AN13075

i.MX RT1170 异构图形流水线

Rev. 0 — 12/2020 Application Note

URL：https://www.nxp.com/docs/en/application-note/AN13075.pdf

目录

1 介绍

i.MX RT1170 crossover MCU 有多种选项，可在把图形内容发送到显示器之前生成、组合和混合图形内容：

- 2D 矢量图形GPU：该引擎可以渲染可缩放的矢量图形并合成和操作位图。
- PXP图形加速器：这个2D引擎可以操作和混合位图，其主要功能是缩放、固定角度旋转和色彩空间转换。
- LCDIF：显示控制器允许创建多达八个显示层，提供支持多格式的动态混合功能。

本应用笔记简要概述了每个引擎以及如何初始化它们、独立使用它们，最后介绍了一个用例，说明如何统一使用它们以获得性能和资源提升。每个阶段都有一个相关的软件项目，以便于理解。

2 2D 矢量图形GPU

i.MX RT1170 上的 GPU 是 Verisilicon 的 GC355 GPU 的一个实例。这是平台上唯一能够根据输入几何信息生成图形的加速器。

用户用图形 API 与此引擎交互。本应用笔记重点介绍 VGLite API。OpenVG 将在稍后提及。

2.1 VGLite

VGLite是一个轻量级的 2D 图形 API，具有较小的内存占用和较低的 CPU 开销。

VGLite 上的两个主要绘图函数是 vg_lite_draw（渲染矢量图基元）和 vg_lite_blit（渲染位图，也称为光栅图像）。

2.1.1 VGLite矢量渲染流水线

要渲染矢量图形，您需要以下六项信息：

1. 目标：这是将保存生成的渲染图像的目标缓冲区。
2. 路径数据：由一组坐标和路径段组成，用于定义将要渲染的几何图形的形状。
3. 填充规则：指定选择以纯色填充几何形状的方案。
4. 变换：通过 3x3 矩阵提供仿射变换。
5. 颜色：路径颜色使用 32bpp 值（alpha、红色、绿色和蓝色，每个使用 8 位）定义。路径覆盖的每个像素都将具有这种颜色。
6. 混合规则：路径将根据这些规则混合到扩展缓冲区内容。

从相关软件打开 **NXPLogo** 项目：这是一个说明如何使用信息块的增量示例，在此示例中，我们将使用矢量图形渲染 NXP 标志。代码分为七个步骤，由 TEST_STEP 宏控制。 第一步 (INITIAL_LOGO) 渲染 NXP 标志如下：

图 1. 第一步预期渲染结果：使用矢量图形的 NXP 标志

2.1.1.1 路径数据

路径数据是最重要的信息。 它定义了要渲染的形状。 路径数据以一对结构化数据片的形式提供:

1. 操作码：定义操作
2. 参数：操作使用的二维坐标数据。

字母N可以使用 10 个操作码定义，如下所示：

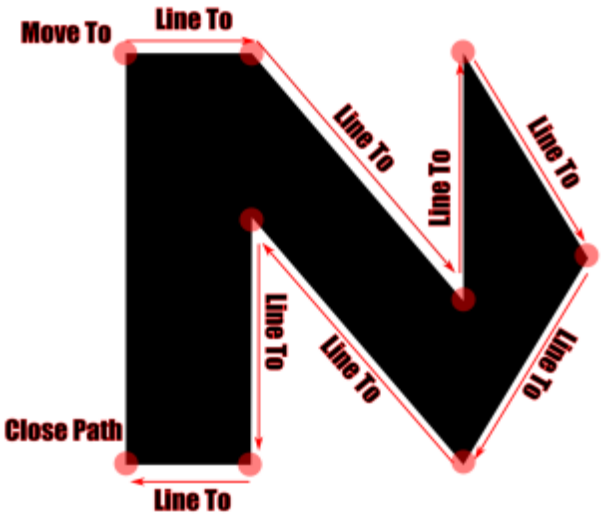

图 2. 字母N路径数据：命令

1. 移动到：字母N的起始点。
2. 八行 To 操作码定义 N 字母形状。
3. 关闭路径操作码，指示必须关闭打开的路径。

上面的每个操作码都使用以下坐标数据：

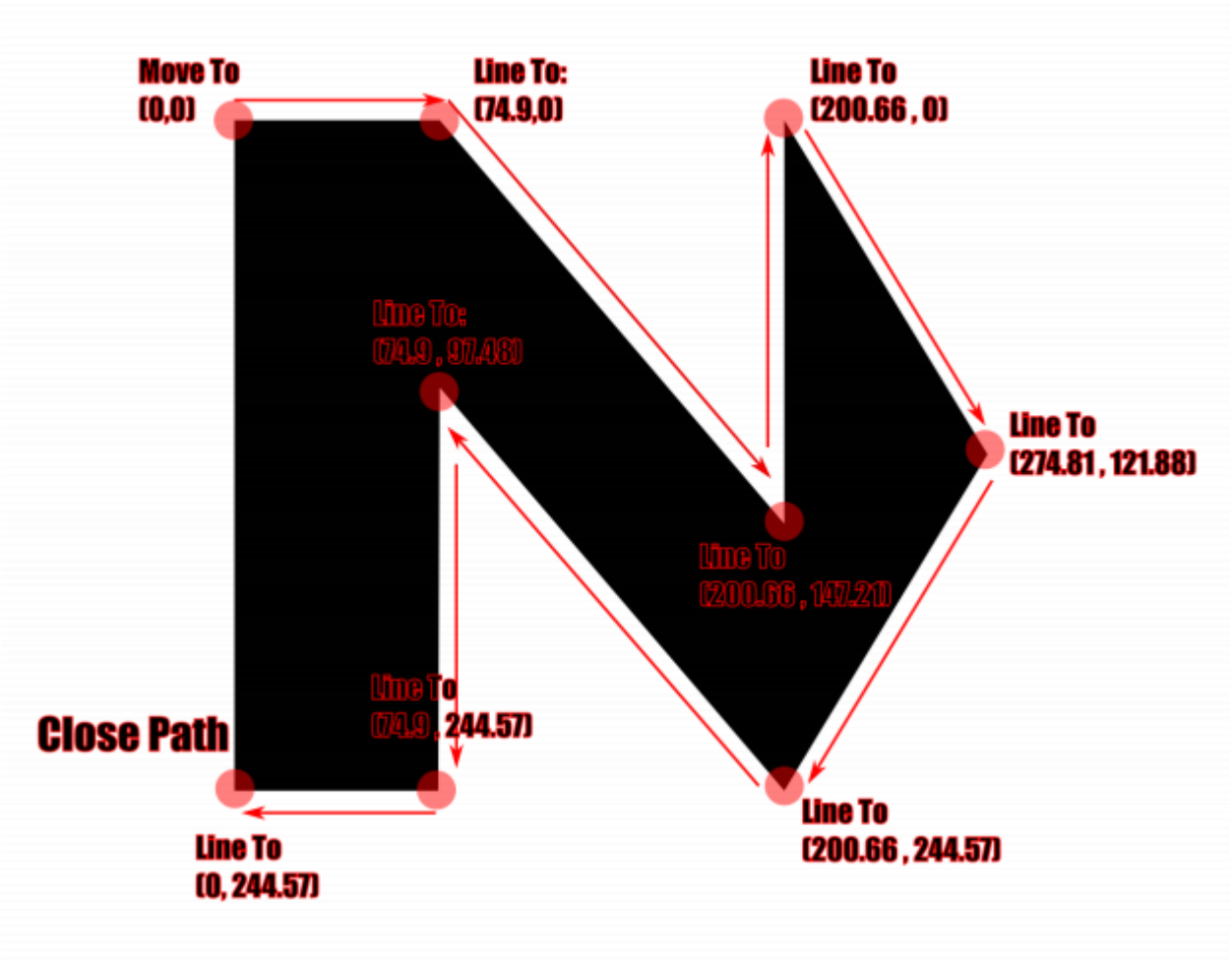

图 3. 字母N路径数据：命令和坐标

1. 移动到：使用两个坐标信息（x 和 y）。
2. 连接到：每次使用两个坐标点，它将定义一条从前一个点到新指定点的线。.
3. 关闭路径不需要使用任何额外的数据。

此路径的信息存储在代码中的 nPathData 数组中。

X 字母的定义与 N 字母类似。 Move to 操作码之后是一系列 Line To 操作码，最后是 Close Path 操作码：

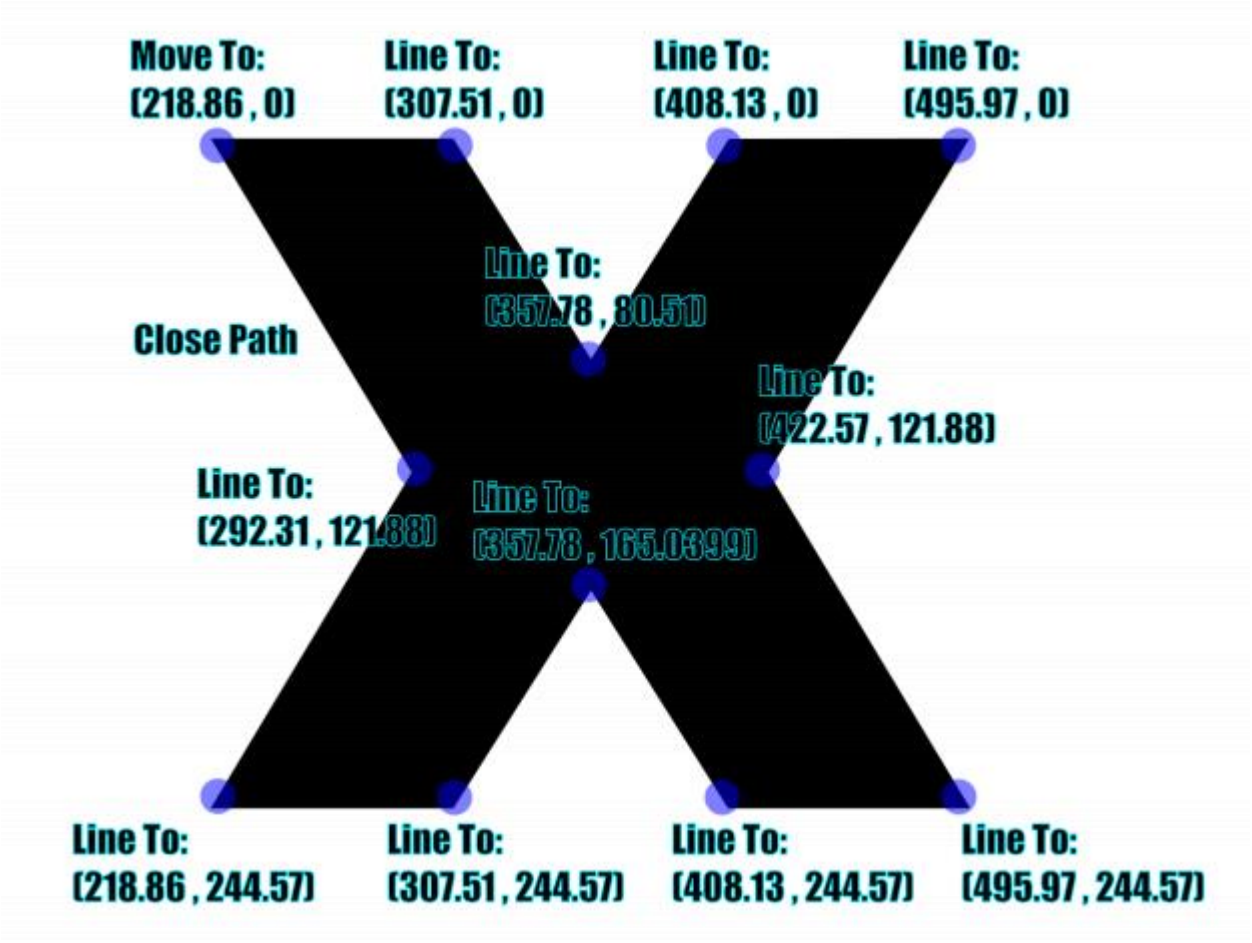

图 4. X 字母路径数据：命令和坐标

此路径的信息存储在代码中的 xPathData 数组中。

P字母更有趣。 它包含曲线和一个内部路径，允许我们稍后展示填充规则。

我们最初将其定义为在单个路径内定义的两个轮廓，一个外部图形和一个内部图形：

1. 移动到: (514.92, 0)
2. 连接到: (648.62, 0)
3. 三次曲线到: (648.62, 191.07)，三次曲线有两个控制点: c1 (740.62, 0) 和 c2 (740.62, 191.07)
4. 连接到: (514.92, 191.07)
5. 连接到: (514.92, 244.57)
6. 连接到: (440.06, 121.88)
7. 连接到: (514.92, 0)

