# AN14848

## i.MX RT700 OpenVG introduction

Rev. 1.0 — 10 November 2025 Application note

**Document information**

| Information | Content |
|---|---|
| Keywords | AN14848, i.MX RT700, Vector graphics, VG Lite, embedded systems, GPU acceleration |
| Abstract | This document introduces OpenVG implementation on the NXP i.MX RT700 platform, detailing its hardware-accelerated 2D vector graphics capabilities for embedded systems. It covers the i.MX RT700 graphics architecture, OpenVG API layering over VGLite, and provides practical guidance for developers on rendering, transformations, paint techniques, and image handling. |

# 1 Introduction

The i.MX RT700 features up to five computing cores designed to power smart AI-enabled edge devices such as wearables, consumer medical, smart home, and HMI devices. Its compute subsystem includes a primary Arm Cortex-M33 running at 325 MHz. Includes 2.5D GPU with vector graphics acceleration and frame buffer compression and the NXP eIQ Neutron NPU accelerating AI workloads by up to 172x and integrates up to 7.5 MB of onboard SRAM.

This application note is designed to guide developers through the practical implementation of OpenVG for embedded graphics, starting from basic rendering concepts to advanced UI designs.

It covers:

- An overview of the i.MX RT700 graphics architecture and how OpenVG is layered over the VGLite engine.
- A step-by-step tutorial on using OpenVG. This includes drawing shapes, applying transformations, strokes, gradients, and image blitting.
- Detailed code examples with explanations of parameters and expected outputs.

# 2 OpenVG

This section introduces OpenVG, a cross-platform API tailored for hardware-accelerated 2D vector graphics. It explains the advantages of vector graphics over raster images, particularly in embedded systems, and highlights OpenVG role in delivering scalable, high-performance visuals. The section also outlines the architecture of OpenVG on the NXP i.MX RT700 platform, detailing its integration with VGLite and GPU hardware to enable efficient rendering for modern user interfaces.

## 2.1 Overview

OpenVG (Open Vector Graphics) is a cross-platform API designed specifically for hardware-accelerated 2D vector graphics. Developed by the Khronos Group, OpenVG provides a standardized interface for rendering scalable graphics such as paths, shapes, gradients, and fonts—making it ideal for applications that require smooth, resolution-independent visuals.

OpenVG is well-suited for user interfaces in automotive clusters, smart appliances, medical devices, and industrial controls, where crisp visuals and efficient rendering are critical. Its architecture supports antialiasing, alpha blending, and transformations, enabling rich graphical experiences with minimal overhead.

On platforms like the NXP i.MX RT700 with integrated GPU support, OpenVG unlocks the full potential of the hardware by providing a streamlined path to high-performance 2D graphics. Developers can build sophisticated UIs and visualizations while maintaining low power consumption and high responsiveness.

## 2.2 Vector graphics vs raster images

Graphics in digital systems are typically represented in two fundamental ways: vector graphics and raster images. Understanding the distinction between these formats is essential when working with graphics APIs like OpenVG.

Vector graphics are composed of geometric primitives, such as points, lines, curves, and polygons. These elements are defined mathematically. It allows vector images to be resolution-independent: they can be scaled to any size without losing clarity or introducing pixelation. This makes them ideal for rendering crisp shapes, icons, fonts, and UI elements.

OpenVG is optimized for vector graphics, providing hardware-accelerated rendering of paths, transformations, gradients, and antialiasing. This enables smooth and efficient drawing of complex scenes, especially in embedded systems where performance and memory are constrained.

In contrast, raster images (also known as bitmap images) are made up of a grid of individual pixels, each representing a specific color. Common formats include JPEG, PNG, and BMP. Raster graphics are resolution-dependent, meaning scaling them can result in visible artifacts or blurring.

While OpenVG's primary focus is vector rendering, it also includes support for raster image manipulation. This allows developers to integrate bitmap images into vector-based scenes, apply transformations, blend modes, and even perform basic image processing. This hybrid capability is useful for applications that combine UI elements with photographic content or textures.

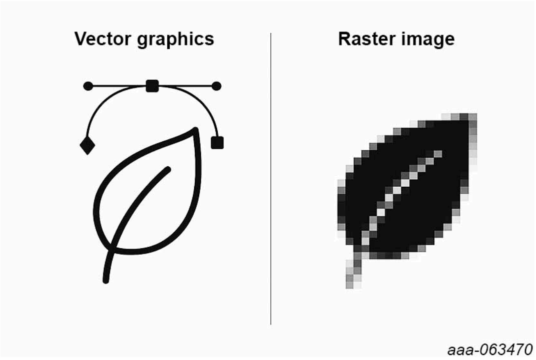


**Figure 1. Vector graphics vs raster images**

## 2.3 OpenVG architecture on i.MX RT700

The NXP i.MX RT700 features an integrated 2.5D GPU designed to accelerate vector graphics rendering. This GPU is accessed through a low-level graphics driver known as VGLite, which provides direct control over the hardware capabilities for drawing paths, shapes, and handling image data.

OpenVG sits above VGLite as a high-level API, offering a standardized and portable interface for developers. Instead of interacting directly with the GPU through VGLite, applications written using OpenVG can apply its abstraction layer to simplify development while still benefiting from the full performance of hardware acceleration. The architecture is made up of:

- Application layer: Your application code interacts with OpenVG functions to define paths, transformations, gradients, and image operations.
- OpenVG API: This layer translates high-level vector graphics commands into GPU-friendly instructions.
- VGLite driver: Acts as the bridge between OpenVG and the GPU, handling low-level rendering tasks and memory management.
- GPU Hardware: Executes the rendering operations efficiently, offloading the CPU and enabling smooth, high-performance graphics.
- EGL: is used to create and manage the rendering context for OpenVG, ensures that what OpenVG draws via VGLite is properly displayed on screen.

