AN14210

Learning VGLite API Programming on i.MX RT Series

Rev. 1 — 26 February 2024 Application note

Document information

Information	Content
Keywords	AN14210, VGLite, i.MX RT, GPU2D
Abstract	This document introduces how to use VGLite API for graphic programming with several examples.

1 Introduction

The VGLite API is a platform-independent API that supports 2D vector and raster rendering. It can be used as the interface for 2D GPU drivers in i.MX RT500, RT1160, and RT1170 series chips.

This document introduces how to use VGLite API for graphic programming with several examples. These examples can be found on https://github.com/nxp-appcodehub/gs-vglite_examples_rt1170, which are based on MCUX SDK 2.14.0. For references when programming, see the *i.MX RT VGLite API Reference Manual* (document IMXRTVGLITEAPIRM).

2 Architecture of VGLite examples

Figure 1 shows the architecture of VGLite examples. VGLite is in the */middleware/vglite/* folder under SDK installation root, mainly including:

- VGLite API: A set of functions, types, structures, and enumerations for 2D vector/raster rendering, which is defined in *vg_lite.h*.
- VGLite Hardware Abstract Layer (HAL): Abstract the GPU hardware-related common operations for the kernel driver.
- Kernel Driver: Receive the request from VGLite API to manipulate the GPU hardware, expected to execute in the kernel space.
- Elementary API: Wrap the VGLite API but provide a simple and intuitive method to load up and manipulate the graphic resources, not covered in this note.

Files involving board-level initialization and operations are in each example folder. All examples are in the */boards/evkbmimxrt1170/vglite_examples/* folder under SDK installation root, including:

- VGLite_Support: Support the initialization of GPU and VGLite memory.
- VGLite_Window: Provide a frame buffer to be blit or drawn by the VGLite API.
- Display_Support: Achieve the initialization and configuration of the display panel.

Other files related to display are in the */components/video/display/* folder under SDK installation root. They are used to drive display controller, display panel, and so on.

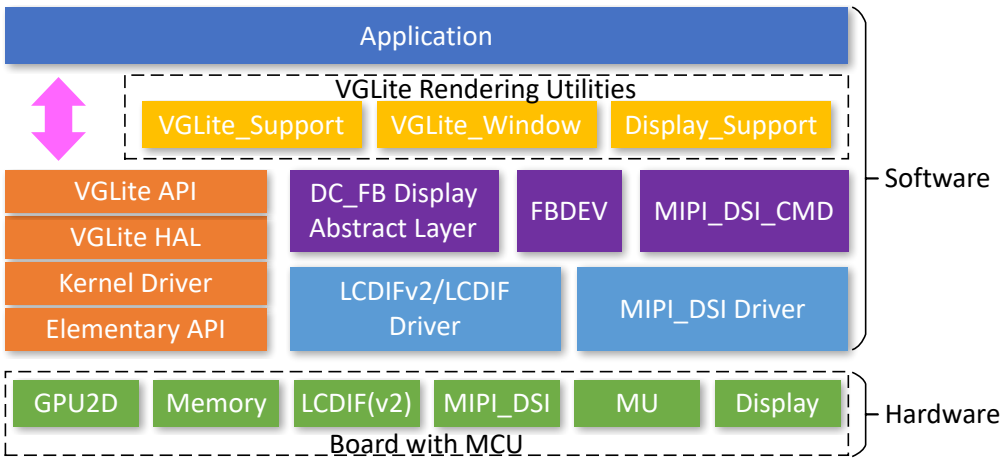

Figure 1. Architecture of VGLite Examples

3 Initialization/Deinitialization

No matter vector drawing or raster blitting, the `vg_lite_init()` function is essential for every task/thread to allocate memory for the command buffer and a tessellation buffer. Conversely, `vg_lite_close()` is used to deallocate the memory previously initialized.

`vg_lite_init()` includes two parameters specifying tessellation buffer size, `tessellation_width` and `tessellation_height`, which must be a multiple of 16. It's recommended to set the tessellation buffer size to cover the most common path size. If the tessellation buffer size is (0, 0), vector drawing APIs such as `vg_lite_draw()` cannot be used.

The command buffer size is set to 64 kB by default by `vg_lite_init()`, which can be changed by `vg_lite_set_command_buffer_size()` if necessary.

The `vg_lite_allocate()` is used to allocate the render buffer before either drawing or blitting functions. The only input parameter of this function is a pointer to the structure `vg_lite_buffer_t`, whose `width`, `height` and `format` must be initialized in advance. The `stride` of this buffer is filled by this function automatically. Conversely, `vg_lite_free()` deallocates the render buffer previously allocated.

Taking 01_SimplePath as an example, the above initialization functions are called before drawing:

```
error = vg_lite_init(OFFSCREEN_BUFFER_WIDTH, OFFSCREEN_BUFFER_HEIGHT);
error = vg_lite_set_command_buffer_size(VGLITE_COMMAND_BUFFER_SZ);
error = vg_lite_allocate(&renderTarget);
```

Once an error happens, the corresponding deinitialization work is executed by the below code:

```
vg_lite_free(&renderTarget);
vg_lite_close();
```

4 Clear

`vg_lite_clear()` clears/fills the entire buffer or partial rectangle of the buffer with a given color.

Before vector drawing or raster blitting, this function is called to prepare the background. Most of the examples apply this function to fill the rendered area with `0xFFFFFFFF` (white):

```
vg_lite_clear(&renderTarget, NULL, 0xFFFFFFFF);
```

And fill the full screen with `0xFFFF0000` (blue):

```
vg_lite_clear(rt, NULL, 0xFFFF0000);
```

5 Color type

For the above `vg_lite_clear()` function, its input color type is 32-bit structure `vg_lite_color_t`. As shown in Figure 2, its format is RGBA8888. This format includes red, green, blue, and alpha channels. Each channel has 8 bits. Red channel is in the least significant bits and alpha channel is in the most significant bits, as described in Section 10.

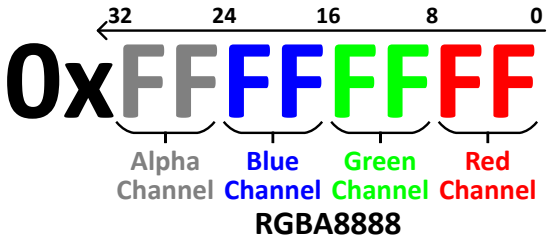

Figure 2. RGBA8888 format

Most VGLite APIs use this structure to define line or paint color, such as `vg_lite_blit()`, `vg_lite_blit_rect()`, `vg_lite_draw()`, `vg_lite_set_stroke()`, `vg_lite_draw_pattern()`, `vg_lite_draw_linear_gradient()`, and `vg_lite_draw_radial_gradient()`, which will be described later.

But some other VGLite functions have different color types:

- `vg_lite_set_grad()` (Section 17) configures colors of linear gradient in an array. These colors are `uint32_t` type with BGRA8888 format, as shown in Figure 3, instead of structure `vg_lite_color_t` with RGBA8888 format.

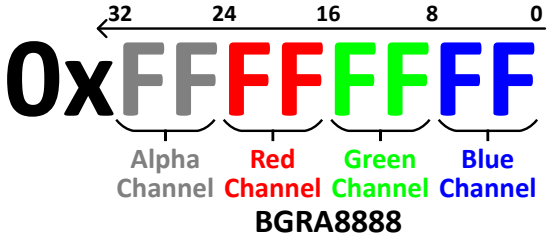

Figure 3. BGRA8888 format

- `vg_lite_set_rad_grad()` (Section 18) and `vg_lite_set_linear_grad()` (Section 19) configure colors of gradient with four float values corresponding to red, green, blue, and alpha channels. These values are in the range of [0, 1.0], which are mapped to an 8-bit pixel value [0, 255] actually.

6 Raster blitting

Blitting is applied to copy the contents of a source buffer to a destination buffer.

Call the function `vg_lite_blit()` to achieve this part, with input source and destination buffer addresses `target`, transformation matrix `matrix`, `color`, blend mode `blend`, and `filter`. These input parameters are introduced in details later.

In addition, examples usually obtain the target buffer by the function `VGLITE_GetRenderTarget()`. After blitting the contents to it, this target buffer will be switched to display when calling the function `VGLITE_SwapBuffers()`. These two functions are from the *vglite_window.c* file, instead of the core VGLite API.

Most examples show the contents on the displayer following the code snippet below:

```
vg_lite_buffer_t *rt = VGLITE_GetRenderTarget(&window);
......
vg_lite_blit(&renderTarget, &dropper, &matrix, VG_LITE_BLEND_SRC_OVER,
 0xFF000000, mainFilter);
VGLITE_SwapBuffers(&window);
```

7 Transformation

Transformation is based on a 3 x 3 matrix (structure `vg_lite_matrix_t`), which achieves translation, rotation and scaling by `vg_lite_translate()`, `vg_lite_rotate()`, and `vg_lite_scale()`. Before calling these functions, `vg_lite_identity()` is needed to reset the input transformation matrix. All transformations are combined until the next time calling `vg_lite_identity()` to reset this transformation matrix.

For all transformation functions, only one coordinate system exists, that is the final plane/surface coordinate system. The (0, 0) is the upper-left corner, the positive x-axis is the right direction, and the positive y-axis is the downward direction. The transformation center of every object is always its upper-left corner point.