   内部图形将在相同的路径结构中定义：
8. 移动到: (514.92, 61.14)
9. 连接到: (613.96, 61.14)
10. 三次曲线到: (613.96. 130.44) ，控制点位于 (639.4, 61.14) 和 (639.4, 130.44)
11. 连接到: (514.92, 130.44)

12. 连接到: (514.92, 61.14)

    最后，您关闭路径：

13. 关闭路径

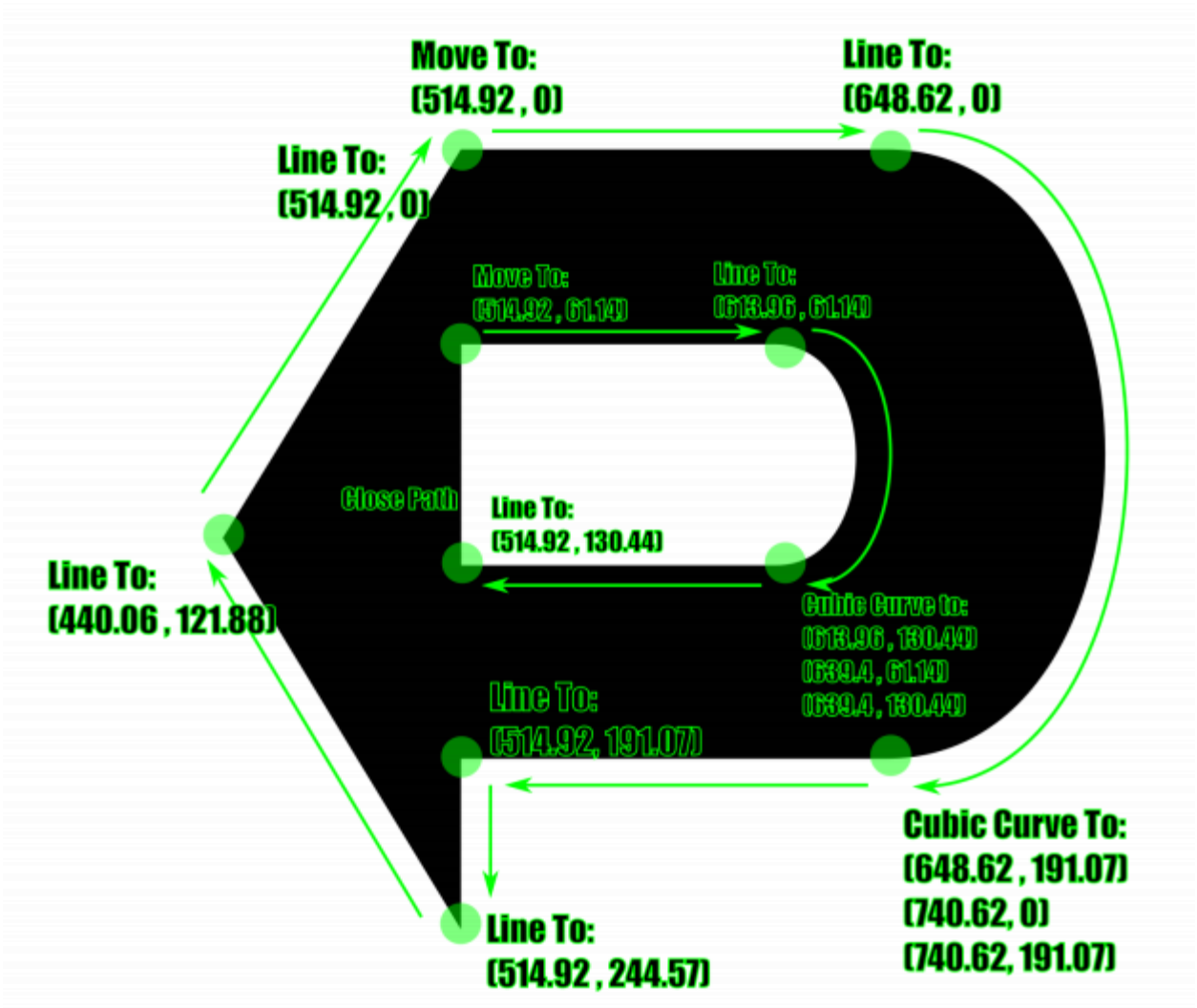

图 5. P 字母路径数据：命令和坐标

此路径的信息存储在代码中的 pPathData 数组中。

准备好每个路径的信息后，下一步是初始化 vg_lite_path_t 结构体：

```
vg_lite_init_path(&nPath, VG_LITE_FP32, VG_LITE_HIGH, sizeof(nPathData), nPathData, 0, 0, 720, 1280);
vg_lite_init_path(&xPath, VG_LITE_FP32, VG_LITE_HIGH, sizeof(xPathData), xPathData, 0, 0, 720, 1280);
vg_lite_init_path(&pPath, VG LITE FP32, VG LITE HIGH, sizeof(pPathData), pPathData, 0, 0, 720, 1280);
```

将三个字母渲染到显示器的代码如下：

```
vg_lite_identity(&matrix);
vg_lite_draw(rt, &nPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_NONE, 0xFF000000);
vg_lite_draw(rt, &xPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_NONE, 0xFF000000);
vg_lite_draw(rt, &pPath, VG LITE FILL EVEN ODD, &matrix, VG LITE BLEND NONE, 0xFF000000);
```

注意我们如何三次调用 vg_lite_draw（每个字母一次）。这是因为我们的路径是分开的。我们可以通过一次绘制调用来渲染它，但是我们的计划是改变路径颜色，因此从一开始我们就把它分成了几个绘制调用。

下图是 i.MXRT 1170 渲染结果直观的副本：

图 6. 实际渲染结果

2.1.1. 2 填充规则

我们将使用 P 字母来展示填充规则如何影响路径内部形状的解释方式。 将 TEST_STEP 宏更改为 FILL_RULE。

填充规则允许在路径中切孔，并允许 P 字母在中心有空间。

我们将 P 字母的填充规则从 VG_LITE_FILL_EVEN_ODD 更改为 VG_LITE_FILL_NON_ZERO， 如下：

```
vg_lite_draw(rt, &pPath, VG_LITE_FILL_NON_ZERO, &matrix, VG_LITE_BLEND_NONE, 0xFF000000);
```

不考虑路径中的内孔：

图 7. 填充规则 VG_LITE_FILL_NON_ZERO 导致 P 路径被完全填充

2.1.1.3 变换

变换是 GPU 的一个强大功能。它允许您为 2D 仿射变换定义 3x3 矩阵（仿射变换是线性变换后跟平移）。它还使我们能够描述投影变换。

VGLite 上有矩阵对象和函数，但先不使用它们来直观了解矩阵是如何定义的。将 TEST_STEP 宏更改为 TRANSFORM_INTUITION。

馈送到 VGLite 的矩阵由 9 个浮点值组成，排列在 3x3 数组中。

$$\begin{bmatrix} m00 & m01 & m02 \\ m10 & m11 & m12 \\ m20 & m21 & m22 \end{bmatrix}$$

图 8. VGLite 矩阵

矩阵按行主顺序排列，这意味着内存中的矩阵将如下所示：

Address	00	04	08	0C	10	14	18	1C	20
Element	m00	m01	m02	m10	m11	m12	m20	m21	m22

图 9. 内存中的矩阵

这与创建有九个位置的单个数组相同：

浮点数型 用户矩阵[9] = {1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0}；

VGLite 向量是列向量。在构建转换矩阵时考虑这一点。让我们构建一个旋转矩阵。假设我们要绕原点旋转 28 度。请记住，对于 VGLite 的坐标系，X 轴指向右侧，Y 轴指向下方，正角度顺时针旋转。

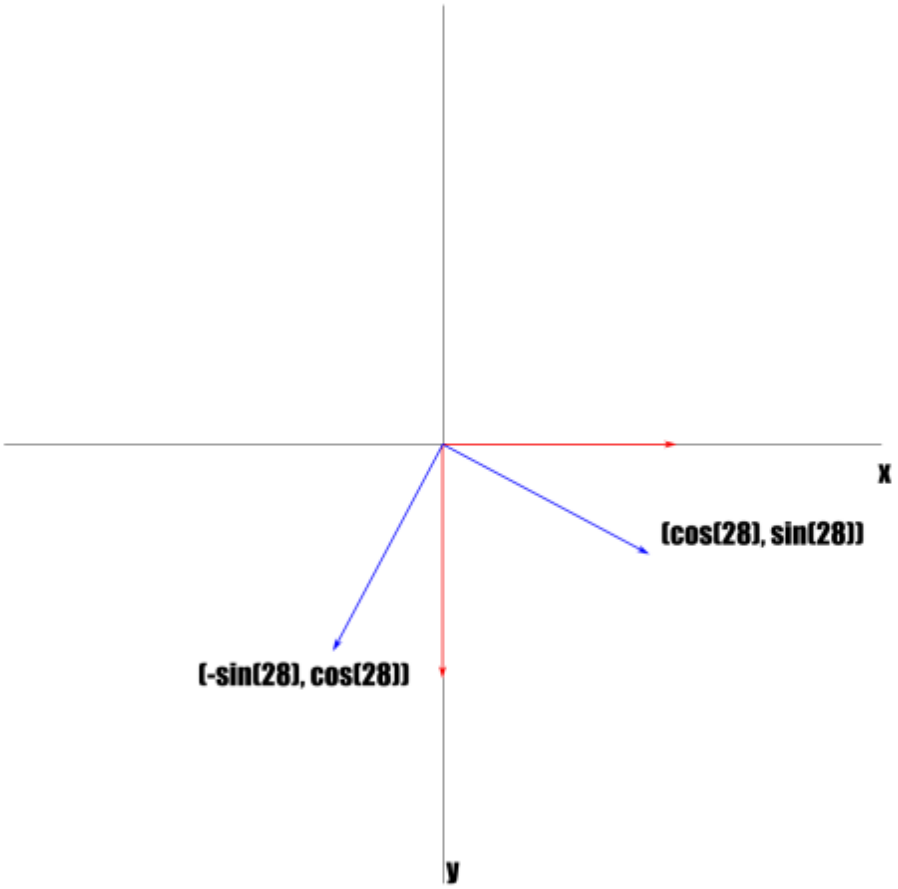

图 10. 基向量的预期变换

现在，为了构建矩阵，我们放置：

- X 轴到第一列的变换坐标。
- Y 轴到第二列的变换坐标。
- 由于我们没有翻转，我们将第三列保留为 (0,0,1)。

$$\begin{bmatrix} \cos(28) & -\sin(28) & 0 \\ \sin(28)) & \cos(28) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

