AN13075

i.MX RT1170 Heterogeneous Graphics Pipeline

Rev. 0 — 12/2020 Application Note

Contents

1 Introduction

The i.MX RT1170 crossover MCU features several options to generate, compose, and blend graphics content before sending it to a display:

- 2D Vector Graphics GPU: This engine can render scalable vector graphics and compose and manipulate bitmaps.
- PXP Graphics Accelerator: This 2D engine can manipulate and blend bitmaps, its main features are scaling, fixed-angle rotation and color space conversion.
- LCDIF: The display controller allows you to create up to eight display layers, offering on-the-fly blending capabilities with multi-format support.

This application note gives a brief overview of each one of the engines, how to initialize them, use them independently, and finally, it introduces a use case on how to use them in unison to get a performance and resource boost. Each stage has an associated software project to make things easier.

2 2D vector graphics GPU

The GPU on the i.MX RT1170 is an instance of the GC355 GPU from Verisilicon. This is the only accelerator on the platform capable of procedurally generating graphics based on input geometry.

The user interfaces with this engine using a graphics API. This application note focuses on the VGLite API. OpenVG will be available at a later date.

2.1 VGLite

VGLite is a lightweight 2D graphics API, designed to have a small memory footprint and low CPU overhead.

The two main drawing functions on VGLite are vg_lite_draw, which renders vector graphics primitives, and vg_lite_blit, which renders bitmaps, also known as raster images.

2.1.1 VGLite vector rendering pipeline

To render vector graphics, you need the following six pieces of information:

1. Target: This is the target buffer that will hold the resulting rendered image.
2. Path data: Consists of a set of coordinates and path segments that define the shape of the geometry that will be rendered.
3. Fill rule: Specifies the scheme selected to fill the geometric shapes with a solid color.
4. Transformation: An affine transformation is provided via a 3x3 matrix.
5. Color: The path color is defined using a 32bpp value (alpha, red, green, and blue, each using eight bits). Each pixel covered by the path will have this color.
6. Blend rules: The path will be blended to the extended buffer content based on these rules.

Open the **NXPLogo** project from the associated software: This is an incremental example explaining how to use the information pieces, in this sample we will render the NXP logo using vector graphics. The code is divided into seven steps controlled by the TEST_STEP macro. The first step (INITIAL_LOGO) renders the NXP logo as follows:

Figure 1. First step expected render result: the NXP Logo using vector graphics

2.1.1.1 Path data

The path data is the most important piece of information. It defines the shape of what you want to render. The path data is provided as a pair of structured data pieces:

1. Opcode: defines the operation
2. Arguments: 2D coordinate data consumed by the operation.

The N letter can be defined using 10 opcodes as follows:

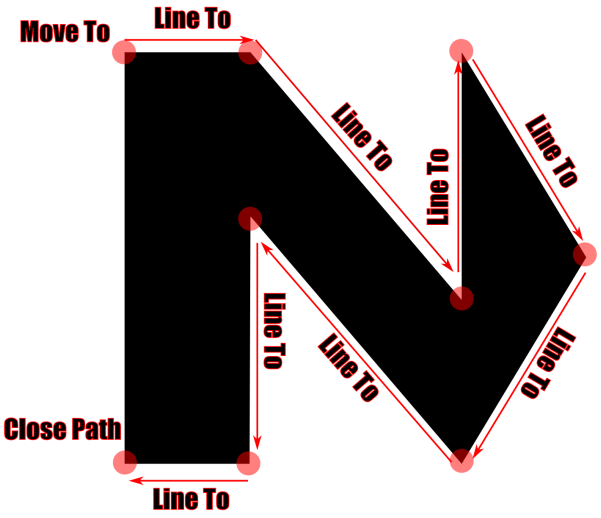

Figure 2. N Letter path data: commands

1. Move To: Initial point for the N letter.
2. Eight Line To opcodes defining the shape of the N letter.
3. Close Path opcode, instructing that the open paths must be closed.

Each one of the opcodes above consumes the following coordinate data:

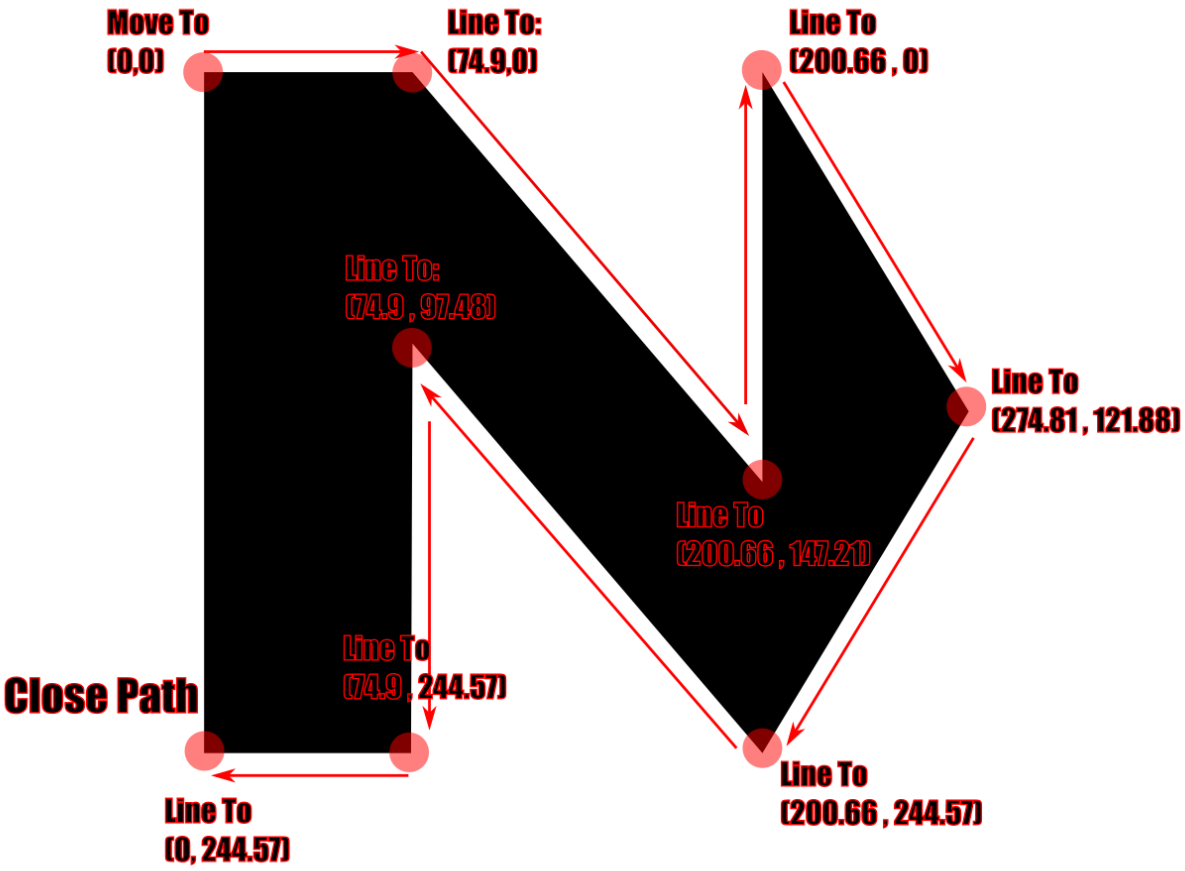

Figure 3. N Letter path data: commands and coordinates

1. Move To: consumes two coordinates (x and y).
2. Line To: consumes each two coordinates and it will define a line from the previous point to the newly specified point.
3. The Close Path does not consume any additional data.

The information for this path is stored in the nPathData array in the code.

The X letter is defined similarly to the N letter. The Move to opcode is followed by a series of Line To opcodes and finally the Close Path opcode:

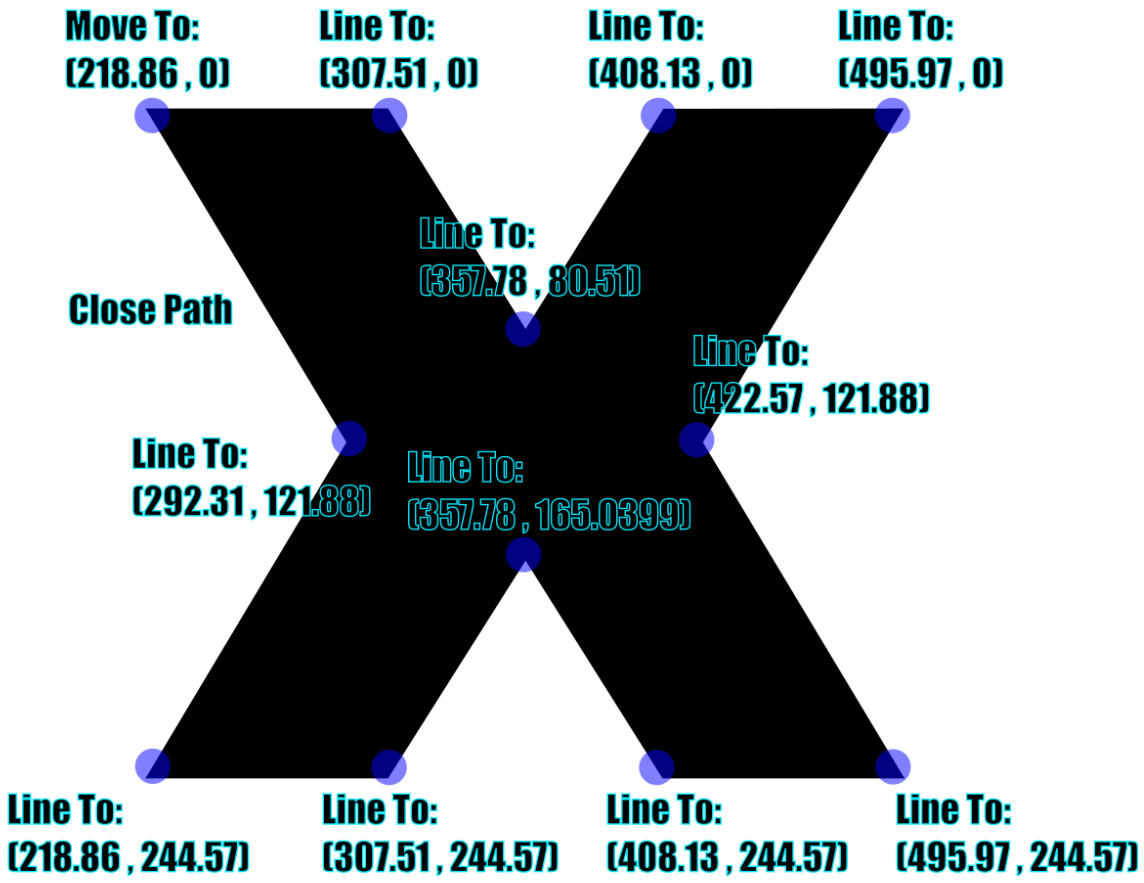

Figure 4. X Letter path data: commands and coordinates

Information for this path is stored in the xPathData array in the code.

