AN14210

使用 **VGLite API** 在 **i.MX RT** 系列的图形编程

Rev. 1 — **2024**年**2**月**26**日 应用笔记

文档信息

信息	内容
关键词	AN14210, VGLite, i.MX RT, GPU2D
摘要	本文通过若干例程介绍了如何使用 VGLite API 进行图形编程。

1 简介

VGLite API 是独立于平台的 API，支持 2D 矢量和光栅的渲染。它可以用作 i.MX RT500、RT1160 和 RT1170 系列芯片中 2D GPU 驱动程序的接口。

本文通过若干例程介绍了如何使用 VGLite API 进行图形编程。这些例程已发布在 https://github.com/nxp-appcodehub/gs-vglite_examples_rt1170，基于 MCUX SDK 2.14.0。建议编程时参考 i.MX RT VGLite API Reference Manual（文档 IMXRTVGLITEAPIRM）。

2 VGLite 例程架构

VGLite 项目架构如 图 1 所示。VGLite 位于 SDK 根目录下的 /middleware/vglite/ 文件夹中，主要包括：

- VGLite API：一系列用于 2D 矢量/光栅渲染的函数、类型、结构体、枚举体，一般在 vg_lite.h 中。
- VGLite Hardware Abstract Layer (HAL 硬件抽象层)：为内核驱动抽象出与 GPU 硬件相关的操作。
- Kernel Driver（内核驱动）: 接收 VGLite API 的请求并操作 GPU，通常在内核空间中执行。
- Elementary API：封装了 VGLite API，提供了一种简单直观的方法来加载和操作图形资源，本文未涉及此部分。

涉及板级初始化与操作的文件位于各例程文件夹内。所有例程位于 SDK 根目录下的 /boards/evkbmimxrt1170/vglite_examples/ 文件夹中，包括：

- VGLite_Support：进行 GPU 和 VGLite 内存的初始化。
- VGLite_Window：提供帧缓冲区，用于 VGLite API 作矢量/光栅的渲染。
- Display_Support：实现显示屏的初始化和配置。

其他与显示相关的文件位于 SDK 根目录下的 /components/video/display/ 文件夹中，用于驱动显示控制器、显示屏等。

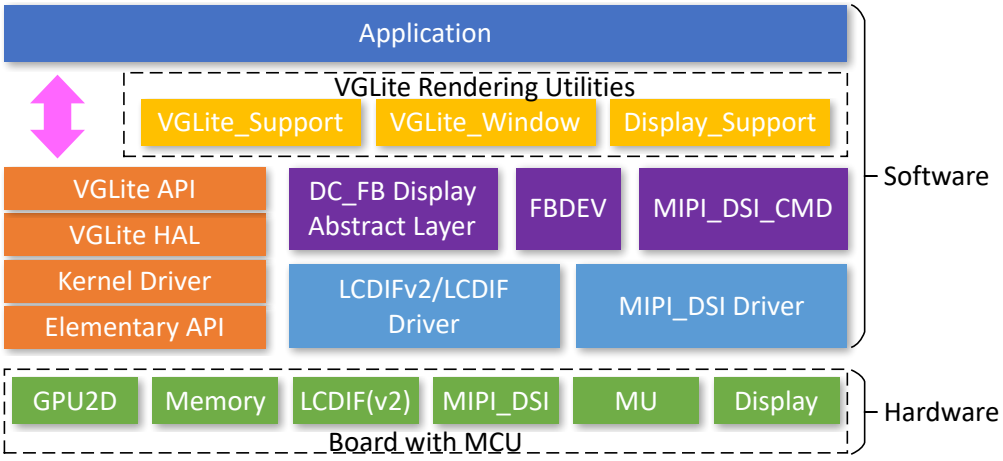

图 1. VGLite 例程架构

3 初始化/反初始化

无论是矢量还是光栅的渲染，每个任务/线程都需要先调用 vg_lite_init() 函数，以此为命令缓冲区和曲面细分缓冲区分配内存。相反，vg_lite_close() 用来释放分配的内存。

vg_lite_init() 包含两个参数来指定曲面细分缓冲区大小：tessellation_width 和 tessellation_height，它们必须是 16 的倍数。建议设置其大小以恰好覆盖常用路径的尺寸。如果曲面细分缓冲区大小为 (0, 0)，则无法使用矢量绘制的 API，如 vg_lite_draw()。

vg_lite_init() 设置默认命令缓冲区大小为 64 KB，后面可再通过 vg_lite_set_command_buffer_size() 函数更改。

vg_lite_allocate() 用于在渲染函数之前分配其缓冲区。该函数只有一个输入参数，为指向 vg_lite_buffer_t 结构体的指针，其 width、height 和 format 属性必须提前初始化。此函数也会根据 width 与 format 自动计算 stride 属性。与此函数相反，vg_lite_free() 会释放其分配的渲染缓冲区。

以例程 01_SimplePath 为例，在渲染之前会调用上述几个初始化函数：

```
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);
```

一旦发生错误，就会执行相应的清理工作：

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

4 清理

vg_lite_clear() 使用指定颜色来清除/填充整个缓冲区或缓冲区的某个矩形区域。

在渲染之前，通常会调用此函数来设置背景色。大多数示例调用此函数来用 0xFFFFFFFF（白色）填充渲染区域：

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

大多数例程通常使用 0xFFFF0000（蓝色）填充全屏：

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

5 颜色类型

对于上述 vg_lite_clear() 函数，其输入颜色由 32 位的 vg_lite_color_t 结构体设置。如 图 2 所示，其格式为 RGBA8888（红、绿、蓝、alpha 通道，每个通道占 8 位。红色通道在最低位，alpha 通道在最高位，详见 章节 10。

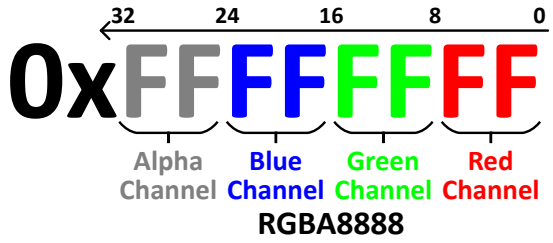

图 2. RGBA8888 格式

大多数 VGLite 函数都使用此结构体来定义线条或填充颜色的格式，例如 `vg_lite_blit()`、`vg_lite_blit_rect()`、`vg_lite_draw()`、`vg_lite_set_stroke()`、`vg_lite_draw_patte` 和 `vg_lite_draw_radial_gradient()`，这些函数稍后均会讲解。

但也有一些 VGLite 函数有其他的颜色格式：

- `vg_lite_set_grad()` (章节 17) 使用数组来定义渐变的颜色。数组元素类型为 `uint32_t`，格式为 图 3 所示的 BGRA8888，而不是上述的 RGBA8888 格式的 `vg_lite_color_t` 结构体。

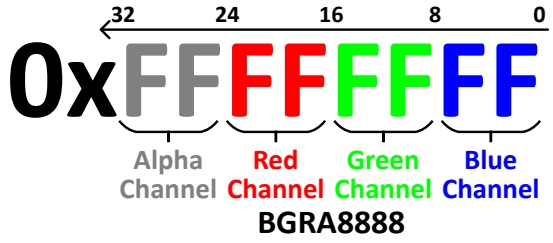

图 3. BGRA8888 格式

- `vg_lite_set_rad_grad()` (章节 18) 和 `vg_lite_set_linear_grad()` (章节 19) 使用四个浮点数来配置渐变的颜色，分别对应红色、绿色、蓝色和 alpha 通道。这些值取值范围是 [0, 1.0]，但实际上 VGLite 会将其映射到 [0, 255] 区间，即 8 位像素值。

6 光栅传输 (Raster Blitting)

`vg_lite_blit()` 函数用于将源缓冲区的内容复制到目标缓冲区中。此函数的输入参数包括：源缓冲区地址`source`、目标缓冲区地址`target`、变换矩阵`matrix`、填充颜色`color`、混合模式`blend`和过滤器`filter`。这些参数稍后会详细介绍。