Taking 08_BlitColor for example, `vg_lite_translate()` is called three times to set a matrix, to place the source buffer storing image to specified locations, and `vg_lite_rotate()` is called in a loop. Figure 4 describes the following four steps:

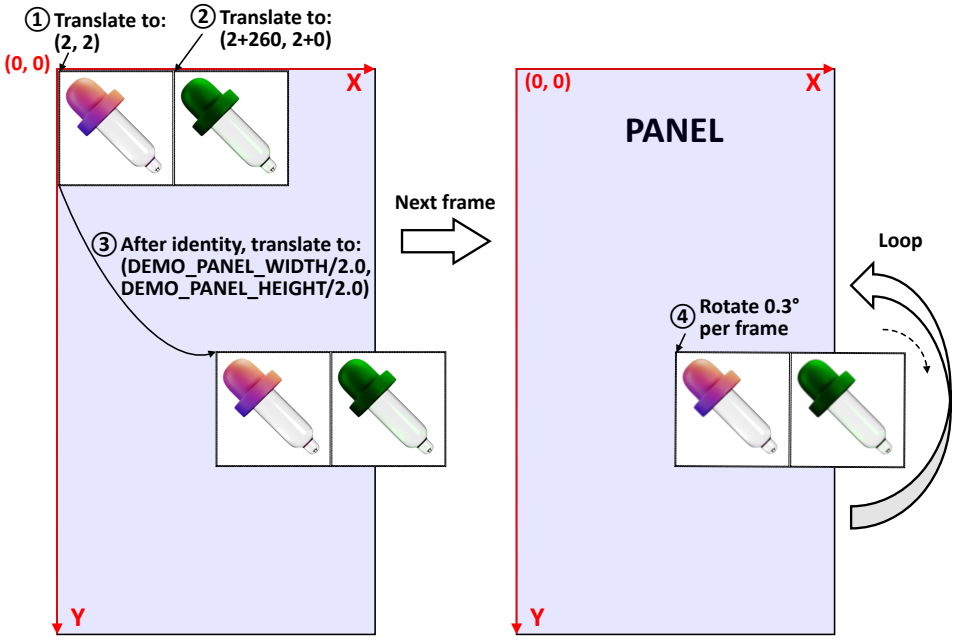

Figure 4. Translation and rotation

1. The first time the translation parameter is set to (2, 2), to blit the upper left corner of the source buffer `dropper` to the (2, 2) position of the target buffer `renderTarget`.

```
vg_lite_identity(&matrix);
vg_lite_translate(2, 2, &matrix);
vg_lite_blit(&renderTarget, &dropper, &matrix, VG_LITE_BLEND_SRC_OVER,
 0xFF000000, mainFilter);
```

2. The second time the translation parameter is set to (260, 0), which is added to the previous value of the matrix, (2+260, 2+0). Then this source buffer is blit to this location of the target buffer.

```
vg_lite_translate(260, 0, &matrix);
vg_lite_blit(&renderTarget, &dropper, &matrix, VG_LITE_BLEND_SRC_OVER,
 0xFF00FF00, mainFilter);
```

3. Then `vg_lite_identity()` is used to clear the previous value of the transformation matrix. So, in the third step, copy the buffer `renderTarget` to `rt` at (`DEMO_PANEL_WIDTH/2.0`, `DEMO_PANEL_HEIGHT/2.0`), which is the center of the displayer.

```
vg_lite_identity(&matrix);
```

```
vg_lite_translate(DEMO_PANEL_WIDTH/2.0, DEMO_PANEL_HEIGHT/2.0, &matrix);
vg_lite_blit(rt, &renderTarget, &matrix, VG_LITE_BLEND_SRC_OVER, 0,
 mainFilter);
```

4. `vg_lite_rotate()` is called in a loop to rotate contents of the source buffer when blitting it to target buffer. Its input `rAngle` increments by 0.3 degree each time.

```
vg_lite_rotate(rAngle, &matrix);
rAngle += 0.3;
```

In addition, the `vg_lite_scale()` function is called in 10_Glyphs to scale a glyph **%** up four times, the code snippet is:

```
vg_lite_scale(4.0, 4.0, &matrix);
/* Blit the buffer to the framebuffer so we can see something on the screen */
error = vg_lite_blit(rt, &bufferToBlit, &matrix, VG_LITE_BLEND_SRC_OVER,
 0xFF00FF00, mainFilter);
```

8 Rectangle blitting

VGLite also supports blitting a rectangular area of the source buffer with `vg_lite_blit_rect()` function, instead of blitting a whole source buffer with regular `vg_lite_blit()`. This function needs an additional array `rect`. `rect[0]` and `rect[1]` are x, y coordinates of the upper left corner of the rectangular area, `rect[0]` and `rect[1]` are width and height of this rectangular area.

12_BlitRect uses it to render four numbers in four times, from the source image that completely contains numbers 0-9. The array `rect` is configured four times, to select corresponding rectangular areas of four numbers, as shown in Figure 5.

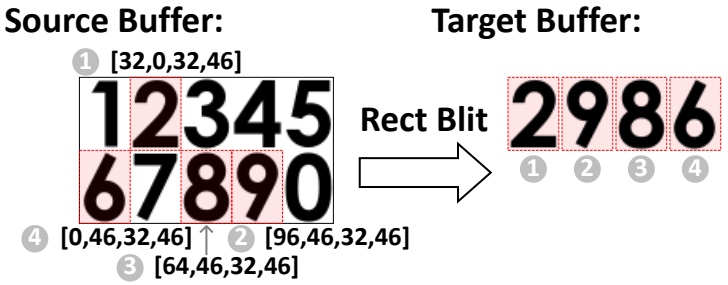

Figure 5. Rectangle blitting

1. Blit the **2** in the sub-rectangle area defined by [32, 0, 32, 46] in the source buffer to the target buffer at (0, 0).

```
vg_lite_identity(&matrix);
rect[0] = 32; rect[1] = 0; rect[2] = 32; rect[3] = 46;
vg_lite_blit_rect(rt, &glyphBuffer, rect, &matrix, VG_LITE_BLEND_SRC_OVER, 0,
 mainFilter);
```

2. Blit the **9** in the sub-rectangle area defined by [96, 46, 32, 46] in the source buffer to the target buffer at (34, 0).

```
vg_lite_translate(34,0,&matrix);
rect[0] = 96; rect[1] = 46; rect[2] = 32; rect[3] = 46;
vg_lite_blit_rect(rt, &glyphBuffer, rect, &matrix, VG_LITE_BLEND_SRC_OVER, 0,
 mainFilter);
```

3. Blit the **8** in the sub-rectangle area defined by [64, 46, 32, 46] in the source buffer to the target buffer at (34+34, 0).

```
vg_lite_translate(34,0,&matrix);
rect[0] = 64; rect[1] = 46; rect[2] = 32; rect[3] = 46;
vg_lite_blit_rect(rt, &glyphBuffer, rect, &matrix, VG_LITE_BLEND_SRC_OVER, 0,
 mainFilter);
```

4. Blit the **6** in the sub-rectangle area defined by [0, 46, 32, 46] in the source buffer to the target buffer at (34+34+34, 0).

```
vg_lite_translate(34,0,&matrix);
rect[0] = 0; rect[1] = 46; rect[2] = 32; rect[3] = 46;
vg_lite_blit_rect(rt, &glyphBuffer, rect, &matrix, VG_LITE_BLEND_SRC_OVER, 0,
 mainFilter);
```

A transformation matrix `matrix` also works when blitting numbers to specified locations of the target buffer.

9 Pixel buffer

Above operations such as blitting, transformation, clear, are all based on the pixel buffer. No matter the source buffer storing an image or the target buffer to be rendered, the buffer features are defined by the structure `vg_lite_buffer_t`, mainly including:

- width: The width of the buffer in pixels.
- height: The height of the buffer in pixels.
- stride: The stride in bytes. It means the address difference of adjacent pixels in the upper and lower rows. It is related to buffer width, color format, and alignment.
- tiled: The buffer data layout in memory, with two choices:
  - `VG_LITE_LINEAR`: Linear (scanline) layout, which is more intuitive.
  - `VG_LITE_TILED`: Data is organized in 4x4 pixel tiles (the buffer address and stride must be 64 bytes aligned for it). This layout has a good rendering performance for the source buffer in the rotation.
- format: The color format of the buffer. It defines color channels and corresponding bit widths, which is described in detail later.
- memory: Pointer to the start address of the memory.
- address: GPU address.
- image_mode: The blitting image mode, described in details later.