The P letter is more interesting. It contains curves and an inner path that will allow us to showcase the fill rule later.

Let's initially define it as two outlines defined within a single path, an outer figure, and an inner figure. The outer figure is defined by the following six opcodes:

1. Move To: (514.92, 0)
2. Line To: (648.62, 0)
3. Cubic Curve To: (648.62, 191.07). A cubic curve has two control points: c1 (740.62, 0) and c2 (740.62, 191.07).
4. Line To: (514.92, 191.07)
5. Line To: (514.92, 244.57)
6. Line To: (440.06, 121.88)
7. Line To: (514.92, 0)

   The inner figure will be defined within the same path structure:

8. Move To: (514.92, 61.14)
9. Line To: (613.96, 61.14)
10. Cubic Curve To: (613.96. 130.44) with control points on (639.4, 61.14) and (639.4, 130.44)
11. Line To: (514.92, 130.44)

12. Line To: (514.92, 61.14)

    Finally, you close the path:

13. Close Path

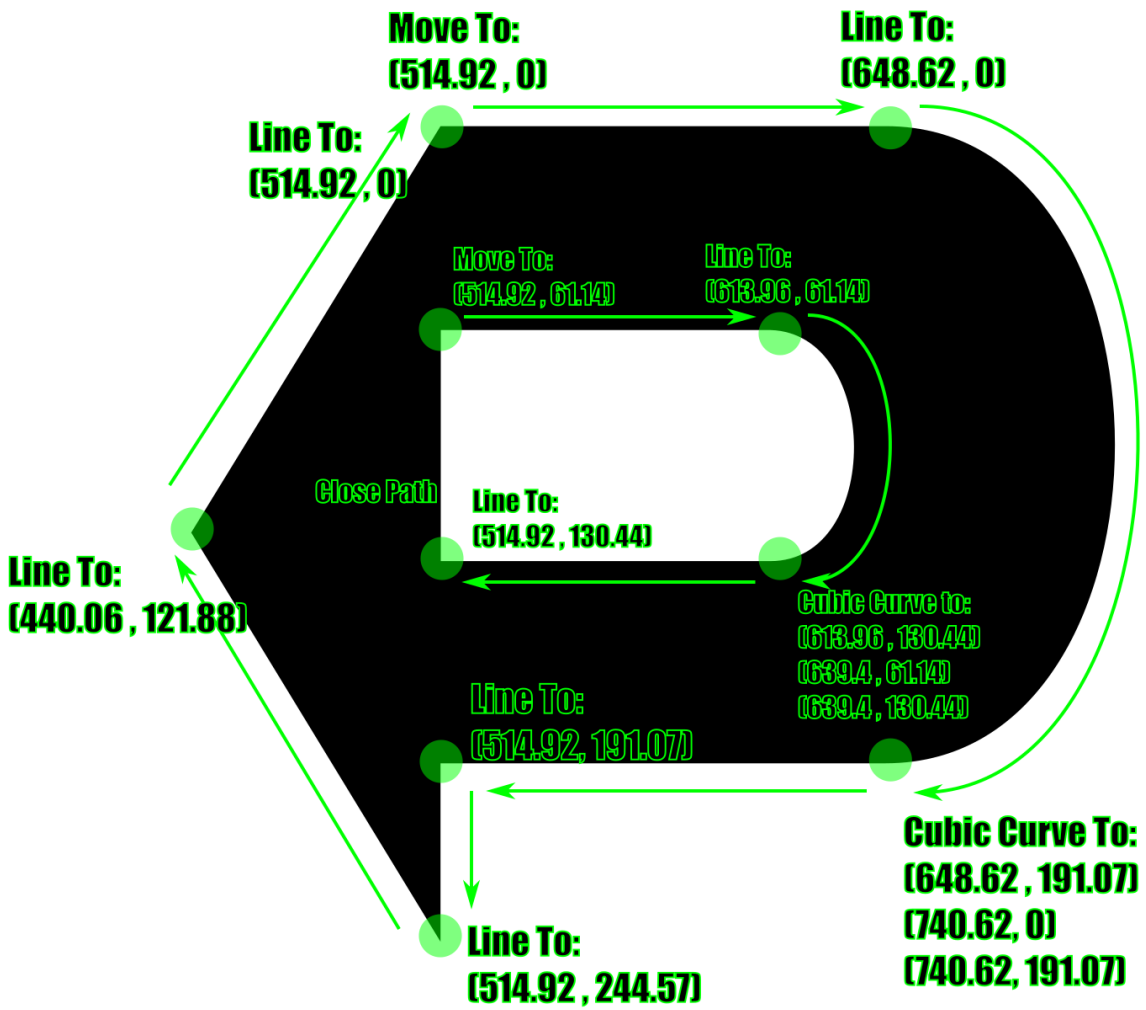

Figure 5. P Letter path data: commands and coordinates

The information for this path is stored in the pPathData array in the code.

With the information for each path ready, the next step is to initialize the vg_lite_path_t structures:

```
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);
```

The code to render the three letters to the display is as follows:

```
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);
```

Notice how we call vg_lite_draw three times (one time per letter). This is because our paths are separated. We could render it with a single draw call, but as we plan to change path colors, we separated it into several draw calls from the beginning.

The following figure is a direct copy of the render result from the i.MXRT 1170:

Figure 6. Actual render result

2.1.1.2 Fill rule

We will use the P letter to showcase how the fill rule affects how the inner shapes within a path are interpreted. Change the TEST_STEP macro to FILL_RULE.

The fill rule allows you to cut holes into paths and that allows the P letter to have the space in the center.

We change the fill rule for the P letter from VG_LITE_FILL_EVEN_ODD to VG_LITE_FILL_NON_ZERO as follows:

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

The inner hole in the path will not be considered:

Figure 7. Fill rule VG_LITE_FILL_NON_ZERO causes the P path to be completely filled

2.1.1.3 Transformation

Transformation is a powerful feature of the GPU. It allows you to define a 3x3 matrix for 2D affine transformations (an affine transformation is a linear transformation followed by a translation). It also enables us to describe projective transformations.

There are matrix objects and functions on VGLite, but let's start by not using them to gain an intuition of how matrices are defined. Change the TEST_STEP macro to TRANSFORM_INTUITION.

The matrices fed to VGLite consist of 9 floating-point values, arranged in a 3x3 array.

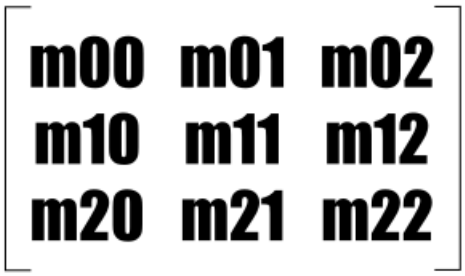

Figure 8. VGLite matrix

The matrix is arranged in a row major order, which means that your matrix in the memory would look like this:

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

Figure 9. Matrix in the memory

This is the same as creating a single array with nine positions:

float userMatrix[9] = {1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0};

The VGLite vectors are column vectors. Consider this when building your transformation matrices. Let's build a rotation matrix. Let's say that we want to rotate by 28 degrees around the origin. Remember that for the VGLite's coordinate system, where the X axis points to the right and the Y axis points down, the positive angles rotate clockwise.

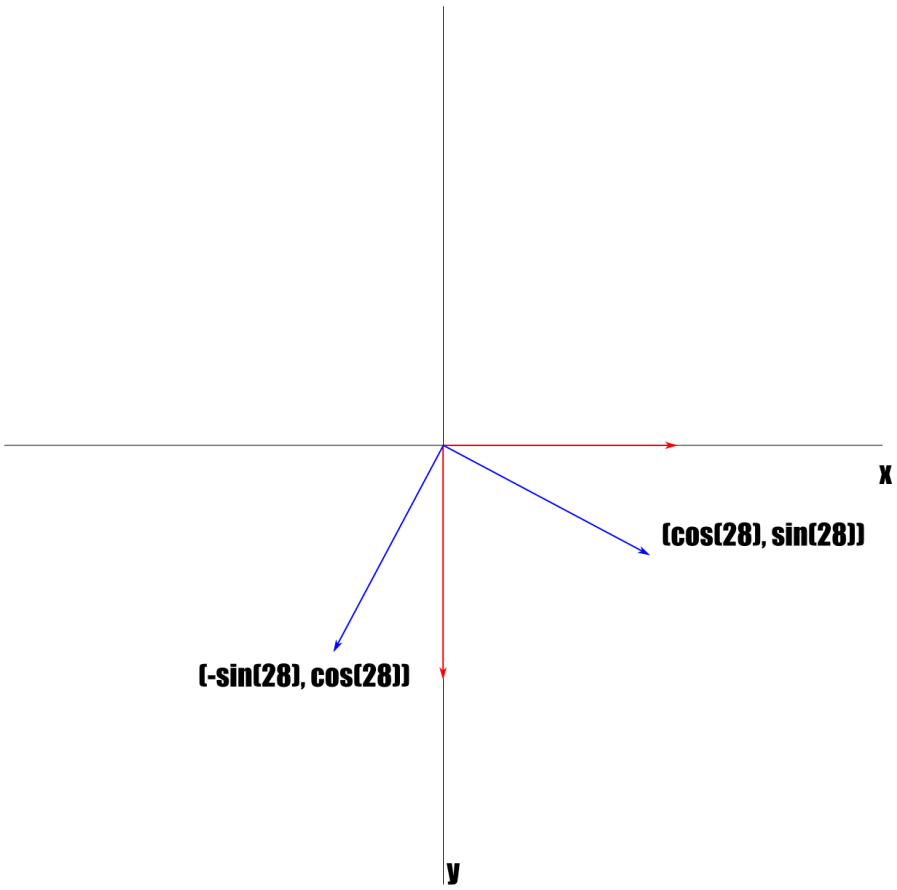

Figure 10. Expected transformation of the basis vectors

Now, to build the matrix, we place:

- The transformed coordinates of the X axis to the first column.
- The transformed coordinates of the Y axis to the second column.
- As we do not have translation, we will leave the third column as (0,0,1).

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

Figure 11. Rotation Matrix

With this, we can now store the values in the array positions:

```
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);
```

The rendering result with this transformation is as follows:

Figure 12. Transformation rendering result

It matches exactly what we expected.