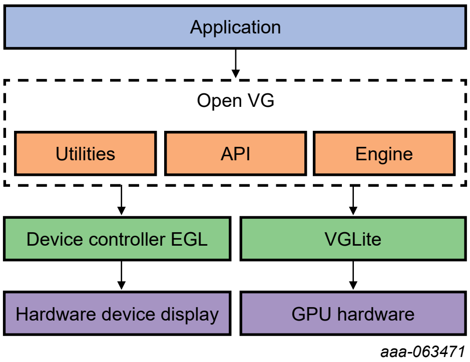


**Figure 2. OpenVG Architecture**

## 3 Basic initialization

Before rendering any graphics, the system must go through a series of initialization steps to prepare the GPU, display, and rendering context. Here is a simplified overview of the process:

1. Prepare the VGLite controller: The i.MX RT700 uses VGLite as the low-level driver to control its 2D GPU. This step ensures that the GPU is ready for rendering operations.
2. Create a native display and window: Using the framebuffer interface, a native display and window are created. They are required for EGL to bind rendering surfaces.
3. Initialize EGL and OpenVG: This step sets up the EGL environment and binds it to OpenVG. It involves several substeps:
   a. Get and initialize the EGL display
   b. Bind the OpenVG API
   c. Choose EGL configuration
   d. Create EGL surface and context
4. Set OpenVG Defaults: Basic rendering settings are applied to ensure consistent behavior.
5. Clear the screen and verify: A simple clear and buffer swap is performed to verify that everything works.

```
//Step 1: Prepare the VGLite Controller
status = BOARD_PrepareVGLiteController();

//Step 2: Create Native Display and Window
fbDpy = fbGetDisplay(NULL);
fbWin = fbCreateWindow(fbDpy, 0, 0, DEMO_WINDOW_WIDTH, DEMO_WINDOW_HEIGHT);

//Step 3: Initialize EGL and OpenVG
initialize_openvg(fbWin, &eglDisplay, &eglSurface, &eglContext);
eglDisplay = eglGetDisplay(EGL_DEFAULT_DISPLAY);
eglInitialize(eglDisplay, NULL, NULL);
eglBindAPI(EGL_OPENVG_API);
choose_config(eglDisplay, &eglConfig)
eglSurface = eglCreateWindowSurface(...);
eglContext = eglCreateContext(...);
eglMakeCurrent(eglDisplay, eglSurface, eglSurface, eglContext);

//Step 4: Set OpenVG Defaults
vgSeti(VG_MATRIX_MODE, VG_MATRIX_PATH_USER_TO_SURFACE);
vgLoadIdentity();
vgSeti(VG_BLEND_MODE, VG_BLEND_SRC_OVER);
vgSeti(VG_FILL_RULE, VG_NON_ZERO);
vgSetfv(VG_CLEAR_COLOR, 4, (VGfloat[]){1.0f, 1.0f, 1.0f, 1.0f})
```

```
//Step 5: Clear the Screen and Verify
vgClear(0, 0, DEMO_WINDOW_WIDTH, DEMO_WINDOW_HEIGHT);
eglSwapBuffers(eglDisplay, eglSurface);
```

This setup ensures that the GPU is ready, the rendering context is active, and the screen is prepared for drawing operations.

## 4 First drawing and path operations.

This section provides an overview of shape construction in OpenVG, covering manual path creation, drawing commands, and the use of VGU utilities for simplified geometry generation. It explains how paths define vector shapes through segment commands, demonstrates rectangle drawing using basic instructions, and introduces VGU functions for quickly rendering common shapes like lines, circles, and arcs.

### 4.1 Path overview

This section introduces how to draw basic shapes using OpenVG. It covers path creation, drawing commands, and the use of VGU utilities.

In OpenVG, a path is a fundamental building block for drawing shapes. It represents a sequence of drawing commands and coordinate points that define geometric figures like lines, curves, and shapes.

Think of a path as a vector blueprint—it tells the GPU how to draw something, but not what color or how thick. Those visual properties are handled separately by paints and styles.

There are several Path commands. The following table describes each segment command and the results of the operations.

**Table 1. Segment commands**

| VGPathSegment | Coordinates | Description |
|---|---|---|
| VG_CLOSE_PATH | none | Closes the current subpath by connecting the last point to the starting point. |
| VG_MOVE_TO | x0,y0 | Moves the "pen" to a new position without drawing. Starts a new subpath. |
| VG_LINE_TO | | Draws a straight line from the current point to (x0, y0). |
| VG_HLINE_TO | x0 | Draws a horizontal line to x0 (y remains unchanged). |
| VG_VLINE_TO | y0 | Draws a vertical line to y0 (x remains unchanged). |
| VG_QUAD_TO | x0,y0,x1,y1 | Draws a quadratic Bézier curve using (x0, y0) as control and (x1, y1) as end. |
| VG_CUBIC_TO | x0,y0,x1,y1, x2, y2 | Draws a cubic Bézier curve with two control points and an end point |
| VG_SQUAD_TO | x1,y1 | Draws a smooth quadratic Bézier curve using the previous control point. |
| VG_SCUBIC_TO | x1,y1, x2, y2 | Draws a smooth cubic Bézier curve using the previous control point. |
| VG_SCCWARC_TO | rh,rv,rot,x0,y0 | Draws a small counterclockwise elliptical arc to (x0, y0). |
| VG_SCWARC_TO | | Draws a small clockwise elliptical arc to (x0, y0). |
| VG_LCCWARC_TO | | Draws a large counterclockwise elliptical arc to (x0, y0). |
| VG_LCWARC_TO | | Draws a large clockwise elliptical arc to (x0, y0). |