图 11. 旋转矩阵

有了这个，我们现在可以将值存储在数组位置中：

```
userMatrix[0] = cosf(angleInRadians);
userMatrix[1] = -sinf(angleInRadians);
userMatrix[3] = sinf(angleInRadians);
userMatrix[4] = cosf(angleInRadians);
vg_lite_draw(rt,  &nPath, VG_LITE_FILL_EVEN_ODD,  (vg_lite_matrix_t *)&userMatrix,
VG_LITE_BLEND_NONE,  0xFF000000);
  vg_lite_draw(rt,  &xPath, VG_LITE_FILL_EVEN_ODD,  (vg_lite_matrix_t *)&userMatrix,
VG_LITE_BLEND_NONE,  0xFF000000);
  vg_lite_draw(rt,  &pPath, VG_LITE_FILL_EVEN_ODD,  (vg_lite_matrix_t *)&userMatrix,
VG_LITE_BLEND_NONE,  0xFF000000);
```

这种变换的渲染结果如下：

图 12. 变换渲染结果

它完全符合我们的预期。

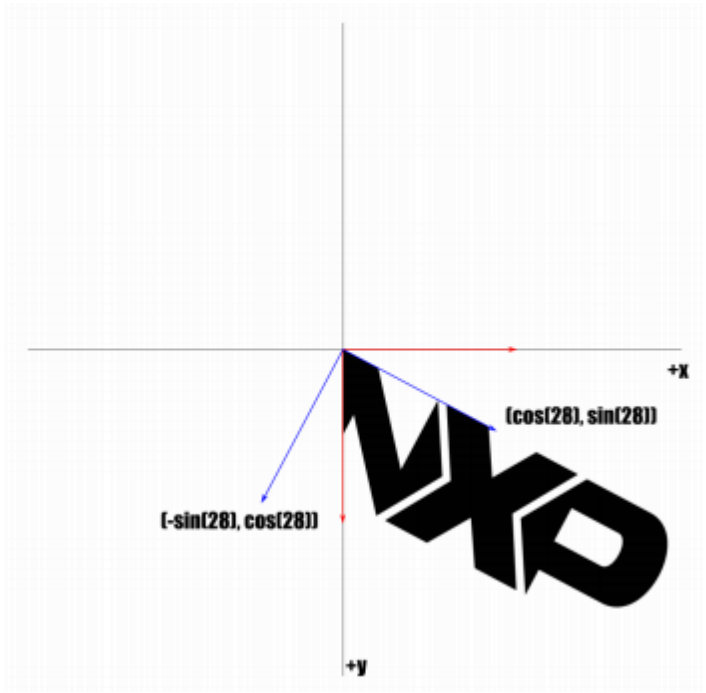

图 13. 渲染结果匹配基向量变换

既然您已经了解了背后发生的事情，请使用 vg_lite_matrix_t 结构和 vg_lite_rotate 函数来实现相同的目的。将 TEST_STEP 宏更改为 TRANSFORM_VG_LITE。

实现与上一步相同转换的代码简化如下：

```
vg_lite_rotate(28.0f, &matrix);
vg_lite_draw(rt, &nPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_NONE, 0xFF000000);
vg_lite_draw(rt, &xPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_NONE, 0xFF000000);
vg_lite_draw(rt, &pPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_NONE, 0xFF000000);
```

2.1.1.4 颜色

绘制时，您可以为每个路径指定一种颜色。 要将 NXP 标志与其真实颜色相匹配，请将 TEST_STEP 宏更改为 COLOR。 为每个字母指定不同的颜色：

```
vg_lite_draw(rt, &nPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_NONE, 0xFF10B4E8);
vg_lite_draw(rt, &xPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_NONE, 0xFFD9AE8B);
vg_lite_draw(rt, &pPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_NONE, 0xFF21D1C9);
```

这种颜色配置的渲染结果如下：

图 14. 显色结果

2.1. 1.5 混合

颜色参数中的 alpha 字段和混合函数都定义了 alpha 对路径和目标缓冲区的影响。 将 TEST_STEP 宏更改为 BLEND。

最常见的混合函数配置是 VG_LITE_BLEND_SRC_OVR，它对于每个像素，实现以下等式来定义像素的最终颜色值：

S + (1 - Sa) * D

“S”是路径颜色，“D”是存储在目标缓冲区中的颜色，“Sa”是路径的alpha值。

在下面的代码中，目标缓冲区被设置为纯红色，并且使用 0x7F 的 alpha 值渲染路径，这大约是 0.5 的不透明度：

```
vg_lite_clear(rt, NULL, 0xFF0000FF);
vg_lite_draw(rt, &nPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_SRC_OVER, 0x7F10B4E8);
vg_lite_draw(rt, &xPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_SRC_OVER, 0x7FD9AE8B);
vg_lite_draw(rt, &pPath, VG_LITE_FILL_EVEN_ODD, &matrix, VG_LITE_BLEND_SRC_OVER, 0x7F21D1C9);
```

每个字母颜色的结果是：

N:

- 基色：红： 0xE8 = 232, 绿：0x B4 = 180, 蓝： 0x10 = 16
- 结果颜色
  - 红 = 232 + (255 - 127) * 255 = 255 = 0xFF
  - 绿 = 180 + (255 - 127) * 0 = 184 = 0xB4
  - 蓝 = 16 + (255 - 127) * 0 = 16 = 0x10
- ABGR 中的结果颜色 = **0xFF10B4FF**

X:

- 基色：红： 0x8B = 139, 绿： 0xAE = 174, 蓝：0xD9 = 217
- 结果颜色
  - 红 = 139 + (255 - 127) * 255 = 255 = 0xFF
  - 绿 = 174 + (255 - 127) * 0 = 174 = 0xAE
  - 蓝 = 217 + (255 - 127) * 0 = 217 = 0xD9

- ABGR 中的结果颜色 = **0xFFD9AEFF**

P:

- 基色：红： 0xC9 = 201, 绿：0xD1 = 209, 蓝： 0x21 = 33
- 结果颜色
  - 红 = 201 + (255 - 127) * 255 = 255 = 0xFF
  - 绿 = 209 + (255 - 127) * 0 = 209 = 0xD1
  - 蓝 = 33 + (255 - 127) * 0 = 33 = 0x21
- ABGR 中的结果颜色 = **0xFF21D1FF**

让我们将这些颜色与一些渲染目标进行比较：

图 15. 颜色对比

每个字母上方的矩形是用我们计算的颜色所设定的，字母的颜色是我们缓冲区的实际结果。如您所见，它们几乎相同，但您会注意到略有不同。 这是因为我们的目标缓冲区被配置为 RGB565 并且在颜色转换过程中丢失了一些颜色信息。

2.1.1.6 线性渐变

您可以为每个路径中的颜色定义线性渐变。为此，请使用名为 vg_lite_linear_gradient_t 的附加结构和绘制函数变体 vg_lite_draw_gradient。 将 TEST_STEP 宏更改为 LINEAR_GRADIENTS。

当您创建渐变时，它将生成一个 256x1 的图像，然后该图像将自动应用于路径。您可以使用矩阵控制渐变的大小和方向。 每个梯度都有它的 3x3 矩阵。

对于每个渐变来说，都要定义颜色和停止点。停止点是 256x1 图像上的偏移量。 颜色被分配给这些停止点，停止点之间的颜色被内插。

我们将使用下面的三个渐变来定义 NXP 字母的颜色。 每个渐变都有自己的停止信息和颜色信息：

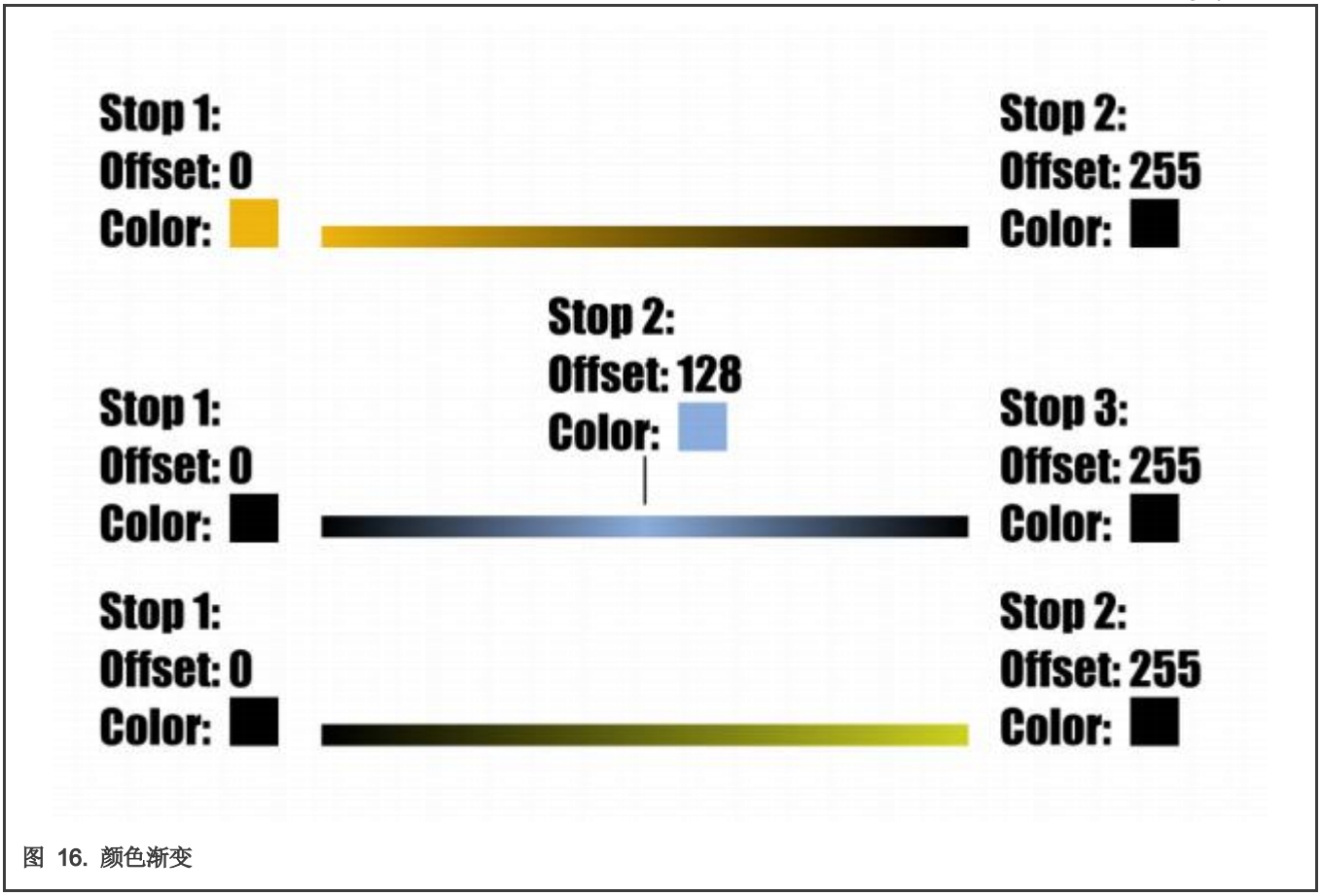

图 16. 颜色渐变

这可以转换为以下代码：

```
uint32_t nStopColors[] = {0xFFE8B410, 0xFF000000};
uint32_t nStops[] = {0, 255};
uint32_t xStopColors[] = {0xFF000000, 0xFF8BAED9, 0xFF000000};
uint32_t xStops[] = {0, 128, 255};
uint32_t pStopColors[] = {0xFF000000, 0xFFC9D121};
uint32_t pStops[] = {0, 255};
To initialize the gradient structures with this information, use the following procedures:
vg_lite_init_grad(&nGradient);
gradientMatrix = vg_lite_get_grad_matrix(&nGradient);
vg_lite_identity(gradientMatrix);
vg_lite_set_grad(&nGradient, 2, nStopColors, nStops);
vg_lite_update_grad(&nGradient);
```