另外，示例通常通过 `VGLITE_GetRenderTarget()` 函数来获取目标缓冲区。将图像复制到目标缓冲区后，再调用函数 `VGLITE_SwapBuffers()` 可切换该目标缓冲区以在屏幕上显示。这两个函数均来自 vglite_window.c 文件，而不是核心 VGLite API。

大多数示例使用以下代码段以在屏幕上显示内容：

```
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 变换 (Transformation)

变换基于 vg_lite_matrix_t 结构体中的 3 x 3 变换矩阵，利用vg_lite_translate()、vg_lite_rotate()、vg_lite_scale() 等函数实现平移、旋转和缩放。调用这些函数之前，需要先使用 vg_lite_identity() 重置变换矩阵。直到下次再调用 vg_lite_identity() 函数时，之前的所有变换操作都会被合并。

对于所有变换函数，均只有一个坐标系，即最终的平面坐标系。该坐标系的原点为左上角，向右为 x 轴正方向，向下为 y 轴正方向。每个对象的变换中心都是该对象的左上角点。

以 08_BlitColor 为例，调用 3 次 vg_lite_translate() 来设置变换矩阵，以此将存储图像的源缓冲区放置到指定位置，并循环调用 vg_lite_rotate() 来旋转图像。图 4 对以下四步进行了描述：

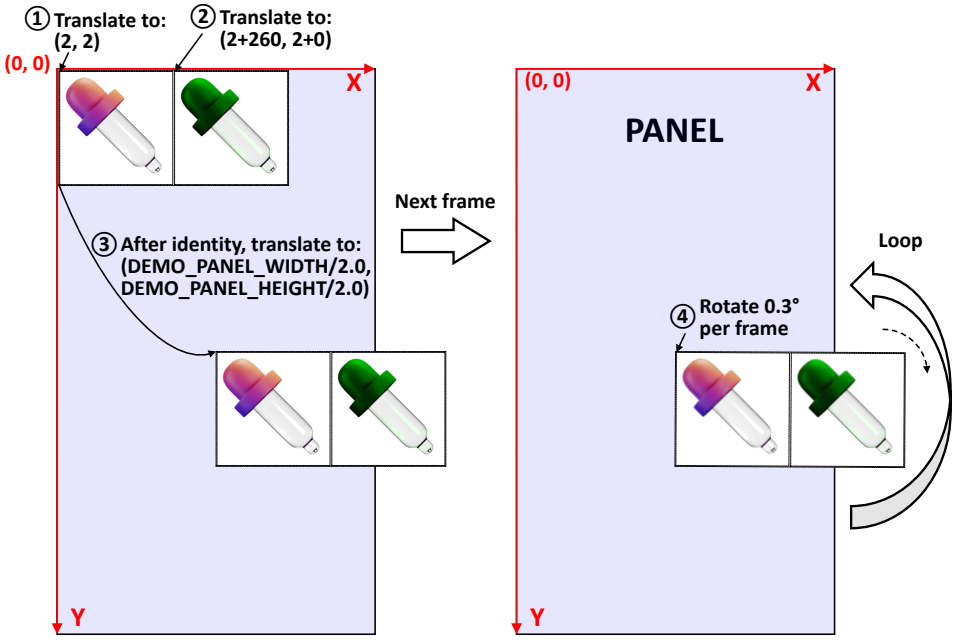

图 4. 平移与旋转

1. 第一次将平移参数设置为 (2, 2)，即将源缓冲区 dropper 的左上角复制到目标缓冲区 renderTarget 的 (2, 2) 坐标处。

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

2. 第二次平移参数设置为 (260, 0)，它会与矩阵的当前值相加，得到 (2+260, 2+0)。然后输入的源缓冲区被复制到目标缓冲区的 (2+260, 2+0) 坐标处。

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

3. 然后使用 vg_lite_identity() 清除了变换矩阵当前的值。因此，第三次平移直接将源缓冲区 renderTarget 放置到目标缓冲区 rt 的（DEMO_PANEL_WIDTH/2.0，DEMO_PANEL_HEIGHT/2.0）坐标处，即屏幕的中心。

```
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. 循环调用 vg_lite_rotate()，其输入变量 rAngle 每次都会递增 0.3 度，以此在复制源缓冲区图像到目标缓冲区时，连续旋转其内容。

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

另外，vg_lite_scale() 函数在 10_Glyphs 中被调用，其将字符 '%' 放大了四倍，主要代码段为：

```
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 矩形传输 (Rectangle Blitting)

VGLite 还支持使用 vg_lite_blit_rect() 函数来复制局部的矩形区域，而不是使用上述的vg_lite_blit() 来复制整个源缓冲区。vg_lite_blit_rect() 函数需要一个额外的数组 rect，rect[0] 和rect[1] 是矩形区域左上角的x 与y 坐标，rect[0] 和rect[1] 是矩形区域的宽度与高度。

12_BlitRect中有一个包含数字 0 - 9 的源图像数据，其使用 vg_lite_blit_rect() 函数从源图像中分别复制四个矩形区域到目标缓冲区，各区域包含一个数字。输入数组 rect 分别被配置四次，以选择四个数字所对应的矩形区域，如 图 5 所示。

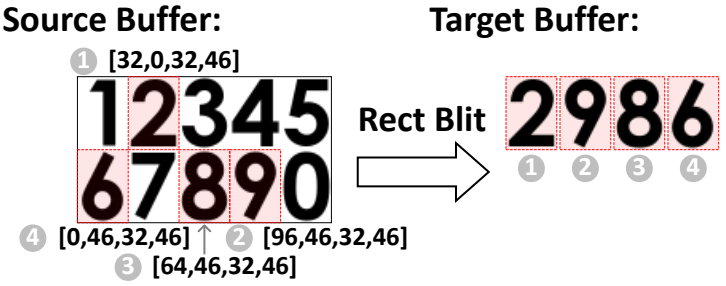

图 5. 矩形传输

1. 将大小为 (32, 46) 的矩形区域 2 从源缓冲区的 (32, 0) 坐标复制到目标缓冲区的 (0, 0) 坐标处。

```
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. 将大小为 (32, 46) 的矩形区域 9 从源缓冲区的 (96, 46) 坐标复制到目标缓冲区的 (34, 0) 坐标处。

```
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. 将大小为 (32, 46) 的矩形区域 8 从源缓冲区的 (64, 46) 坐标复制到目标缓冲区的 (34+34, 0) 坐标处。

```
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. 将大小为 (32, 46) 的矩形区域 6 从源缓冲区的 (0, 46) 坐标复制到目标缓冲区的 (34+34+34, 0) 坐标处。

```
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);
```

上述代码中，变换矩阵 `matrix` 用于将矩形区域复制到目标缓冲区的指定位置。

9 像素缓冲区

以上各操作，无论是存储图像的源缓冲区还是要渲染的目标缓冲区，缓冲区特征均由结构体 `vg_lite_buffer_t` 定义，其属性主要包括：

- width：缓冲区宽度（以像素为单位）。
- height：缓冲区高度（以像素为单位）。
- stride：步长，以字节为单位，表示上下行相邻像素的地址差，与缓冲区宽度、颜色格式、对齐方式等有关。
- tiled：缓冲区数据在内存中的布局，有两种选择：
  - VG_LITE_LINEAR：从左到右、从上到下的线性布局，更直观。
  - VG_LITE_TILED：数据以 4x4 像素块的形式排列（缓冲区地址和步长必须为 64 字节对齐）。这种布局在旋转缓冲区内容时具有较好的性能。
- format：缓冲区的颜色格式。它定义了颜色通道和相应的位宽，稍后将详细描述。
- memory：指向数据起始地址的指针。
- address：GPU 地址。
- image_mode：图像模式，稍后详细介绍。