### 4.2 Simple rectangle

To draw a rectangle manually, define a path with commands and coordinates.

```
VGPath rectPath = vgCreatePath(
    VG_PATH_FORMAT_STANDARD,
    VG_PATH_DATATYPE_F,
    1.0f, 0.0f,
    segmentHint, coordHint,
    VG_PATH_CAPABILITY_ALL
);

VGubyte segments[] = {
    VG_MOVE_TO_ABS,
    VG_LINE_TO_ABS,
    VG_LINE_TO_ABS,
    VG_LINE_TO_ABS,
    VG_CLOSE_PATH
};

VGfloat coords[] = {
    50, 50,          // Move to top-left
    150, 50,         // Line to top-right
    150, 130,        // Line to bottom-right
    50, 130          // Line to bottom-left
    // Close path connects back to (50, 50)
};
```

The segments define what we are going to do (move, draw, close) and the coordinates define where to top it (x, y positions). Once the path is defined, it is possible to draw it:

```
vgAppendPathData(rectPath, 5, segments, coords);
vgSetPaint(paint, VG_FILL_PATH);
vgDrawPath(rectPath, VG_FILL_PATH);
```

Figure 3 illustrates how a rectangle is constructed using OpenVG path commands. It begins with a **MoveTo** command to set the starting point, followed by three LineTo segments to define the edges. The shape is completed with a **ClosePath** command that connects the last point back to the start. Each vertex is labeled with its coordinates and the corresponding drawing instruction.

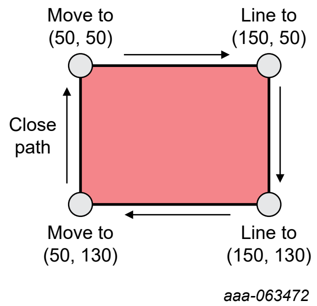


**Figure 3. Rectangle draw**

 

### 4.3 Vector graphics utilities VGU

VGU is a companion utility library for OpenVG that simplifies the creation of common geometric shapes. While OpenVG provides the core API for rendering paths and graphics, VGU offers helper functions to generate predefined shapes without manually specifying path segments and coordinates.

This is especially useful for developers who want to quickly draw basic shapes like rectangles, circles, and ellipses, or perform operations like arc creation and curve fitting, without constructing paths from scratch. VGU provides functions to create the following shapes:

**Table 2. Functions**

| Function | Description |
|---|---|
| vguLine() | Creates a straight line between two points. |
| vguRect() | Generates a rectangle given position and size. |
| vguRoundRect() | Similar to vguRect() but with rounded corners. |
| vguEllipse() | Creates an ellipse or circle based on width and height. |
| vguArc() | Generates an elliptical arc segment with specified angles and radio. |

The following peace of code is a representative example of how you can use the library to draw a simple circle:

```
VGPath circlePath = vgCreatePath(...);
vguEllipse(circlePath, 100, 100, 80, 80); // Circle with radius 40
vgSetPaint(fillPaint, VG_FILL_PATH);
vgDrawPath(circlePath, VG_FILL_PATH);
```

## 5 Paint and stroke fundamentals

OpenVG separates how shapes are drawn (geometry) from how they look (style). This is achieved through paint objects and stroke properties that define the visual appearance of paths.

### 5.1 Paint objects

A VGPaint object defines the color or gradient used to fill or stroke a path. It is possible to create multiple paint objects and assign them to different drawing operations.

```
VGPaint fillPaint = vgCreatePaint();
vgSetParameteri(fillPaint, VG_PAINT_TYPE, VG_PAINT_TYPE_COLOR);
VGfloat color[] = {1.0f, 0.0f, 0.0f, 1.0f}; // RGBA: Red
vgSetParameterfv(fillPaint, VG_PAINT_COLOR, 4, color);
```

`VG_PAINT_TYPE_COLOR`: Specifies solid color. RGBA values range from 0.0 to 1.0. It is possible to create gradient paints (linear or radial), this section focuses on solid colors. Gradients are covered in the sections below.

### 5.2 Fill vs stroke

OpenVG allows users to fill the interior of a path or stroke its outline. These are controlled using:

- `VG_FILL_PATH`: Fills the shape.
- `VG_STROKE_PATH`: Draws the outline.

```
vgSetPaint(fillPaint, VG_FILL_PATH);    // Apply fill paint
```

```
vgSetPaint(strokePaint, VG_STROKE_PATH); // Apply stroke paint
```

The same paint object can be used for both fill and stroke, or separate ones for different styles. Figure 4 demonstrates the visual differences between OpenVG fill and stroke rendering modes. The first rectangle uses only VG_FILL_PATH, showing a solid interior. The second uses `VG_STROKE_PATH`, outlining the shape without filling it. The third combines both, displaying a filled shape with a visible border. This highlights how paint assignment affects path rendering.