当梯度被初始化时，它们必须被转换到它们必须在的位置。

图 17. 梯度变换

首先, 每个渐变必须水平转换到每个字母的开头。

移动是不够的，因为每个字母都大于 256。我们还必须缩放梯度。 这就是梯度矩阵派上用场的地方：

N 字母的宽度为 274.81，因此我们需要在水平轴上将渐变缩放 1.073。

X 字母的宽度为 277.140，比例尺必须为 1.0825。 我们还必须将其转换为水平轴上的 218.86。

P 字母的宽度为 277.380，因此比例尺必须为 1.0835，并在水平轴上转换为 440。

梯度变换很重要。 没有它，渲染结果将如下：

图 18. 没有梯度变换的渲染结果

2.1.2 VGLite光栅流水线

VGLite 还支持通过 vg_lite_blit* 函数绘制图像。 基本 blit 函数需要以下信息：

- 目标：保存 blit 函数结果的目标缓冲区。
- 源：生成目标缓冲区的原始数据缓冲区。
- 转换：以 3x3 矩阵形式提供，它允许您在将源组合为目标之前对其进行转换。有了这个，可以实现平移、缩放、旋转、剪切和透视等变换。
- 混合规则：指定如何使用 alpha 将源组合到目标中。
- 颜色：您可以在将源缓冲区渲染到目标之前乘以颜色。这对于使用A8或A4源格式渲染彩色字体很有用。在本文档中，我们不会使用此参数。

- 过滤器：如果硬件允许，您可以对位图用双线性过滤。在本文档中，我们不会使用任何类型的过滤。

2.1.2.1 简版 blit

从相关软件打开项目 VGLiteBlit。 我们将使用相关软件一一查看这些信息。 将 TEST_STEP 宏设置为 SIMPLE_BLIT。

前三个步骤的目标是将这个位图渲染到屏幕上：

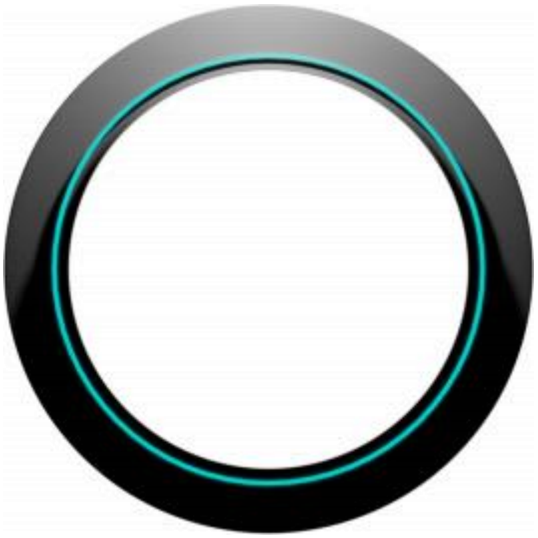

图 19. 位图渲染

此应用程序的目标缓冲区将由 VGLITE_GetRenderTarget 函数提供。此函数不是核心 VGLite API 的一部分。它的目的是为您提供一个 vg_lite_buffer_t，该 vg_lite_buffer_t 包裹在将发送到显示器的帧缓冲区周围。

此步骤的源缓冲区是一个 vg_lite_buffer_t，格式为 VG_LITE_RGBA4444，宽度和高度为 720 像素。

要将其渲染到屏幕上，请使用 vg_lite_blit 函数：

```
vg_lite_identity(&matrix);
vg_lite_blit(rt, &dial, &matrix, VG_LITE_BLEND_NONE, 0, VG_LITE_FILTER_POINT);
```

VGLITE_GetRenderTarget 返回的渲染目标是 rt，dial 是我们计划 blit 的位图。

vg_lite_blit 函数的一个有用功能是它会自动执行从源格式到目标格式的色彩空间转换。 在这种情况下，它将从 RGBA4444 转换为 RGB565。

当您渲染到屏幕时，您将看到：

图 20. 将 RGBA4444 缓冲区的 blit 渲染到 RGB565 缓冲区的结果，存在重色带

这看起来不像我们预期的那样（渐变不平滑）。 这种视觉伪像称为色带。

这是因为我们将 RGBA4444 缓冲区渲染为 RGB565 缓冲区。 这些缓冲区都没有足够的颜色信息来描述平滑的颜色渐变。

将 TEST_STEP 宏更改为 NO_BANDING。 在此配置下，我们会将 RGBA8888 缓冲区渲染为 RGB565 缓冲区。

更改的结果看起来更好：

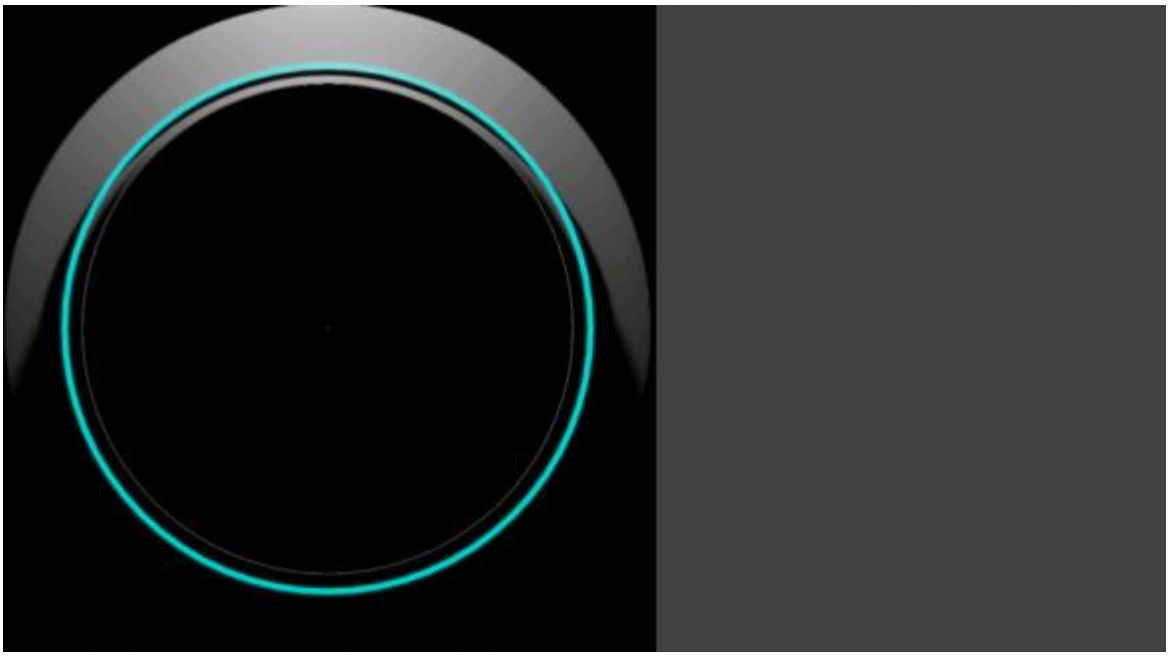

图 21. RGBA8888 缓冲区到 RGB565 缓冲区的 blit 渲染结果，存在轻微色带

它仍然不完美，因为我们的 RGB565 目标缓冲区仍然没有足够的信息来完全消除条带。

要完全删除它，请将渲染目标配置为 kVIDEO_PixelFormatXRGB8888。为此，请打开 display_support.h 文件并从以下内容更改以下宏指令：

```
#define DEMO_BUFFER_PIXEL_FORMAT kVIDEO_PixelFormatRGB565
#define DEMO_BUFFER_BYTE_PER_PIXEL 2
```

对此:

```
#define DEMO_BUFFER_PIXEL_FORMAT kVIDEO_PixelFormatXRGB8888
#define DEMO_BUFFER_BYTE_PER_PIXEL 4
```

再次渲染，最后，结果跟我们预期的一样：

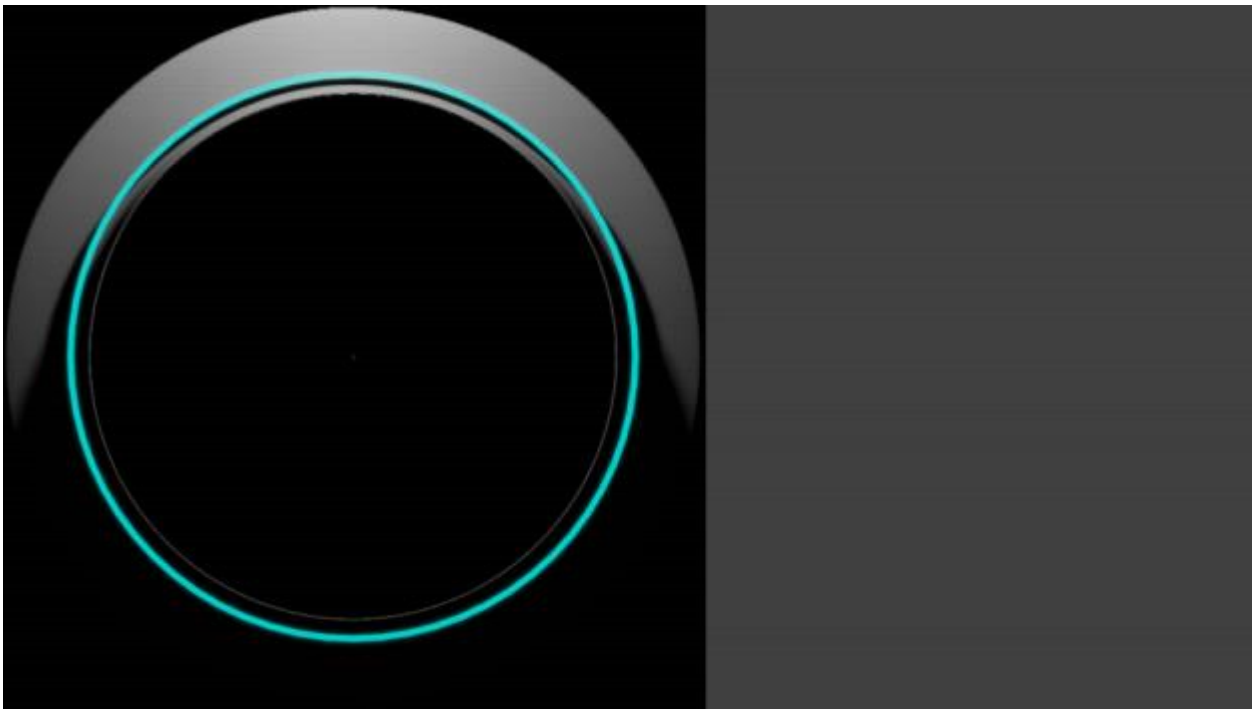

图 22. 将 RGBA8888 缓冲区的 blit 渲染结果为 RGBA8888，无色带

这是有代价的。从第一步到这一步，我们的内存使用量大幅上升：

我们的初始步骤为表盘使用了 1 个 RGBA4444 缓冲区（720*720*2 字节，或大约 1 MB）和 2 个 RGB565 目标缓冲区，每个缓冲区的大小为 720*1280*2 字节。这大约是 4.5 MB 的总和。