For vector drawing, call `vg_lite_allocate()` to allocate the memory for `vg_lite_buffer_t` after setting its width, height, and format. The memory and address of buffer are the address of memory allocated by `vg_lite_allocate()`. And this function calculates and sets stride automatically. Most vector drawing examples use the following code snippet to achieve the above process, such as 01_SimplePath, 02_QuadraticCurves, and 03_Stroked_CubicCurves, and so on.

```
buffer.width = OFFSCREEN_WIDTH;
buffer.height = OFFSCREEN_HEIGHT;
buffer.format = VG_LITE_RGBA8888;
vg_lite_allocate(&buffer);
```

This method also applies to raster blitting example, which sets the three parameters of the buffer and then calls `vg_lite_allocate()`. Original image data must be copied to the allocated memory of buffer manually. The code snippet of 08_BlitColor follows this process:

```
/* Load the image data to a vg_lite_buffer */
dropper.width  = IMG_WIDTH;
```

```
dropper.height = IMG_HEIGHT;
dropper.format = IMG_FORMAT;
vg_lite_allocate(&dropper);
uint8_t * buffer_memory = (uint8_t *) dropper.memory;
uint8_t  *pdata = (uint8_t *) BGRA8888_Data;
for (j = 0; j < dropper.height; j++)
{
    memcpy(buffer_memory, pdata, dropper.stride);
    buffer_memory += dropper.stride;
    pdata += dropper.stride;
}
```

But there are two memory blocks storing the same image: original data, and another copy in the allocated memory of buffer. There is another method to avoid memory waste, in which the memory pointer of the buffer directly points to the original image data, instead of calling `vg_lite_allocate()` to allocate new memory. In this way, the stride of buffer must be set manually. The code snippet of 13_PatternFill and 14_UI applies this method:

```
buffer->width = width;
buffer->height = height;
buffer->stride = stride;
buffer->format = format;
/* "Copy" image data in the buffer */
buffer->handle = NULL;
buffer->memory = imm_array;
buffer->address = (uint32_t)imm_array;
```

10 Color format

VGLite includes but not limits to the following color formats, which are applied to both source image and render target.

- `VG_LITE_RGBA8888`
- `VG_LITE_RGBA5551`
- `VG_LITE_RGBA4444`
- `VG_LITE_RGBA2222`
- `VG_LITE_RGB565`
- `VG_LITE_YUYV`
- `VG_LITE_A8`
- `VG_LITE_A4`
- `VG_LITE_L8`

R, **G**, **B**, and **A** mean red, green, blue, and alpha channels respectively. The numbers such as 8, 5, 4, 2 indicates the bit width of channel at the corresponding position. For instance, **RGBA5551** means the color with 5-bit red, 5-bit green, 5-bit blue, and 1-bit alpha channels. **A8** or **A4** means only 8-bit or 4-bit alpha channel describing transparency, without RGB channels. **L8** means 8-bit luminance, discarding color information.

There are other color formats such as `VG_LITE_ABGR8888`, `VG_LITE_ARGB8888`, `VG_LITE_BGRA8888`. Similar to `VG_LITE_RGBA8888`, they have the same bit widths, but channels are arranged differently. In addition, there is `VG_LITE_RGBX8888` format. **X** means that the alpha channel is all `0xFF` (opaque).

Figure 6 shows the bit width and arrangement of the channels of these color formats.

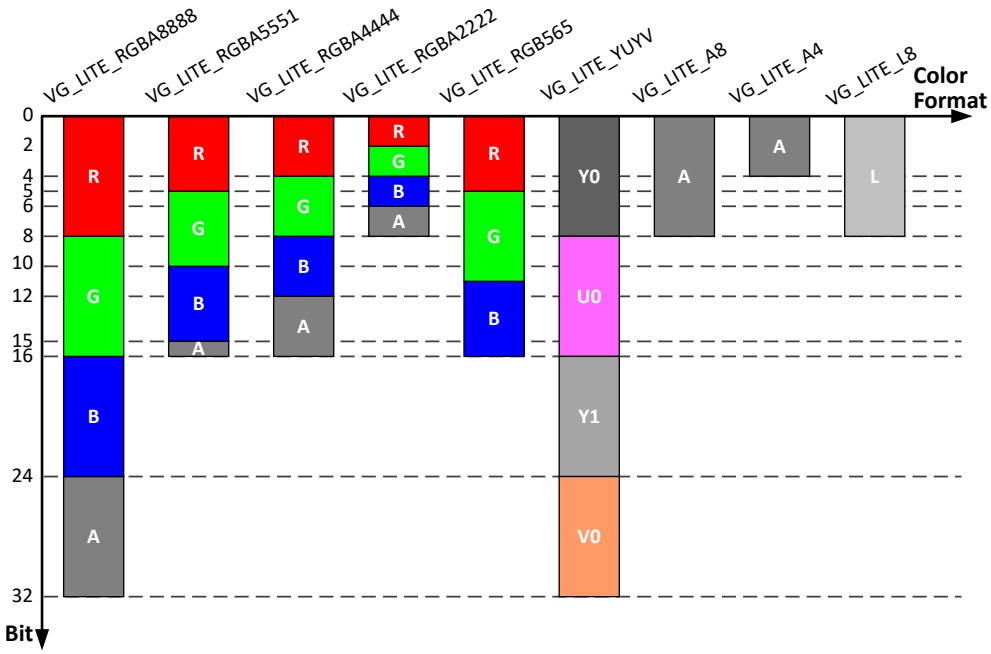

Figure 6. Bit width and arrangement of color formats

Different color formats have different bit widths, strides, and pixel alignment requirements. 08_BlitColor displays the same image with above-mentioned different color formats, switched through the macro `DROPPER_FORMAT` and then blit to the displayer, there is the code snippet:

```
#if(DROPPER_FORMAT==DROPPER565)
uint8_t  *pdata = (uint8_t *) BGR565_Data;
#elif (DROPPER_FORMAT==DROPPER4444)
uint8_t  *pdata = (uint8_t *) BGRA4444_Data;
#elif (DROPPER_FORMAT==DROPPER8888)
uint8_t  *pdata = (uint8_t *) BGRA8888_Data;
......
dropper.stride = IMG_STRIDE;
dropper.format = IMG_FORMAT;
......
vg_lite_blit(&renderTarget, &dropper, &matrix, VG_LITE_BLEND_SRC_OVER,
 0xFF000000, mainFilter);
```

Figure 7 shows the display effects of these color formats.

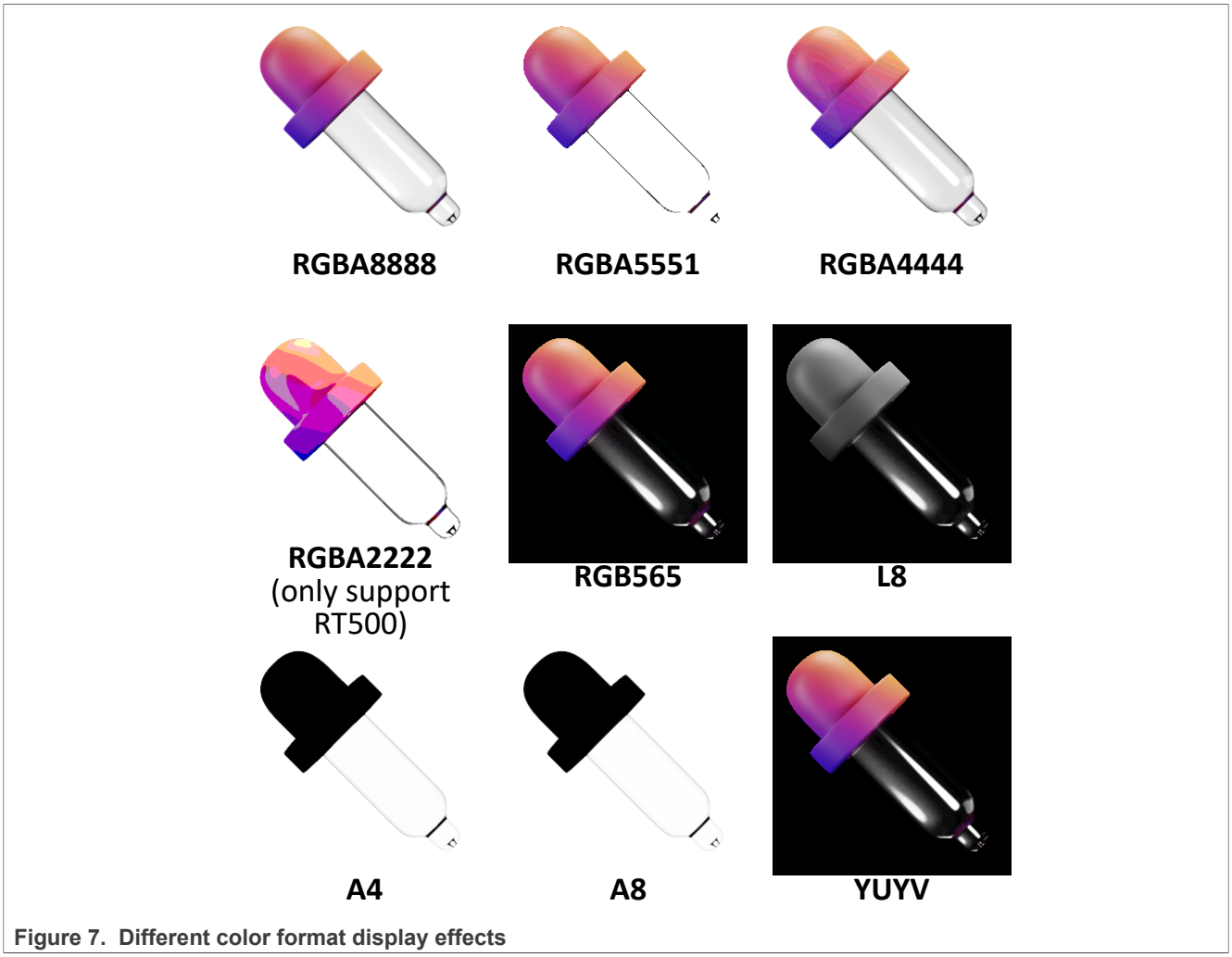

Figure 7. Different color format display effects

As shown in Figure 7, there are some features of various color formats:

- Channels with lower bit width occupy less memory, but also degrade the image quality, resulting in obvious color bandings (especially comparing `VG_LITE_BGRA8888` and `VG_LITE_BGRA2222`).
- **BGR565**, **L8**, and **YUYV** have no alpha channel to mask the background. So that the original background defined by RGB channels appears, instead of being transparent. In this example, the background is black.
- **A8** and **A4** only have alpha data that is invisible without RGB channels, so that this example sets their `image_mode` to `VG_LITE_MULTIPLY_IMAGE_MODE` to paint it black. The image mode is described in details later.
- **BGRA5551** does not support semi-transparency since the alpha value is only 0 (transparent) or 1 (opaque).