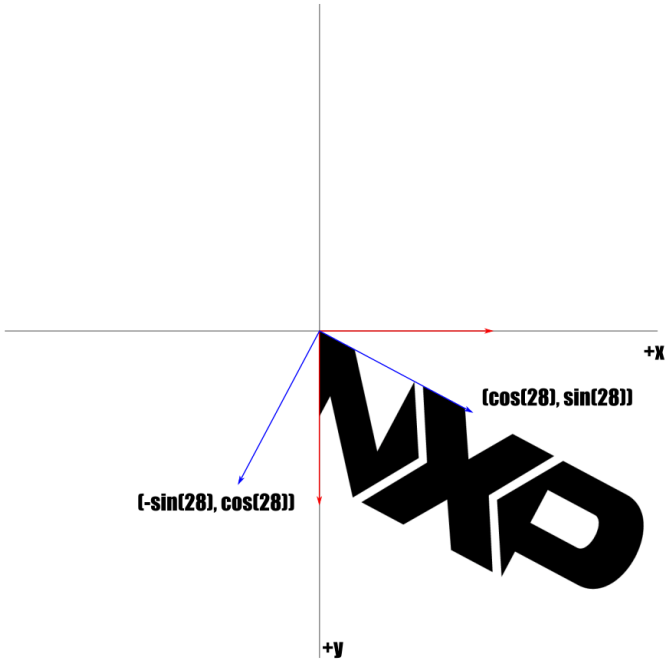

Figure 13. Render result matches the basis vectors transformation

Now that you have an idea of what is happening under the hood, use the vg_lite_matrix_t structure and the vg_lite_rotate function to achieve the same thing. Change the TEST_STEP macro to TRANSFORM_VG_LITE.

The code to achieve the same transformation from the previous step is simplified as follows:

```
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 Color

When drawing, you can specify a color for each path. To match the NXP logo with its true colors, change the TEST_STEP macro to COLOR. Specify a different color for each letter:

```
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);
```

The render result of such color configuration is as follows:

Figure 14. Color rendering result

2.1.1.5 Blend

Both the alpha field within the color parameter and the blend function define the effect that the alpha will have on both the path and the destination buffer. Change the TEST_STEP macro to BLEND.

The most common blend function configuration is VG_LITE_BLEND_SRC_OVR, which, for each pixel, implements the following equation to define the final color value of the pixel:

S + (1 - Sa) * D

“S” is the path color, “D” is the color stored in the target buffer, and “Sa” is the alpha value of the path.

In the following code, the target buffer is set to a solid red color and the path is rendered with an alpha value of 0x7F, which is roughly an opacity of 0.5:

```
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);
```

The result for each letter’s color would be:

N:

- Base color: red: 0xE8 = 232, green: 0x B4 = 180, blue: 0x10 = 16
- Resulting colors
  - Red = 232 + (255 - 127) * 255 = 255 = 0xFF
  - Green = 180 + (255 - 127) * 0 = 184 = 0xB4
  - Blue = 16 + (255 - 127) * 0 = 16 = 0x10
- Resulting color in ABGR = **0xFF10B4FF**

X:

- Base color: red: 0x8B = 139, green: 0xAE = 174, blue: 0xD9 = 217
- Resulting colors
  - Red = 139 + (255 - 127) * 255 = 255 = 0xFF
  - Green = 174 + (255 - 127) * 0 = 174 = 0xAE
  - Blue = 217 + (255 - 127) * 0 = 217 = 0xD9

- Resulting color in ABGR = **0xFFD9AEFF**

P:

- Base color: red: 0xC9 = 201, green: 0xD1 = 209, blue: 0x21 = 33
- Resulting colors
  - Red = 201 + (255 - 127) * 255 = 255 = 0xFF
  - Green = 209 + (255 - 127) * 0 = 209 = 0xD1
  - Blue = 33 + (255 - 127) * 0 = 33 = 0x21
- Resulting color in ABGR = **0xFF21D1FF**

Let's compare those colors against a dump of our render target:

Figure 15. Color compared

The rectangles above each letter were set using the colors that we calculated, and the letters are the actual result from our buffer. As you can see they are nearly the same, but you will notice a slight difference. This is because our target buffer is configured as RGB565 and some color information is lost during the color conversion process.

2.1.1.6 Linear gradients

You can define linear gradients for the colors in each path. To do this, use an additional structure called vg_lite_linear_gradient_t and a draw function variant vg_lite_draw_gradient. Change the TEST_STEP macro to LINEAR_GRADIENTS.

When you create a gradient, it will generate a 256x1 image that will be then applied to the path automatically. You can control the size and orientation of the gradient using a matrix. Each gradient has its own 3x3 matrix.

For each gradient, define the colors and stops. Stops are offsets on the 256x1 image. Colors are assigned to those stops and the colors between the stops will be interpolated.

We will use the three gradients below to define the color of the NXP letters. Each gradient has its own stop and color information:

Figure 16. Color gradients

This can be translated to the following code:

```
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);
```

When the gradients are initialized, they must be transformed to be located where they must be.

Figure 17. Gradient transformation

For starters, each gradient must be translated horizontally to the start of each letter.

The translation is not sufficient, because each letter is bigger than 256. We have to scale the gradient as well. This is where the gradient matrix comes in handy:

The N letter has a width of 274.81, so we need to scale our gradient by 1.073 on the horizontal axis.

The X letter has a width of 277.140 and the scale must be 1.0825. We also have to translate it to 218.86 on the horizontal axis.

The P letter has a width of 277.380 so the scale must be 1.0835 and translated to 440 on the horizontal axis.

The gradient transform is important. Without it, you get the following rendering result:

Figure 18. Rendering result without gradient transformation

2.1.2 VGLite raster pipeline

VGLite also supports drawing images via the vg_lite_blit* functions. The base blit function requires the following pieces of information:

- Target: The target buffer that will hold the result of the blit function.
- Source: The buffer that will be composed into the target buffer.
- Transformation: Provided as a 3x3 matrix, it allows you to transform the source before composing it into the target. With this, you can achieve transformations like translation, scaling, rotating, shearing, and perspective.
- Blend Rule: Specify how alpha is used to compose the source into the target.
- Color: You can multiply your source buffer by a color before rendering it into the target. This is useful for rendering colored fonts using an A8 or A4 source format. In this document, we will not use this parameter.

- Filter: If the hardware allows, you can enable bilinear filtering for the bitmaps. In this document, we will not use any kind of filtering.

2.1.2.1 Simple blit

Open the project **VGLiteBlit** from the associated software. We will look at these pieces of information one by one using the associated software. Set the TEST_STEP macro to SIMPLE_BLIT.

The objective of the first three steps will be to render this bitmap to the screen:

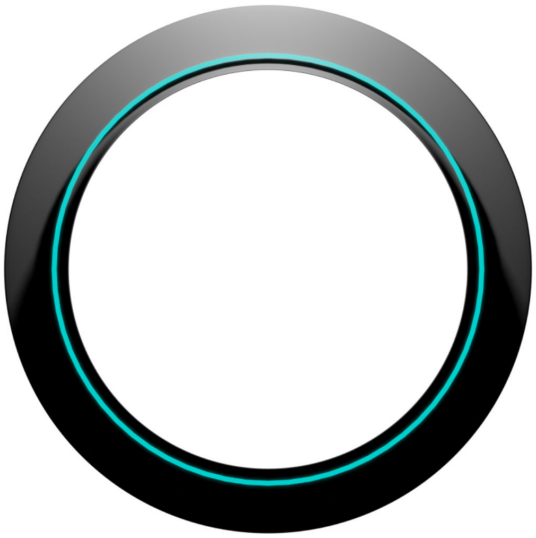

Figure 19. Bitmap render

The target buffer for this application will be provided by the VGLITE_GetRenderTarget function. This function is not part of the core VGLite API. Its goal is to give you a vg_lite_buffer_t that is wrapped around the framebuffer that will be sent to the display.

The source buffer for this step is a vg_lite_buffer_t with a VG_LITE_RGBA4444 format and 720 pixels of width and height.

To render it to the screen, use the vg_lite_blit function:

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

The render target returned by VGLITE_GetRenderTarget is rt and dial is the bitmap we plan to blit.

A useful feature of the vg_lite_blit function is that it automatically performs color space conversion from the source format to the target format. In this case, it will convert from RGBA4444 to RGB565.

When you render to your screen, you will see this:

Figure 20. Render result of a blit of a RGBA4444 buffer to RGB565 buffer, heavy color banding present

That does not look like we expected it to look (the gradient is not smooth). This visual artifact is called color banding.

This is because we are rendering a RGBA4444 buffer to a RGB565 buffer. None of those buffers have enough color information to describe a smooth color gradient.

Change the TEST_STEP macro to NO_BANDING. Under this configuration, we will render a RGBA8888 buffer to a RGB565 buffer.

The result of this change looks better:

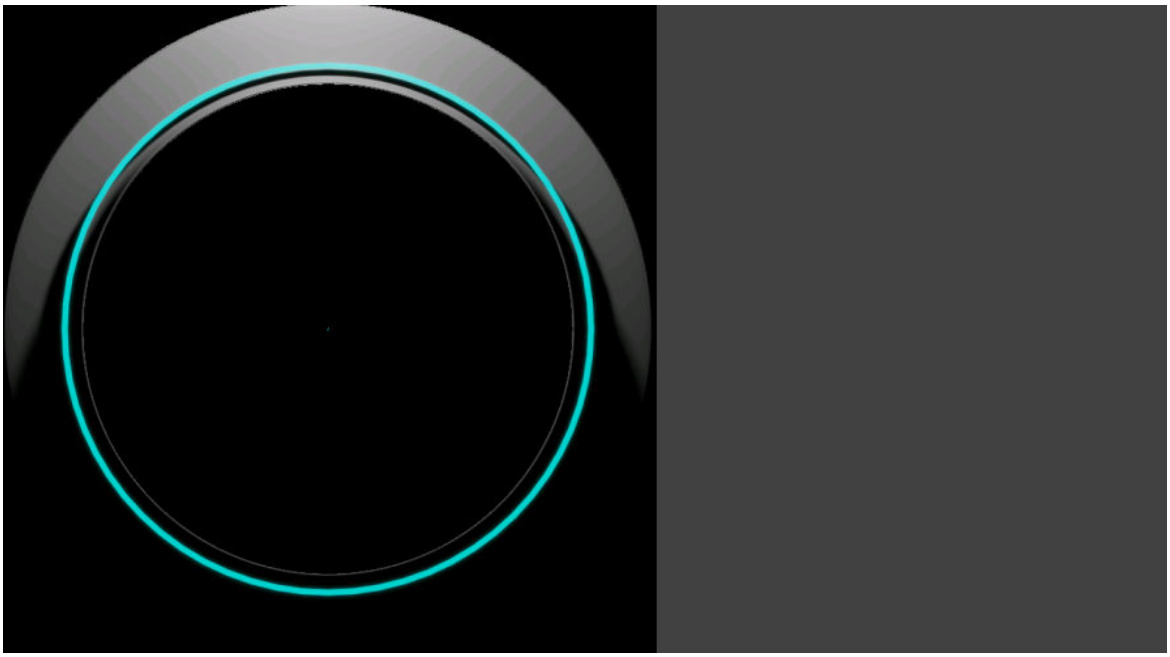

Figure 21. Render result of a blit of a RGBA8888 buffer to RGB565 buffer, slight color banding present

It is still not perfect because our RGB565 target buffer still does not have enough information to completely remove the banding.

To completely remove it, configure your render target as kVIDEO_PixelFormatXRGB8888. To do this, open the *display_support.h* file and change the following macros from the following:

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

To this:

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

Render it again and finally, the result is what we expected:

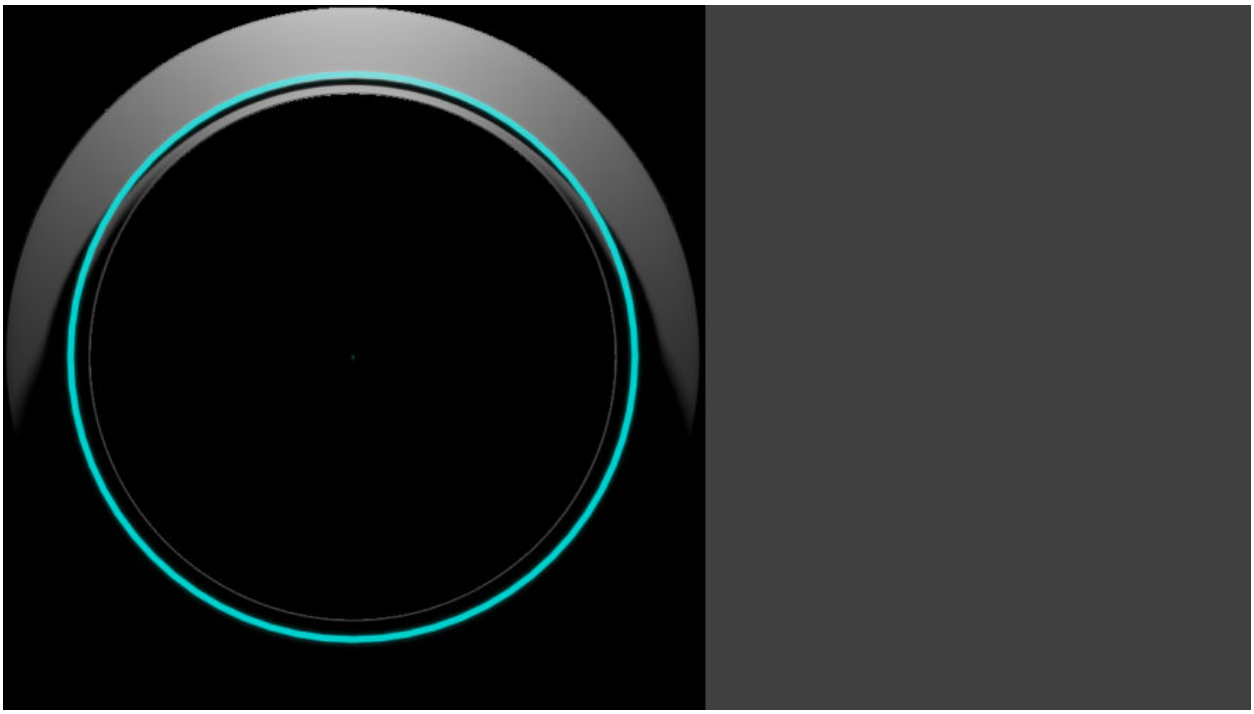

Figure 22. Render result of a blit of a RGBA8888 buffer to a RGBA8888, no color banding

This comes at a cost. From the first step to this one, our memory usage has risen substantially:

Our initial step used 1 RGBA4444 buffer for the dial (720*720*2 bytes, or around 1 MB) and 2 RGB565 target buffers, each with a size of 720*1280*2 bytes. This equals to a sum of approximately 4.5 MB.

When we took all the steps to remove the banding, we ended up with 1 RGBA8888 buffer for the dial (720*720*4 bytes or close to 2 MB) and 2 XRGB8888 target buffers, each with a size of 720*1280*4 bytes. This equals to a sum of 9 MB.

This will also impact the system bandwidth usage, as more memory must be moved each frame.

In the LCDIF chapter, you will learn how to remove banding artifacts without having to pay this memory usage and bandwidth cost.

2.1.2.2 Blending

The dial that we are rendering is not blended with the background. Change the TEST_STEP macro to BLENDING.

In the redraw function, before calling the render function, we are clearing the target buffer with a gray color: vg_lite_clear(rt, NULL, 0xFF404040). However, when the dial was rendered, the whole background of the 720x720 image was interpreted as black. This is because we had blending disabled. By changing the blend rule to VG_LITE_BLEND_SRC_OVER, you will get the following output:

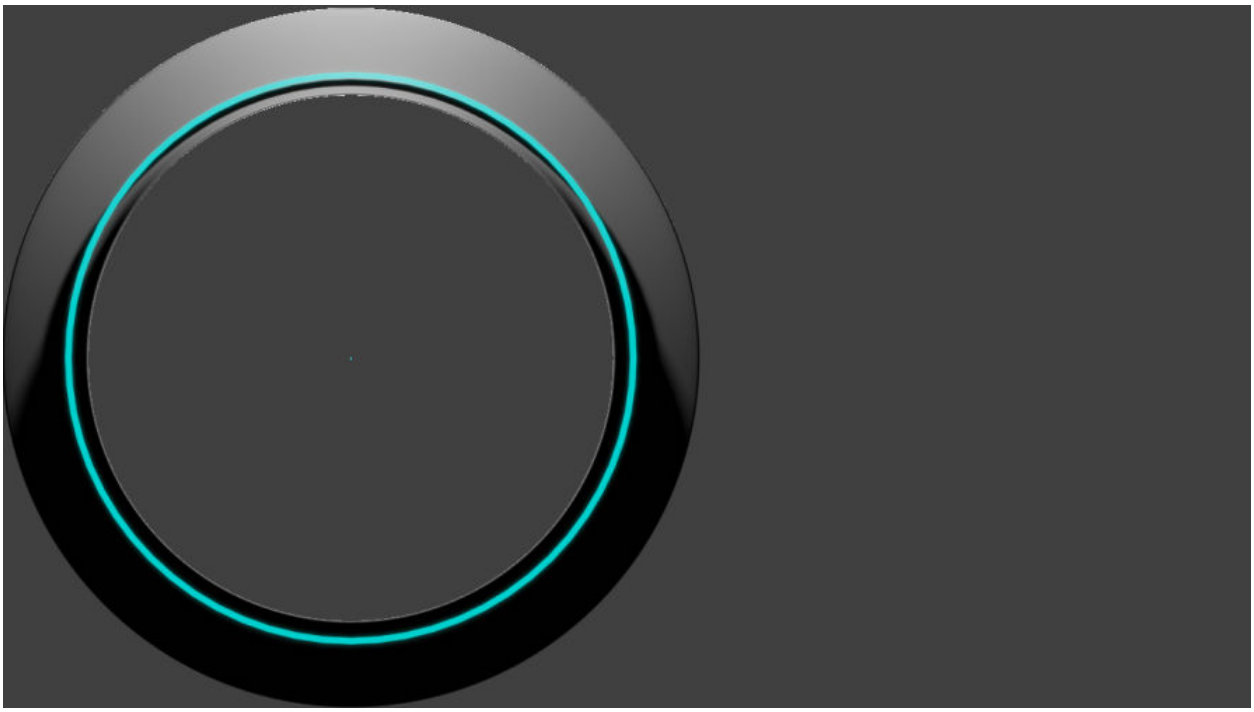

Figure 23. Blending

2.1.2.3 Transformation

The procedure to transform bitmaps is the same than the one to transform paths. Use a 3x3 matrix to describe the affine transformation that you want to effectuate on the bitmap. The matrix objects and functions to do so are the same. Change the TEST_STEP macro to SIMPLE_TRANSFORMATION.

In this step, we rotate a buffer by 45 degrees:

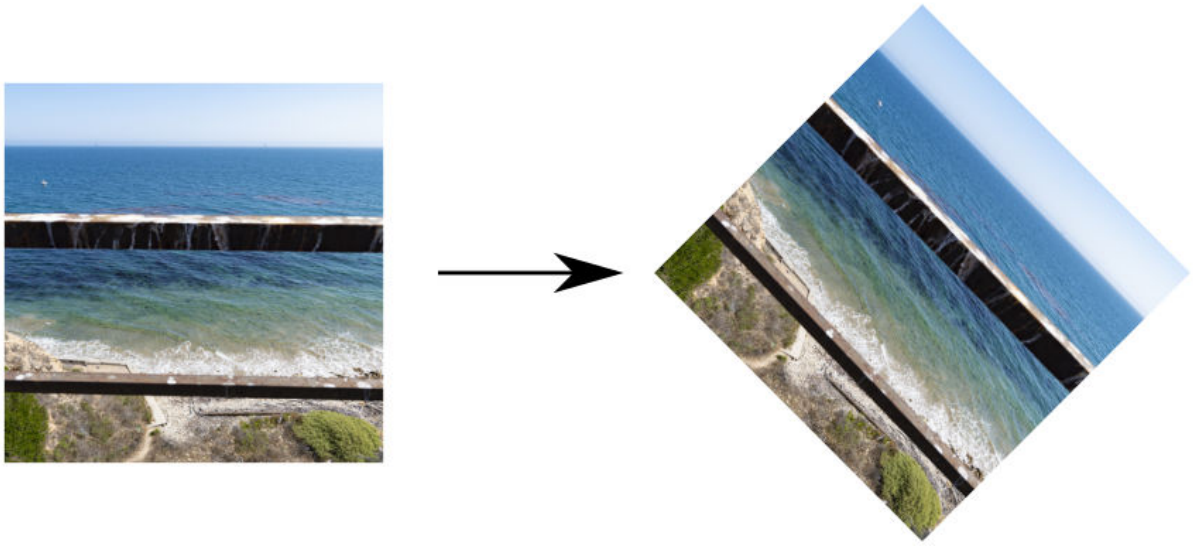

Figure 24. Buffer rotation

However, by having a 3x3 matrix, we can achieve even more complex transformations, like projections.

Change the TEST_STEP macro to PERSPECTIVE_2_5_D. In this step, we build a projective transformation that maps the following points:

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

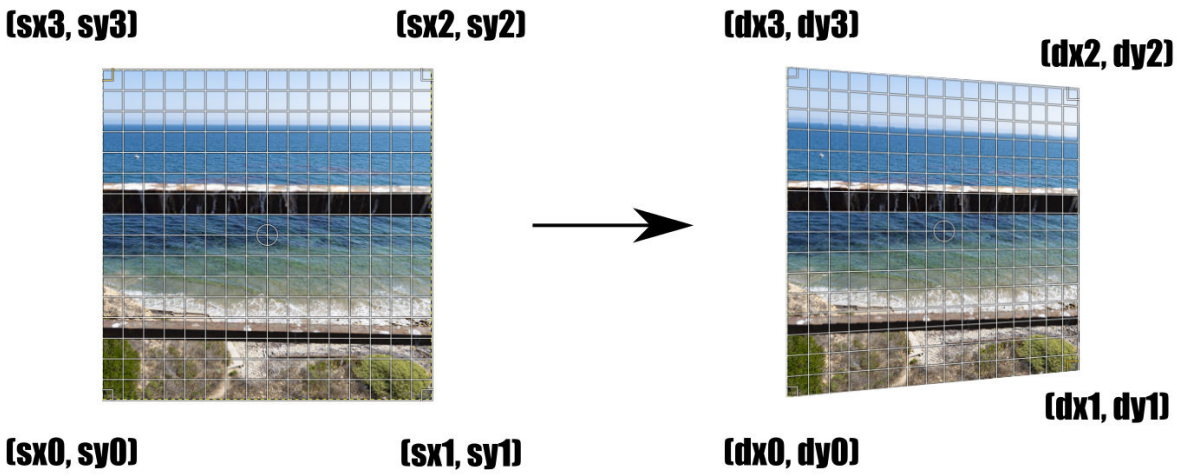

Figure 25. Projective transformation

The image below is a direct snapshot of the result of the transformation.

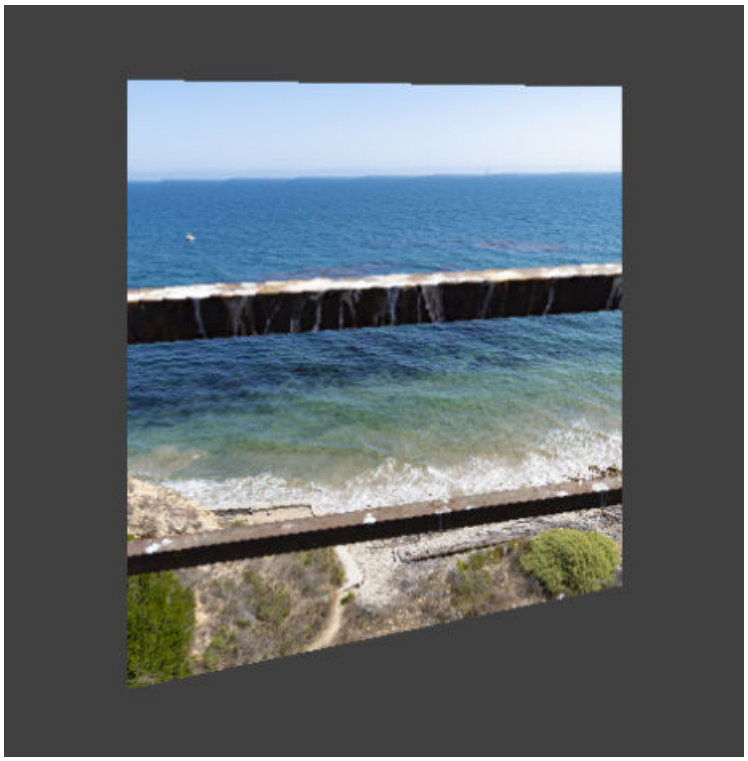

Figure 26. Transformation result snapshot

To achieve this, we ported the vguComputeWarpQuadToQuad from OpenVG's utility libraries.

2.1.2.4 Raster and vector operations

Because vg_lite_draw also uses vg_lite_buffer_t structures as targets, it allows you to mix the calls of these rendering functions to achieve interesting effects. Change the TEST_STEP macro to VECTOR_AND_RASTER.

In this step, we create a vg_lite_buffer_t where we will render a vector-based landscape:

Figure 27. Vector-based landscape

On top of that same buffer, we render an alpha mask with the VG_LITE_BLEND_DST_IN blend configuration. This enables us to discard all pixels that are not covered by the white pixels on the mask.

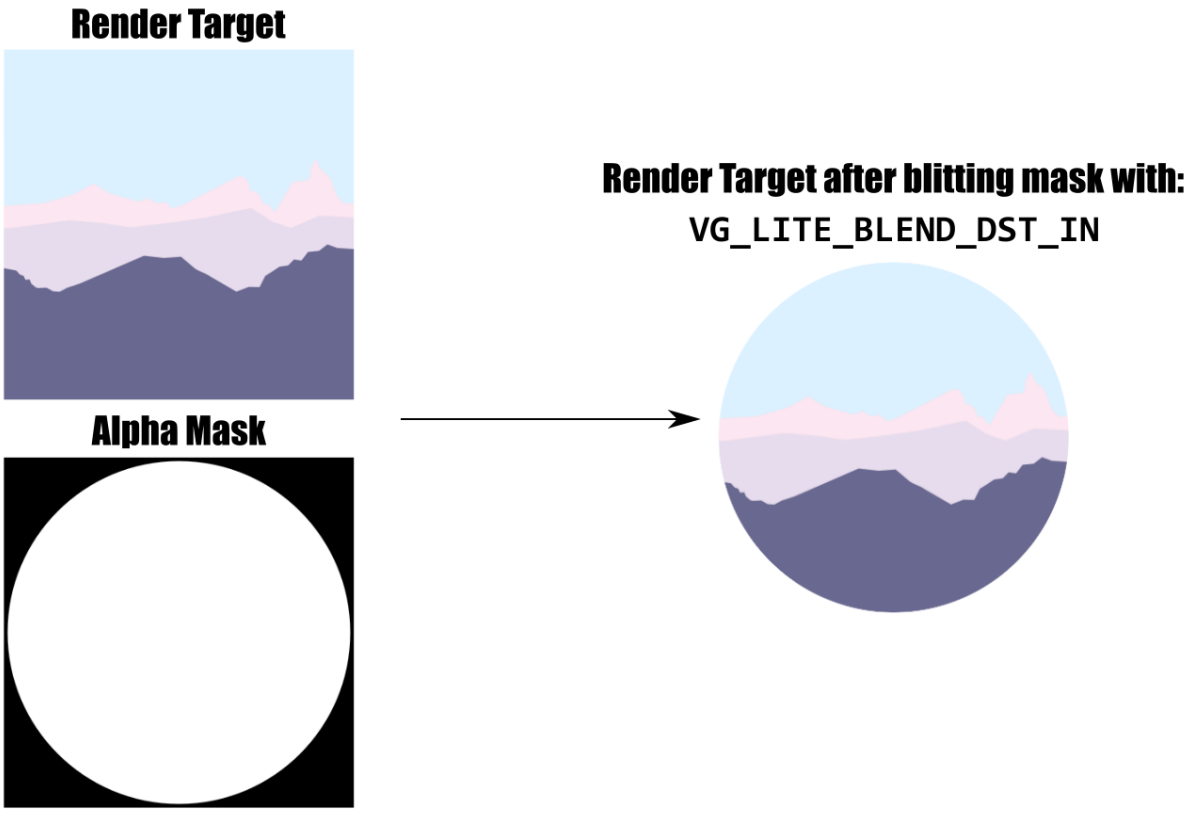

Figure 28. Alpha mask

The result in your target will look like this:

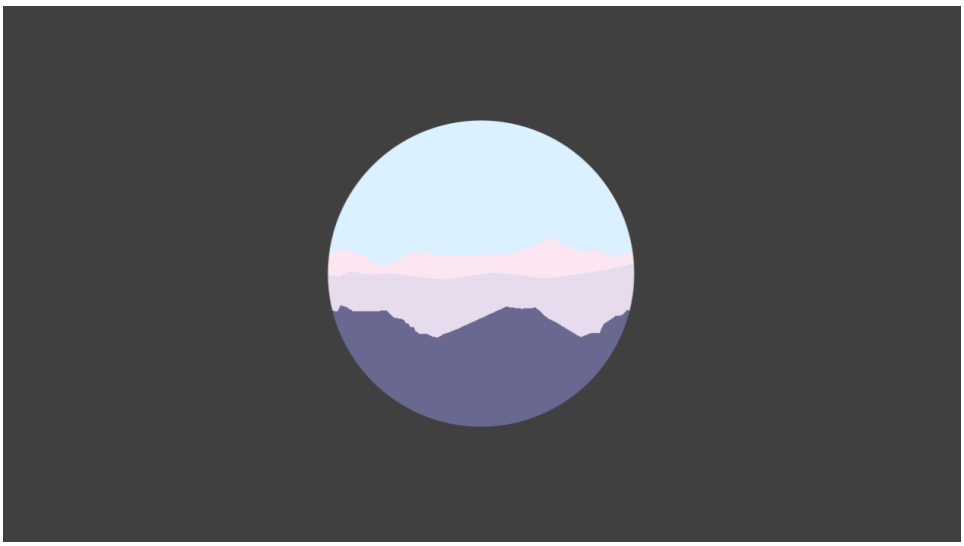

Figure 29. Target result

3 PXP 2D accelerator

Pixel Processing Pipeline (PXP) is a powerful 2D accelerator capable of composing graphics assets before sending them to the display controller. Its most important functions are blitting, alpha blending, color space conversion, fixed angle rotation, and scaling.

This chapter is brief, because the PXP is already well documented and there are multiple samples already included in the SDK. We will limit our use of PXP to blending its Alpha Surface (AS) with the Processed Surface (PS). Keep in mind that this module has far more capabilities than the ones we use here.

Open the “LCDIF_PXP” project from the application note software file.