当我们采取所有步骤去除条带时，我们最终得到了 1 个用于表盘的 RGBA8888 缓冲区（720*720*4 字节或接近 2 MB）和 2 个 XRGB8888 目标缓冲区，每个缓冲区的大小为 720*1280*4 字节。这等于 9 MB 的总和。

这也会影响系统带宽的使用，因为每帧必须移动更多的内存。

在 LCDIF 章节中，您将学习如何去除带状伪影并且无需支付此内存使用量和带宽成本。

2.1.2.2 混合

我们渲染的表盘没有与背景混合。 将 TEST_STEP 宏更改为 BLENDING。

在重绘函数中，在调用渲染函数之前，我们用灰色清除目标缓冲区：vg_lite_clear(rt, NULL, 0xFF404040)。然而，当表盘被渲染时，720x720 图像的整个背景被阐释为黑色。这是因为我们禁用了混合。 通过将混合规则更改为 VG_LITE_BLEND_SRC_OVER，您将获得以下输出：

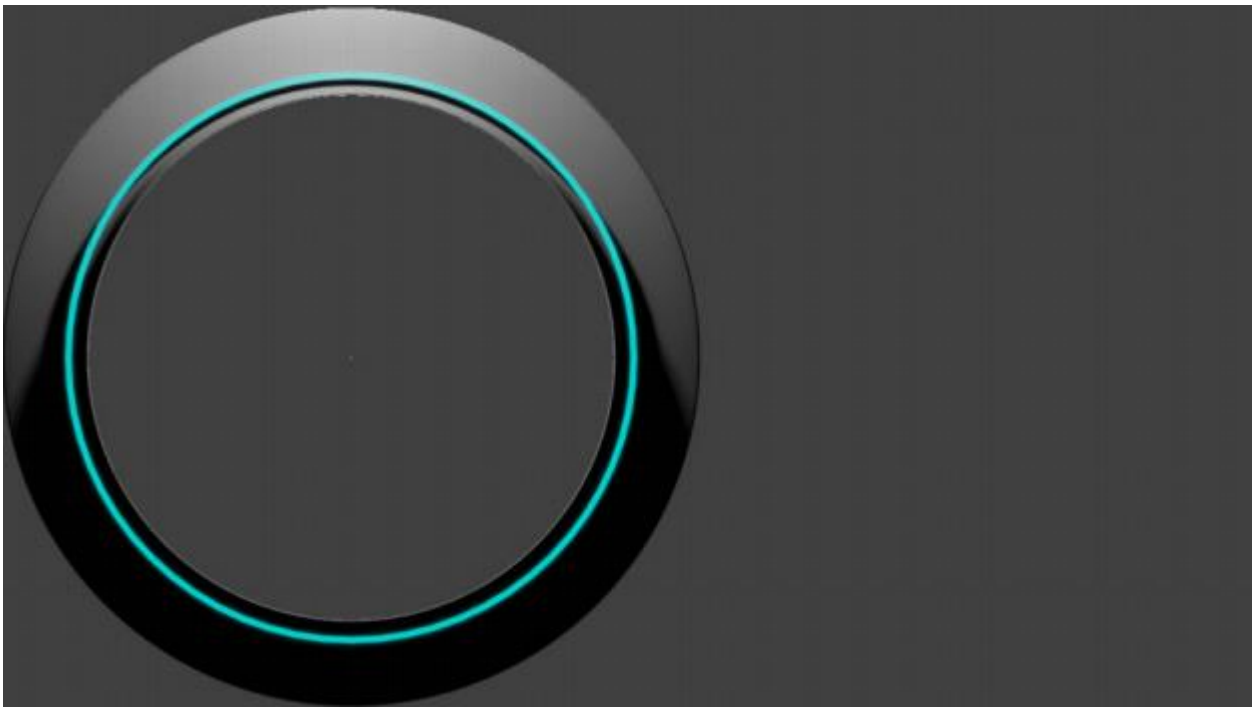

图 23. 混合

2.1.2.3 变换

转换位图的过程与转换路径的过程相同。使用 3x3 矩阵来描述要在位图上实现的仿射变换。执行此操作的矩阵对象和函数是相同的。 将 TEST_STEP 宏更改为 SIMPLE_TRANSFORMATION。

在这一步中，我们将缓冲区旋转 45 度：

图 24. 缓冲区旋转

但是，通过使用 3x3 矩阵，我们可以实现更复杂的转换，例如投影。

将 TEST_STEP 宏更改为 PERSPECTIVE_2_5_D。 在这一步中，我们构建了投影变换，映射以下点：

- (sx0, sy0) 到 (dx0, dy0)
- (sx1, sy1) 到 (dx1, dy1)
- (sx2, sy2) 到 (dx2, dy2)
- (sx3, sy3) 到 (dx3, dy3)

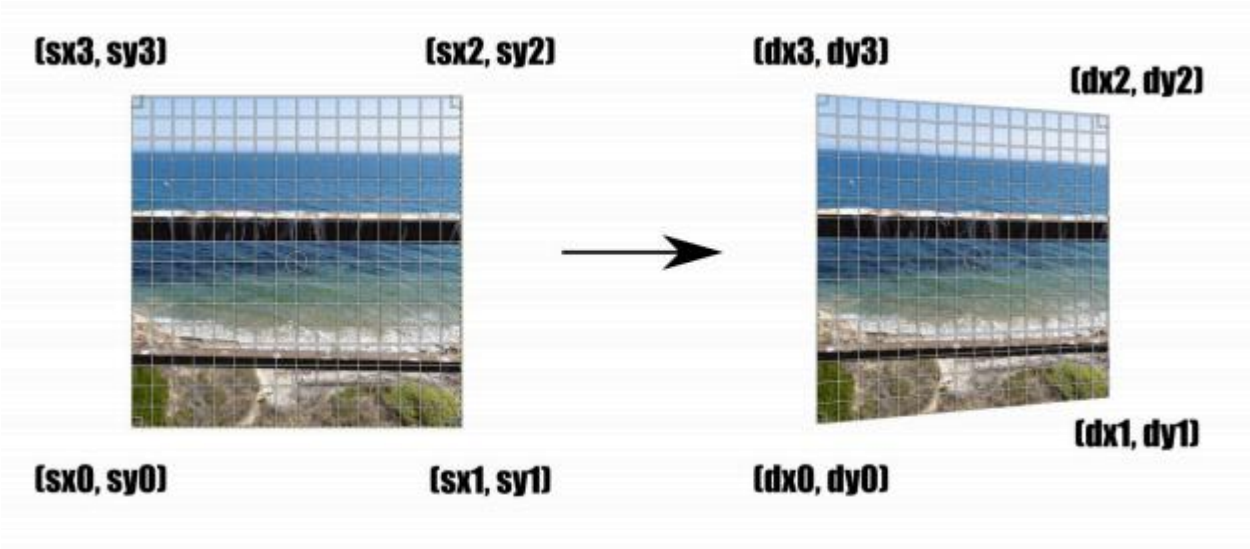

图 25. 投影变换

下图是变换结果的直接快照。

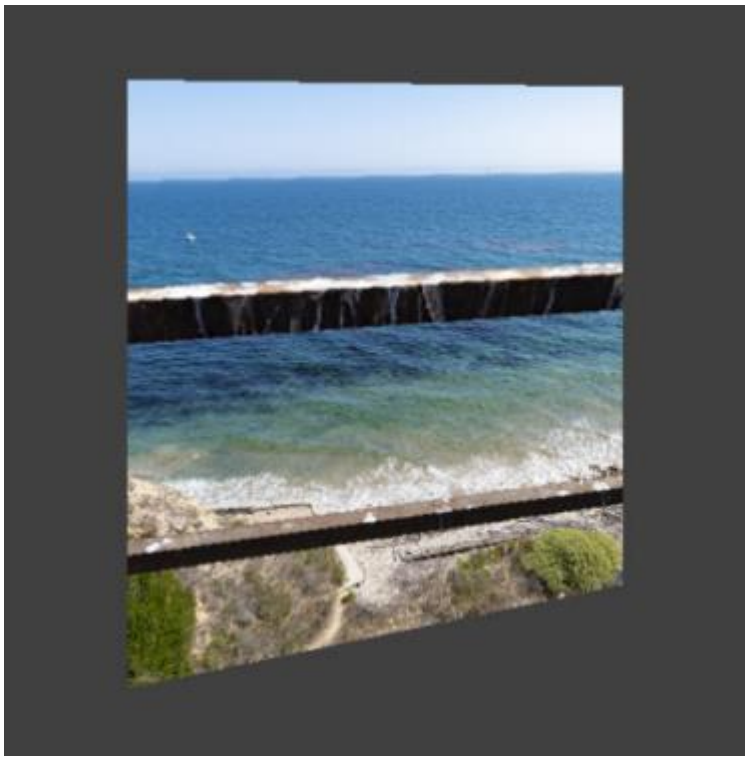

图 26. 变换结果快照

为了实现这一点，我们从OpenVG的实用程序库中移植了vguComputeWarpQuadToQuad。

2.1.2.4 光栅和矢量操作

因为 vg_lite_draw 也使用 vg_lite_buffer_t 结构作为目标，它允许你混合调用这些渲染函数来实现有趣的效果。 将 TEST_STEP 宏更改为 VECTOR_AND_RASTER。