对于矢量绘图，在设置其宽度 `width`、高度 `height` 和颜色格式 `format` 后，调用 `vg_lite_allocate()` 为 `vg_lite_buffer_t` 结构体分配内存。缓冲区的内存地址将是 `vg_lite_allocate()` 分配的内存地址，该函数将自动计算并设置步长 `stride`。大多数矢量绘图的例程都使用以下代码来实现上述过程，例如 01_SimplePath、 02_QuadraticCurves 和 03_Stroked_CubicCurves 等：

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

此方法也适用于渲染光栅位图的示例，即设置缓冲区的宽度 `width`、高度 `height` 和颜色格式 `format` 三个参数，然后调用 `vg_lite_allocate()`。原始的图像数据也需要手动复制到缓冲区分配的内存中。08_BlitColor 中的代码段使用了这一方法：

```
/* 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;
}
```

但是以上方法中，有两个内存块都存储相同的图像数据：一份是源图像，另一份是缓冲区分配的内存中的副本。下述方法可以避免这种内存浪费，即将缓冲区的内存指针直接指向源图像数据，而不是调用 `vg_lite_allocate()` 来为 `vg_lite_buffer_t` 结构体分配新内存，不过这样就必须手动设置该结构体的 `stride` 属性。13_PatternFill 和 14_UI 的代码段使用了这一方法：

```
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 颜色格式 (Color Format)

VGLite 包括但不限于以下颜色格式，源图像和渲染缓冲区均采用这些格式。

- 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、A 分别表示红色、绿色、蓝色和 alpha 通道。8、5、4、2 等数字表示相应位置通道的位宽。例如“RGBA5551” 表示具有 5 位红色、5 位绿色、5 位蓝色和 1 位 Alpha 通道的颜色。A8 或 A4 表示仅描述透明度的 8 位或 4 位 alpha 通道，没有 RGB 通道。L8 表示 8 位亮度，丢弃了颜色信息。

还有其他与 VG_LITE_RGBA8888 类似的颜色格式，例如 VG_LITE_ABGR8888、VG_LITE_ARGB8888、VG_LITE_BGRA8888。它们对应通道的位宽相同，但通道的排列顺序不同。另外，还有形如 VG_LITE_RGBX8888 的颜色格式，其中 X 表示 alpha 通道全部为 0xFF，即不透明。

图 6 显示了这些颜色格式的通道位宽和排列方式：

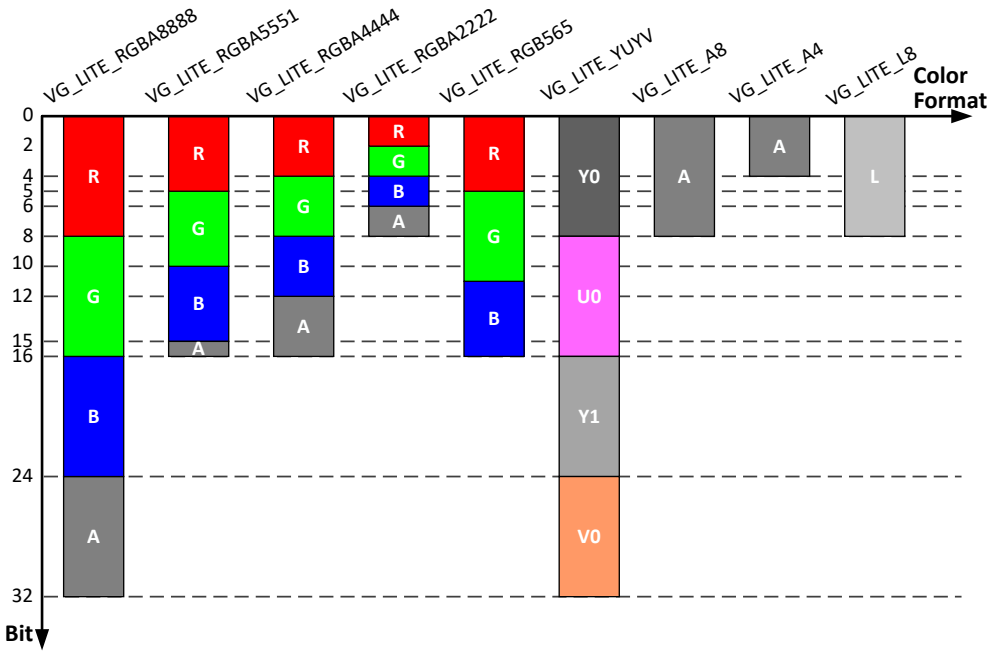

图 **6.** 颜色格式的通道位宽和排列方式

不同的颜色格式对应于不同的位宽、步长和像素对齐要求。08_BlitColor 采用上述不同的颜色格式显示同一张图片，通过宏 DROPPER_FORMAT 进行切换，再调用 vg_lite_blit() 函数显示，关键代码段如下：

```
#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);
```

图 7 展示了这些颜色格式的显示效果。

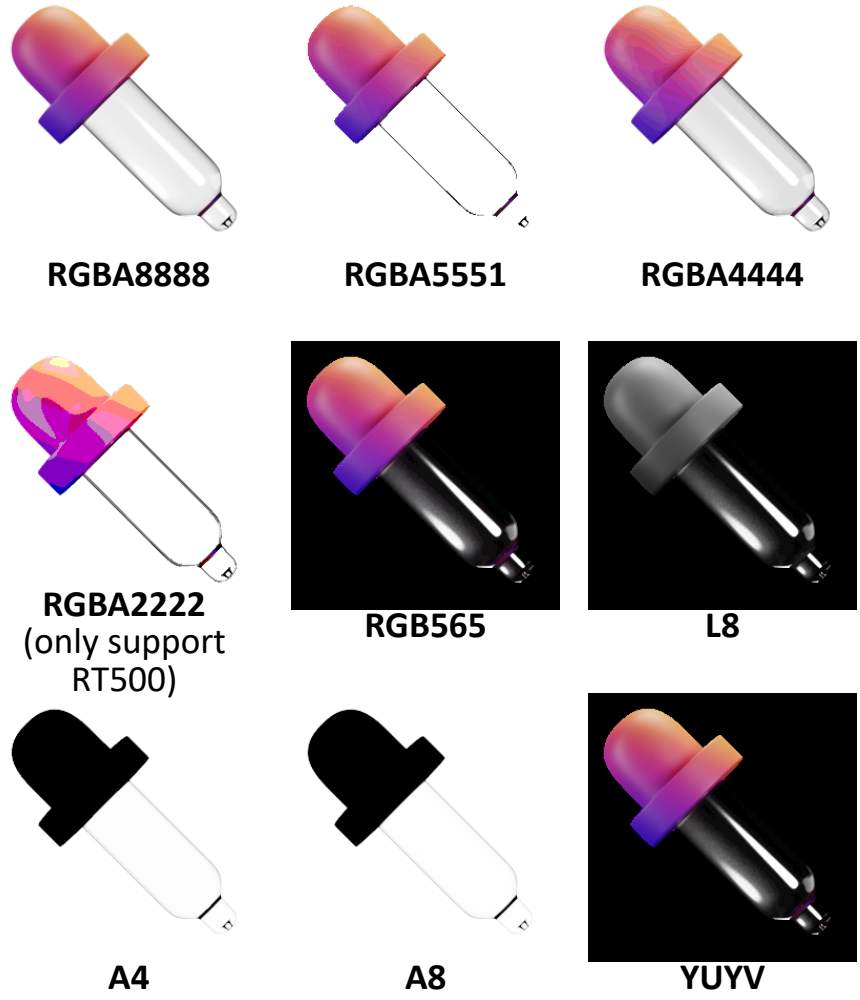

图 7. 不同颜色格式的显示效果

图 7 中, 不同的颜色格式有不同的特点：

- 较低位宽的通道占用较少的内存，但也会降低图像质量，导致出现明显的色带（尤其对比 VG_LITE_BGRA8888 和 VG_LITE_BGRA2222）。
- BGR565、L8、YUYV 没有 alpha 通道来遮盖背景。这样就会出现由 RGB 通道定义的原始背景，而非透明。在此图中，背景是黑色的。
- A8 和 A4 只有 alpha 数据，在没有 RGB 通道的情况下不可见，因此本例程将 `image_mode` 属性设置为 `VG_LITE_MULTIPLY_IMAGE_MODE` 以将其填充为黑色。稍后将详细描述此属性。
- BGRA5551 不支持半透明，因为 alpha 值仅为 0（透明）或 1（不透明）。

11 图像模式 (Image Mode)

每个 `vg_lite_buffer_t` 结构体都需使用三种图像模式之一：

- VG_LITE_NORMAL_IMAGE_MODE：直接使用混合模式（blending mode，下文介绍）绘制图像。默认选项，最常用。
- VG_LITE_NONE_IMAGE_MODE：忽略图像输入。

• VG_LITE_MULTIPLY_IMAGE_MODE：图像与填充颜色相乘。填充颜色在 vg_lite_blit() 函数中指定。

在 08_BlitColor中，两种图像模式分别应用于两个图像，如 图 8 所示。第一个为使用 VG_LITE_NORMAL_IMAGE_MODE 的图像，第二个为使用 VG_LITE_MULTIPLY_IMAGE_MODE 的图像。对于第二张图像，填充颜色为 0xFF00FF00（绿色），因为其他通道的值都乘以 0x00，所以结果图像中只保留了绿色通道。

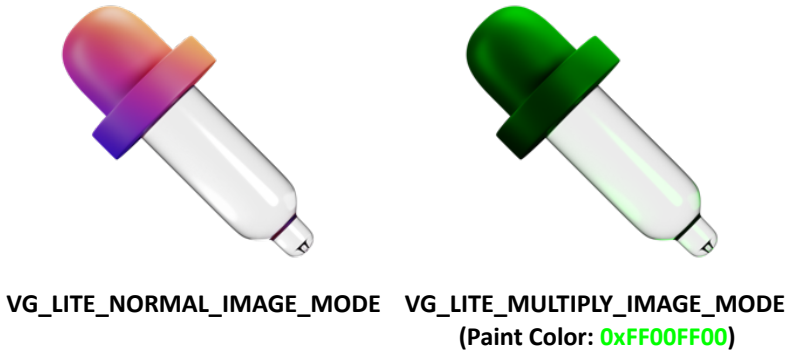

图 8. 不同的图像模式

VG_LITE_MULTIPLY_IMAGE_MODE 也有助于使不可见的颜色格式可见，即 A4、A8。此外，若想显示字符，alpha 数据 + VG_LITE_MULTIPLY_IMAGE_MODE 是不错的搭配，因为 alpha 数据占用的内存较少，而且使用 VG_LITE_MULTIPLY_IMAGE_MODE 可以轻松更改字符颜色。

在 10_Glyphs中，字符 % 的 alpha 数据就被绘制为 0xFF00FF00（绿色），并通过以下代码进行渲染显示：

