NXP Semiconductors

Document Number: IMXGRAPHICUG
Rev. 0, 06/2018

i.MX Graphics User's Guide

Contents

Chapter 1 Introduction 6
Chapter 2 i.MX G2D API 6
2.1 Overview 6
2.2 Enumerations and structures 6
2.3 G2D function descriptions 10
2.4 Support of new operating system in G2D 16
2.5 Sample code for G2D API usage 16
2.6 Feature list on multiple platforms 19
Chapter 3 i.MX EGL and OGL Extension Support 20
3.1 Introduction 20
3.2 EGL extension support 20
3.3 OpenGL ES extension support 23
3.4 Extension GL_VIV_direct_texture 29
3.5 Extension GL_VIV_texture_border_clamp 32
Chapter 4 i.MX Framebuffer API 35
4.1 Overview 35
4.2 API data types and environment variables 35
4.3 API description and syntax 37
Chapter 5 OpenCL 44
5.1 Overview 44
5.2 Vivante OpenCL implementation 51
5.3 Optimization for OpenCL embedded profile 53
5.4 OpenCL Debug messages 56
5.5 Instruction Cache Availablity for i.MX Graphics 56
Chapter 6 OpenVX Introduction 58
6.1 Overview 58
6.2 Designing framework of OpenVX 58
6.3 OpenVX extension implementation 60
6.4 OpenCL functions compatible with Vivante vision 63
Chapter 7 Vulkan 66
7.1 OverView 66
7.2 Vivante Extension Support for Vulkan 66
Chapter 8 Multiple GPUs and Virtualization 68
8.1 Overview 68
8.2 Multi-GPU configurations 68

8.3 GPU affinity configuration 68
8.4 OpenCL on multi-GPU device 68
8.5 GPU virtualization configuration 69
Chapter 9 G2D compositor on Weston 70
9.1 Overview 70
9.2 Enabe G2D compositor 70
Chapter 10 XServer Video Driver 71
10.1 EXA driver 71
10.2 XRandR 72
Chapter 11 Advanced GPU Configuration 83
11.1 GPU Scaling Governor 83
11.2 GPU Device Cooling 83
Chapter 12 Vivante Software Tool Kit 83
12.1 Vivante Tool Kit overview 83
12.2 vEmulator 85
12.3 vShader 96
12.4 vCompiler 104
12.5 vTexture 108
12.6 vProfiler and vAnalyzer 112
12.7 Debug and performance counters 126
Chapter 13 GPU Tools 128
13.1 gpuinfo tool 128
13.2 gmem_info tool 130
13.3 Apitrace user guide 131
Chapter 14 GPU Memory Introduction 136
14.1 GPU memory overview 136
14.2 GPU memory pools 136
14.3 GPU memory allocators 136
14.4 GPU reserved memory 137
14.5 GPU memory base address 137
Chapter 15 Application Programming Recommendations 139
15.1 Understand the system configuration and target application 139
15.2 Optimize off chip data transfer such as accessing off-chip DDR memory/mobile DDR memory 139
15.3 Avoid W-Clipping issue in the Application Program 139
15.4 Avoid GPU hang and data corruption when use occlusion query 140
15.5 Avoid random cache or memory accesses 140

