 169
Alpha masks........6, 19, 41-43
Color paint........94, 151
Double-buffered rendering........19, 21
EGL........2, 16, 18-20, 42, 148-150
EGL_OPENVG_API........19
EGL_OPENVG_BIT........2, 19
EGL_OPENVG_IMAGE........20
eglBindAPI........2, 19
EGLBoolean........19-21
EGLClientBuffer........20
EGLConfig........2, 18-20, 42
EGLContext........19, 20
eglCreateContext........2, 19
eglCreatePbufferFromClientBuffer........2, 20
eglCreateWindowSurface........2, 19, 20
eglDestroyContext........2, 20
EGLDisplay........19-21, 148
EGLenum........19, 20
eglGetCurrentContext........2, 20
EGLint........19, 20
eglMakeCurrent........2, 20
EGLSurface........19-21
eglSwapBuffers........2, 21
Ellipses....7, 8, 46, 49, 51, 58, 71, 156, 157, 162-169, 184, 186
Endianness........53, 112
Fill rule........5, 76, 77, 79, 88
Flash........1, 183
Gradient paint........4, 90, 93, 98, 99, 124, 174
Handles....16, 28, 29, 44, 58, 60, 62, 64-71, 73-75, 88, 90-92, 95, 106, 115-123, 126, 128, 134, 136, 138-141, 153-155, 157, 159
Image warping........7, 160, 161, 182
Java2D........1
khronos_float........8, 170
khronos_int16_t........7, 170
khronos_int32_t........8, 170
khronos_int8_t........7, 170
khronos_types.h........7, 8, 170
khronos_uint32_t........8, 170
khronos_uint8_t........7, 170
NativeWindowType........20
OpenGL........2, 10, 16, 33, 36, 147
OPENVG_VERSION_1_0........1, 7, 170
openvg.h........7, 147, 148, 170, 181
Pattern paint.....6, 16, 24, 80, 82-84, 86, 89, 90, 105, 106, 108, 115, 124, 125, 179
PostScript........1, 183
Premultiplied alpha.....9, 12, 13, 23, 45, 90, 94, 95, 100-

103, 106, 107, 111, 112, 118, 123, 126, 131, 132, 142, 143, 171, 184
Scaling.........................11, 13, 34, 35, 79, 86, 87, 162, 163
Scissoring..................6, 17, 23, 41, 45, 123, 127, 128, 171
Shearing...34, 35, 79, 86, 87
Single-buffered rendering..19
Smooth segments..48, 53, 184
Stops..100-104, 184
Surface coordinates..........................5, 33-35, 86-90, 122
SVG..1, 53, 86, 143, 163, 183
Tiling of images........5, 14, 22, 26, 93, 106-108, 135-139, 171, 175, 180
Transformations.....5, 6, 16, 17, 30, 32-36, 38-40, 68, 74, 79, 86, 87, 90, 122-125, 150, 159-162, 164, 165, 168, 169, 184
Translation......................33-35, 39, 40, 86, 135, 137, 168
User coordinates...5, 33, 70, 79, 80
VG_A_8.......................................110, 112, 113, 140, 175
VG_ABSOLUTE......................................54, 55, 172, 173
VG_ALPHA..................................26, 132, 133, 141, 176
VG_BAD_HANDLE_ERROR..18, 28, 29, 44, 62, 64-68, 70, 71, 73-75, 88, 91, 92, 95, 106, 116-123, 126, 128, 134, 136, 138-141, 170
VG_BLEND_ADDITIVE..................................144, 176
VG_BLEND_DARKEN.............................143, 144, 176
VG_BLEND_DST_IN...144, 176
VG_BLEND_DST_OVER..................................144, 176
VG_BLEND_LIGHTEN.............................143, 144, 176
VG_BLEND_MODE......................3, 22, 24, 26, 144, 171
VG_BLEND_MULTIPLY...........................143, 144, 176
VG_BLEND_SCREEN...............................143, 144, 176
VG_BLEND_SRC....................................24, 26, 144, 176