```
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);
```

如果现在需要红色字符 %，只需 vg_lite_blit() 中的 0xFF00FF00（绿色）更改为 0xFF0000FF（红色）即可，如 图 9 所示。

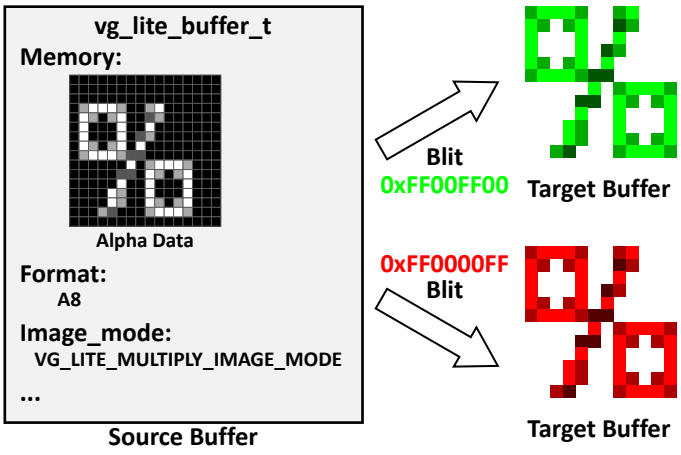

图 9. 不同的字符颜色

12 混合模式 (Blending Mode)

当将源 (src) 图像复制到目标 (dst) 图像时，有不同的混合模式（blending mode）：（S 和 D 表示源和目标的颜色通道，Sa 和 Da 表示源和目标的 Alpha 通道。

- VG_LITE_BLEND_SRC_OVER：在 dst 上显示 src，并且 dst 只能在 Sa 不为 1 的区域显示。公式为 S + (1 - Sa) * D。
- VG_LITE_BLEND_DST_OVER：与 VG_LITE_BLEND_SRC_OVER 相反，在 src 上方显示 dst，因此 src 只能在 Da 不为 1 的区域看到。公式为 (1 – Da) * S + D。
- VG_LITE_BLEND_SRC_IN：在 src 和 dst 重叠的区域显示 src，Da 会应用于 src 图像，其公式为 Da * S。
- VG_LITE_BLEND_DST_IN：与 VG_LITE_BLEND_SRC_OVER 相反，在 src 和dst 重叠的区域显示 dst，Sa 也应用于 dst 图像，其公式为 Sa * D。
- VG_LITE_BLEND_SCREEN：同时显示 src 和 dst，结果的颜色相比 S、D 一般更浅，其效果类似于将多个摄影幻灯片同时投影到同一位置。公式为 1 - [(1 - S) * (1 - S)] = S + D – S * D。
- VG_LITE_BLEND_MULTIPLY：同时显示 src 和 dst，然后替换 src 和 dst 重叠的区域为 src 乘以 dst。公式为 S * (1 – Da) + D * (1 – Sa) + S * D。
- VG_LITE_BLEND_ADDITIVE：简单将 src 和 dst 相加，公式为 S + D。
- VG_LITE_BLEND_SUBTRACT：与 VG_LITE_BLEND_ADDITIVE 模式相反，用 dst 减去 src，公式为 D * (1 - Sa)。

对于 09_AlphaBehavior 中的源缓冲区，左上四分之一的区域为不透明蓝色，其他区域为透明度为 0x7F 的黑色，通过以下代码设置：

```
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;
```

目标缓冲区被填充为不透明红色：

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

然后，使用上述各种混合模式，将此源缓冲区复制到目标缓冲区八次。