在这一步中，我们创建了一个 vg_lite_buffer_t，我们将在其中渲染基于矢量的景观：

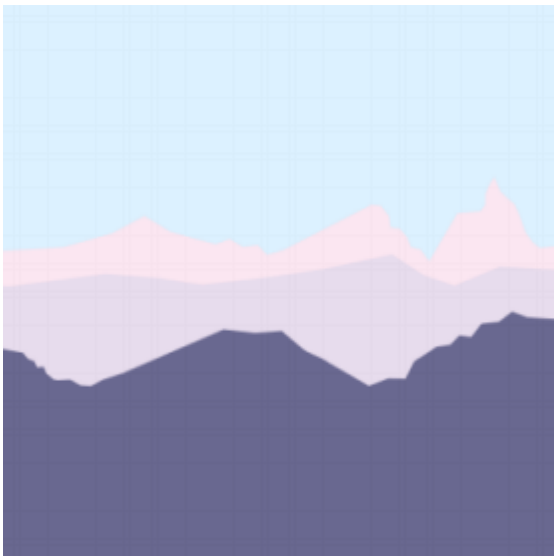

图 27. 基于矢量的景观

在同一个缓冲区之上，我们使用 VG_LITE_BLEND_DST_IN 混合配置渲染一个 alpha 蒙版。这让我们能够

丢弃所有未被蒙版上的白色像素覆盖的像素。

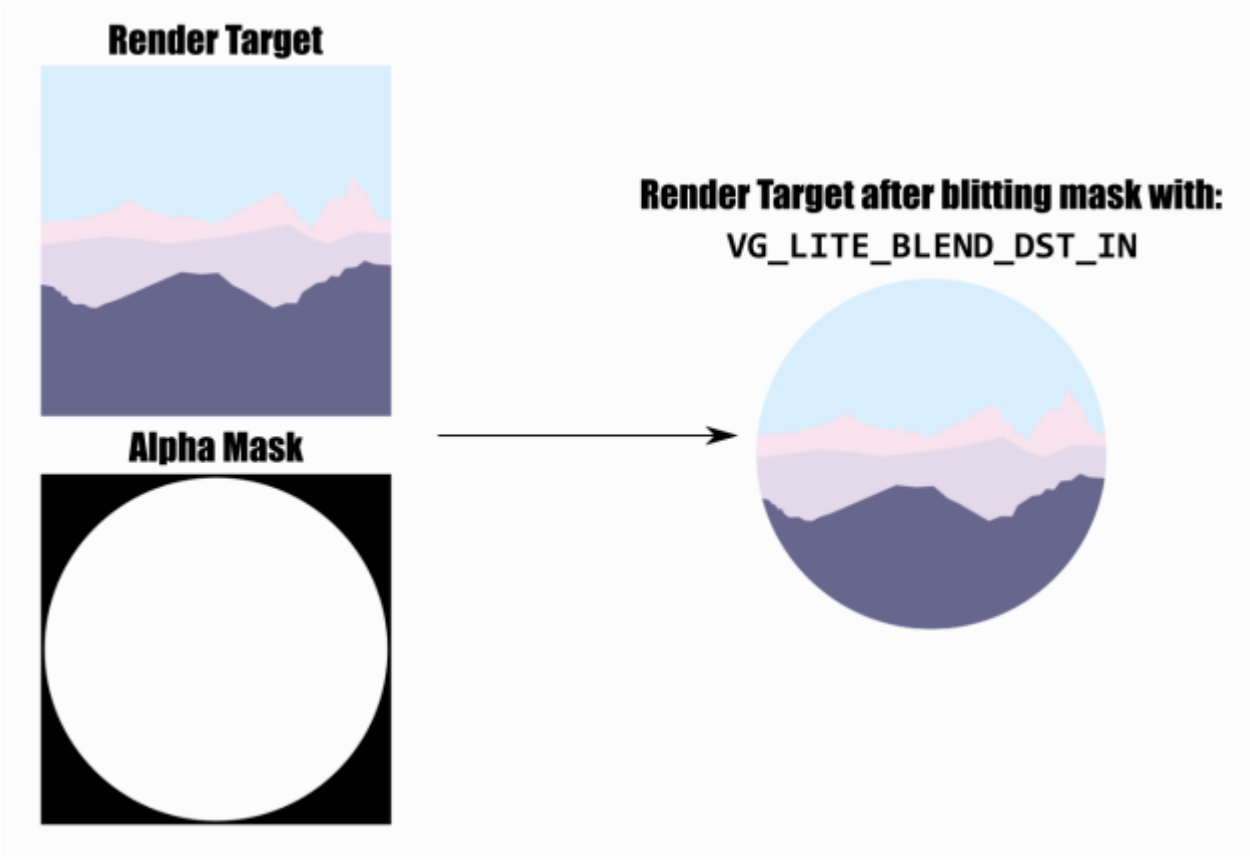

图 28. Alpha 蒙版

您的目标中的结果将如下所示：

图 29. 目标结果

3 PXP 2D 加速器

像素处理管道(PXP) 是一种强大的 2D 加速器，能够在将图形资源发送到显示控制器之前对其进行组合。它最重要的功能是 blitting、alpha 混合、色彩空间转换、固定角度旋转和缩放。

本章很简短，因为 PXP 已经有很好的文档记录，并且 SDK 中已经包含了多个示例。此处我们将对 PXP在把 Alpha 表面（AS）与处理过的表面（PS）混合方面的使用做有限的讲解。请记住，该模块的实际功能，与我们在这里使用到的模块功能相比，会有更多。

从应用笔记软件文件中打开“LCDIF_PXP”项目。

对于此示例，我们将使用 PXP 填充发送到显示控制器的缓冲区。总共有三个缓冲区：两个用于第 0 层（将不断重绘）和一个用于第 1 层（将最初填充）。

第0层配置为 720x640 大小和 ARGB8888 颜色格式。 我们将分配两个缓冲区，因此 PXP 仅渲染到当前未显示的缓冲区。

第1层配置大小为 144x394 和 ARGB8888 颜色格式。

下图对图层进行颜色编码，以显示我们想要显示的图层：

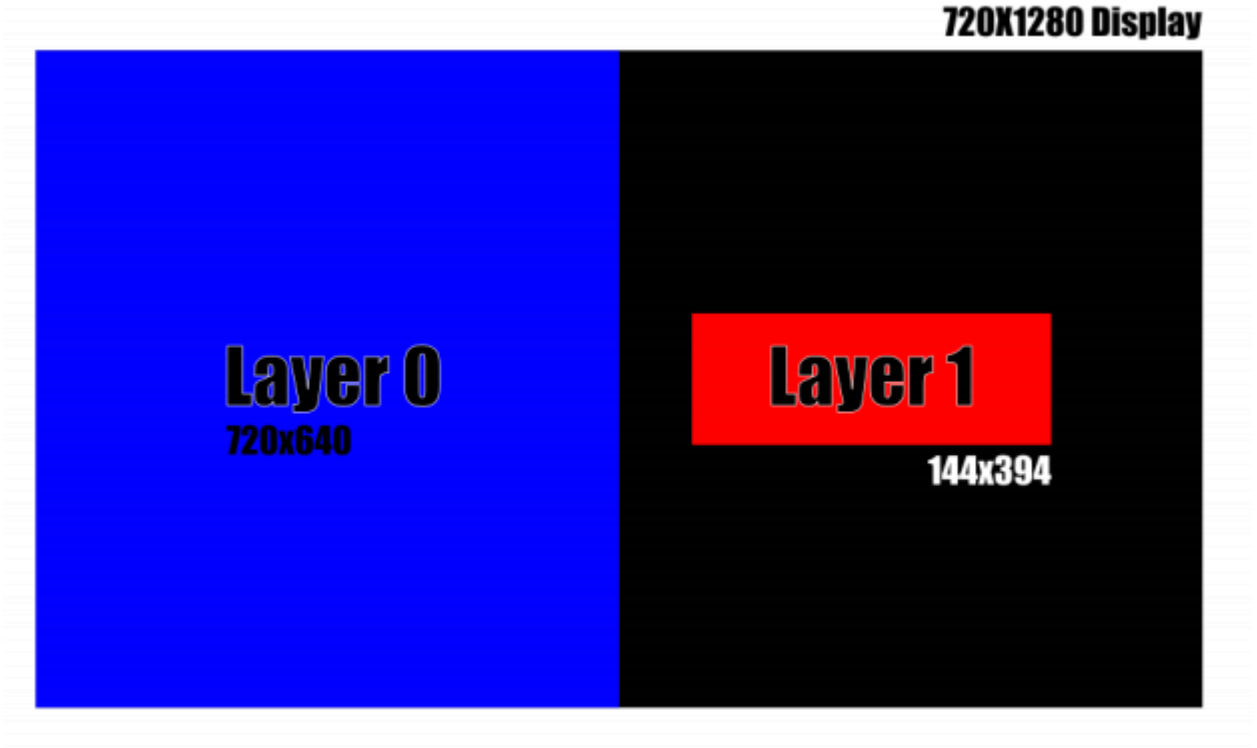

图 30. 图层颜色代码

第0层布局分为三个 PXP blit 操作：

第一个 blit 将 PS 配置为指向背景图像。 AS 将指向一个标签，输出缓冲区将指向第 0 层的后台缓冲区（后台缓冲区是当前未显示的缓冲区）。

这将在输出缓冲区中混合 AS 和 PS。

图 31. 混合 AS 和 PS

当的一个blit 结束时，第二个blit准备。在这种情况下，PS 和输出缓冲区都将指向 第0层后台缓冲区，而 AS 将指向标签编号 2。这有效地获取了我们之前的渲染结果并在这组合了新标签。

图 32. 相同的 PS 和输出缓冲区

对于最后的第 0 层合成步骤，我们做同样的事情，但我们使用第三个标签。

图 33. 第 0 层的第三个 blit

对于第 1 层，我们执行单个 blit 来填充其缓冲区，并且我们再也不会用到它。

显示和填充两个图层的结果如下：

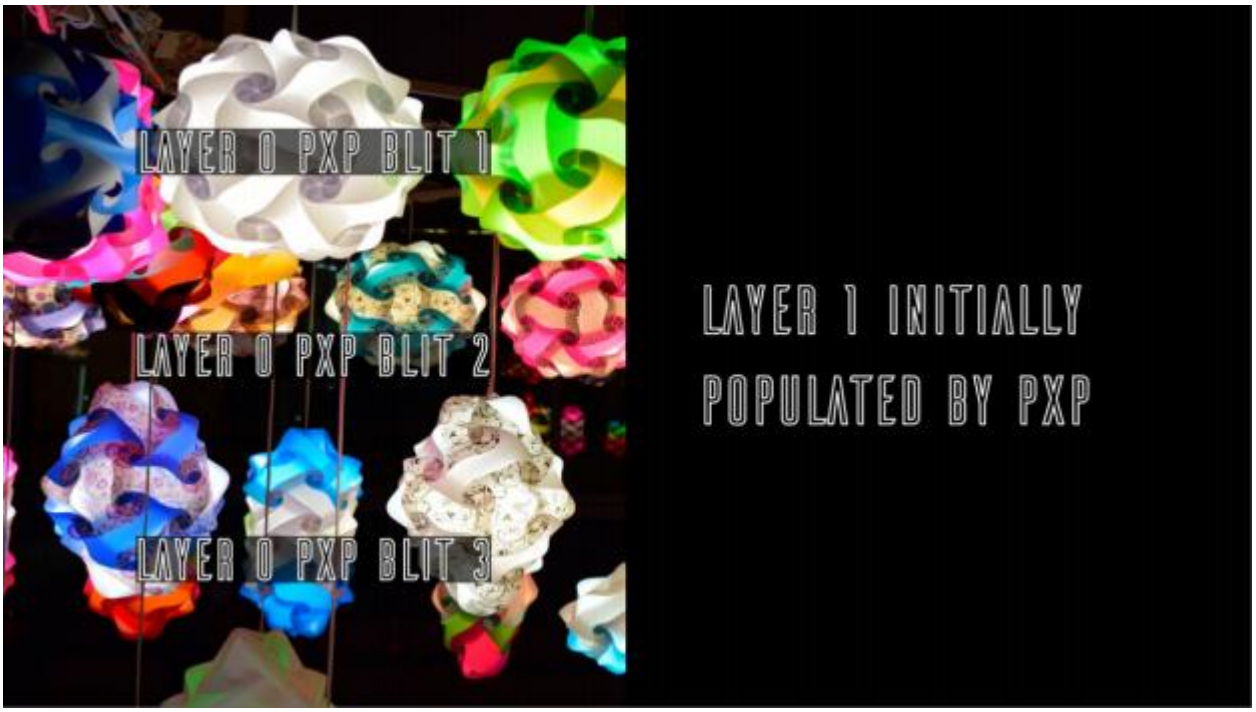

图 34. 显示两个图层的结果

4 LCDIFV2 显示控制器

LCDIF 模块获取内存缓冲区并将它们发送到显示控制器。 i.MXRT1170 的 LCDIFv2 模块最多支持八层，可以即时混合，无需任何其他加速器干预。

每个层都可以用不同的颜色格式、大小和位置进行配置，并且它可以从内存映射中的任何位置获取缓冲区。

在支持的格式中，有一种特别有用（Index8BPP）。这允许您使用颜色查找表和索引数组定义每像素 32 位的图像。用这种方法有一个优势，就是无需为此花费内存即可定义 ARGB8888 缓冲区。