VG_BLEND_SRC_IN...144, 176
VG_BLEND_SRC_OVER.......................24, 26, 144, 176
VG_BLUE..26, 133, 141, 176
VG_BW_1.....110, 112, 113, 118, 119, 126, 128, 132, 140, 175
VG_CAP_BUTT..26, 82, 85, 174
VG_CAP_ROUND..82, 85, 174
VG_CAP_SQUARE................................82, 85, 89, 174
VG_CLEAR_COLOR........................22, 26, 45, 117, 171
VG_CLEAR_MASK..42-44, 172
VG_CLOSE_PATH. 51, 52, 54, 55, 66, 69, 70, 78, 83, 84, 153-156, 158, 172
VG_COLOR_RAMP_SPREAD_PAD.........93, 101, 175
VG_COLOR_RAMP_SPREAD_REFLECT......101, 175
VG_COLOR_RAMP_SPREAD_REPEAT........101, 175
VG_CUBIC_TO.......................51, 52, 55, 69, 83, 172, 173
VG_DRAW_IMAGE_MULTIPLY.6, 122, 124, 176, 184
VG_DRAW_IMAGE_NORMAL.....6, 26, 122-124, 176, 184
VG_DRAW_IMAGE_STENCIL.............6, 122, 124, 176
VG_EVEN_ODD..26, 87, 174
VG_EXTENSIONS.....................................148, 149, 177
VG_FALSE....8, 26, 41, 42, 45, 69, 84, 131, 165, 166, 170
VG_FILL_MASK..43, 44, 172
VG_FILL_PATH.............................88, 89, 91, 124, 174
VG_FILL_RULE..................................22, 26, 87, 88, 171
VG_FILTER_CHANNEL_MASK..........23, 26, 132, 171
VG_FILTER_FORMAT_LINEAR..23, 26, 131, 171, 184
VG_FILTER_FORMAT_PREMULTIPLIED.....23, 131, 132, 171
VG_GREEN..26, 133, 141, 176
VG_HARDWARE_ACCELERATED................145, 177
VG_HARDWARE_UNACCELERATED.........145, 177
VG_HLINE_TO............52, 53, 55, 68, 154, 155, 172, 173
VG_ILLEGAL_ARGUMENT_ERROR....18, 24, 25, 28, 29, 37, 38, 44, 45, 61, 66, 67, 71, 73-75, 88, 92, 115, 117-120, 122, 126-130, 134, 136, 138-141, 146, 170
VG_IMAGE_FORMAT........................116, 145, 176, 185
VG_IMAGE_FORMAT_INVALID.........................185
VG_IMAGE_FORMAT_QUERY.......................145, 176
VG_IMAGE_HEIGHT...............................116, 117, 176
VG_IMAGE_IN_USE_ERROR.......16, 18, 44, 106, 117-123, 126, 128, 134, 136, 138-141, 171
VG_IMAGE_MODE........................22, 26, 122-124, 171
VG_IMAGE_QUALITY..22, 26, 106, 113-115, 123, 171, 176
VG_IMAGE_QUALITY_BETTER.....106, 113, 114, 176
VG_IMAGE_QUALITY_FASTER......26, 106, 113, 114, 176
VG_IMAGE_QUALITY_NONANTIALIASED.....106, 113-115, 176
VG_IMAGE_WIDTH..116, 176
VG_INTERSECT_MASK...................................43, 172
VG_INVALID_HANDLE........2, 15, 44, 60, 91-93, 106, 115, 170
VG_JOIN_BEVEL...85, 174
VG_JOIN_MITER....................................26, 85, 89, 174
VG_JOIN_ROUND...85, 174
VG_LCCWARC_TO......................................52, 55, 173
VG_LCWARC_TO..52, 55, 173
VG_LINE_TO.51, 52, 55, 68-70, 72, 78, 83, 84, 153, 158, 172, 173
VG_lL_8......................................110, 113, 132, 140, 175
VG_lRGBA_8888.................110, 112, 113, 118, 126, 175
VG_lRGBA_8888_PRE........110, 112, 113, 118, 126, 175
VG_lRGBX_8888.................................110, 112, 113, 175
VG_MASKING..23, 26, 42, 171
VG_MATRIX_FILL_PAINT_TO_USER..27, 36, 87, 88, 90, 172
VG_MATRIX_IMAGE_USER_TO_SURFACE. .27, 36-38, 172
VG_MATRIX_MODE..............................22, 26, 36, 171

VG_MATRIX_PATH_USER_TO_SURFACE.....27, 36, 87-89, 172
VG_MATRIX_STROKE_PAINT_TO_USER.27, 36, 89, 90, 172