11 Image mode

One of three image modes is involved in each buffer to be blit:

- `VG_LITE_NORMAL_IMAGE_MODE`: Image drawn with blending mode (introduced later). This is the most used option.
- `VG_LITE_NONE_IMAGE_MODE`: Image input is ignored.
- `VG_LITE_MULTIPLY_IMAGE_MODE`: Image is multiplied with paint color. The paint color is specified in the function `vg_lite_blit()`.

In 08_BlitColor, two image modes are applied in two images. As shown in Figure 8, the first one shows normal image with `VG_LITE_NORMAL_IMAGE_MODE`, the second one shows painted image with `VG_LITE_MULTIPLY_IMAGE_MODE`. For the second image, the paint color is `0xFF00FF00` (green), then the new image only reserves the green channel, because the values of the other channels are multiplied by `0x00`.

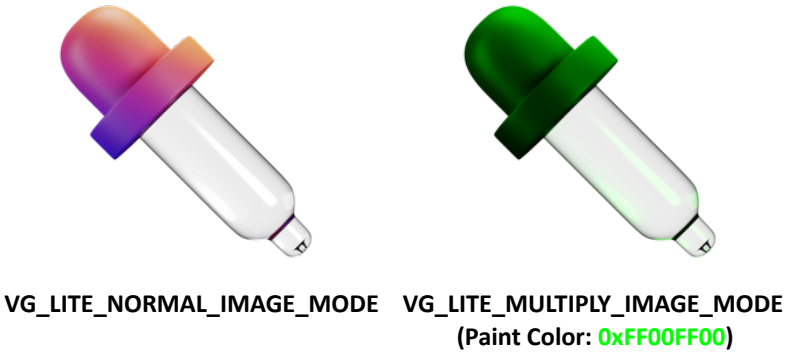

Figure 8. Different image mode

`VG_LITE_MULTIPLY_IMAGE_MODE` is also helpful to make invisible color formats visible, such as **A4** and **A8**. In addition, alpha data + `VG_LITE_MULTIPLY_IMAGE_MODE` is a good combination for displaying glyphs, as alpha data occupies less memory, and this image mode makes it easy to change the color of glyphs.

In 10_Glyphs, the alpha data of glyph **%** is painted to `0xFF00FF00` (green), and blit by the following code snippet:

```
bufferToBlit.format = VG_LITE_A8;
bufferToBlit.image_mode = VG_LITE_MULTIPLY_IMAGE_MODE;
vg_lite_blit(rt, &bufferToBlit, &matrix, VG_LITE_BLEND_SRC_OVER, 0xFF00FF00,
 mainFilter);
```

If a red glyph **%** is needed now, change `0xFF00FF00` (green) in `vg_lite_blit()` to `0xFF0000FF` (red) simply, as shown in Figure 9.

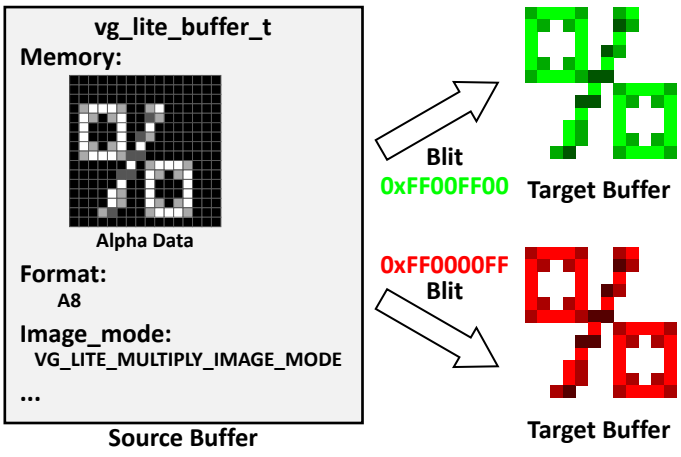

Figure 9. Different glyph color

12 Blending mode

When blitting the source (src) image to the destination (dst) image, there are different blending modes defining the final effect: (S and D mean the source and destination color channels, Sa and Da mean the source and destination alpha channels).

- `VG_LITE_BLEND_SRC_OVER`: Show src over dst, and dst can only be shown in the area where Sa is not 1. The formula is `S + (1 - Sa) * D`.
- `VG_LITE_BLEND_DST_OVER`: Contrary to `VG_LITE_BLEND_SRC_OVER` mode, show dst over src, so src can only be seen in the area where Da is not 1. The formula is `(1 - Da) * S + D`.
- `VG_LITE_BLEND_SRC_IN`: Show src where src and dst overlap, and Da is also applied to *src* image, whose formula is `Da * S`.
- `VG_LITE_BLEND_DST_IN`: Contrary to `VG_LITE_BLEND_SRC_OVER` mode, show *dst* where *src* and *dst* overlap, and *Sa* is also applied to *dst* image, whose formula is `Sa * D`.
- `VG_LITE_BLEND_SCREEN`: Show both *src* and *dst* images, and the result color is at least as light as either of S or D, whose effect is similar to projecting multiple photographic slides simultaneously onto a single screen. The formula is `1 - [(1 - S) * (1 - S)] = S + D - S * D`.
- `VG_LITE_BLEND_MULTIPLY`: Shows both *src* and *dst* images. The*src* image is multiplied by *dst* image, which then replaces the area where *src* and *dst* overlap. The formula is `S * (1 - Da) + D * (1 - Sa) + S * D`.
- `VG_LITE_BLEND_ADDITIVE`: Simply add images of *src* and *dst*, and the formula is `S + D`.
- `VG_LITE_BLEND_SUBTRACT`: Contrary to `VG_LITE_BLEND_ADDITIVE` mode, subtract *src* from *dst*, with the formula of `D * (1 - Sa)`.

For the source buffer in 09_AlphaBehavior, the upper left quarter is opaque blue, and the other area is in black with transparency `0x7F`, set by the following code:

```
buffer_memory = (uint32_t *) bufferToBlit.memory;
/* Alpha value is 0x7f, color value is 0 */
for ( i = 0; i < TEST_RASTER_BUF_WIDTH * TEST_RASTER_BUF_HEIGHT; i++)
buffer_memory[i] = 0x5f000000;
/* Blue color */
for ( i = 0; i < TEST_SMALL_SIZE; i++)
for ( j = 0; j < TEST_SMALL_SIZE; j++)
buffer_memory[i * TEST_RASTER_BUF_WIDTH + j] = 0xFF0000FF;
```

And the destination buffer is cleared in opaque red:

```
vg_lite_clear(rt, NULL, 0xFF0000FF);
```

Then, this source buffer is blit to the destination buffer eight times side by side, with these different blending modes.

```
vg_lite_identity(&matrix);
/*Blit with VG_LITE_BLEND_SRC_OVER blending */
vg_lite_blit(rt, &bufferToBlit, &matrix, VG_LITE_BLEND_SRC_OVER, 0, mainFilter);
vg_lite_translate(TEST_RASTER_BUF_WIDTH, 0, &matrix);
/*Blit with VG_LITE_BLEND_DST_OVER blending */
vg_lite_blit(rt, &bufferToBlit, &matrix, VG_LITE_BLEND_DST_OVER, 0, mainFilter);
vg_lite_translate(-TEST_RASTER_BUF_WIDTH, TEST_RASTER_BUF_HEIGHT, &matrix);
/*Blit with VG_LITE_BLEND_SRC_IN blending */
vg_lite_blit(rt, &bufferToBlit, &matrix, VG_LITE_BLEND_SRC_IN, 0, mainFilter);
......
```

The source image, destination image, and the result image after blitting are shown in Figure 10. The color range in Figure 10 is shown as [0, 1.0] for the convenience of calculation, the real range in this example is still [0, 255]. The color sequence in Figure 10 is (R, G, B) for intuitive display, and the real color format is `VG_LITE_BGRA8888`.