15.6 Optimize your use of system memory 140
15.7 Target a fixed frame rate that is visibly smooth.....140
15.8 Minimize GL state changes 141
15.9 Batch primitives to minimize the number of draw calls 141
15.10 Perform calculations per vertex instead of per fragment/pixel 141
15.11 Enable early-Z, hierarchical-Z and back face culling 141
15.12 Use branching carefully 142
15.13 Do not use static or stack data as vertex data - use VBOs instead 142
15.14 Use dynamic VBO if data is changing frame by frame 142
15.15 Tessellate your data so that Hierarchical Z (HZ) can do its job 143
15.16 Use dynamic textures as a texture cache (texture atlas).....143
15.17 If you use many small triangle strips, stitch them together 143
15.18 Specify EGL configuration attributes precisely 143
15.19 Use aligned texture/render buffers 143
15.20 Disable MSAA rendering unless high quality is needed 144
15.21 Avoid partial clears 144
15.22 Avoid mask operations 144
15.23 Use MIPMAP textures.....144
15.24 Use compressed textures if constricted by RAM/ROM budget 144
15.25 Draw objects from near to far if possible 144
15.26 Avoid indexed triangle strips. 144
15.27 Vertex attribute stride should not be larger than 256 bytes 145
15.28 Avoid binding buffers to mixed index/vertex array 145
15.29 Avoid using CPU to update texture/buffer contexts during render 145
15.30 Avoid frequent context switching.....145
15.31 Optimize resources within a shader 145
15.32 Avoid using glScissor Clear for small regions 145
15.33 Use PRE to accelerate data transfer 145
15.34 i.MX 8QuadMax dual-GPU performance 146
Chapter 16 Demo Framework.....147
16.1 Summaries.....147
16.2 Introduction 149
16.3 Design overview 150
16.4 High level overview 151
16.5 Demo application details 152
16.6 Demo playback.....156

16.7 Helper Class Overview 157
16.8 FslBuild scripts 165
16.9 Android SDK+NDK on windows build guide 166
16.10 Ubuntu build guide 171
16.11 Windows build guide 175
16.12 Yocto build guide 179
16.13 FslContentSync.py notes 187
16.14 Known limitations 188
16.15 Upgrading samples from earlier SDKs 189
16.16 What's new 190
Chapter 17 Environment Variables Summary 193
17.1 Environment variable for drivers and HAL 193
17.2 Environment variable for compiler 194

Chapter 1 Introduction

The purpose of this document is to provide information on graphic APIs and driver support. Each chapter describes a specific set of APIs or driver integration as well as specific hardware acceleration customization. The target audiences for this document are developers writing graphics applications or video drivers.

Chapter 2 i.MX G2D API

2.1 Overview

The G2D Application Programming Interface (API) is designed to be easy to understand and to use the 2D Bit blit (BLT) function. It allows the user to implement the customized applications with simple interfaces. It is hardware and platform independent for i.MX 2D Graphics.

G2D API supports the following features but is not limited to these:

- Simple BLT operation from source to destination
- Alpha blending for source and destination with Porter-Duff rules
- High-performance memory copy from source to destination
- Up-scaling and down-scaling from source to destination
- 90/180/270 degree rotation from source to destination
- Horizontal and vertical flip from source to destination
- Enhanced visual quality with dither for pixel precision-loss
- High performance memory clear for destination
- Pixel-level cropping for source surface
- Global alpha blending for source only
- Asynchronous mode and sync
- Contiguous memory allocator
- Support VG engine

The G2D API document includes a detailed interface description and sample code for reference.

The API is designed with C-Style coding and can be used in both C and C++ applications.

G2D API supports the following features but is not limited to these:

- Multi source blit

2.2 Enumerations and structures

This chapter describes all enumeration and structure definitions in G2D.

2.2.1 g2d_format enumeration

This enumeration describes the pixel format for source and destination.

Table 1. g2d_format enumeration

Name	Numeric	Description
G2D_RGB565	0	RGB565 pixel format
G2D_RGBA8888	1	32-bit RGBA pixel format
G2D_RGBX8888	2	32-bit RGBX without alpha blending
G2D_BGRA8888	3	32-bit BGRA pixel format
G2D_BGRX8888	4	32-bit BGRX without alpha blending

G2D_BGR565	5	16-bit BGR565 pixel format
G2D_ARGBA8888	6	32-bit ARGB pixel format
G2D_ABGR8888	7	32-bit ABGR pixel format
G2D_XRGB8888	8	32-bit XRGB without alpha
G2D_XBGR8888	9	32-bit XBGR without alpha
G2D_RGB888	10	24-bit RGB
G2D_NV12	20	Y plane followed by interleaved U/V plane
G2D_I420	21	Y, U, V are within separate planes
G2D_YV12	22	Y, V, U are within separate planes
G2D_NV21	23	Y plane followed by interleaved V/U plane
G2D_YUYV	24	Interleaved Y/U/Y/V plane
G2D_YVYU	25	Interleaved Y/V/Y/U plane
G2D_UYVY	26	Interleaved U/Y/V/Y plane
G2D_VYUY	27	Interleaved V/Y/U/Y plane
G2D_NV16	28	Y plane followed by interleaved U/V plane
G2D_NV61	29	Y plane followed by interleaved V/U plane

2.2.2 g2d_blend_func enumeration

This enumeration describes the blend factor for source and destination.