For this sample, we will use PXP to populate the buffers that will be sent to the display controller. In total, there will be three buffers: two for layer 0 (which will be constantly redrawn) and one for layer 1 (which will be initially populated).

Layer 0 is configured with a size of 720x640 and an ARGB8888 color format. We will allocate two buffers, so the PXP only renders to a buffer that is not currently being shown.

Layer 1 is configured with a size of 144x394 and an ARGB8888 color format.

The picture below color codes the layers as we want to display them:

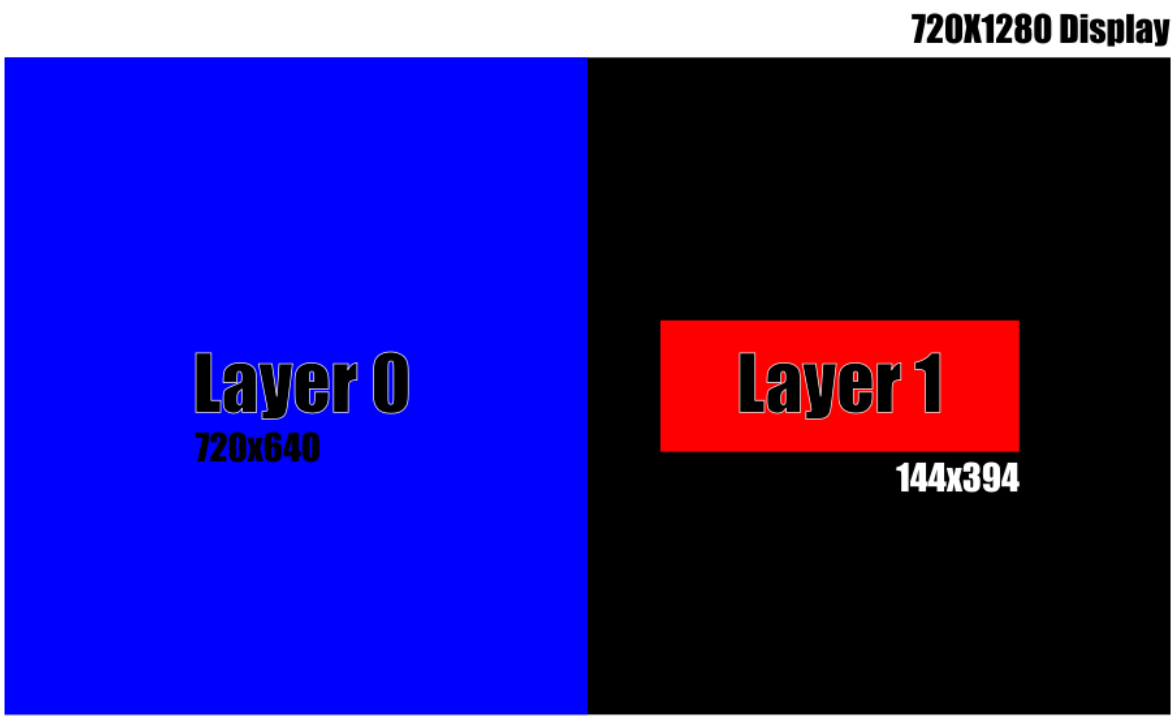

Figure 30. Layer color codes

Layer 0 composition is divided into three PXP blit operations:

The first blit configures the PS to point to a background image. The AS will point to a label and the output buffer will point to the backbuffer for layer 0 (backbuffer is the buffer not currently shown).

This will blend the AS and the PS in the output buffer.

Figure 31. Blending AS and PS

When that blit ends, a second blit is prepared. In this case, the PS and the output buffer will both point to the layer0 backbuffer and the AS will point to the label number 2. This effectively takes our previous render result and composes the new label on top of it.

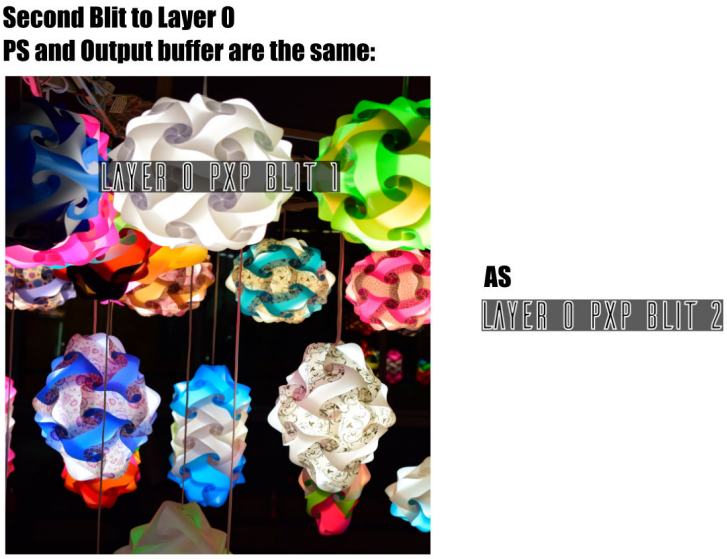

Figure 32. Same PS and output buffers

For the final layer 0 composition step, we do the same, but we use a third label.

Figure 33. Third blit to layer 0

For layer 1, we execute a single blit to populate its buffer, and we never touch it again.

The result of both layers being shown and populated is as follows:

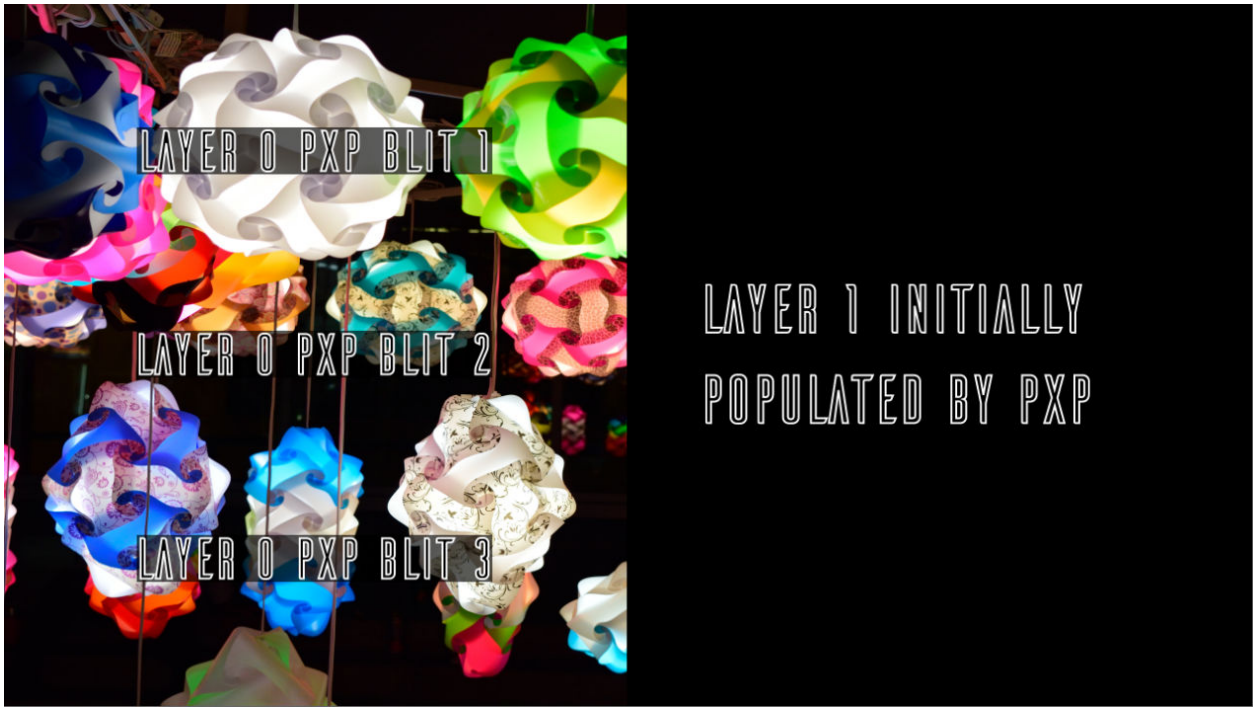

Figure 34. Result with both layers shown

4 LCDIFV2 display controller

The LCDIF module fetches memory buffers and sends them to the display controller. i.MXRT1170's LCDIFv2 module supports up to eight layers that can be blended on the fly, without any other accelerator intervention.

Each layer can be configured using different color formats, sizes, and positions and it can fetch buffers from any location in your memory map.

Among the supported formats, there is a particularly useful one (Index8BPP). This allows you to define a 32-bit-per-pixel image using a color look-up table and an index array. Using this approach, you can get the advantages of defining an ARGB8888 buffer without having to spend the memory doing it.

See the dial of section 2.1.2.1 and the color-banding artifacts we encountered. Let's revisit it again.

Instead of saving our image as an array of 720x1280 uint32_t, where each element of the array represents the color of each pixel in our image, we will save the image using two arrays. Our first array will be a 256 uint32_t array, where we will store as many colors as we can for our image. The second array will contain 720x1280 uint8_t values, where each value is an index to the first array.

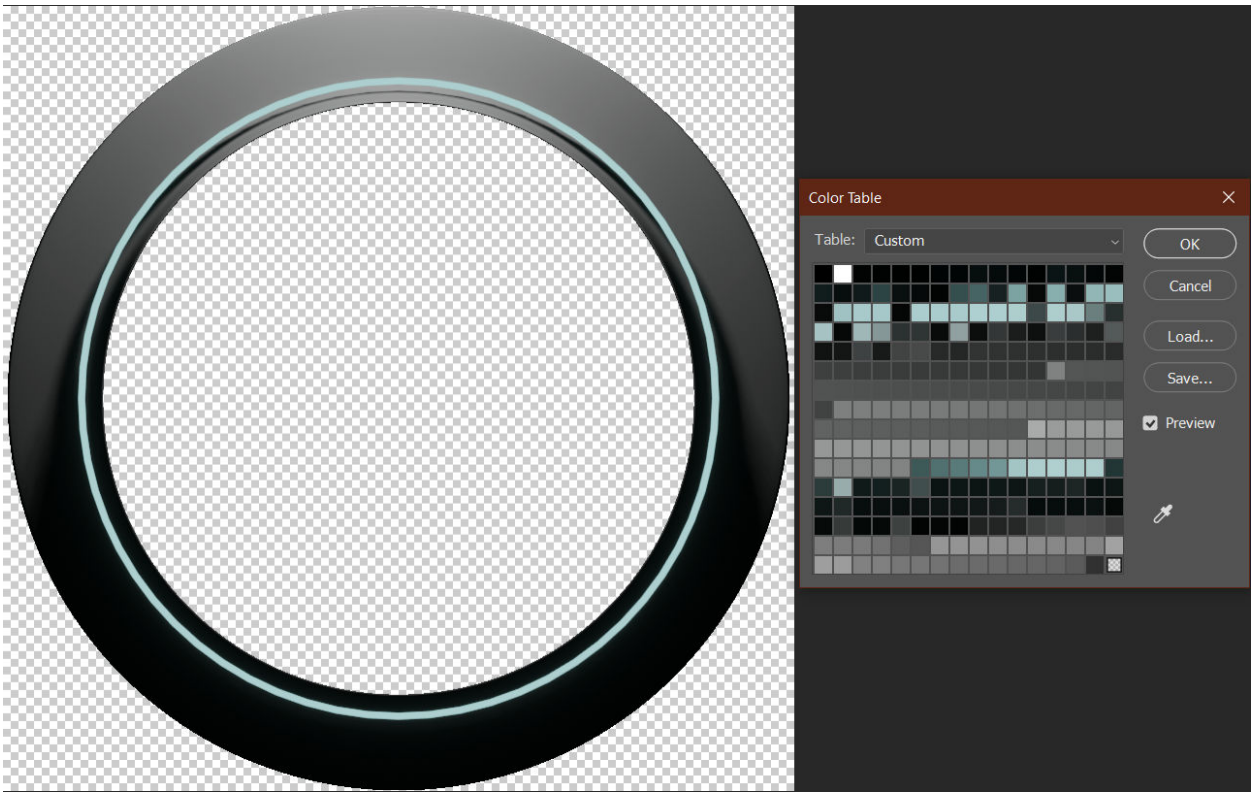

Figure 35. The pallete to the right of the dial is the color look-up table

Any recent image manipulation software allows you to create a color-lookup table. In the above figure, to the right of the dial, you can see the color table created from the image.

The LCDIFV2 module will fetch the appropriate color based on the indices from the uint8_t array. If your image does not have more than 256 colors (or it does not exceed this number by much), then the visual result will be equal or close to your original image.

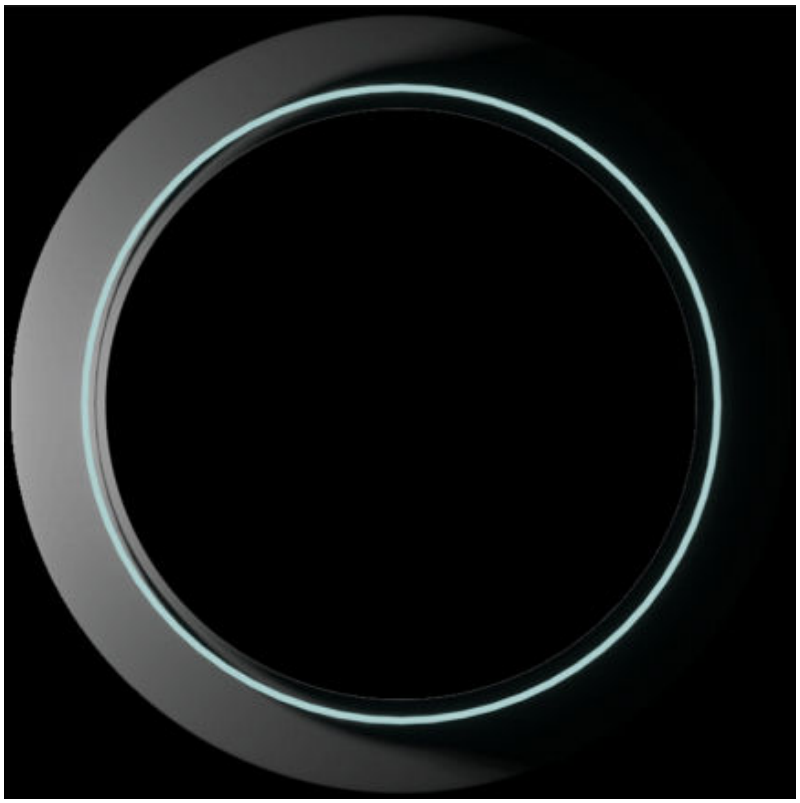

Figure 36. Render result of a Indexed color format

The “LCDIF_VGLite” project included in the software accompanying this application note shows you how to use the Index8Bpp color format on one layer and VGLite on another layer.

For layer 0, we will have a single 720x720 Index8BPP buffer. For layer 1, we will create two 416x416 ARGB8888 framebuffers.

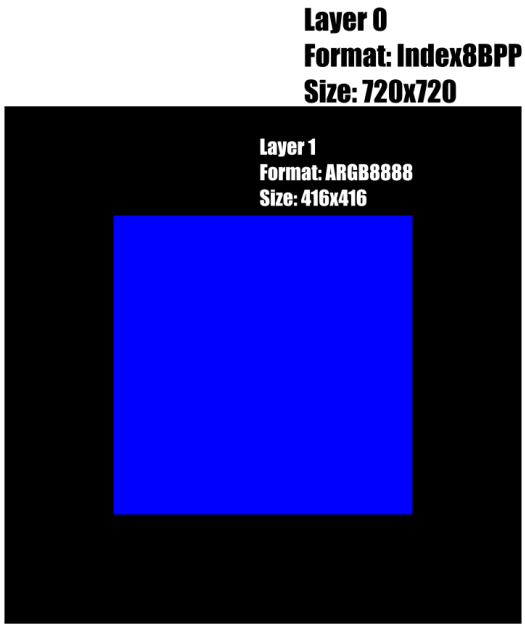

Figure 37. LCDIF_VGLite project layer configuration

The InitLCDIFV2 function will configure the layers as shown in the image above. One important consideration here is that we added a blend configuration so that layer 1 can be blended with layer 0 in the final step.

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