```
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);
......
```

源图像、目标图像以及结果图像如 图 10 所示。为了更计算方便，图中的颜色范围设置为 [0, 1.0]，但实际范围仍是 [0, 255]。为了直观展示，图 10 中的颜色格式表示为 (R, G, B)，实际仍为 VG_LITE_BGRA8888。

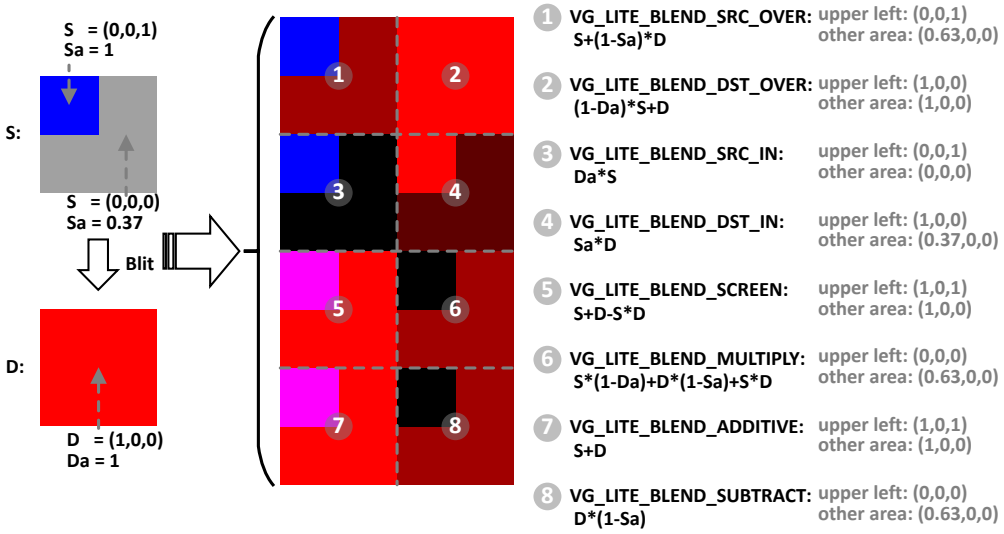

图 **10.** 不同混合模式的显示效果

13 路径

VGLite提供了一系列用于绘制路径的操作码，包括直线、二次曲线、三次曲线、圆弧等，且同时支持绝对坐标点和相对坐标点。表 1 展示了一些常用的操作码，不同操作码具有不同的参数：

表 **1.** 常用的路径绘制操作码

操作码	参数	描述
0x00	None	停止，闭合一个开放路径。
0x02	x, y	移动到给定端点 (x, y)，闭合开放路径。
0x03	Δx, Δy	利用给定的相对坐标 (Δx, Δy)，移动到 (startx + Δx, starty + Δy)。闭合开放路径。
0x04	x, y	画直线到给定坐标点 (x, y)。
0x05	Δx, Δy	利用给定的相对坐标 (Δx, Δy)，画直线到 (startx + Δx, starty + Δy)。
0x06	cx, cy, x, y	利用给定的控制点 (cx, cy)，绘制二次曲线到给定终点 (x, y)。
0x07	Δcx, Δcy, Δx, Δy	利用给定的相对坐标 (Δcx, Δcy) 所计算的 (startx + Δcx, starty + Δcy) 控制点，绘制二次曲线到终点 (startx + Δx, starty + Δy)，其由给定的相对坐标 (x, y) 计算得到。
0x08	cx_1, cy_1, cx_2, cy_2, x, y	利用给定的控制点 1 (cx1, cy1) 和控制点 2 (cx2, cy2)，绘制二次曲线到给定终点 (x, y)。
0x09	Δcx_1, Δcy_1, Δcx_2, Δcy_2, Δx, Δy	利用给定的相对坐标 (Δcx1, Δcy1) 和 (Δcx2, Δcy2)，计算控制点 1 (startx + Δcx1, starty + Δcy1) 和控制点 2 (startx + Δcx2, starty + Δcy2)，利用两个控制点绘制三次曲线到终点 (startx + Δx, starty + Δy)，终点由给定的相对坐标 (Δx, Δy) 计算得到。

操作码和参数必须具有相同的对齐方式。当参数类型为整数类型时，操作码的类型通常设置为与参数类型相同。但如果参数类型为 float，则操作码类型将设置为 uint32_t以保持相同的对齐方式。可以通过以下代码片段中的联合体来实现此方法：

```
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},
};
```

例程 01_SimplePath 简单绘制了一个三角形，使用 vg_lite_path_t 结构体描述了路径信息、边界框等。

```
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
};
```

路径数据数组 pathData 由操作码和后面的坐标参数组成：

```
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,
};
```

在 01_SimplePath 中，pathData 包含三个操作码：

- 2：移动到后续两个坐标（x 和 y）指定的点。
- 4：从前一个点绘制一条线到后续两个坐标（x 和 y）指定的点。
- 0：完成上面定义的路径，并闭合开放的路径。

02_QuadraticCurves 与 01_SimplePath 除了操作码其他基本相同，其 pathData 数组为：

```
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,
};
```

其中，操作码 6 用于绘制二次曲线，后跟一个控制点和一个终点的 x 与 y 坐标。

图 11 显示了 01_SimplePath 和 02_QuadraticCurves。由于操作码 6，第二个三角形拥有两条弯曲的边。图 11 中，路径端点由大灰点指定，二次曲线的控制点由小灰点和虚线标记。

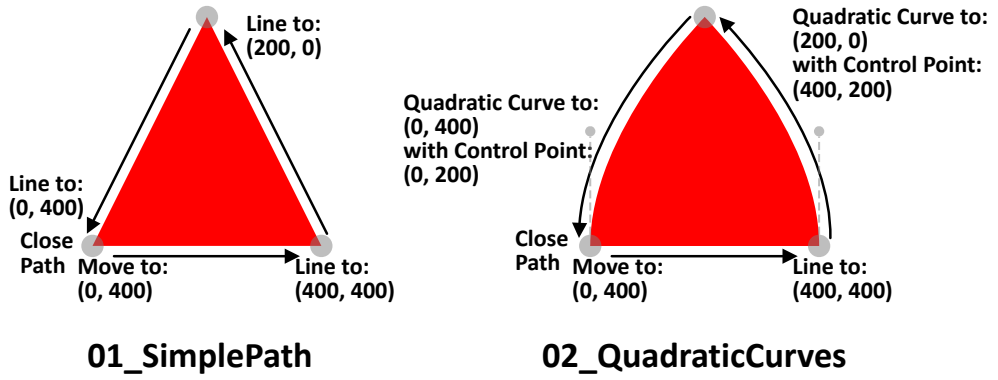

图 **11.** 由直线与二次曲线构成的路径

此外，03_Stroked_CubicCurves 通过操作码 6 绘制了三次曲线。该操作码需要三个点（六个参数）：控制点 1、控制点 2 和终点，例如：

```
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,
    ......
};
```

图 12 显示了 03_Stroked_CubicCurves的结果，三次曲线对应的控制点 1 和控制点 2 用数字、小灰点和虚线标记，路径端点由大灰点表示。.

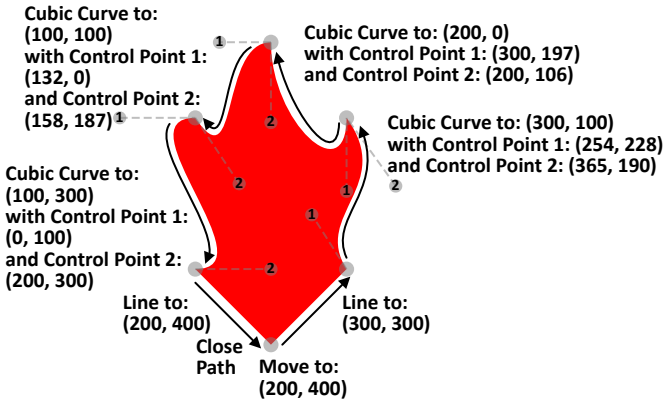

图 **12.** 三次曲线构成的路径

14 矢量绘制

当设置好路径数据后，`vg_lite_draw()` 将其绘制到指定的缓冲区，输入参数包括目标缓冲区`target`、路径`path`、变换矩阵`matrix`、混合模式`blend`、填充颜色`color`，填充规则`fill_rule`（稍后详细介绍）等。

例如，使用此函数绘制 01_SimplePath 中的三角形并填充其为 0xFF0000FF（红色）：

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

15 填充规则 (Fill Rule)

vg_lite_draw() 涉及两个填充规则，以此判断一个点是否在路径区域内，若在区域内，则将填充该区域：

- VG_LITE_FILL_NON_ZERO：非零填充。假设有一条射线从某一点出发到任意方向的无穷远处。计数从零开始，当路径从左到右穿过射线时加一，当路径从右到左穿过射线时减一。仅当结果不为零时，该点才位于路径区域内。
- VG_LITE_FILL_EVEN_ODD：奇偶填充。假设有一条射线从某一点出发到任意方向的无穷远处。计数从零开始，当路径从任意一侧与射线相交时加一。仅当结果为奇数时，该点才位于路径区域内。

07_FillRules 用上述两条填充规则分别绘制同一路径两次，代码如下：