请参阅第 2.1.2.1 节的表盘和我们遇到的带状伪影。 让我们再次回顾它。

我们将使用两个数组保存图像，而不是将图像保存为 720x1280 uint32_t 数组（其中数组的每个元素代表图像中每个像素的颜色）。我们的第一个数组将是一个 256 uint32_t 数组，我们将在其中为图像存储尽可能多的颜色。 第二个数组将包含 720x1280 个 uint8_t 值，其中每个值都是第一个数组的索引。

图 35. 表盘右侧的调色板是颜色查找表

任何最新的图像处理软件都允许您创建颜色查找表。 在上图中，在表盘的右侧，您可以看到根据图像创建的颜色表。

LCDIFV2 模块将根据 uint8_t 数组中的索引获取适当的颜色。 如果您的图像没有超过 256 种颜色（或没有超过这个数字太多），那么视觉效果将等于或接近您的原始图像。

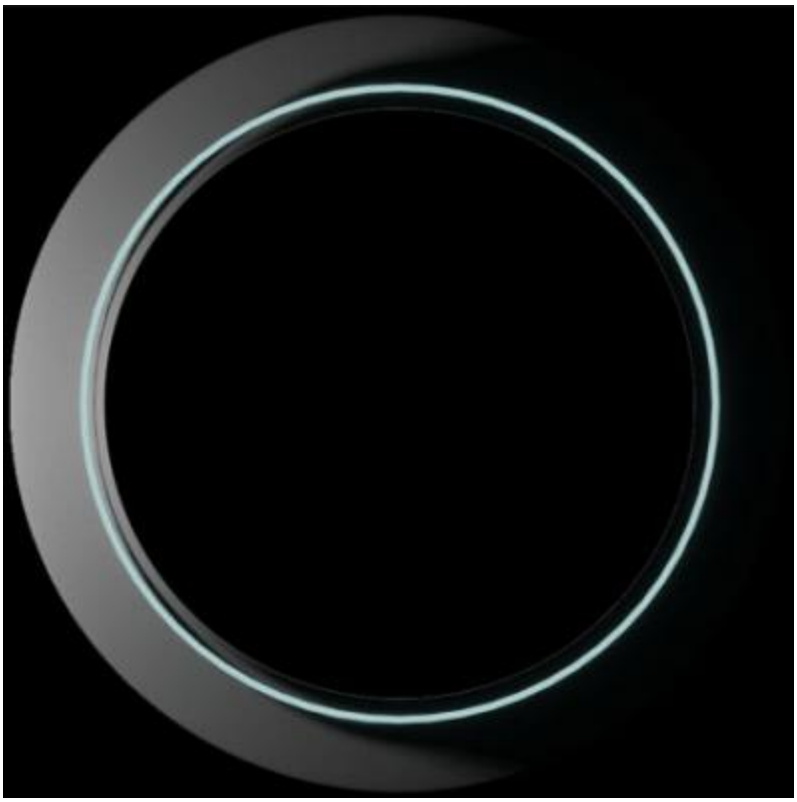

图 36. 索引颜色格式的渲染结果

本应用笔记附随的软件中包含的“LCDIF_VGLite”项目向您展示了如何在一层上使用 Index8Bpp 颜色格式，在另一层上使用 VGLite。

对于第 0 层，我们将有一个 720x720 Index8BPP 缓冲区。 对于第 1 层，我们将创建两个 416x416 ARGB8888 帧缓冲区。

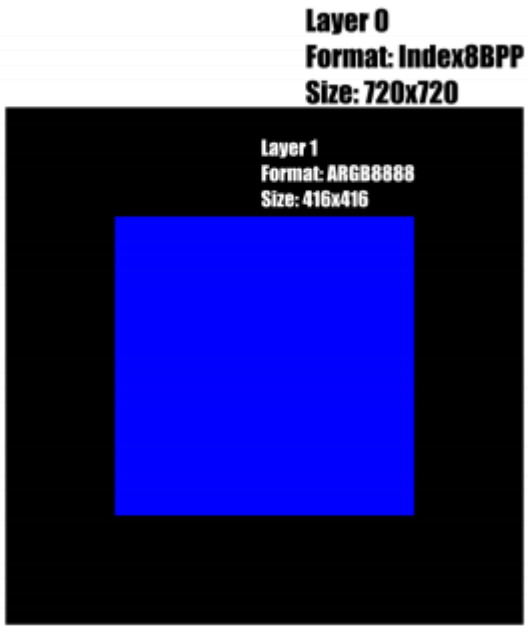

图 37. LCDIF_VGLite 工程层配置

nitLCDIFV2 函数将配置层，如上图所示。这里的一个重要考虑因素是我们添加了一个混合配置，以便第 1 层可以在最后一步与第 0 层混合。

```
const lcdifv2_blend_config_t blendConfig0 = {
.globalAlpha = 255,
.alphaMode = kLCDIFV2_AlphaEmbedded,
};
const lcdifv2_blend_config_t blendConfig1 = {
.globalAlpha = 255,
.alphaMode = kLCDIFV2_AlphaEmbedded,
};
```

kLCDIFV2_AlphaEmbedded 模式使用图像上的 alpha 来混合图像。

使用以下函数将颜色查找表复制到第 0 层配置：

LCDIFV2_SetLut(LCDIFV2, 0, **Dial720Indexed8bpp_CLUT**, 255, false)；

第 1 层缓冲区由 vg_lite_buffer_t 结构包裹。 通过这种方式，我们将能够渲染到这些缓冲区。

renderTarget[0].width = APP_LAYER1_WIDTH;

renderTarget[0].height = APP_LAYER1_HEIGHT;

renderTarget[0].stride = APP_LAYER1_WIDTH*4;

renderTarget[0].format = VG_LITE_BGRA8888;

renderTarget[0].memory = (void *)vgLiteFrameBuffers[0];

renderTarget[0].address = (uint32_t)vgLiteFrameBuffers[0];

renderTarget[1].width = APP_LAYER1_WIDTH;

renderTarget[1].height = APP_LAYER1_HEIGHT;

renderTarget[1].stride = APP_LAYER1_WIDTH*4;

renderTarget[1].format = VG_LITE_BGRA8888;

renderTarget[1].memory = (void *)vgLiteFrameBuffers[1];

renderTarget[1].address = (uint32_t)vgLiteFrameBuffers[1];

redrawVGLite例程执行以下绘制操作：

1. 通过将整个 416x416 缓冲区设置为纯蓝色来清除。
2. 它将 alpha 掩码应用于缓冲区，并使用第 2.1.2.4 节中描述的技术为其赋予圆形形状。

显示屏上显示的结果如下所示：

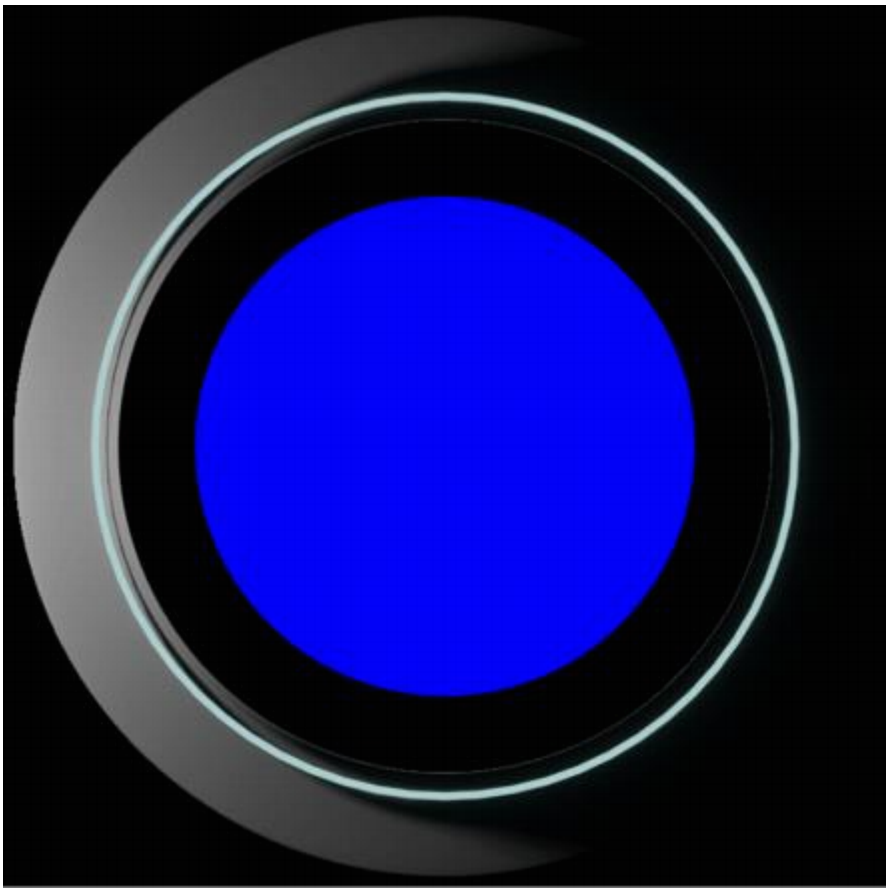

图 38. 结果

5 示例应用程序实现

第 4 章和第 5 章介绍了如何在 PXP 和 VGLite 中使用 LCDIF。

在接下来的三个部分中，我们将展示应用程序如何通过协同使用三个引擎来节省资源和提高性能。

我们将介绍的用例有以下要求：

- 复合的 1280x720 背景。
- 一个 1280x48 半透明顶栏。
- 四个图标。
- 一个带有矢量内容的窗口。

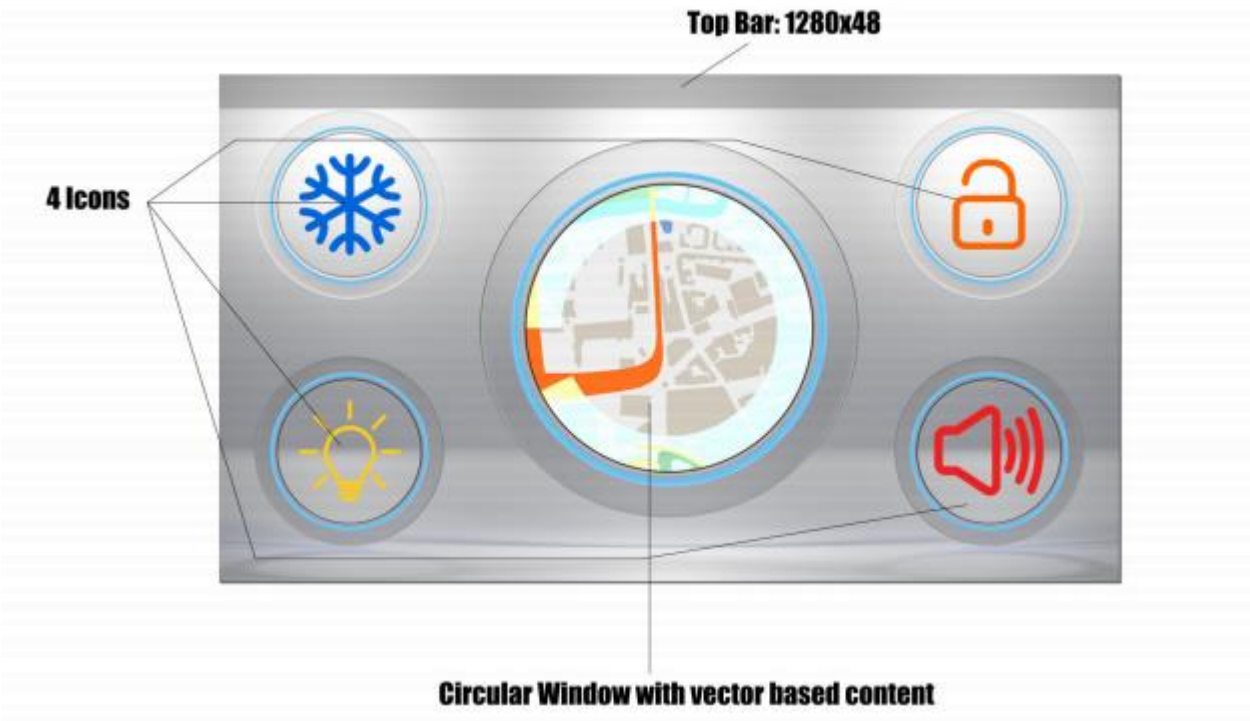

图 39. 示例应用程序说明

5.1 仅VGLite实现

打开“地图”项目。 此示例包括 VGLite-only 实现，其中所有元素都使用 VGLite 呈现。本案例的渲染资源如下：

- 显示：两个 720x1280 帧缓冲区，每像素 32 位，ARGB8888
- 背景位图：720x1280，每像素 32 位，ARGB8888
- Icon1 位图：144x144，每像素 16 位，RGBA4444
- Icon2 位图：144x144，每像素 16 位，RGBA4444
- Icon3 位图：128x155，每像素 16 位，RGBA4444
- Icon4 位图：128x103，每像素 16 位，RGBA4444
- 顶条位图：48x1280，每像素 16 位，RGBA4444
- 矢量渲染目标：416x416，每像素 32 位，ARGB8888