The kLCDIFV2_AlphaEmbedded mode uses the alpha on the image to blend the images.

The color lookup table is copied to the layer 0 configuration using the following function:

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

Layer 1 buffers are wrapped by vg_lite_buffer_t structures. In this way, we will be able to render to those buffers.

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];

The redrawVGLite routine performs the following drawing operations:

1. It clears by setting the whole 416x416 buffer to a solid blue color.
2. It applies an alpha mask to the buffer and gives it a round shape using the same technique as the one described in section 2.1.2.4.

The result shown on the display will look as follows:

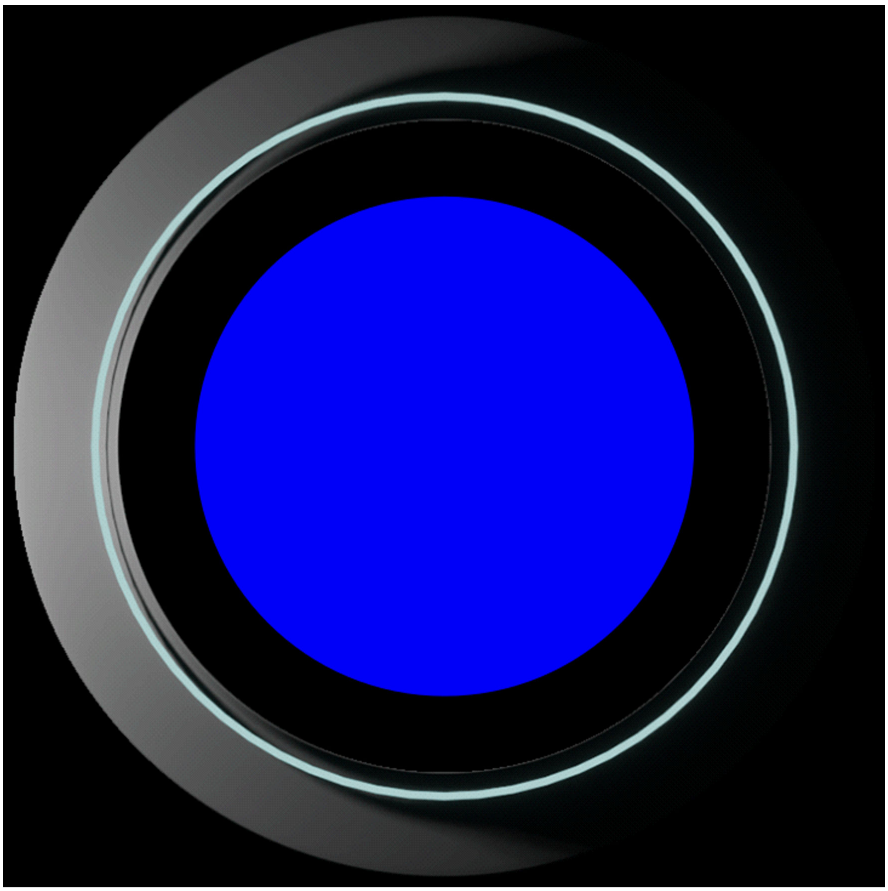

Figure 38. Result

5 Sample application implementations

Chapters 4 and 5 provide an introduction on how to use LCDIF with both PXP and VGLite.

In the following three sections, we will showcase how an application can benefit from using the three engines in unison to save resources and increase performance.

The use case we will introduce has the following requirements:

- Complex 1280x720 background.
- One 1280x48 semi-transparent top bar.
- Four icons.
- One window with vector-based content.

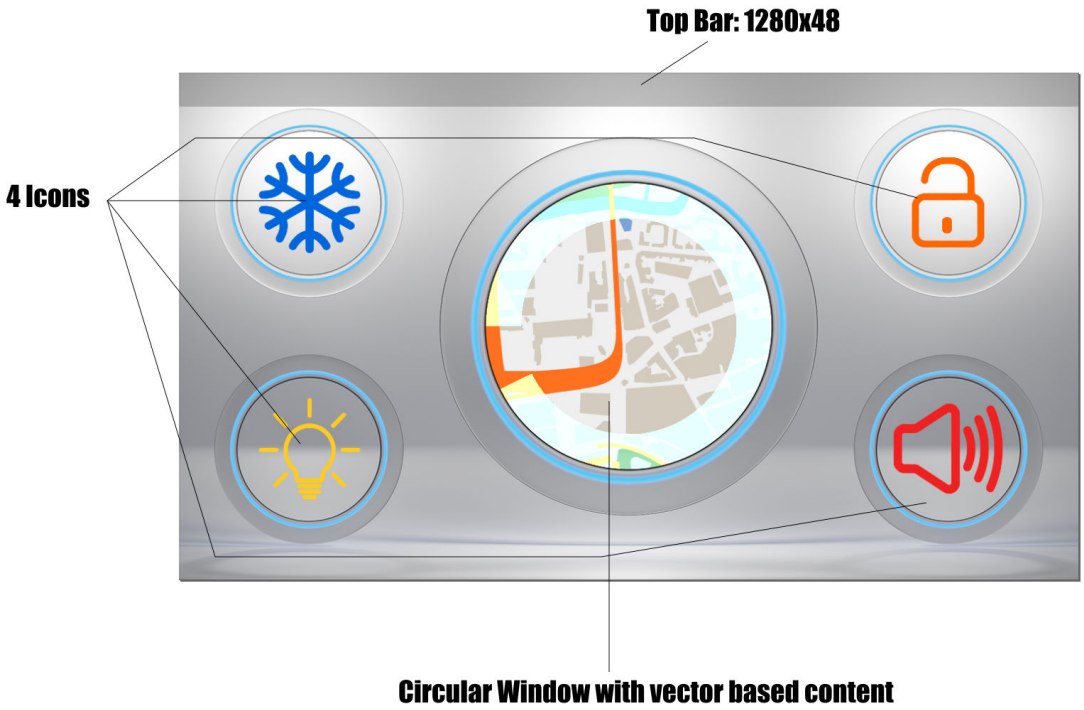

Figure 39. Sample application description

5.1 VGLite only implementation

Open the “Map” project. This example includes the VGLite-only implementation, in which all elements are rendered using VGLite. The render resources for this case are as follows:

- Display: two 720x1280 framebuffers, 32 bits per pixel, ARGB8888
- Background bitmap: 720x1280, 32 bits per pixel, ARGB8888
- Icon1 bitmap: 144x144, 16 bits per pixel, RGBA4444
- Icon2 bitmap: 144x144, 16 bits per pixel, RGBA4444
- Icon3 bitmap: 128x155, 16 bits per pixel, RGBA4444
- Icon4 bitmap: 128x103, 16 bits per pixel, RGBA4444
- Top bar bitmap: 48x1280, 16 bits per pixel, RGBA4444
- Vector render target: 416x416, 32 bits per pixel, ARGB8888

The per-frame draw process is as follows:

- Render the background to the display back-buffer using vg_lite_blit.
- Render the map to the offscreen target.
- Render the circular dial to the same offscreen target as the map.
- Compose the alpha mask in the offscreen target to achieve the circular shape.
- Render the offscreen target to the display back-buffer.
- Render the icons.
- Render the top bar.

- Swap the display back-buffer and the front-buffer to show the new frame.

Note: The map is rendered using a rendering library called "Elementary". This library uses VGLite to render vector objects and bitmaps. It is divided into two parts: an offline tool that converts SVG to Elementary structures and a runtime that interprets those structures into VGLite-usable code. The runtime source code is provided with the i.MXRT1170 SDK. For the offline tool, contact the NXP community.

The rendering result looks like this:

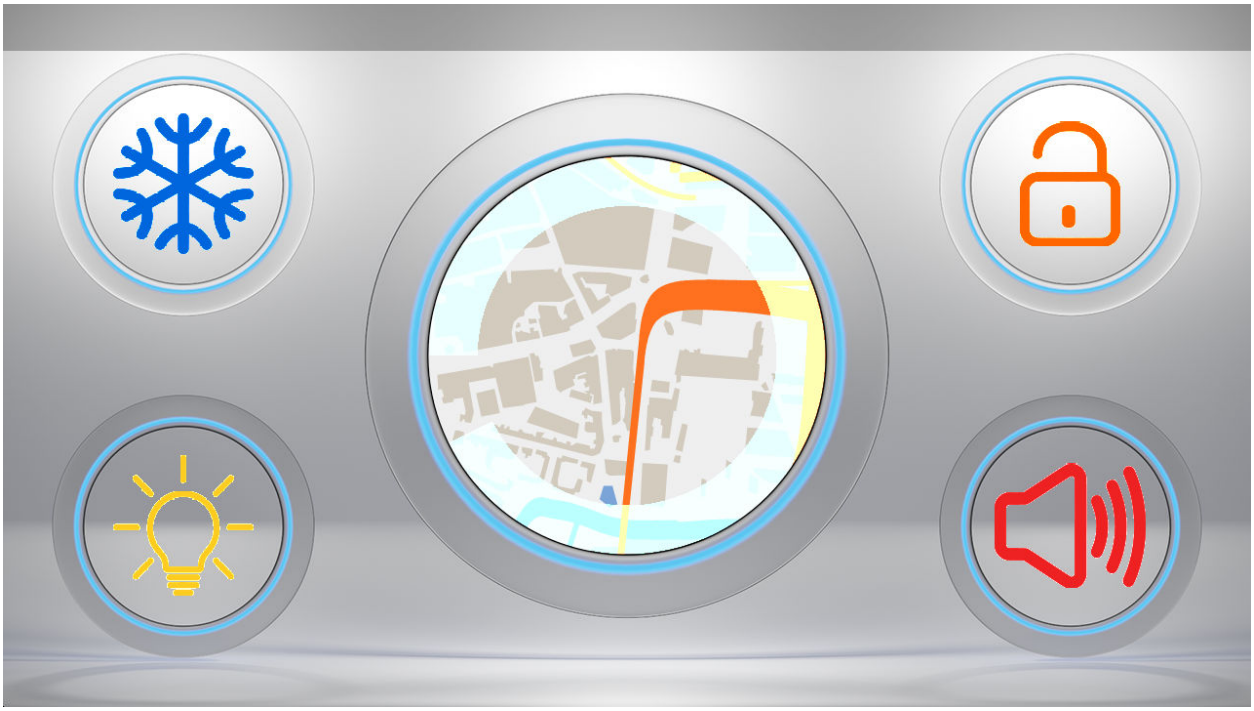

Figure 40. VGLite-only rendering result

The performance for this implementation is 14 FPS.

The memory used is as follows (all buffers are located in SDRAM):

- Framebuffers: 7 MB
- Background: 3.5 MB
- Icon1: 0.04 MB
- Icon2: 0.04 MB
- Icon3: 0.037 MB
- Icon4: 0.025 MB
- Top bar: 0.1172 MB
- Vector render target: 0.66 MB
- Total: 11.4 MB

5.2 VGLite + PXP implementation

The VGLite blit functions and PXP compose functions have a certain degree of overlap. Besides the matrix transformations (only available on VGLite), both engines' raster capabilities are similar. You can take advantage of this and distribute the load between the two engines.

In this case, we will use VGLite to render the map to an offscreen buffer and, while VGLite finishes, we will use PXP to render the rest of the screen.

The rendering resources for this case are the same as for the VGLite-only case with one additional 416x416 32 bits per pixel offscreen target.

Open the “Map_PXP_SingleTask” project and navigate to the redrawPXP function:

- While the code waits for the next display VSYNC signal, VGLite will render the map to an offscreen render target. In this case, we will have two offscreen render targets, so the PXP only composes the buffer that we are not rendering to.