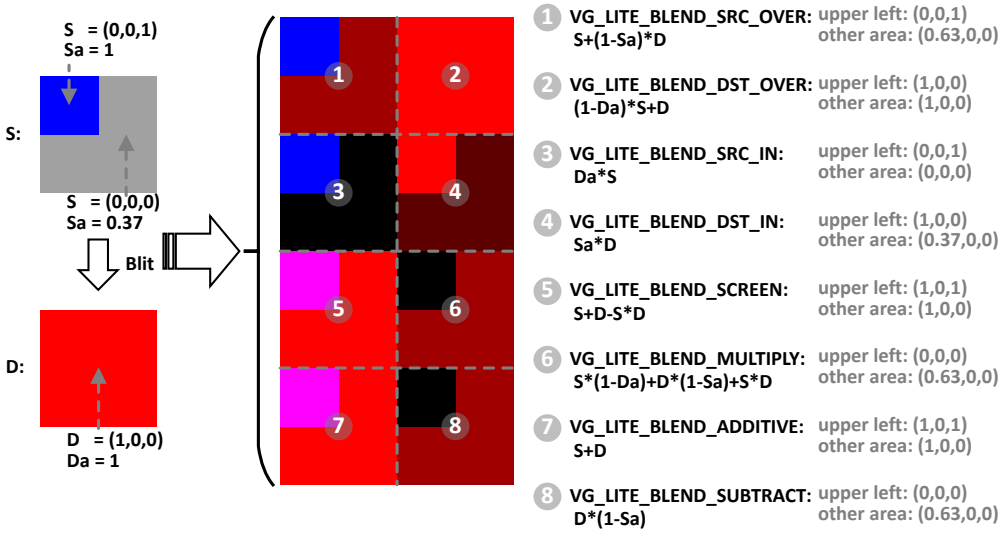

Figure 10. Different blending mode display effects

13 Path control

VGLite provides a series of opcodes for path drawing, such as line, quadratic curve, cubic curve, arc. Both absolute and relative points are supported. There are some common opcodes in Table 1. Different opcodes have different arguments:

Table 1. Common opcodes for path drawing

Opcode	Arguments	Description
`0x00`	None	End. Close any open path.
`0x02`	x, y	Move to the given vertex (x, y). Close any open path.
`0x03`	Δx, Δy	Move to ($start_x$ + Δx, $start_y$ + Δy) by the given relative point (Δx, Δy). Close any open path.
`0x04`	x, y	Draw a line to the given point (x, y).
`0x05`	Δx, Δy	Draw a line to ($start_x$ + Δx, $start_y$ + Δy) by the given relative point (Δx, Δy).
`0x06`	cx, cy, x, y	Draw a quadratic curve to the given endpoint (x, y) using one control point (cx, cy).
`0x07`	Δcx, Δcy, Δx, Δy	Draw a quadratic curve to ($start_x$ + Δx, $start_y$ + Δy) by the given relative endpoint (Δx, Δy), using one control point ($start_x$ + Δcx, $start_y$ + Δcy), calculated by the given relative point (Δcx, Δcy).
`0x08`	cx_1, cy_1, cx_2, cy_2, x, y	Draw a cubic curve to the given endpoint (x, y) using two control points: (cx_1, cy_1) and (cx_2, cy_2).
`0x09`	Δcx_1, Δcy_1, Δcx_2, Δcy_2, Δx, Δy	Draw a cubic curve to ($start_x$ + Δx, $start_y$ + Δy) by the given relative endpoint (Δx, Δy), using two control points, ($start_x$ + Δcx_1, $start_y$ + Δcy_1) and ($start_x$ + Δcx_2, $start_y$ + Δcy_2), calculated by given relative points (Δcx_1, Δcy_1) and (Δcx_2, Δcy_2).

The opcode and arguments must have the same alignment. When the argument type is integer type, the type of opcode is set to be the same as the argument type. But if the argument type is `float`, the opcode type is set to be `uint32_t` to maintain the same alignment. The following code snippet uses a union to achieve this method:

```
union opcode_coord {
float    coord;
uint32_t opcode;
};
static int32_t pathData[] = {
{.opcode=2}, 0.5f,  50.5f,    // Move to (0.5, 50.5)
{.opcode=4}, 50.5f, 50.5f,    // Line from (0.5, 50.5) to (50.5, 50.5)
{.opcode=4}, 25.5f, 0.5f,     // Line from (50.5, 50.5) to (25.5, 0.5)
{.opcode=4}, 0.5f,  50.5f,    // Line from (25.5, 0.5) to (0.5, 50.5)
{.opcode=0},
};
```

The example 01_SimplePath draws a triangle simply, using the structure `vg_lite_path_t` to describe path data, bounding box, quality, and so on.

```
static vg_lite_path_t path = {
    {0, 0,            // left,top
    400, 400},        // right,bottom
    VG_LITE_HIGH,     // quality
    VG_LITE_S32,
    {0},              // uploaded
    sizeof(pathData), // path length
    pathData,         // path data
    1                 // path changed
};
```

The path data array `pathData` consists of drawing opcodes and following arguments:

```
static int32_t pathData[] = {
    2, 0, 400,    // Move to (0, 400)
    4, 400, 400,  // Line from (0,400) to (400, 400)
    4, 200, 0,    // Line from (400, 400) to (200, 0)
    4, 0, 400,    // Line from (200, 0) to (0, 400)
    0,
};
```

In 01_SimplePath, `pathData` includes three opcodes:

- *2*: Move to the point specified by the following two coordinates (x and y).
- *4*: Draw a line from the previous point to the new point defined by the following two coordinates.
- *0*: Finish the path defined above, and close any open path.

02_QuadraticCurves is the same as 01_SimplePath except for the drawing opcode, whose array `pathData` is:

```
static int32_t pathData[] = {
    2, 0, 400,            //Move to (0, 400)
    4, 400, 400,          //Line from (0,400) , to (400, 400)
    6, 400, 200, 200, 0,  //Quadratic Curve from (400, 400) to (200, 0) with
 control point in (400, 200)
    6, 0,200, 0, 400,     //Quadratic Curve from (200, 0) to (0, 400) with
 control point in (0, 200)
    0,
};
```

The new opcode *6* is used to draw the quadratic curve, requiring a control point and an end point.

Figure 11 shows the different results between 01_SimplePath and 02_QuadraticCurves. The second triangle owns two curved edges because of the opcode *6*. As shown in Figure 11, the vertexes of path are marked with big gray dots, and the control points of the quadratic curves are marked with small gray dots and dashed lines.

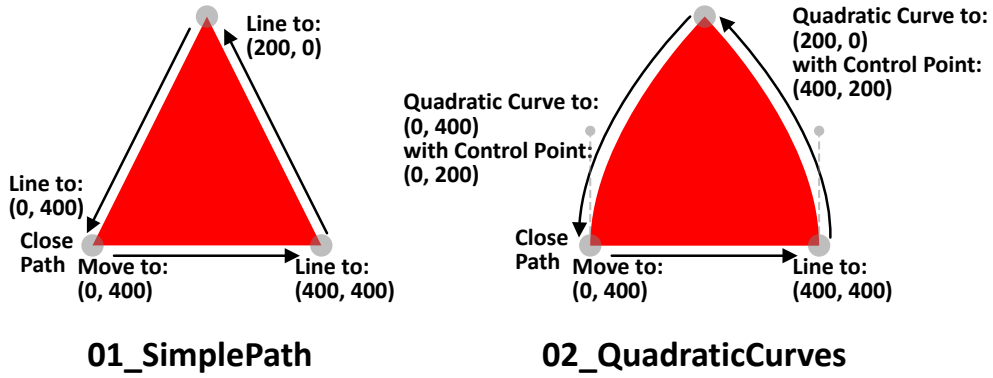

Figure 11. Line path and quadratic curve path

Furthermore, 03_Stroked_CubicCurves draws the cubic curve by the opcode *8*. This opcode needs six arguments: control point 1, control point 2, and end point, such as:

```
static int32_t pathData[] = {
    ......
    // Cubic Curve from (300, 300) to (300, 100)
    // with control point 1 in (254, 228)
    // and control point 2 in (365, 190)
    8, 254, 228, 365, 190, 300, 100,
    ......
};
```

Figure 12 shows the result of 03_Stroked_CubicCurves, with the corresponding control points 1 and 2 of the cubic curves marked by number, small gray dots, and dashed lines.

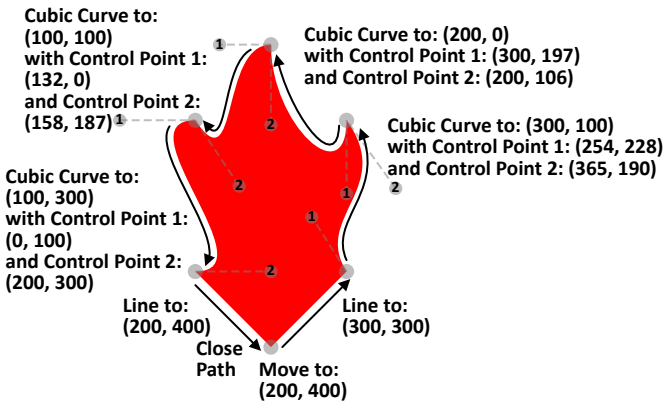

Figure 12. Cubic curve path

14 Vector drawing

Using path array, `vg_lite_draw()` draws contents to specified buffer, with buffer to be drawn `target`, path data `path`, transformation matrix `matrix`, blending mode `blend`, fill color `color`, and specified fill rule `fill_rule` (described in detail later), and so on.

For example, the triangle in 01_SimplePath is drawn and filled with `0xFF0000FF` (red) by the below code:

```
vg_lite_draw(&renderTarget, &path, VG_LITE_FILL_EVEN_ODD, &matrix,
 VG_LITE_BLEND_NONE, 0xFF0000FF);
```

15 Fill rule

Two fill rules are involved in `vg_lite_draw()`, judging whether one point is inside the path area, which is drawn:

- `VG_LITE_FILL_NON_ZERO`: Non-zero fill rule. Suppose that there is a ray from one point to infinity in any direction. When starting with a count of zero, add one when the path crosses the ray from left to right and subtract one when the path crosses it from right to left. This point is inside the path area only if the result is not zero.
- `VG_LITE_FILL_EVEN_ODD`: Even-odd fill rule. Suppose that there is a ray from one point to infinity in any direction. When starting with a count of zero, add one when the path crosses the ray from any side. This point is inside the path area only if the result is odd.