```
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);
```

图 13 显示了两种不同的效果。对于 VG_LITE_FILL_NON_ZERO 模式，内部的菱形区域会被填充，因为有两个路径段从左到右穿过射线，计数大于零（即位于内部）。但是，当使用 VG_LITE_FILL_EVEN_ODD 模式时，该菱形区域不会被填充，因为有偶数个线段穿过该射线，计数为偶数（即位于外部）。

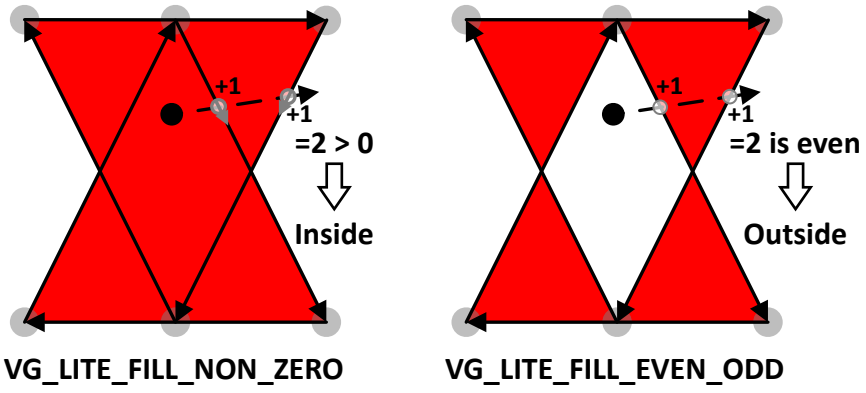

图 13. 两种填充规则

16 线条 (Stroke)

路径的线条（stroke）由 vg_lite_set_stroke() 函数配置，包括 color、width、end cap style、line join style、dash pattern、miter limit、dash phase 等属性。03_Stroked_CubicCurves 在循环中显示了由不同 end cap style 与 line join style 组合的九种线条：

```
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);
```

end cap style 包括三种：

- VG_LITE_CAP_BUTT：每条线段都垂直于每个端点的切线。
- VG_LITE_CAP_ROUND：添加一个半圆，其直径等于以每个端点为中心的线宽。
- VG_LITE_CAP_SQUARE：在每个端点追加一个矩形，其垂直长度等于线宽，平行长度等于线宽的一半。

图 14 中，红线表示真实路径数据，黑色和灰色是绘制的线条，其中灰色是通过 end cap style 添加的（灰色只是为了突显，这些区域的颜色与实际线条的颜色其实相同，下同）。

图 **14.** 三种 **cap styles**

即使一个线段长度为零，VG_LITE_CAP_ROUND 和 VG_LITE_CAP_SQUARE 也会为其添加 end cap，以此使该段可见。但是若选择 VG_LITE_CAP_BUTT 时，该段不可见。

line join style 决定了两条线交点的样式，也包括三种：

- VG_LITE_JOIN_MITER：延伸两个段的外边缘直到它们相交，以此连接它们。如果选择这种连接方式，则需要注意 vg_lite_set_stroke() 函数的

  ```
  stroke_miter_limit
  ```

  输入参数，因为较小的值可能会限制延伸长度。
- VG_LITE_JOIN_ROUND：附加一个扇形的圆，圆心为交点，直径等于线宽。
- VG_LITE_JOIN_BEVEL：用直线连接两个线段外边缘的两点。

在 图 15 中，红线表示真实路径，黑色和灰色是绘制的线条，其中灰色是通过 line join style 添加的。

图 **15.** 三种 **join styles**

vg_lite_set_stroke() 函数的输入参数 lines_dash_pattern 是数组，数组中各元素交替表示虚线中不同实线段和空白段的长度。如果数组元素数量为奇数，则忽略最后一个元素。

Dash pattern 由 03_Stroked_CubicCurves中的数组 stroke_dash_pattern决定，虚线由数组 stroke_dash_pattern 定义，如 图 16 所示。第一个元素 30.0f 表示第一条实现长度为 30，第二个元素 20.0f 表示后跟长度为 20 的空白段。以此类推，再后面是长度为 50 的实线和长度为 25 的空白段。然后，后续的线条将不断重复这四个元素。

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

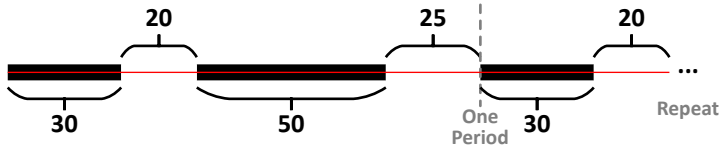

图 16. Dash pattern 显示效果

stroke_dash_phase 参数定义了虚线的起点。03_Stroked_CubicCurves 将其设置为 4，表示虚线将跳过长度为 4 的部分，如 图 17 所示。

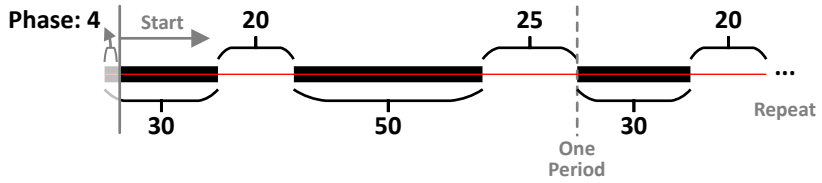

图 17. Dash phase is 为 4 时

如果 stroke_dash_phase 大于虚线数组中的所有元素的长度和，则将跳过第一个周期中的段，剩余的值继续跳过后续周期中的段。如 图 18 所示，假如 stroke_dash_phase 设置为 129，则先跳过第一个周期中长度为 125 的部分，然后再跳过第二个周期中长度为 4 的部分。这种情况下，显示效果实际上和 stroke_dash_phase 为 4 时的效果一样。

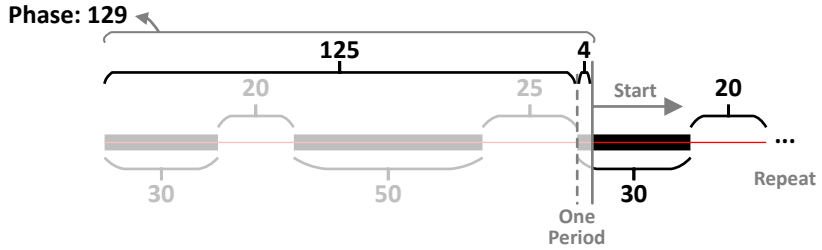

图 18. Dash phase 为 129 时

17 线性渐变 (Linear Gradient)

除了上述的纯色填充，路径也可以用线性渐变或径向渐变填充。

线性渐变的特征由 vg_lite_linear_gradient_t 结构体描述，主要包括：

- colors：定义渐变颜色的数组，32 位 ARGB 格式的 uint32_t 类型，而不是通常的 32 位 ABGR 格式的 vg_lite_color_t 结构。
- count：颜色的数量。
- stops：颜色节点偏移量，从 0 到 255。
- matrix：渐变的变换矩阵。

通常该结构体由 vg_lite_init_grad() 函数初始化，由 vg_lite_set_grad() 设置，由 vg_lite_update_grad() 函数更新。

线性渐变的变换矩阵通过 vg_lite_get_grad_matrix() 函数得到。vg_lite_identity()、vg_lite_translate()、vg_lite_scale()、vg_lite_rotate() 等涉及变换的函数均适用于此矩阵，以对相应的线性渐变做变换操作。

然后调用 vg_lite_draw_gradient() 来绘制路径，其由线性渐变填充。线性渐变之外的路径区域将用最近的节点颜色填充。

在 04_LinearGradient中，使用数组 stops 定义了三个节点，分布在 0、127 和 255。另一个数组 ramps 以 BGRA8888 格式定义了相应的颜色（分别为红、绿、蓝），相关代码如下：

```
/* 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};
```

以下代码执行初始化、配置、更新操作：

```
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);
```

然后获取渐变的变换矩阵，并通过变换函数将其放置在路径的区域，最后绘制：