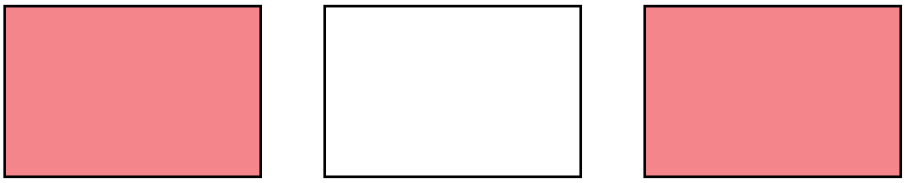


**Figure 4. Fill vs stroke**

## 5.3 Stroke properties

Stroke settings define how the outline of a path is rendered, including end cap style, line join style, width, dash pattern.

The end cap style includes:

- `VG_CAP_BUTT`: Each segment with a line is perpendicular to the tangent at each endpoint.
- `VG_CAP_ROUND`: Append a semicircle with a diameter equal to the line width centered on each endpoint.
- `VG_CAP_SQUARE`: Append a rectangle at each endpoint, whose vertical length is equal to the line width, and the parallel length is equal to half the line width.

As shown in Figure 5, the red lines mean the real path data, the black and the gray are the drawn strokes where the gray is added by end cap styles (the gray color is used to highlight, these areas have the same color as the real path).


**Figure 5. Cap styles**

The line join style determines the style of the intersection point of two lines, including:

- `VG_JOIN_MITER:` Connect the two segments by extending their outer edges until they meet. If this join style is selected.
- `VG_JOIN_ROUND`: Append a wedge-shaped portion of a circle, centered at the intersection point, whose diameter is equal to the line width.
- `VG_JOIN_BEVEL`: Connect two points of the outer border of two segments with a straight line.

As shown in Figure 6, the red lines mean the true path, the black and gray lines are the drawn strokes where the gray is added by line join styles.


**Figure 6. Join Styles**

It is possible to create a dash pattern that defines alternating segments of "on" and "off" strokes, for example, dashed or dotted lines. They can be specified by using an array of floats, where each value represents the length of a dash or gap. The following examples create a patter with 10 pixels on and 5 off as shown in Figure 7.

```
VGfloat dashPattern[] = {10.0f, 5.0f}; // 10 pixels on, 5 pixels off
```

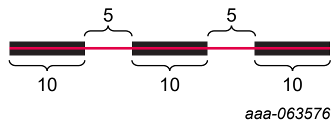


**Figure 7. Dash Pattern**

On the software related to this document, a function to generate multiple dash patterns is included.

```
// Different dash patterns
    VGfloat solid[] = {};                                           // Solid line
    VGfloat dots[] = {2.0f, 8.0f};                                 // Small dots
    VGfloat dashes[] = {15.0f, 10.0f};                             // Regular dashes
    VGfloat dotDash[] = {3.0f, 5.0f, 15.0f, 5.0f};               // Dot-dash
    VGfloat complex[] = {10.0f, 3.0f, 3.0f, 3.0f, 3.0f, 8.0f}; // Complex
 pattern

    struct {
        VGfloat* pattern;
        int count;
        const char* name;
        float yOffset;
    } patterns[] = {
        {NULL, 0, "Solid", 400},
        {dots, 2, "Dots", 350},
        {dashes, 2, "Dashes", 300},
        {dotDash, 4, "Dot-Dash", 250},
        {complex, 6, "Complex", 200}
    };

    for (int i = 0; i < 5; i++)
    {
        vgLoadIdentity();
        vgTranslate(0, patterns[i].yOffset);

        if (patterns[i].pattern == NULL)
        {
            vgSetfv(VG_STROKE_DASH_PATTERN, 0, NULL); // Solid
        }
        else
        {
            apply_advanced_dash_pattern(patterns[i].name, patterns[i].pattern,
  patterns[i].count);
```

```
        }

        vgDrawPath(testLine, VG_STROKE_PATH);
    }
```

The expected result is shown on Figure 8. From top to bottom, it displays a solid line, small dotted line, regular dashed line, a dot-dash combination, and a complex repeating pattern. Each style is defined by a sequence of visible and invisible segments, allowing expressive and customizable outlines in vector graphics.

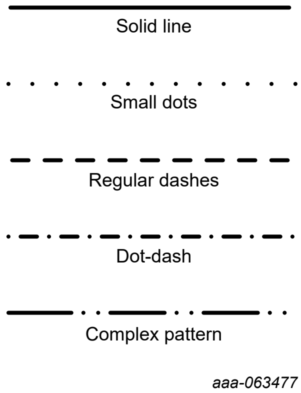


**Figure 8. Dash pattern example**

## 6 Matrix transformation

OpenVG uses transformation matrices (3x3) to manipulate how graphics are positioned, scaled, and rotated on the screen. These transformations are essential for building dynamic and responsive user interfaces.

OpenVG operates with two main coordinate systems:

- User coordinates: The logical space where you shapes and paths are defined.
- Surface coordinates: The physical pixel space of the display.

There are three main functions to apply transformation to a current matrix: translation *vgTranslate(),* scaling *vgScale(),* rotation *vgRotate().* Transformations are applied in reverse order of how they are written. For example:

```
vgTranslate(100, 100);
vgRotate(30);
vgScale(2, 2);
```

This means that the object is first scaled, then rotated, then translated.

On the example reference there is an animation where a "house" shape rotates around a central point in a circular orbit, while also spinning on its own axis. This demonstrates how matrix transformations can be combined to create dynamic motion and rotation effects.