每帧绘制过程如下：

- 用 vg_lite_blit 将背景渲染到显示后台缓冲区。
- 将地图渲染到屏幕外目标。
- 将圆形表盘渲染到与地图相同的屏幕外目标。
- 在屏幕外目标中组合 alpha 蒙版以实现圆形。
- 将屏幕外目标渲染到显示后台缓冲区。
- 渲染图标。
- 渲染顶部栏。

- 交换显示后台缓冲区和前台缓冲区以显示新帧。

注意：该地图使用称为“Elementary”的渲染库进行渲染。这个库使用 VGLite 渲染矢量对象和位图。它分为两部分：将SVG转换为基本结构的离线工具和将这些结构解释为VGLite可用代码的运行时。运行时源代码随i.MXRT1170 SDK提供。对于离线工具，请联系NXP社区。

渲染结果是这样的：

图 40. VGLite-only渲染结果

此实现的性能为 14 FPS。

使用的内存如下（所有缓冲区都位于SDRAM中）：

- 帧缓冲区：7 MB
- 背景：3.5 MB
- 图标 1：0.04 MB
- 图标 2：0.04 MB
- 图标 3：0.037 MB
- 图标 4：0.025 MB
- 顶部栏：0.1172 MB
- 矢量渲染目标：0.66 MB
- 总计：11.4 MB

5.2 VGLite + PXP 实现

VGLite blit 函数和 PXP compose 函数有一定的重叠。除了矩阵转换（仅在 VGLite 上可用），两个引擎的光栅功能是相似的。您可以利用这一点并在两个引擎之间分配负载。

在这种情况下，我们会用 VGLite 把地图渲染到屏幕外缓冲区，当 VGLite 完成时，我们用 PXP 渲染屏幕的其余部分。

这种情况下的渲染资源与 VGLite-only 情况相同，只有一个额外的 每像素416x416 32 位的屏幕外目标。

打开“Map_PXP_SingleTask”项目并导航到redrawPXP函数：

- 当代码等待下一个显示 VSYNC 信号时，VGLite 会将地图渲染到屏幕外渲染目标。在这种情况下，我们会有两个离屏渲染目标，因此 PXP 只组成我们不渲染的缓冲区。
- 当 VSYNC 信号到达时，我们准备渲染到显示后台缓冲区并使用 PXP 处理表面渲染背景，使用 Alpha 表面渲染我们当前未使用的 VGLite 的屏幕外渲染目标。
- 然后我们使用 PXP 来渲染其余的元素。

渲染结果是这样的：

图 41. VGLite 和 PXP 渲染结果

乍一看，它看起来与仅使用 VGLite 的情况相同，但图标轮廓的混合方式不同。在撰写本文档时，最新版本的 VGLite 驱动程序 (3.0.4) 不会预乘 alpha 值。

此实现的性能为 21 FPS。 使用的内存为 12 MB。

5.3 VGLite + PXP + LCDIF 实现

LCDIFV2 有八层，这样能让我们轻松地对渲染过程进行分段，允许开发人员仅刷新那些在那一刻必须刷新的图层。开发人员还可以控制每个图层的混合模式、全局 Alpha 和位置。 在这个用例中，我们可以将渲染分成六层，如下图所示。

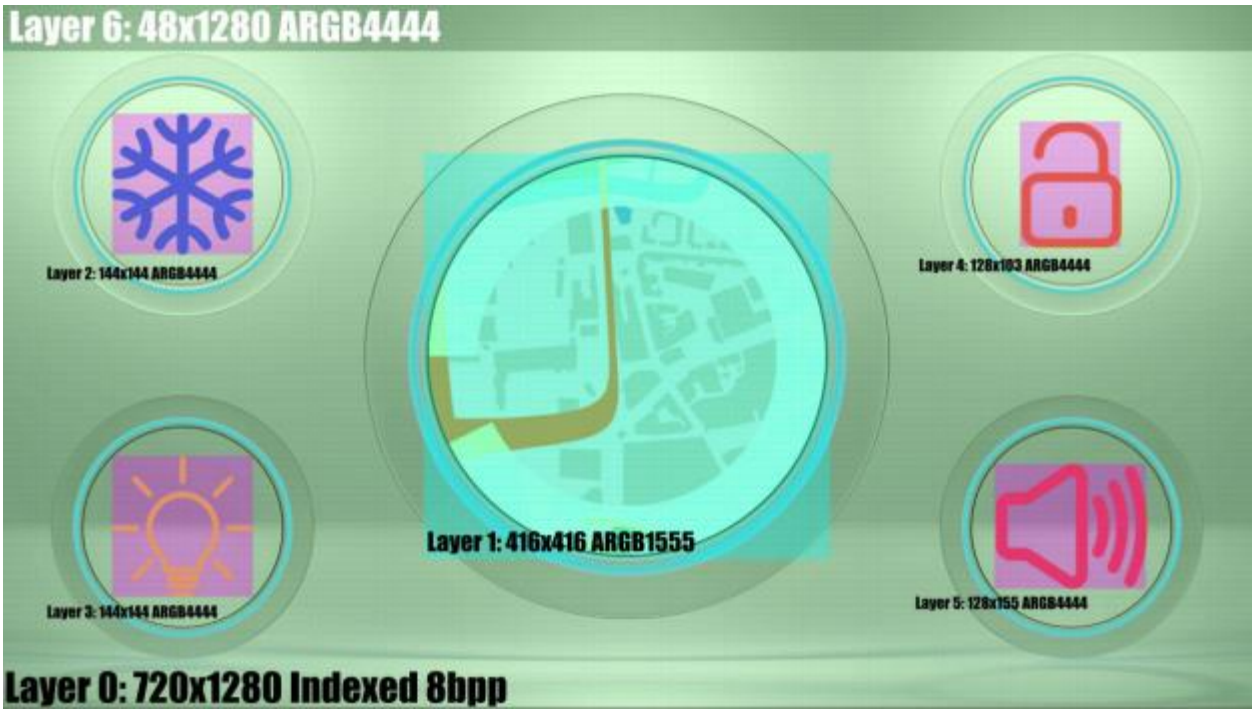

图 42. 渲染分割

打开“Map_PXP_LCDIF_SingleTast”项目。 这里面有所需的代码，代码的作用是设置多个图层并将它们混合在一起。项目建立在之前的示例之上，并显示了具有多层格式的 Elementary、VGLite、PXP 和 LCDIF。

项目使用以下资源：

- 第 0 层：720x1280，索引 8 bpp 帧缓冲区
- 背景位图：720x1280，每像素 32 位，ARGB8888
- 第 1 层：两个 416x416、ARGB1555 帧缓冲区
- 矢量渲染目标：416x416，每像素 32 位，ARGB8888
- 第 2 层：144x144，每像素 16 位，RGBA4444 帧缓冲
- Icon1 位图：144x144，每像素 16 位，RGBA4444
- 第 3 层：144x144，每像素 16 位，RGBA4444 帧缓冲区
- Icon2 位图：144x144，每像素 16 位，RGBA4444
- 第 4 层：128x155，每像素 16 位，RGBA4444 帧缓冲
- Icon3 位图：128x155，每像素 16 位，RGBA4444
- 第 5 层：128x103，每像素 16 位，RGBA4444 帧缓冲
- Icon4 位图：128x103，每像素 16 位，RGBA4444
- 第 6 层：128x103，每像素 16 位，RGBA4444 帧缓冲
- 顶条位图：48x1280，每像素 16 位，RGBA4444

此应用程序的渲染过程是最简单的：

- 每层都有自己的帧缓冲区。对于每一层，还分配一个位图作为内容的来源。在 APP_InitLcdif 中初始化 LCDIFV2 时，将位图复制到帧缓冲区。
- 初始化 LCDIF 层后，使用 VGLite 将地图渲染到屏幕外渲染目标。

- 将屏幕外目标渲染到第 1 层后台缓冲区。

每层都可以由 i.MXRT1170 中的任何模块（CPU、GPU、PXP和eDMA）填充。开发人员可以完全自由地选择使用哪个引擎。

渲染结果如下所示：

图 43. LCDIFv2, VGLite 和 PXP 渲染结果

如果您密切注意，您会注意到表盘后面高光处的色带。这是意料之中的，因为索引背景的调色板仅有 256 种颜色。这 256 种颜色仍然可以在这种情况下生成可接受的输出。

此实现的性能为 44 FPS。

使用的内存如下：

- 第 0 层：0.88 MB
- 背景位图：0.88 MB
- 第 1 层：0.66 MB
- 矢量渲染目标：1.3 MB
- 第 2 层：0.04 MB
- 图标 1 位图：0.04 MB
- 第 3 层：0.04 MB
- Icon2 位图：0.04 MB
- 第 4 层：0.037 MB
- Icon3 位图：0.037 MB
- 第 5 层：0.025 MB
- Icon4 位图：0.025 MB
- 第 6 层：0.1172 MB

- 顶栏位图：0.1172 MB

使用的总内存为 4.2384 MB。

从最初的实现到最终的实现，性能提高了三倍，但是内存使用量减半。在最后一种情况下，图标和顶部栏不再每帧渲染，但这体现了使用多层的优点和设置它们的简单性。

这只是一个假设的示例应用程序，但是在平台上采用不同的加速器是一个很好的练习。我们鼓励您了解每个功能的功能，以便充分利用 i.MXRT1170。

How To Reach Us

Home Page:

nxp.com

Web Support:

nxp.com/support

Information in this document is provided solely to enable system and software implementers to use NXP products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. NXP reserves the right to make changes without further notice to any products herein.

NXP makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does NXP assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in NXP data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. NXP does not convey any license under its patent rights nor the rights of others. NXP sells products pursuant to standard terms and conditions of sale, which can be found at the following address: nxp.com/SalesTermsandConditions.

Security — Customer understands that all NXP products may be subject to unidentified or documented vulnerabilities. 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.

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

For sales office addresses, please send an email to: salesaddresses@nxp.com

Date of release: 12/2020

Document identifier: AN13075

arm