VG_MAX_COLOR_RAMP_STOPS..5, 23, 27, 101-103, 171
VG_MAX_DASH_COUNT............4, 23, 27, 85, 86, 171
VG_MAX_FLOAT...................................1, 9, 23, 27, 172
VG_MAX_IMAGE_BYTES.........5, 23, 27, 114, 115, 171
VG_MAX_IMAGE_HEIGHT.....5, 23, 27, 114, 115, 171
VG_MAX_IMAGE_PIXELS.......5, 23, 27, 114, 115, 171
VG_MAX_IMAGE_WIDTH......5, 23, 27, 114, 115, 171
VG_MAX_KERNEL_SIZE..........6, 23, 27, 134, 136, 171
VG_MAX_SCISSOR_RECTS...............3, 23, 27, 41, 171
VG_MAX_SEPARABLE_KERNEL_SIZE.......6, 23, 27, 134, 138, 171
VG_MAXINT...1, 9, 41, 85, 101, 114, 115, 134, 135, 170
VG_MAXSHORT...1, 9, 170
VG_MOVE_TO. .51-53, 55, 70, 72, 78, 83, 153-156, 158, 172, 173
VG_NO_ERROR..18, 170
VG_NON_ZERO..87, 174
VG_OUT_OF_MEMORY_ERROR................17, 18, 170
VG_PAINT_COLOR..........93-95, 97, 101-103, 174, 184
VG_PAINT_COLOR_RAMP_SPREAD_MODE.....93, 102, 103, 174
VG_PAINT_COLOR_RAMP_STOPS.93, 102, 103, 174
VG_PAINT_LINEAR_GRADIENT..............93, 98, 174
VG_PAINT_PATTERN_TILING_MODE..93, 108, 174
VG_PAINT_RADIAL_GRADIENT......93, 99, 100, 174
VG_PAINT_TYPE..............93, 94, 98-100, 106, 108, 174
VG_PAINT_TYPE_COLOR...................93, 94, 106, 174
VG_PAINT_TYPE_LINEAR_GRADIENT. .94, 98, 174
VG_PAINT_TYPE_PATTERN............94, 106, 108, 174
VG_PAINT_TYPE_RADIAL_GRADIENT.94, 99, 100, 174
VG_PATH_BIAS..63, 64, 174
VG_PATH_CAPABILITY_ALL...................59, 60, 173
VG_PATH_CAPABILITY_APPEND_FROM.....59, 60, 65, 173
VG_PATH_CAPABILITY_APPEND_TO.....59, 60, 65, 66, 153-155, 157, 159, 173
VG_PATH_CAPABILITY_ERROR.....18, 65-68, 70, 71, 73-75, 170
VG_PATH_CAPABILITY_INTERPOLATE_FROM.... 59, 60, 70, 173
VG_PATH_CAPABILITY_INTERPOLATE_TO.....59, 60, 70, 173
VG_PATH_CAPABILITY_MODIFY......59, 60, 67, 173
VG_PATH_CAPABILITY_PATH_BOUNDS.....59, 60, 74, 173
VG_PATH_CAPABILITY_PATH_LENGTH.....59, 60, 70, 71, 173
VG_PATH_CAPABILITY_PATH_TRANSFORMED_BOUNDS..59, 60, 75, 173
VG_PATH_CAPABILITY_POINT_ALONG_PATH... 59, 60, 72, 73, 173
VG_PATH_CAPABILITY_TANGENT_ALONG_PATH...59, 60, 72, 73, 173
VG_PATH_CAPABILITY_TRANSFORM_FROM..59, 60, 68, 173
VG_PATH_CAPABILITY_TRANSFORM_TO. .59, 60, 68, 173
VG_PATH_DATATYPE......53, 54, 63, 66, 67, 145, 172, 173, 176, 185
VG_PATH_DATATYPE_F.................................54, 172
VG_PATH_DATATYPE_INVALID.........................185
VG_PATH_DATATYPE_QUERY.......................145, 176
VG_PATH_DATATYPE_S_16..............................54, 172
VG_PATH_DATATYPE_S_32..............................54, 172
VG_PATH_DATATYPE_S_8................................54, 172
VG_PATH_FORMAT......3, 50, 60, 63, 66, 172, 173, 184
VG_PATH_FORMAT_STANDARD.......3, 50, 60, 172, 184
VG_PATH_NUM_COORDS.........................63, 64, 174
VG_PATH_NUM_SEGMENTS.....................63, 64, 174
VG_PATH_SCALE..63, 174
VG_PIXEL_LAYOUT..........................23, 26, 32, 171, 172
VG_PIXEL_LAYOUT_BGR_HORIZONTAL....32, 172
VG_PIXEL_LAYOUT_BGR_VERTICAL...........32, 172
VG_PIXEL_LAYOUT_RGB_HORIZONTAL....32, 172
VG_PIXEL_LAYOUT_RGB_VERTICAL...........32, 172
VG_PIXEL_LAYOUT_UNKNOWN............26, 32, 172
VG_QUAD_TO.............................52, 55, 69, 83, 172, 173