1. The loop runs from 0 degrees to 360 degrees in steps of 10, simulating 36 frames of animation:

   ```
   for (int frame = 0; frame < 360; frame += 10) // 36 frames
   ```

2. Each frame starts by clearing the screen to prepare for fresh drawing:

   ```
   vgClear(0, 0, DEMO_WINDOW_WIDTH, DEMO_WINDOW_HEIGHT);
   ```

3. A dashed circle is drawn to visualize the orbit:

```
draw_circle(centerX, centerY, radius, NULL, orbitColor, 1.0f, dashPattern,
 2);
```

4. The house is moved and rotated using a series of transformations:
   a. The first vgRotate(frame) makes the house orbit.
   b. The second vgRotate(frame * 3) makes the house spin as it orbits.
   c. The final vgTranslate(-20, -25) centers the shape visually.

```
vgLoadIdentity();                    // Reset matrix
vgTranslate(centerX, centerY);       // Move origin to center
vgRotate(frame);                     // Rotate around center (orbit)
vgTranslate(radius, 0);              // Move outward along orbit
vgRotate(frame * 3);                 // Spin the house itself
vgTranslate(-20, -25);               // Center the house shape
vgDrawPath(house, VG_FILL_PATH);     // Draw the house
```

5. The frame is rendered to the screen and a short delay (~50 ms) creates a smooth frame rate (~20 FPS):

```
eglSwapBuffers(eglDisplay, eglSurface);
vTaskDelay(pdMS_TO_TICKS(50));
```

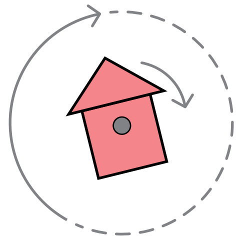


**Figure 9. House rotating**

## 7 Fill rules

When rendering complex or self-intersecting shapes, OpenVG uses fill rules to determine which areas of a path should be filled. These rules define how the interior of a shape is calculated based on the direction and overlap of its segments.

Non-zero winding rule (`VG_NON_ZERO`):

This rule counts how many times a path winds around a point. If the result is non-zero, the point is considered inside the shape and gets filled.

- Direction matters: Clockwise adds +1, counterclockwise subtracts -1.
- Useful for complex shapes with overlapping segments.

Even-Odd Rule (`VG_EVEN_ODD`):

This rule counts how many times a path crosses a line drawn from a point. If the count is odd, the point is inside; if even, it is outside.

- Direction does not matter.
- Simpler and often used for star-like or nested shapes.

The `demo_fill_rules()` function demonstrates both rules using a self-intersecting star shape:

```
VGPath path1 = create_intersecting_path(150, 300, 80);
VGPath path2 = create_intersecting_path(350, 300, 80);

// Non-Zero fill rule
vgSeti(VG_FILL_RULE, VG_NON_ZERO);
vgDrawPath(path1, VG_FILL_PATH | VG_STROKE_PATH);

// Even-Odd fill rule
vgSeti(VG_FILL_RULE, VG_EVEN_ODD);
vgDrawPath(path2, VG_FILL_PATH | VG_STROKE_PATH);
```

Figure 10 compares the two fill rules supported by OpenVG: Non-zero winding and even-odd. The left shape uses the non-zero rule, filling all regions enclosed by overlapping paths. The right shape uses the even-odd rule, which creates hollow areas based on the number of path crossings.

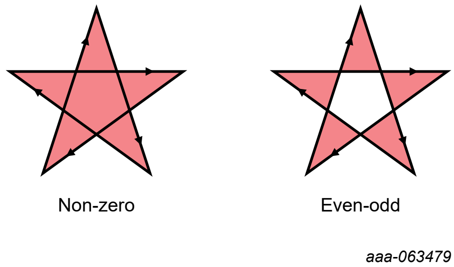


**Figure 10. Fill rules**

# 8 Gradients and advanced paint

OpenVG supports advanced paint types beyond solid colors, allowing for smooth transitions and textured fills using gradients and patterns. These techniques enhance visual richness and realism in embedded UIs.

## 8.1 Linear gradient

A linear gradient is a smooth transition between two or more colors along a straight line. It is commonly used to simulate lighting, depth, or visual transitions in UI elements. In OpenVG, gradients are applied using a VGPaint object configured with the `VG_PAINT_TYPE_LINEAR_GRADIENT` type. Follow this step to create a linear gradient.

1. Create the paint object: tells OpenVG that the paint will be a linear gradient rather than a solid color or pattern using the `VG_PAINT_TYPE_LINEAR_GRADIENT` definition.

   ```
   VGPaint gradientPaint = vgCreatePaint();
   vgSetParameteri(gradientPaint, VG_PAINT_TYPE,
   VG_PAINT_TYPE_LINEAR_GRADIENT);
   ```

2. Define the gradient vector: The gradient vector determines the direction and length of the gradient.

   ```
   VGfloat gradientVector[] = {x0, y0, x1, y1};
   vgSetParameterfv(gradientPaint, VG_PAINT_LINEAR_GRADIENT, 4, gradientVector);
   ```

   a. x0, y0: The start point of the gradient.
   b. x1, y1: The end point of the gradient.

The gradient will interpolate colors from the start to the end point. For example:

- Horizontal gradient: {0, 0, 200, 0}
- Vertical gradient: {0, 0, 0, 200}
- Diagonal gradient: {0, 0, 200, 200}

3. Set color stops: color stops define the colors and their positions along the gradient vector.

```
VGfloat colorStops[] = {
    0.0f, 1.0f, 0.0f, 0.0f, 1.0f, // Red at offset 0.0
    1.0f, 0.0f, 0.0f, 1.0f, 1.0f  // Blue at offset 1.0
};
vgSetParameterfv(gradientPaint, VG_PAINT_COLOR_RAMP_STOPS, 10, colorStops);
```

Each stop consists of:

- Offset: A value between 0.0 and 1.0 indicating position along the gradient.
- RGBA: Red, Green, Blue, Alpha values (0.0 to 1.0).