```
/* 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);
```

图 19 是显示结果，其中两条灰色虚线标记了线性渐变的边界。显然，路径的左右两侧是在线性渐变的外部，分别使用了最近的节点颜色填充（纯红色和纯蓝色）。

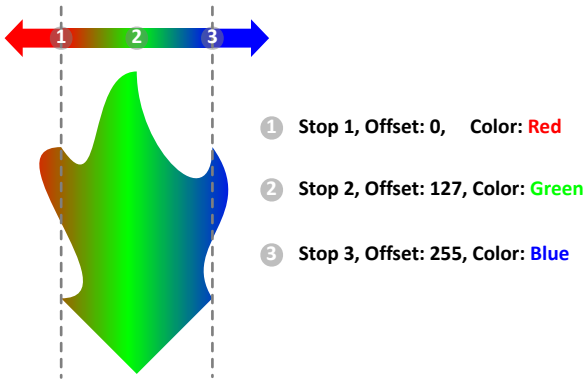

图 **19.** 线性渐变

这种方法绘制的线性渐变实际上是通过软件模拟的，vg_lite_draw_gradient() 底层实际是通过 vg_lite_draw_pattern() 实现的（后面会详细介绍）。i.MX RT500、RT1160、RT1170 均支持此方法。然而后面两种渐变绘制方法（章节 18 和 章节 19）是基于硬件实现的，并不支持 RT500。

18 径向渐变 (Radial Gradient)

径向渐变由 vg_lite_radial_gradient_t 结构体定义，主要包括：

- count：颜色数量，最多有 256 种。
- matrix：径向渐变的变换矩阵。
- RadialGradient：径向渐变参数。
- vgColorRamp：渐变的节点和对应颜色。
- SpreadMode：渐变传播模式，有四种：
  - VG_LITE_RADIAL_GRADIENT_SPREAD_FILL：用黑色填充渐变的外部区域。
  - VG_LITE_RADIAL_GRADIENT_SPREAD_PAD：用最近的节点颜色填充外部区域。
  - VG_LITE_RADIAL_GRADIENT_SPREAD_REPEAT：用重复的渐变填充外部区域。
  - VG_LITE_RADIAL_GRADIENT_SPREAD_REFLECT：用反转的渐变填充外部区域。

在 05_RadialGradient 中，展示了四个并排的小矩形，由径向渐变填充，各小矩形具有不同的 SpreadMode。

以下代码设置 vgColorRamp 参数，每个元素包含一个节点和其相应的颜色。每元素中的五个子元素分别代表节点偏移量、红色通道、绿色通道、蓝色通道和 alpha 通道值。虽然颜色值的范围是 [0, 1.0]，但实际上它们映射为 8 位，即范围为 [0, 255]。

```
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}
};
```

radialGradient 参数由 vg_lite_radial_gradient_parameter_t 结构体定义，包括五个数字。第一个数字是渐变半径，接下来的两个数字是渐变中心的 x 和 y 坐标，最后两个数字是焦点的 x 和 y 坐标。在此例程中，渐变半径设置为 256，渐变中心坐标等于焦点，均位于 (256, 256)。 在这种情况下，径向渐变是一个同心圆：

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

调用 vg_lite_set_rad_grad() 和 vg_lite_update_rad_grad() 函数分别配置和更新径向渐变：

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

vg_lite_get_rad_grad_matrix() 获取径向渐变的变换矩阵，vg_lite_identity() 、vg_lite_translate() 等变换函数都适合该矩阵。 在放置径向渐变于路径内之后，调用 vg_lite_draw_radial_gradient() 函数进行绘制：

```
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);
```

一旦发生错误，就会通过调用 vg_lite_clear_radial_grad() 函数来执行径向渐变的清理工作：

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

图 20 为显示结果，其中黑色虚线划分了四个矩形。两条灰色虚线标记了径向渐变的边界。四个矩形分别应用四种 SpreadMode，使得外部区域分别填充不同的颜色或渐变。

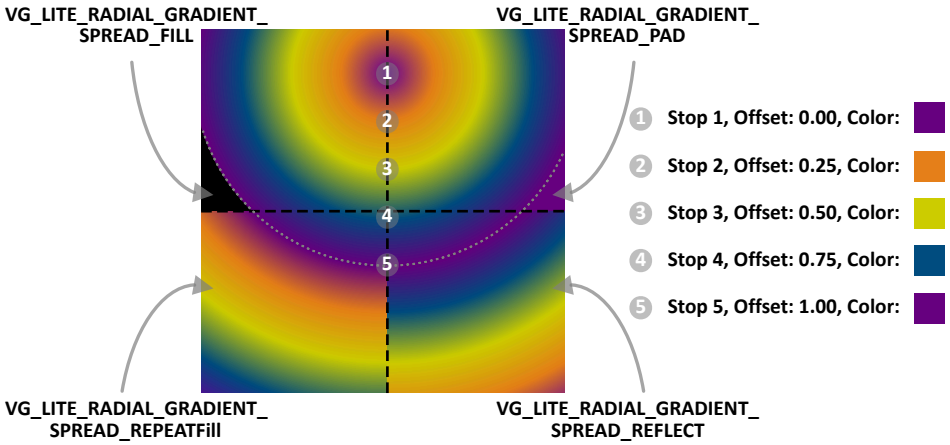

图 **20.** 径向渐变

19 扩展线性渐变 (Extended Linear Gradient)

与径向渐变类似，线性渐变还有另一个结构体 vg_lite_linear_gradient_ext_t，定义了扩展的线性渐变的参数，主要包括：

- count：颜色数量，最多 256 种。
- matrix：线性渐变的变换矩阵。
- LinearGradient：线性渐变参数。
- vgColorRamp：渐变的节点和对应颜色。
- SpreadMode：渐变扩散模式，与径向渐变的四种扩散模式相同。

涉及线性渐变的两个结构体（vg_lite_linear_gradient_t 和 vg_lite_linear_gradient_ext_t）间存在以下差异：

- vg_lite_linear_gradient_t 分别在两个数组中设置颜色和节点，颜色类型为 uint32_t，如 0xFFFF0000，节点范围为 [0, 255]。但 vg_lite_linear_gradient_ext_t 将颜色和节点都存储在vg_lite_color_ramp_t 结构体数组中，其中节点和颜色的四个通道（RGBA）都是浮点型，范围为 [0, 1.0]。
- vg_lite_linear_gradient_t 始终使用最近的节点颜色来填充渐变的外部区域。而 vg_lite_linear_gradient_ext_t 有四种不同的扩展模式来定义外部区域的填充模式。

在 06_LinearExtGradient 中有四个并排的小矩形，使用扩展线性渐变填充，各小矩形有着不同的扩展模式。05_RadialGradient 和 06_LinearExtGradient 的代码结构类似。

对于 vgColorRamp 参数，以下数组中也有五个元素，用于设置该线性渐变的五个节点和相应的颜色。每行五个成员分别代表节点偏移量和 R、G、B、A 颜色通道：

```
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}
};
```

linearGradient 参数影响线性渐变的方向。它由 vg_lite_linear_gradient_parameter_t 结构体定义，包含四个数字。前两个数字是起点的 x 和 y 坐标，后两个数字是终点的x 和 y 坐标。在该例程中，线性渐变的径向方向设置为从 (160, 100) 到 (480, 100)：

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

然后 vg_lite_set_linear_grad() 和 vg_lite_update_linear_grad() 函数用于配置和更新线性渐变。

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

该扩展线性渐变的变换矩阵通过 vg_lite_get_linear_grad_matrix() 函数获得，并将其放置在路径内。vg_lite_draw_linear_gradient() 函数最终绘制它：