VG_RED...26, 133, 141, 176
VG_RELATIVE................................54, 55, 66, 172, 173
VG_RENDERER.......................................148, 149, 177
VG_RENDERING_QUALITY. 22, 26, 31, 171, 172, 184
VG_RENDERING_QUALITY_BETTER......26, 31, 172
VG_RENDERING_QUALITY_FASTER...........31, 172
VG_RENDERING_QUALITY_NONANTIALIASED. 31, 172
VG_RGBA_s8888_PRE.......110, 112, 113, 118, 126, 175
VG_SCCWARC_TO................52, 55, 155, 156, 158, 173
VG_SCISSOR_RECTS.............................22, 26, 41, 171
VG_SCISSORING..............................23, 26, 41, 45, 171
VG_SCUBIC_TO......................................52, 55, 69, 173
VG_SCWARC_TO...................................52, 55, 158, 173
VG_SET_MASK...43, 172
VG_sL_8.................................110, 112, 113, 132, 140, 175
VG_SQUAD_TO...52, 55, 173
VG_sRGB_565.....................................110, 112, 113, 175
VG_sRGBA_4444.......................................110, 113, 175
VG_sRGBA_5551.......................................110, 113, 175

VG_sRGBA_8888.................110, 112, 113, 118, 126, 175
VG_sRGBX_8888.................................110, 112, 113, 175
VG_STROKE_CAP_STYLE................22, 26, 85, 89, 171
VG_STROKE_DASH_PATTERN. 22, 24, 26, 85, 86, 89, 171
VG_STROKE_DASH_PHASE...22, 26, 86, 89, 171, 184
VG_STROKE_JOIN_STYLE...............22, 26, 85, 89, 171
VG_STROKE_LINE_WIDTH.............22, 26, 84, 89, 171
VG_STROKE_MITER_LIMIT............22, 26, 85, 89, 171
VG_STROKE_PATH.................................87-89, 91, 174
VG_SUBTRACT_MASK...43, 172
VG_TILE_FILL............................22, 26, 93, 107, 171, 175
VG_TILE_FILL_COLOR.........................22, 26, 107, 171
VG_TILE_PAD...107, 175
VG_TILE_REFLECT..107, 175
VG_TILE_REPEAT..107, 175
VG_TRUE...8, 41, 42, 69, 89, 93, 101, 121, 131, 165, 166, 170
VG_UNION_MASK...43, 172
VG_UNSUPPORTED_IMAGE_FORMAT_ERROR..... 18, 115, 118, 119, 127, 129, 170
VG_UNSUPPORTED_PATH_FORMAT_ERROR. .18, 61, 170
VG_VENDOR...148, 177
VG_VERSION......................................148, 149, 177, 184
vgAppendPath.......................................4, 58, 59, 65, 178
vgAppendPathData.......................4, 58, 59, 66, 178, 184
VGbitfield...1, 8, 26, 61, 62, 64, 65, 88, 91, 115, 170, 178, 179
VGBlendMode.....................................6, 14, 26, 144, 176
VGboolean.....1, 8, 14, 26, 70, 86, 93, 102, 103, 121, 140, 153, 165, 166, 170, 178-181, 184
VGbyte...1, 7, 170
VGCapStyle................................3, 4, 14, 26, 84, 85, 174
vgChildImage............................6, 16, 115, 120, 123, 179
vgClear...3, 22, 44, 45, 171, 178
vgClearImage..5, 117, 179
vgClearPath..3, 57, 60, 62, 178
vgColorMatrix...6, 133, 180
VGColorRampSpreadMode.......5, 14, 93, 101-103, 175
vgConvolve...6, 134, 135, 180
vgCopyImage..6, 121, 179
vgCopyPixels..6, 130, 179
vgCreateImage.................5, 114, 115, 123, 127, 129, 179
vgCreatePaint...4, 91, 179
vgCreatePath.............................3, 50, 57, 59-61, 178, 184
vgDestroyImage..............5, 115, 116, 120, 127, 129, 179
vgDestroyPaint...5, 91, 179
vgDestroyPath.................................3, 57, 59, 60, 62, 178