It is possible to add more stops for complex gradients:

```
VGfloat multiStops[] = {
    0.0f, 1.0f, 0.0f, 0.0f, 1.0f, // Red
    0.5f, 0.0f, 1.0f, 0.0f, 1.0f, // Green
    1.0f, 0.0f, 0.0f, 1.0f, 1.0f  // Blue
};
```

4. Apply the gradient to a path: once the gradient is configured, you can apply it to a path:

```
VGfloat colorStops[] = {
    0.0f, 1.0f, 0.0f, 0.0f, 1.0f, // Red at offset 0.0
    1.0f, 0.0f, 0.0f, 1.0f, 1.0f  // Blue at offset 1.0
};
vgSetParameterfv(gradientPaint, VG_PAINT_COLOR_RAMP_STOPS, 10, colorStops);
```

This fills the path using the gradient defined by the vector and color stops. Figure 11 shows three rectangles filled with different types of linear gradients:

- Left: Horizontal gradient from red to blue
- Center: Vertical gradient from red to blue
- Right: Diagonal gradient transitioning from red to green to blue

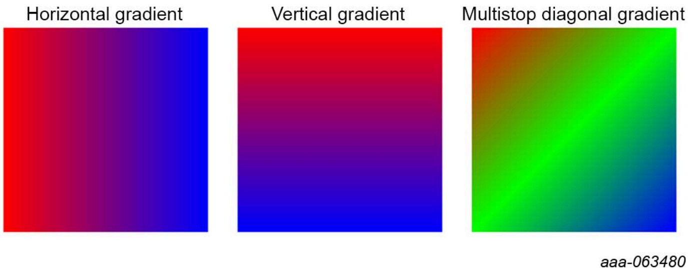


**Figure 11. Linear gradient**

## 8.2 Radial gradient

A radial gradient is a smooth transition between colors that radiates outward from a central point. Unlike linear gradients that interpolate colors along a straight line, radial gradients simulate depth, lighting, or focus effects by blending colors in a circular or elliptical pattern.

This type of gradient is useful for:

- Simulating light sources (for example, glows, highlights)
- Creating circular UI elements with depth
- Emphasizing focal points in a design

Radial gradients are applied using a VGPaint object configured with the `VG_PAINT_TYPE_RADIAL_GRADIENT` type. The gradient is defined by a center point, a focal point, and a radius.

1. Create the paint object: tells OpenVG that the paint will be a radial using the `VG_PAINT_TYPE_RADIAL_GRADIENT` definition:

```
VGPaint paint = vgCreatePaint();
vgSetParameteri(paint, VG_PAINT_TYPE, VG_PAINT_TYPE_RADIAL_GRADIENT);
```

2. Defines the radial gradient parameters:

```
VGfloat radialGradient[5] = {
    200.0f, 200.0f,   // Center of the gradient (cx, cy)
    200.0f, 200.0f,   // Focal point (fx, fy) — same as center for symmetric
 gradient
    200.0f            // Radius — covers the entire circle area
};
vgSetParameterfv(paint, VG_PAINT_RADIAL_GRADIENT, 5, radialGradient);
```

3. Define color ramp stops: Color ramp stops specify how colors transition along the gradient. Here is an example with three colors: red, green, and blue:

```
VGfloat rampStops[] = {
    0.0f, 1.0f, 0.0f, 0.0f, 1.0f, // Red at position 0.0
    0.5f, 0.0f, 1.0f, 0.0f, 1.0f, // Green at position 0.5
    1.0f, 0.0f, 0.0f, 1.0f, 1.0f  // Blue at position 1.0
};
vgSetParameterfv(paint, VG_PAINT_COLOR_RAMP_STOPS, 15, rampStops);
```

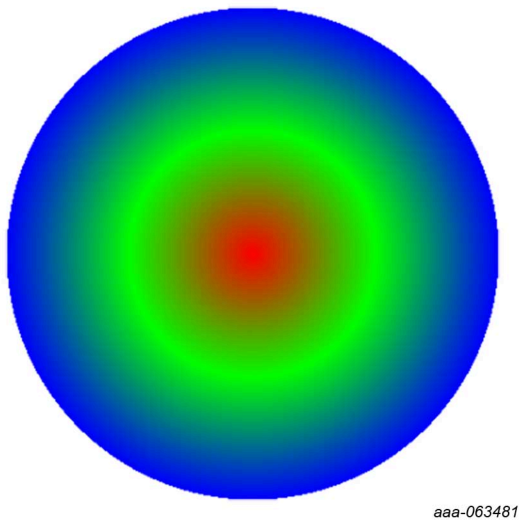


**Figure 12. Radial gradient**

## 9 Image handling

In OpenVG, images are managed using the VGImage type. These objects store pixel data and can be rendered, transformed, or used as paint sources. OpenVG supports several formats:

- `VG_sRGB_565`: 16-bit RGB, no alpha

- `VG_sRGBA_8888`: 32-bit RGBA
- `VG_sRGBA_8888_PRE`: Premultiplied alpha
- `VG_lRGBx_8888`: Luminance-based

We are going to use the NXP logo to show the image capabilities:

1. Load the Image into VGImage: Assuming the image is stored in memory or loaded from flash, it is possible to use `vgImageSubData()` to populate a VGImage object:

```
VGImage nxpLogo = vgCreateImage(
    VG_sRGBA_8888,    // Format with alpha
    128, 64,          // Width and height of the logo
    VG_IMAGE_QUALITY_BETTER
);

// Assuming 'logoData' is a pointer to the raw RGBA pixel data
vgImageSubData(
    nxpLogo,
    logoData,
    128 * 4,          // Stride: width * bytes per pixel
    VG_sRGBA_8888,
    0, 0,
    128, 64
);
```

2. Position and scale the logo: To center and scale the logo on screen, use:

```
vgSeti(VG_MATRIX_MODE, VG_MATRIX_IMAGE_TO_USER);
VGfloat matrix[9] = {
    1.5f, 0, 0,
    0, 1.5f, 0,
    100.0f, 50.0f, 1
};
vgLoadMatrix(matrix);
```

This scales the logo by 1.5× and positions it at (100, 50) in user space.