07_FillRules draws one path twice with the above two fill rules separately by the following code:

```
vg_lite_draw(&renderTarget, &path, VG_LITE_FILL_NON_ZERO, &matrix,
 VG_LITE_BLEND_NONE, 0xFF0000FF);
vg_lite_draw(&renderTarget, &path, VG_LITE_FILL_EVEN_ODD, &matrix,
 VG_LITE_BLEND_NONE, 0xFF0000FF);
```

Figure 13 shows the two different effects. For `VG_LITE_FILL_NON_ZERO` mode, the inner diamond area is filled because both two path segments cross the ray from left to right, so the count is greater than 0, which means that this point is inside the path area. However, when using `VG_LITE_FILL_EVEN_ODD` mode, an even number of segments crosses this ray, the count is even, which means that this point is outside the path area, so this diamond area is not filled.

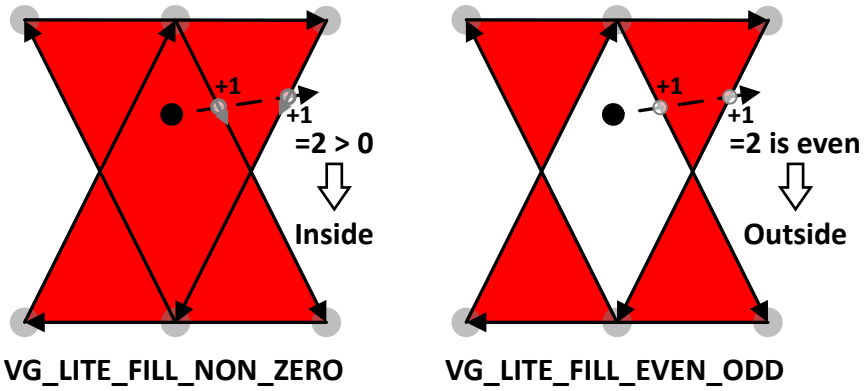

Figure 13. Two fill rules

16 Stroke

The path stroke is configured by function `vg_lite_set_stroke()`, including end cap style, line join style, width, dash pattern, miter limit, dash phase, color, and so on.

03_Stroked_CubicCurves shows nine strokes with different combinations of end cap styles and line join styles in a loop:

```
for (int i = 0; i < 3; i++)
    for (int j = 0; j < 3; j++)
        vg_lite_set_stroke(&paths[index], capStyles[i], joinStyles[j],
 10.0f, 5, stroke_dash_pattern, sizeof(stroke_dash_pattern) /
 sizeof(stroke_dash_pattern[0]), 4.0f, 0xff000000);
```

The end cap style includes:

- `VG_LITE_CAP_BUTT`: Each segment with a line is perpendicular to the tangent at each endpoint.
- `VG_LITE_CAP_ROUND`: Append a semicircle with a diameter equal to the line width centered around each endpoint.
- `VG_LITE_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 14, 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 just for highlight, and these areas have the same color as the line actually, same as below).

Figure 14. Three cap styles

Even if the length of one segment is zero, `VG_LITE_CAP_ROUND` and `VG_LITE_CAP_SQUARE` styles make this segment visible as end caps are added. But this segment is invisible when `VG_LITE_CAP_BUTT` style is selected.

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

- `VG_LITE_JOIN_MITER`: Connect the two segments by extending their outer edges until they meet. If this join style is selected, the input parameter `stroke_miter_limit` of the `vg_lite_set_stroke()` function needs attention, as a small value may limit the miter length.
- `VG_LITE_JOIN_ROUND`: Append a wedge-shaped portion of a circle, centered at the intersection point, whose diameter is equal to the line width.
- `VG_LITE_JOIN_BEVEL`: Connect two points of the outer border of two segments with a straight line.

As shown in Figure 15, 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 15. Three join styles

`vg_lite_set_stroke()` function's input parameter `stroke_dash_pattern` is a sequence of lengths of alternating non-blank and blank dash segments. If there are an odd number of elements, the final element is ignored.

Dash pattern is defined by the array `stroke_dash_pattern` in 03_Stroked_CubicCurves, as illustrated in Figure 16. The first element **30.0f** means that the first line segment length is 30, and the second element **20.0f** means that the following part is a blank space of a length of 20. And so on, the following are a line segment with a length of 50 and a blank space with a length of 25. Then subsequent segments keep repeating these four elements.

```
float stroke_dash_pattern[4] = {30.0f, 20.0f, 50.0f, 25.0f};
```

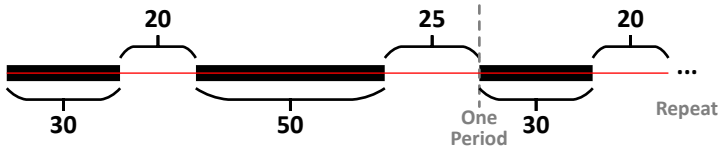

Figure 16. Dash pattern

And the parameter `stroke_dash_phase` defines the starting point in the dash pattern. 03_Stroked_CubicCurves sets it to 4, indicating that the dash skips the first part of length 4, as shown in Figure 17.

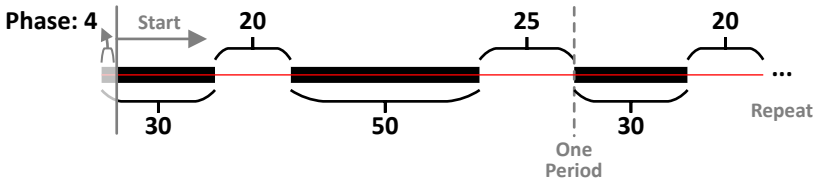

Figure 17. Dash phase is 4

If input `stroke_dash_phase` is greater than the whole length of the dash pattern, segments in the first period are skipped, and the remaining value continues to skip segments in the subsequent periods. For example, if the `stroke_dash_phase` is 129, as shown in Figure 18, the whole part with a length of 125 in the first period is skipped, and then the part of length 4 in the second period is skipped. In this case, the display effect is the same as when `stroke_dash_phase` is 4.

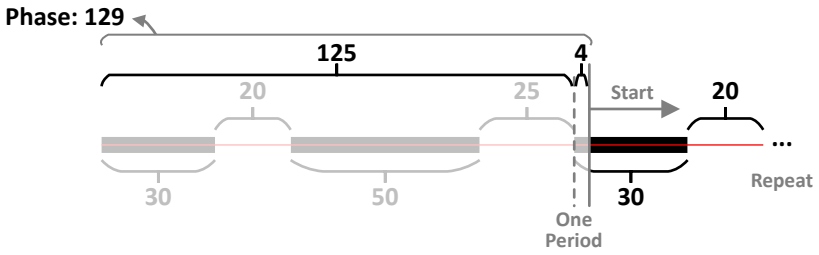

Figure 18. Dash phase is 129

17 Linear gradient

Path can also be filled with a linear or radial gradient color instead of a solid color.

Features of linear gradient are described by the structure `vg_lite_linear_gradient_t`, including:

- colors: Color array for the gradient, `uint32_t` type with 32-bit ARGB format, instead of structure `vg_lite_color_t` with 32-bit ABGR format.
- count: Number of colors.
- stops: Color stop offsets, from 0 to 255.
- matrix: The matrix to transform the gradient.

This structure is initialized by the function `vg_lite_init_grad()` with default configuration, set and updated by functions `vg_lite_set_grad()` and `vg_lite_update_grad()`.

The matrix of linear gradient is gotten by the function `vg_lite_get_grad_matrix ()` . General transformation functions like `vg_lite_identity()`, `vg_lite_translate()`, `vg_lite_scale()` and `vg_lite_rotate()` are suitable for this matrix, to transform the corresponding linear gradient.

Then `vg_lite_draw_gradient()` is called to draw the path filled with linear gradient color. The area of path outside the linear gradient will be filled with the closest stop color.

In 04_LinearGradient, there is an array defining three stops, distributed at 0, 127, and 255. And another array defines corresponding colors (pure red, green, and blue separately) with BGRA8888 format, as shown in the following code snippet:

```
/* Gradient information. The ramps specify the color information at each one of
 the stops */
uint32_t ramps[] = {0xffff0000, 0xff00ff00, 0xff0000ff};

/* Stops define the offset, where in the line those color ramps will be located.
 It can go from 0 to 255 */
uint32_t stops[] = {0, 128, 255};
```

Then initialization, configuration, and update are executed by the following code:

```
vg_lite_init_grad(&grad))

/* Create the gradient using the values from the structures we defined */
vg_lite_set_grad(&grad, 3, ramps, stops);
vg_lite_update_grad(&grad);
```

Get the transform matrix of gradient and place it inside the path, draw it finally:

```
/* Locate the gradient in the gradient coordinate system */
gradientMatrix = vg_lite_get_grad_matrix(&grad);
vg_lite_identity(gradientMatrix);
vg_lite_translate(100,0,gradientMatrix);
vg_lite_scale(200.0/256,1.0f,gradientMatrix);
vg_lite_draw_gradient(&renderTarget, &path, VG_LITE_FILL_EVEN_ODD, &matrix,
 &grad, VG_LITE_BLEND_NONE);
```

Figure 19 shows the display result, where two gray dash lines mark the border of the linear gradient. Obviously, the left and right sides of path are outside gradient, which is filled with pure red and pure blue separately.