vgDrawImage...................5, 6, 35, 90, 122-124, 179, 184
vgDrawPath...............................4, 5, 80, 87-90, 123, 179
VGErrorCode......................................2, 14, 18, 171, 177
VGFillRule.....................................4, 14, 26, 87, 88, 174
vgFinish...2, 21, 90, 177
VGfloat.....1, 3, 8, 18, 23-27, 29, 37-40, 45, 61, 63, 64, 70, 73-75, 84-87, 89, 93-95, 97, 98, 100, 103, 133, 135, 137, 139, 153-156, 158, 160, 161, 170, 177-182
vgFlush...2, 21, 177
vgGaussianBlur........................6, 134, 135, 138, 139, 180
vgGet.....2, 9, 22, 23, 25, 27, 32, 41, 85, 86, 101, 114, 115, 134, 135, 177, 184
vgGetColor...5, 95, 179
vgGetError..2, 17, 18, 152, 177
vgGetImageSubData..............................5, 119, 129, 179
vgGetMatrix...2, 38, 178
vgGetPaint..5, 88, 92, 179
vgGetParameter......2, 22, 23, 27, 28, 58, 62, 92, 103, 116
vgGetParameterf.................................28, 29, 63, 64, 177
vgGetParameterfv....................................28, 29, 97, 177
vgGetParameteri......28, 29, 63, 64, 66, 67, 116, 117, 177
vgGetParameteriv...28, 29, 178
vgGetParameterVectorSize........................2, 28, 29, 177
vgGetParent...6, 121, 179
vgGetPathCapabilities..........................4, 58, 64, 65, 178
vgGetPixels...6, 127-129, 179
vgGetString....................................7, 147-149, 180, 184
vgGetVectorSize...2, 25, 177
VGHandle.....2, 15, 22, 27-29, 58, 91, 113, 170, 173, 174, 176-178
vgHardwareQuery.................................6, 145, 146, 180
VGHardwareQueryResult........6, 14, 145, 146, 177, 180
VGHardwareQueryType..........6, 14, 145, 146, 176, 180
VGImage.....5, 15, 19, 20, 27, 44, 106, 108, 113, 115-121, 123, 126-129, 133, 135, 137, 139, 140, 176, 178-180
VGImageChannel.....6, 14, 132, 133, 140, 141, 176, 180
VGImageFormat5, 14, 109, 111, 115-119, 126-129, 145, 176, 179, 184
VGImageMode.................................6, 14, 26, 122, 176
VGImageParamType................................5, 14, 116, 176
VGImageQuality...........................5, 14, 26, 113-115, 176
vgImageSubData.....................5, 117-119, 127, 179, 184
VGint..1, 8, 9, 24-27, 29, 41, 44, 45, 57, 61, 63, 64, 66, 67, 70, 73, 85, 86, 101, 114-121, 126-130, 134, 135, 137, 146, 153, 170, 177-181
vgInterpolatePath.....................4, 58, 59, 67-70, 178, 184
VGJoinStyle...4, 14, 26, 85, 174
vgLoadIdentity.....................................2, 36, 37, 87, 178
vgLoadMatrix...................................2, 3, 36, 37, 87, 178
vgLookup...6, 139, 140, 180
vgLookupSingle..6, 140, 180
vgMask...3, 42-44, 178
VGMaskOperation.................3, 14, 42-44, 172, 178, 186
VGMatrixMode......................................2, 14, 26, 36, 172
vgModifyPathCoords...........................4, 58, 59, 67, 178

vgMultMatrix...2, 36, 38, 87, 178
VGPaint...4, 15, 27, 90-92, 94, 95, 98, 100, 103, 106, 108, 116, 124, 174, 179
VGPaintMode..........................4, 14, 88, 91, 92, 174, 179
VGPaintParamType......................5, 14, 92, 93, 174, 186
vgPaintPattern.......................................5, 106, 108, 179
VGPaintType...5, 14, 93, 94, 174