3. To draw the image, use:

```
vgDrawImage(nxpLogo);
```

aaa-063482

**Figure 13. NXP logo**

## 9.1 Image filters

Image filters in OpenVG allow users to apply pixel-level transformations to VGImage objects. Filters operate on image data and produce a new image as an output that can then be drawn or used as a paint source. Several built-in image filters are available via the `vgColorMatrix()` and `vgConvolve()` functions. Common filters include:

**Color matrix filters (vgColorMatrix)**

Used to apply transformations like:

- Grayscale
- Sepia
- Color inversion
- Brightness/contrast adjustment

**Convolution filters (vgConvolve)**

Used for:

- Blurring
- Sharpening
- Edge detection

These filters use a kernel matrix to process pixel neighborhoods.

The following instructions show how to apply the color matrix filter (Grayscale) and the convolution kernel (blur) to the NXP logo.

Grayscale (color matrix):

```
VGfloat grayscaleMatrix[20] = {
    0.299f, 0.587f, 0.114f, 0, 0,
    0.299f, 0.587f, 0.114f, 0, 0,
    0.299f, 0.587f, 0.114f, 0, 0,
    0,      0,      0,      1, 0
};
vgColorMatrix(outputImage, inputImage, grayscaleMatrix);
```

Blur (convolution filter):

```
 VGshort kernel[9] = {
    1, 2, 1,
    2, 4, 2,
    1, 2, 1
};
vgConvolve(outputImage, inputImage, 3, 3, 1, 1, kernel, 16, 0, VG_TILE_PAD);
```

Figure 14 shows the results of each filter applied to the NXP logo.


**Figure 14. Image filters**

## 9.2 Image as paint

It is possible to use an image as a paint source, meaning the image is tiled or stretched to fill shapes instead of using solid colors or gradients. This is useful for:

- Creating textured fills
- Repeating patterns (for example, fabric, tiles)
- Simulating materials or branding elements

This technique is like using a texture in OpenGL, but OpenVG handles it through the pattern paint type.

Figure 15 shows an example how the NXP logo fills a rectangle.

```
VGPath rect = createRectanglePath(0, 0, 300, 200); // Custom function to create
 a rectangle path

vgSeti(VG_MATRIX_MODE, VG_MATRIX_PAINT_TO_USER);
VGfloat matrix[9] = {
    1.0f, 0, 0,
    0, 1.0f, 0,
    0, 0, 1
};
vgLoadMatrix(matrix);

vgSetPaint(paint, VG_FILL_PATH);
vgDrawPath(rect, VG_FILL_PATH);
```

aaa-063484

**Figure 15. Image as paint**

## 10 Conclusion

Throughout this application note, a comprehensive exploration of OpenVG on the NXP i.MX RT700 is presented, highlighting its capabilities for efficient, hardware-accelerated 2D vector graphics. Beginning with foundational concepts and advancing through sophisticated rendering techniques, users are now equipped with a solid framework for developing rich graphical interfaces on embedded systems. The topics covered include:

- OpenVG basics: An overview of how OpenVG differs from raster graphics, its architecture on the i.MX RT700, and the initialization of the graphics pipeline.
- Path drawing: Techniques for creating and manipulating vector paths, including rectangles and curves, using both manual commands and VGU utilities.
- Paint and stroke: Methods for applying fills and strokes with customizable properties to enable precise visual styling.
- Transformations: Application of matrix operations to dynamically scale, rotate, and position graphics—essential for responsive UI design.
- Fill rules: Implementation of even-odd and non-zero winding rules to control the rendering of complex shapes.
- Advanced paint techniques: Use of linear and radial gradients to simulate lighting and depth.
- Image handling: Creation and rendering of images, application of transformations, and use of clipping to manage visibility. Additional techniques include applying image filters such as grayscale and blur, and utilizing images as paint sources for textured fills.

## 11 References

*i.MX RT700 Reference Manual* (document IMXRT700RM)

*i.MX RT VGLite API Reference Manual* (document IMXRTVGLITEAPIRM)

*i.MX RT1170 Heterogeneous Graphics Pipeline* (document AN13075)

*Porting VGLite Driver for Bare Metal or Single Task* (document AN13778)

OpenVG 1.1 Lite Specification

## 12 Note about the source code in the document

Example code shown in this document has the following copyright and BSD-3-Clause license:



## 13 Revision history

**Table 3. Revision history**

| Document ID | Release date | Description |
|---|---|---|
| AN14848 v.1.0 | 10 November 2025 | Initial public release |

# Legal information

## Definitions

**Draft** — A draft status on a document indicates that the content is still under internal review and subject to formal approval, which may result in modifications or additions. NXP Semiconductors does not give any representations or warranties as to the accuracy or completeness of information included in a draft version of a document and shall have no liability for the consequences of use of such information.