```
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);
```

该线性渐变的清理工作由以下代码完成：

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

图 21 为显示结果，两条灰色虚线标记了线性渐变的边界。四个矩形被黑色虚线划分，四种展开模式使得四个外部区域被不同的颜色或渐变填充。

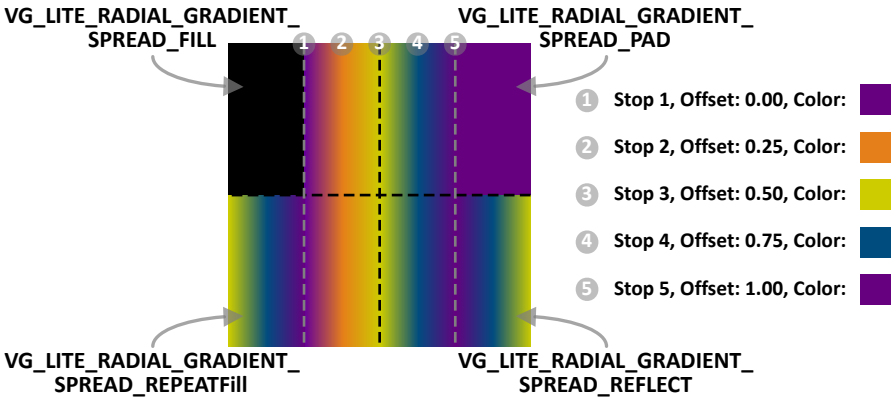

图 **21.** 扩展线性渐变

20 图像填充模式 (Fill Pattern)

除了纯色和渐变之外，当用 `vg_lite_draw_pattern()` 函数而非 `vg_lite_draw()` 函数时，图像也可以用于填充路径。

下面有两种图像填充模式（fill pattern），由 `vg_lite_pattern_mode` 枚举体定义：

- `VG_LITE_PATTERN_COLOR`：用指定的颜色填充图像外部的区域。
- `VG_LITE_PATTERN_PAD`：扩展图像边框的颜色以填充外部区域。

13_PatternFill 分别应用这两种模式填充同一图像，通过下面的代码实现：

```
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);
```

在上面的代码中，矩阵 `matPath` 用于转换路径 `path`。`image` 是存储图像数据的缓冲区，其变换由另一个矩阵 `matrix` 控制。

应用 `VG_LITE_PATTERN_COLOR` 模式时，需要指定图像外部区域的颜色，此处设置为 0xffaabbcc。但在另一模式下，该数值无关紧要，因为外部区域已被图像边框填充。

结果如 图 22 所示，其中灰色虚线标记了图像的轮廓。

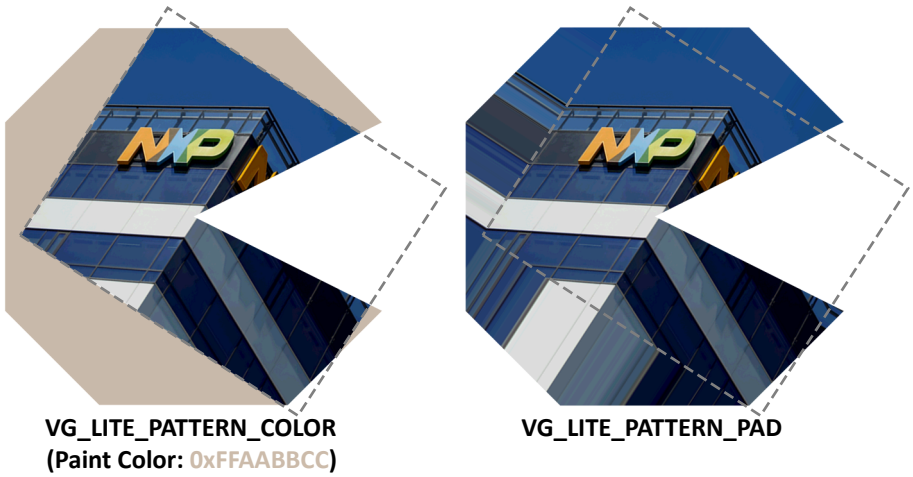

图 22. 两种填充模式

21 多任务

VGLite 基于 FreeRTOS，具有多任务调度功能。因此，使用不同任务显示 GUI 的不同部分是一个可行的解决方案。在实际项目中，可以使用其他任务接收并处理来自传感器的数据，再在 VGLite 任务中将它们显示在 GUI 上。

20_Multitask 创建了两个任务：vglite_task 和 vglite_task2，两个任务分别绘制两个旋转的老虎。每个任务都需要做一遍 VGLite 的初始化工作：

```
// 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 通过调用 VGLITE_GetRenderTarget() 来获取目标缓冲区，并在其上绘制老虎。vglite_task2 依次用蓝色填充 tmp_buf 中的三个缓冲区之一，对应缓冲区索引为 index。然后 vglite_task2 调用 vg_lite_finish()，将命令缓冲区提交给 GPU 并等待它完成。vglite_task2 绘制的缓冲区将被复制到 vglite_task 中的目标缓冲区。然后 vglite_task 调用 VGLITE_SwapBuffers() 来切换 framebuffer，以此实现两只老虎的显示。代码段如下：

```
// 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();
```

结果如 图 23 所示，其中灰色虚线标记的是 vglite_task2 写入的缓冲区内容。

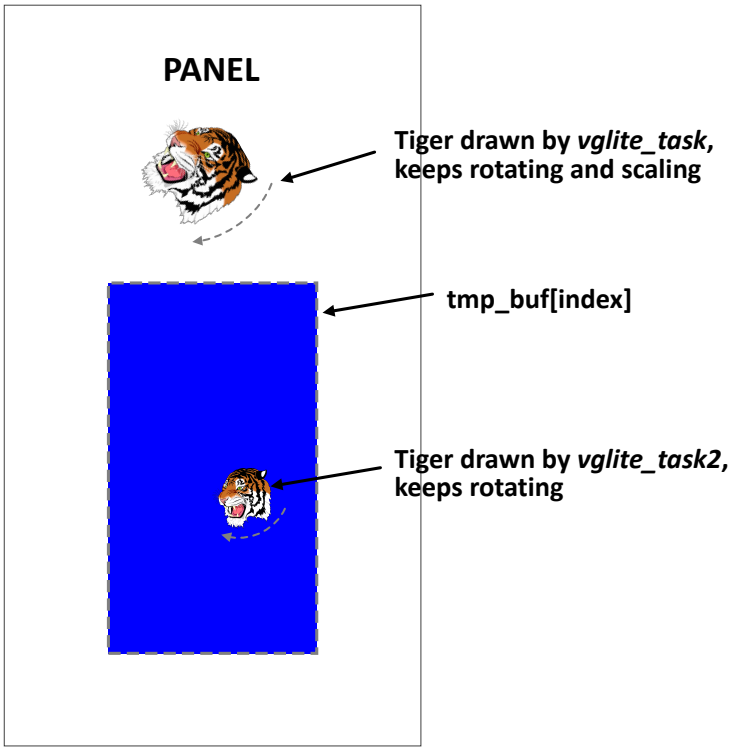

图 23. 多任务

22 参考资料

- i.MX RT VGLite API Reference Manual（文档 IMXRTVGLITEAPIRM）
- i.MX RT1170 Heterogeneous Graphics Pipeline（文档AN13075）
- Porting VGLite Driver for Bare Metal or Single Task（文档 AN13778）
- VGLite Driver Porting Guide（文档 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 修订记录

表 2 汇总了自初始版以来对本文档所做的更改。

表 **2.** 修订记录

文档号	日期	说明
AN14210_ZH v.1	2024 年 2 月 26 日	初次发布

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.

使用 **VGLite API** 在 **i.MX RT** 系列的图形编程

内容

1 简介 2
2 **VGLite** 例程架构 2
3 初始化/反初始化 3
4 清理 3
5 颜色类型 3
6 光栅传输 **(Raster Blitting)** 4
7 变换 **(Transformation)** 5
8 矩形传输 **(Rectangle Blitting)** 6
9 像素缓冲区 7
10 颜色格式 **(Color Format)** 8
11 图像模式 **(Image Mode)** 10
12 混合模式 **(Blending Mode)** 12
13 路径 13
14 矢量绘制 15
15 填充规则 **(Fill Rule)** 16
16 线条 **(Stroke)** 16
17 线性渐变 **(Linear Gradient)** 18
18 径向渐变 **(Radial Gradient)** 20
19 扩展线性渐变 **(Extended Linear Gradient)** 21
20 图像填充模式 **(Fill Pattern)** 23
21 多任务 24
22 参考资料 25
23 **Note about the source code in the document** 26
24 修订记录 26
Legal information 27

Date of release: 2024年2月26日
Document identifier: AN14210_ZH