VGParamType..............................2, 14, 22-25, 172, 177
VGPath.......3, 15, 19, 27, 50, 58, 60-68, 70, 73-75, 87-89, 153-156, 158, 173, 178, 179, 181, 182
VGPathAbsRel...................................3, 14, 54, 55, 57, 172
vgPathBounds.................................4, 58, 59, 73, 74, 178
VGPathCapabilities.............3, 14, 59, 60, 62, 64, 65, 173
VGPathCommand.................................3, 14, 55, 66, 173
VGPathDatatype. .3, 14, 54, 57, 60, 61, 63, 145, 172, 178
vgPathLength.................................4, 58-60, 70, 72, 178
VGPathParamType........................3, 14, 62, 63, 174, 186
VGPathSegment..............................3, 14, 54, 55, 57, 173
vgPathTransformedBounds.................4, 59, 73-75, 179
VGPixelLayout.........................2, 14, 26, 31, 32, 172, 186
vgPointAlongPath................4, 58-60, 71-73, 82, 84, 178
vgReadPixels..6, 128, 129, 179
vgRemovePathCapabilities..................4, 58, 64, 65, 178
VGRenderingQuality.................................2, 14, 31, 172
vgRotate..3, 36, 40, 178
vgScale..3, 36, 39, 178
vgSeparableConvolve....................6, 134, 136, 137, 180
vgSet.......2, 3, 22-25, 27, 31, 32, 36, 41, 42, 45, 80, 84-89, 114, 122, 144, 177
vgSetColor..5, 94, 95, 179
vgSetPaint...5, 88, 91, 93, 179
vgSetParameter.............2, 22-24, 27, 28, 90, 92, 102, 184
vgSetParameterf..27, 28, 177
vgSetParameterfv...........27, 28, 94, 95, 98-100, 103, 177
vgSetParameteri...........27, 28, 94, 98-100, 103, 108, 177
vgSetParameteriv...27, 28, 177
vgSetPixels...6, 125-127, 179
vgShear...3, 36, 39, 178
VGshort.................................1, 7, 9, 135, 137, 170, 180
VGStringID................................7, 14, 148, 149, 177, 180
VGTilingMode......5, 14, 93, 106-108, 135-139, 175, 180
vgTransformPath..............4, 58, 59, 67, 68, 87, 178, 184
vgTranslate...2, 36, 39, 178
VGU_ARC_CHORD...............................3, 157, 158, 181
VGU_ARC_OPEN...157, 181
VGU_ARC_PIE..157, 158, 181
VGU_BAD_HANDLE_ERROR. 152-155, 157, 159, 181
VGU_BAD_WARP_ERROR..............152, 160, 161, 181
VGU_ILLEGAL_ARGUMENT_ERROR.152, 154, 155, 157, 159-161, 181
VGU_NO_ERROR..152, 181
VGU_OUT_OF_MEMORY_ERROR................152, 181
VGU_PATH_CAPABILITY_ERROR.......152-155, 157, 159, 181
VGU_VERSION_1_0......................................7, 152, 181
vgu.h..7, 152, 181
vguArc...............................7, 58, 157-159, 182, 185, 187
VGUArcType......................7, 14, 157-159, 181, 182, 187
VGubyte............1, 7, 57, 66, 139, 140, 149, 170, 178, 180
vguComputeWarpQuadToQuad.................7, 161, 182
vguComputeWarpQuadToSquare................7, 160, 182
vguComputeWarpSquareToQuad................7, 160, 182
vguEllipse...7, 58, 156, 182
VGUErrorCode...7, 14, 152-156, 158, 160, 161, 181, 182
VGuint..............................1, 8, 15, 95, 140, 170, 179, 180
vguLine..7, 58, 153, 181
vguPolygon...7, 58, 153, 181
vguRect..7, 58, 154, 181
vguRoundRect....................................3, 7, 58, 155, 181
vgWritePixels.................................6, 126, 127, 179, 184
VLINE_TO...................52, 53, 55, 68, 154, 155, 172, 173