## Disclaimers

**Limited warranty and liability** — Information in this document is believed to be accurate and reliable. However, NXP Semiconductors does not give any representations or warranties, expressed or implied, as to the accuracy or completeness of such information and shall have no liability for the consequences of use of such information. NXP Semiconductors takes no responsibility for the content in this document if provided by an information source outside of NXP Semiconductors.

In no event shall NXP Semiconductors be liable for any indirect, incidental, punitive, special or consequential damages (including - without limitation - lost profits, lost savings, business interruption, costs related to the removal or replacement of any products or rework charges) whether or not such damages are based on tort (including negligence), warranty, breach of contract or any other legal theory.

Notwithstanding any damages that customer might incur for any reason whatsoever, NXP Semiconductors' aggregate and cumulative liability towards customer for the products described herein shall be limited in accordance with the Terms and conditions of commercial sale of NXP Semiconductors.

**Right to make changes** — NXP Semiconductors reserves the right to make changes to information published in this document, including without limitation specifications and product descriptions, at any time and without notice. This document supersedes and replaces all information supplied prior to the publication hereof.

**Suitability for use** — NXP Semiconductors products are not designed, authorized or warranted to be suitable for use in life support, life-critical or safety-critical systems or equipment, nor in applications where failure or malfunction of an NXP Semiconductors product can reasonably be expected to result in personal injury, death or severe property or environmental damage. NXP Semiconductors and its suppliers accept no liability for inclusion and/or use of NXP Semiconductors products in such equipment or applications and therefore such inclusion and/or use is at the customer's own risk.

**Applications** — Applications that are described herein for any of these products are for illustrative purposes only. NXP Semiconductors makes no representation or warranty that such applications will be suitable for the specified use without further testing or modification.

Customers are responsible for the design and operation of their applications and products using NXP Semiconductors products, and NXP Semiconductors accepts no liability for any assistance with applications or customer product design. It is customer's sole responsibility to determine whether the NXP Semiconductors product is suitable and fit for the customer's applications and products planned, as well as for the planned application and use of customer's third party customer(s). Customers should provide appropriate design and operating safeguards to minimize the risks associated with their applications and products.

NXP Semiconductors does not accept any liability related to any default, damage, costs or problem which is based on any weakness or default in the customer's applications or products, or the application or use by customer's third party customer(s). Customer is responsible for doing all necessary testing for the customer's applications and products using NXP Semiconductors products in order to avoid a default of the applications and the products or of the application or use by customer's third party customer(s). NXP does not accept any liability in this respect.

**Terms and conditions of commercial sale** — NXP Semiconductors products are sold subject to the general terms and conditions of commercial sale, as published at https://www.nxp.com/profile/terms, unless otherwise agreed in a valid written individual agreement. In case an individual agreement is concluded only the terms and conditions of the respective agreement shall apply. NXP Semiconductors hereby expressly objects to applying the customer's general terms and conditions with regard to the purchase of NXP Semiconductors products by customer.

**Export control** — This document as well as the item(s) described herein may be subject to export control regulations. Export might require a prior authorization from competent authorities.

**Suitability for use in non-automotive qualified products** — Unless this document expressly states that this specific NXP Semiconductors product is automotive qualified, the product is not suitable for automotive use. It is neither qualified nor tested in accordance with automotive testing or application requirements. NXP Semiconductors accepts no liability for inclusion and/or use of non-automotive qualified products in automotive equipment or applications.

In the event that customer uses the product for design-in and use in automotive applications to automotive specifications and standards, customer (a) shall use the product without NXP Semiconductors' warranty of the product for such automotive applications, use and specifications, and (b) whenever customer uses the product for automotive applications beyond NXP Semiconductors' specifications such use shall be solely at customer's own risk, and (c) customer fully indemnifies NXP Semiconductors for any liability, damages or failed product claims resulting from customer design and use of the product for automotive applications beyond NXP Semiconductors' standard warranty and NXP Semiconductors' product specifications.

**HTML publications** — An HTML version, if available, of this document is provided as a courtesy. Definitive information is contained in the applicable document in PDF format. If there is a discrepancy between the HTML document and the PDF document, the PDF document has priority.

**Translations** — A non-English (translated) version of a document, including the legal information in that document, is for reference only. The English version shall prevail in case of any discrepancy between the translated and English versions.

**Security** — Customer understands that all NXP products may be subject to unidentified vulnerabilities or may support established security standards or specifications with known limitations. Customer is responsible for the design and operation of its applications and products throughout their lifecycles to reduce the effect of these vulnerabilities on customer's applications and products. Customer's responsibility also extends to other open and/or proprietary technologies supported by NXP products for use in customer's applications. NXP accepts no liability for any vulnerability. Customer should regularly check security updates from NXP and follow up appropriately.

Customer shall select products with security features that best meet rules, regulations, and standards of the intended application and make the ultimate design decisions regarding its products and is solely responsible for compliance with all legal, regulatory, and security related requirements concerning its products, regardless of any information or support that may be provided by NXP.

NXP has a Product Security Incident Response Team (PSIRT) (reachable at PSIRT@nxp.com) that manages the investigation, reporting, and solution release to security vulnerabilities of NXP products.

**NXP B.V.** — NXP B.V. is not an operating company and it does not distribute or sell products.

## Trademarks

Notice: All referenced brands, product names, service names, and trademarks are the property of their respective owners.

**NXP** — wordmark and logo are trademarks of NXP B.V.

## Contents





**Date of release: 10 November 2025**
**Document identifier: AN14848**