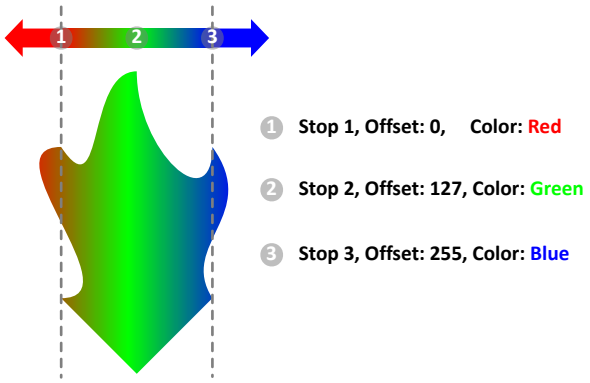

Figure 19. Linear gradient

The gradient drawn by this method is simulated by software, as `vg_lite_draw_gradient()` is achieved by `vg_lite_draw_pattern()` (detailed later). i.MX RT500, RT1160, and RT1170 all support this method, while the latter two gradient-drawing methods (Section 18 and Section 19) are not supported by RT500, which is based on hardware.

18 Radial gradient

Features of radial gradient are defined by structure `vg_lite_radial_gradient_t`, mainly including:

- count: Count of colors, up to 256.
- matrix: The matrix to transform the gradient.
- radialGradient: Radial gradient parameters.
- vgColorRamp: Stops and colors for gradient.
- SpreadMode: The gradient spread mode. There are four spread modes:
  - `VG_LITE_RADIAL_GRADIENT_SPREAD_FILL`: Fill the outside area with black color.
  - `VG_LITE_RADIAL_GRADIENT_SPREAD_PAD`: Fill the outside area with the closest stop color.
  - `VG_LITE_RADIAL_GRADIENT_SPREAD_REPEAT`: Fill the outside area with a repeated gradient.
  - `VG_LITE_RADIAL_GRADIENT_SPREAD_REFLECT`: Fill the outside area with a reflected gradient.

In 05_RadialGradient, four small rectangles with different spread modes are blit side by side, filled with radial gradient.

For parameter `vgColorRamp`, there are five members in the following code. Each member includes one stop and a corresponding color. Five sub members in each member represent stop, red channel, green channel, blue channel, and alpha channel separately. Though the color values are in the range of [0, 1.0], they are mapped to an 8-bit pixel value [0, 255] actually.

```
static vg_lite_color_ramp_t vgColorRamp[] =
{
    {0.0f,  0.4f, 0.0f, 0.6f, 1.0f},
    {0.25f, 0.9f, 0.5f, 0.1f, 1.0f},
    {0.5f,  0.8f, 0.8f, 0.0f, 1.0f},
    {0.75f, 0.0f, 0.3f, 0.5f, 1.0f},
    {1.00f, 0.4f, 0.0f, 0.6f, 1.0f}
};
```

The parameter `radialGradient` is defined by `vg_lite_radial_gradient_parameter_t`, including five numbers. The first number is the gradient radius, the next two numbers are x and y coordinates of the gradient center, and the last two numbers are x and y coordinates of the focal point. In this example, the gradient radius

is 256, and the coordinates of the gradient center are equal to the focal point. Both are at (256, 256). In this case, this radial gradient is a concentric circle:

```
vg_lite_radial_gradient_parameter_t radialGradient = {256.0f, 256.0f, 256.0f,
 256.0f, 256.0f};
```

Functions `vg_lite_set_rad_grad()` and `vg_lite_update_rad_grad()` are called to configure and update the radial gradient respectively:

```
vg_lite_set_rad_grad(&grad, 5, vgColorRamp, radialGradient,
 spreadmode[fcount],1);
vg_lite_update_rad_grad(&grad);
```

`vg_lite_get_rad_grad_matrix()` gets the transformation matrix of the radial gradient, and general transformation functions, such as `vg_lite_identity()`, `vg_lite_translate()`, are suitable for this matrix. After transforming the radial gradient to make it inside the path, the `vg_lite_draw_radial_gradient()` function draws it finally:

```
matGrad = vg_lite_get_rad_grad_matrix(&grad);
vg_lite_identity(matGrad);
......
vg_lite_draw_radial_gradient(fb, &path, VG_LITE_FILL_EVEN_ODD, &matPath, &grad,
 0,
VG_LITE_BLEND_NONE, VG_LITE_FILTER_LINEAR);
```

Once an error happens, a cleaning up work of this radial gradient is executed by calling the function `vg_lite_clear_radial_grad()`:

```
vg_lite_clear_radial_grad(&grad);
```

Figure 20 shows the display result of this example, where four rectangles are divided by black dash lines. And two gray dash lines mark the edge of the radial gradient. Four rectangles apply four spread modes separately, so that four outside areas are filled with different colors or gradients.

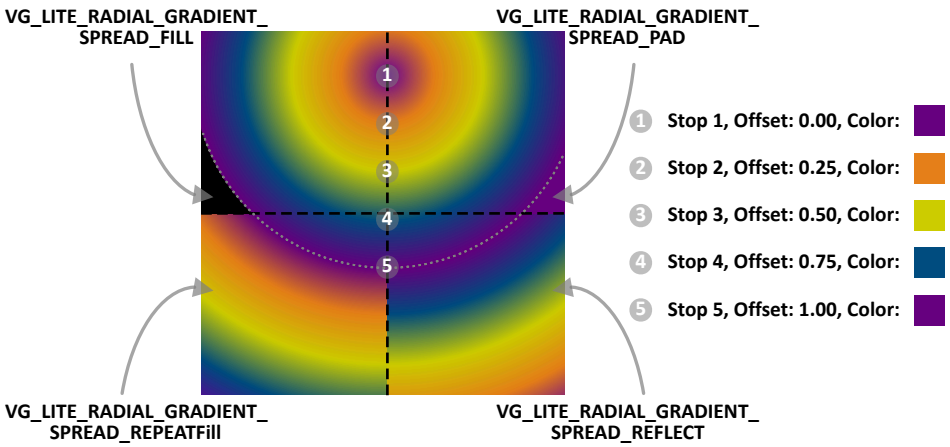

Figure 20. Radial gradient

19 Extended linear gradient

Similar to the radial gradient, there is another structure for the linear gradient, `vg_lite_linear_gradient_ext_t`. It defines the extended linear gradient parameters, including:

- count: Count of colors, up to 256.
- matrix: The matrix to transform the gradient.
- linearGradient: Linear gradient parameters.
- vgColorRamp: Stops and colors for gradient.
- SpreadMode: The gradient spread mode, same as the four spread modes of radial gradient.

There are some differences between the two structures involving linear gradient, `vg_lite_linear_gradient_t` and `vg_lite_linear_gradient_ext_t`:

- `vg_lite_linear_gradient_t` sets colors and stops in two arrays respectively, the type of colors is `uint32_t` with BGRA8888 format, such as 0xFFFF0000 (red), and the range of stops is [0, 255]. But `vg_lite_linear_gradient_ext_t` stores both colors and stops in an array of structure `vg_lite_color_ramp_t`, in which stops and four channels (RGBA) of color are all floating point with a range of [0, 1.0].
- `vg_lite_linear_gradient_t` always uses the closest stop color to fill the outside area of gradient. While `vg_lite_linear_gradient_ext_t` has four different spread modes to define the gradient padding mode.

In 06_LinearExtGradient, four small rectangles are blit side by side, filled with a linear gradient with four different spread modes. The code structures of *05_RadialGradient* and *06_LinearExtGradient* are similar.

For the parameter `vgColorRamp`, the following code sets five stops and the corresponding colors of this linear gradient. Five sub members in each member represent stop, and R, G, B, A channels separately:

```
static vg_lite_color_ramp_t vgColorRamp[] =
            {
            {0.0f,  0.4f, 0.0f, 0.6f, 1.0f},
            {0.25f, 0.9f, 0.5f, 0.1f, 1.0f},
            {0.5f,  0.8f, 0.8f, 0.0f, 1.0f},
            {0.75f, 0.0f, 0.3f, 0.5f, 1.0f},
            {1.00f, 0.4f, 0.0f, 0.6f, 1.0f}
            };
```

The parameter `linearGradient` affects the radial direction for a linear gradient. It is defined by the structure `vg_lite_linear_gradient_parameter_t`, including four numbers. The first two numbers are x and y coordinates of start point, and the last two are x and y coordinates of end point. For this example, set the radial direction of linear gradient from (160, 100) to (480, 100) with the following code:

```
vg_lite_linear_gradient_parameter_t radialGradient = {160.0f, 100.0f, 480.0f,
 100.0f};
```

Then the functions `vg_lite_set_linear_grad()` and `vg_lite_update_linear_grad()` configure and update the linear gradient.

```
vg_lite_set_linear_grad(&grad, 5, vgColorRamp, radialGradient,
 spreadmode[fcount], 1);
vg_lite_update_linear_grad(&grad);
```

The transform matrix of this gradient is gained by the function `vg_lite_get_linear_grad_matrix()` to place it inside the path. The function `vg_lite_draw_linear_gradient()` draws it finally:

```
matGrad = vg_lite_get_linear_grad_matrix(&grad);
```

```
......
vg_lite_draw_linear_gradient(fb, &path, VG_LITE_FILL_EVEN_ODD, &matPath, &grad,
 0, VG_LITE_BLEND_NONE, VG_LITE_FILTER_LINEAR);
```

A cleaning up work of this linear gradient is supported with the following code:

```
vg_lite_clear_linear_grad(&grad);
```

Figure 21 shows the display result of this example, two gray dash lines mark the border of the linear gradient. Four rectangles are divided by black dash lines, four different spread modes result in that four outside areas are filled with different colors or gradients correspondingly.