- When the VSYNC signal arrives, we are ready to render to the display back-buffer and use the PXP Process Surface to render the background and the Alpha Surface to render VGLite’s offscreen render target that we are not currently using.
- Then we use the PXP to render the rest of the elements.

The rendering result looks like this:

Figure 41. VGLite and PXP rendering result

At a first glance, it looks equal to the VGLite-only case, but the icon outlines are blended differently. The latest version of the VGLite drivers (3.0.4) at the time of writing this document does not pre-multiply the alpha values.

The performance for this implementation is 21 FPS. The memory used is 12 MB.

5.3 VGLite + PXP + LCDIF implementation

The LCDIFV2 has eight layers that enable us to easily segment the rendering process, allowing the developer to refresh only those layers that must be refreshed at that moment. The developer can also control the blend mode, global alpha, and position of each layer. In this use case, we can segment the rendering into six layers, as shown in the following figure.

Figure 42. Rendering segmentation

Open the “Map_PXP_LCDIF_SingleTast” project. It contains the code required to set multiple layers and blend them together. It builds on the previous samples and shows the Elementary, VGLite, PXP, and LCDIF with multiple layer formats.

This project uses the following resources:

- Layer 0: 720x1280, indexed 8 bpp framebuffer
- Background bitmap: 720x1280, 32 bits per pixel, ARGB8888
- Layer 1: two 416x416, ARGB1555 framebuffer
- Vector render target: 416x416, 32 bits per pixel, ARGB8888
- Layer 2: 144x144, 16 bits per pixel, RGBA4444 framebuffer
- Icon1 bitmap: 144x144, 16 bits per pixel, RGBA4444
- Layer 3: 144x144, 16 bits per pixel, RGBA4444 framebuffer
- Icon2 bitmap: 144x144, 16 bits per pixel, RGBA4444
- Layer 4: 128x155, 16 bits per pixel, RGBA4444 framebuffer
- Icon3 bitmap: 128x155, 16 bits per pixel, RGBA4444
- Layer 5: 128x103, 16 bits per pixel, RGBA4444 framebuffer
- Icon4 bitmap: 128x103, 16 bits per pixel, RGBA4444
- Layer 6: 128x103, 16 bits per pixel, RGBA4444 framebuffer
- Top bar bitmap: 48x1280, 16 bits per pixel, RGBA4444

The render process for this application is the simplest:

- Each layer has its own framebuffer. For each layer, allocate also a bitmap to be the source of the content. When initializing the LCDIFV2 in APP_InitLcdif, copy the bitmaps to the framebuffers.
- When the LCDIF layers are initialized, use VGLite to render the map to an offscreen render target.

- Render the offscreen target to layer 1 back-buffer.

Each layer can be populated by any of the modules in i.MXRT1170 (CPU, GPU, PXP, and eDMA). The developer has complete freedom on which engine to use.

The rendering result looks like this:

Figure 43. LCDIFv2, VGLite and PXP rendering result

If you pay close attention, you will notice color banding in the highlights behind the dials. This is expected, because the color palette of the indexed background is limited to 256 colors. Those 256 colors are still enough for this case to generate an acceptable output.

The performance of this implementation is 44 FPS.

The memory used is as follows:

- Layer 0: 0.88 MB
- Background bitmap: 0.88 MB
- Layer 1: 0.66 MB
- Vector render target: 1.3 MB
- Layer 2: 0.04 MB
- Icon1 bitmap: 0.04 MB
- Layer 3: 0.04 MB
- Icon2 bitmap: 0.04 MB
- Layer 4: 0.037 MB
- Icon3 bitmap: 0.037 MB
- Layer 5: 0.025 MB
- Icon4 bitmap: 0.025 MB
- Layer 6: 0.1172 MB

- Top bar bitmap: 0.1172 MB

The total memory used is 4.2384 MB.

From the original implementation to the final implementation, the performance tripled while the memory usage was halved.

In the last case, the icons and the top bar were no longer rendered each frame, but this brings the advantages of using several layers and the simplicity to set them.

This is only a hypothetical sample application, but it is a good exercise to introduce the different accelerators on the platform. We encourage you to understand the capabilities of each one so you can get the most out of 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