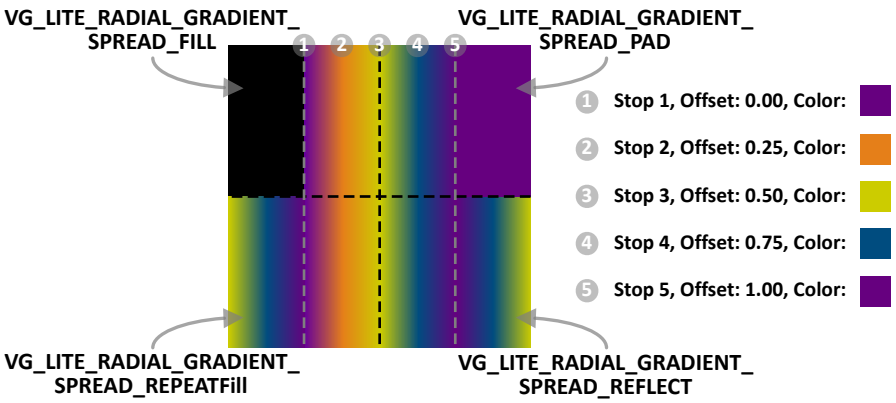

Figure 21. Extended linear gradient

20 Fill pattern

In addition to solid color and gradient, a path can also be filled with an image when replacing the regular `vg_lite_draw()` function with the `vg_lite_draw_pattern()` function.

There are two image fill pattern modes below, defined by the enumeration `vg_lite_pattern_mode`:

- `VG_LITE_PATTERN_COLOR`: Fill the area outside the image with a specified color.
- `VG_LITE_PATTERN_PAD`: Expand the color of the image border to fill the outside area.

13_PatternFill applies these two pattern modes separately, achieved by the code below:

```
vg_lite_draw_pattern(&renderTarget, &path, VG_LITE_FILL_EVEN_ODD, &matPath,
 &image, &matrix, VG_LITE_BLEND_NONE, VG_LITE_PATTERN_COLOR, 0xffaabbcc,
 mainFilter);
vg_lite_draw_pattern(&renderTarget, &path, VG_LITE_FILL_EVEN_ODD, &matPath,
 &image, &matrix, VG_LITE_BLEND_NONE, VG_LITE_PATTERN_PAD, 0xffaabbcc,
 mainFilter);
```

In the above code, a matrix `matPath` is used to transform `path`. In addition, `image` is the source buffer storing image, whose transformation is controlled by another matrix `matrix`.

When applying `VG_LITE_PATTERN_COLOR` mode, the solid color is needed to fill the area outside the image, which is set to 0xffaabbcc in this example. But it's not necessary in another mode, as the area outside is already filled by the image border.

Figure 22 shows the result, in which gray dash lines mark the outline of the images.

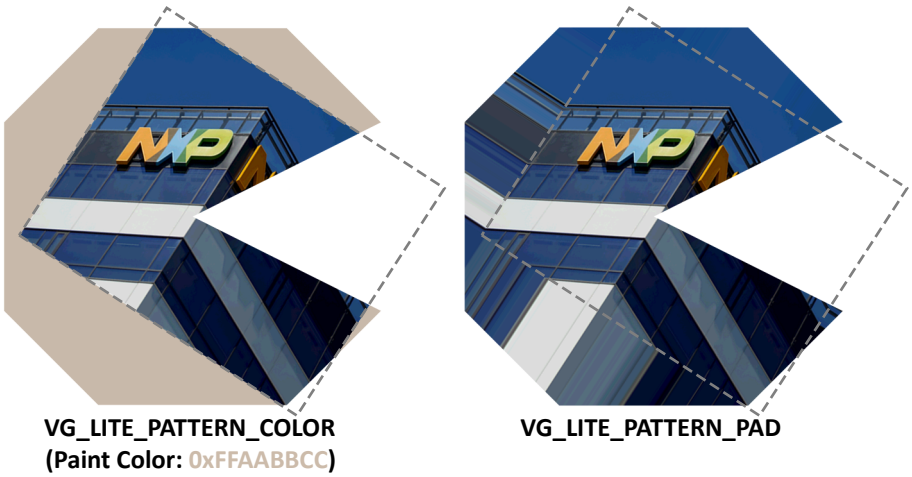

Figure 22. Two fill patterns

21 Multitask

VGLite is based on FreeRTOS that has multitask scheduling capabilities. Displaying different parts of the GUI in various tasks is a feasible solution. And it's common in actual projects that other tasks receive and process the data from sensors, and VGLite task displays them on the GUI.

20_Multitask creates two tasks: `vglite_task` and `vglite_task2`. The two tasks draw two rotating tigers separately. Every task must do the initialization work of VGLite:

```
// vglite_task
// Initialize the draw.
vg_lite_init(TW / 2, TH / 2);
// Set GPU command buffer size for this drawing task.
vg_lite_set_command_buffer_size(VGLITE_COMMAND_BUFFER_SZ);
......
// vglite_task2
// Initialize the draw.
vg_lite_init(DEMO_BUFFER_WIDTH / 2, DEMO_BUFFER_HEIGHT / 2);
// Set GPU command buffer size for this drawing task.
vg_lite_set_command_buffer_size(VGLITE_COMMAND_BUFFER_SZ);
```

`vglite_task` gets the render target buffer by calling `VGLITE_GetRenderTarget()`, and draws tiger on it. `vglite_task2` clears one of three buffers in `tmp_buf` to blue draws on it in turn, specified by `index`. Then `vglite_task2` calls `vg_lite_finish()` to submit the command buffer to the GPU and wait it to complete. This drawn buffer is blit to the target buffer in `vglite_task`. Then `vglite_task` calls `VGLITE_SwapBuffers()` to switch the frame buffer and achieve displaying of two tigers. The code snippet is shown in the following code:

```
// vglite_task
vg_lite_buffer_t *rt = VGLITE_GetRenderTarget(&window);
// Draw the path using the matrix.
vg_lite_clear(rt, NULL, 0xFFFFFFFF);
for (count = 0; count < pathCount; count++)
    vg_lite_draw(rt, &path[count], VG_LITE_FILL_EVEN_ODD, &matrix,
 VG_LITE_BLEND_NONE, color_data[count]);
```

```
vg_lite_blit(rt, &tmp_buf[(index+2) % 3], &mat,VG_LITE_BLEND_NONE, 0,
 VG_LITE_FILTER_POINT);
/* Switch the current framebuffer to be displayed */
VGLITE_SwapBuffers(&window);
......
// vglite_task2
index = index % 3;
// Draw the path using the matrix.
vg_lite_clear(&tmp_buf[index], NULL, 0xFFFF0000);
for (count = 0; count < pathCount; count++)
    error = vg_lite_draw(&tmp_buf[index], &path[count], VG_LITE_FILL_EVEN_ODD,
 &matrix2, VG_LITE_BLEND_NONE, color_data[count]);
index++;
vg_lite_finish();
```

Figure 23 shows the result, where the gray dash lines mark the content of buffer written by `vglite_task2`:

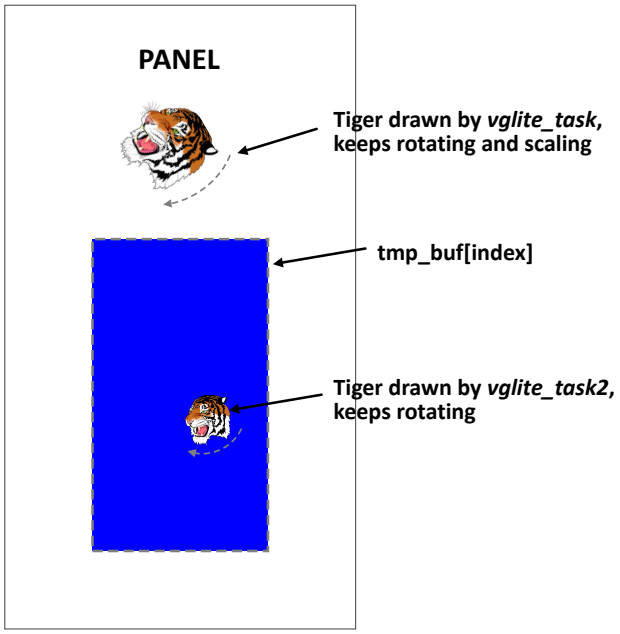

Figure 23. Multitask

22 References

- *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)
- *VGLite Driver Porting Guide* (document IMXRTVGLITEPG)
- OpenVG 1.1 Lite Specification
- Compositing and Blending Level 1

23 Note about the source code in the document

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

Copyright 2024 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

24 Revision history

Table 2 summarizes the revisions to this document.

Table 2. Revision history

Document ID	Release date	Description
AN14210 v.1	26 February 2024	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.

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.

Amazon Web Services, AWS, the Powered by AWS logo, and FreeRTOS — are trademarks of Amazon.com, Inc. or its affiliates.

i.MX — is a trademark of NXP B.V.

SEGGER Embedded Studio — is a trademark of SEGGER Microcontroller GmbH.

Contents

For more information, please visit: https://www.nxp.com

Date of release: 26 February 2024
Document identifier: AN14210