Table 2. g2d_blend_func enumeration

Name	Numeric	Description
G2D_ZERO	0	Blend factor with 0
G2D_ONE	1	Blend factor with 1
G2D_SRC_ALPHA	2	Blend factor with source alpha
G2D_ONE_MINUS_SRC_ALPHA	3	Blend factor with 1 - source alpha
G2D_DST_ALPHA	4	Blend factor with destination alpha
G2D_ONE_MINUS_DST_ALPHA	5	Blend factor with 1 - destination alpha
G2D_PRE_MULTIPLIED_ALPHA	0 x 10	Extensive blend as pre-multiplied alpha
G2D_DEMULTIPLY_OUT_ALPHA	0 x 20	Extensive blend as demultiply out alpha

2.2.3 g2d_cap_mode enumeration

This enumeration describes the alternative capability in 2D BLT.

Table 3. g2d_cap_mode enumeration

Name	Numeric	Description
G2D_BLEND	0	Enable alpha blend in 2D BLT
G2D_DITHER	1	Enable dither in 2D BLT
G2D_GLOBAL_ALPHA	2	Enable global alpha in blend

Note: G2D_GLOBAL_ALPHA is only valid when G2D_BLEND is enabled.

2.2.4 g2d_rotation enumeration

This enumeration describes the rotation mode in 2D BLT.

Table 4. g2d_rotation enumeration

Name	Numeric	Description
G2D_ROTATION_0	0	No rotation
G2D_ROTATION_90	1	Rotation with 90 degree
G2D_ROTATION_180	2	Rotation with 180 degree
G2D_ROTATION_270	3	Rotation with 270 degree
G2D_FLIP_H	4	Horizontal flip
G2D_FLIP_V	5	Vertical flip

2.2.5 g2d_cache_mode enumeration

This enumeration describes the cache operation mode.

Table 5. g2d_cache_mode enumeration

Name	Numeric	Description
G2D_CACHE_CLEAN	0	Clean the cacheable buffer
G2D_CACHE_FLUSH	1	Clean and invalidate cacheable buffer
G2D_GLOBAL_INVALIDATE	2	Invalidate the cacheable buffer

2.2.6 g2d_hardware_type enumeration

This enumeration describes the supported hardware type.

Table 6. g2d_hardware_type enumeration

Name	Numeric	Description
G2D_HARDWARE_2D	0	2D hardware type by default
G2D_HARDWARE_VG	1	VG hardware type

2.2.7 g2d_surface structure

This structure describes the surface with operation attributes.

Table 7. g2d_surface structure

g2d_surface Members	Type	Description
format	g2d_format	Pixel format of surface buffer
planes[3]	Int	Physical addresses of surface buffer
left	Int	Left offset in blit rectangle
top	Int	Top offset in blit rectangle
right	Int	Right offset in blit rectangle

bottom	Int	Left offset in blit rectangle
stride	Int	RGB/Y stride of surface buffer
width	Int	Surface width in pixel unit
height	Int	Surface height in pixel unit
blendfunc	g2d_blend_func	Alpha blend mode
global_alpha	Int	Global alpha value 0~255
clrcolor	Int	Clear color is 32bit RGBA
rot	g2d_rotation	Rotation mode

Notes:

- RGB and YUV formats can be set in source surface, but only RGB format can be set in destination surface.
- RGB pixel buffer only uses planes [0], buffer address is with 16bytes alignment on i.MX 6Quad/Dual/DualLite/Solo/SoloLite, 1 pixel alignment on i.MX 6QuadPlus.
- NV12: Y in planes [0], UV in planes [1], with 64bytes alignment,
- I420: Y in planes [0], U in planes [1], U in planes [2], with 64 bytes alignment
- The cropped region in source surface is specified with left, top, right and bottom parameters.
- RGB stride alignment is 16bytes on i.MX 6Quad/Dual/DualLite/Solo/SoloLite, 1 pixel on i.MX 6QuadPlus, both for source and destination surface.
- NV12 stride alignment is 8bytes for source surface, UV stride = Y stride,
- I420 stride alignment is 8bytes for source surface, U stride=V stride = ½ Y stride.
- G2D_ROTATION_0/G2D_FLIP_H/G2D_FLIP_V shall be set in source surface, and the clockwise rotation degree shall be set in destination surface.
- Application should calculate the rotated position and set it for destination surface.
- The geometry definition of surface structure is described as follows.

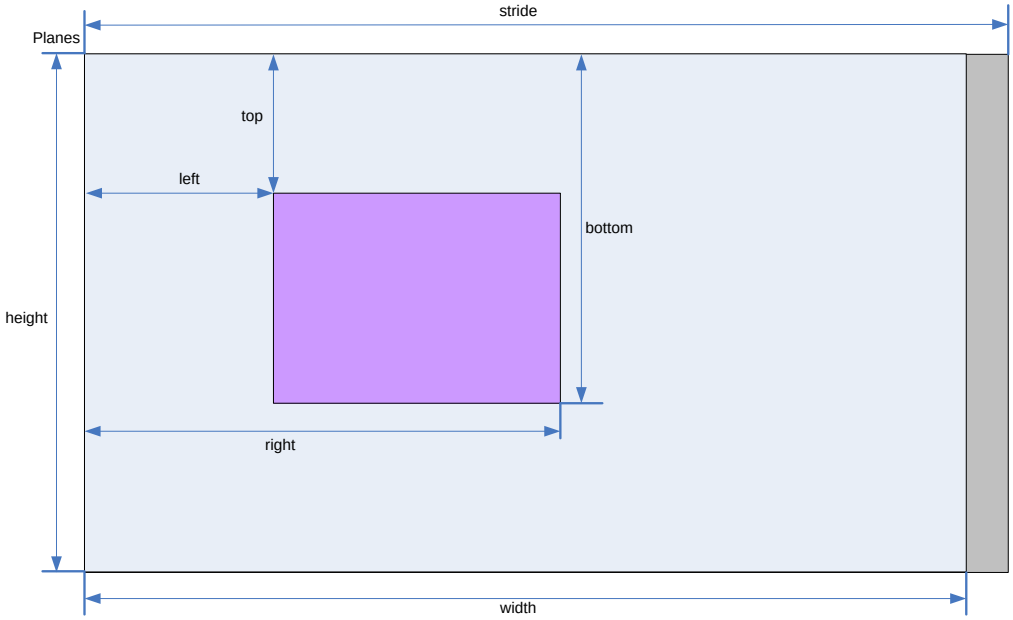


Figure 1 g2d_surface structure

2.2.8 g2d_buf structure

This structure describes the buffer used as G2D interfaces.

Table 8. g2d_buf structure

g2d_buf Members	Type	Description
buf_handle	void *	The handle associated with buffer
buf_vaddr	void *	Virtual address of the buffer
buf_paddr	int	Physical address of the buffer
buf_size	int	The actual size of the buffer

2.2.9 g2d_surface_pair structure

This structure binds one source g2d_surface and one destination g2d_surface as a pair. When doing multi-source blit, they are one-to-one correspondent.

Table 9. g2d_surface_pair structure

g2d_surface_pair Members	Type	Description
s	g2d_surface	Source g2d_surface
d	g2d_surface	Destination g2d_surface

2.2.10 g2d_feature enumeration

This enumeration describes the features in G2D BLT.

Table 10. g2d_feature enumeration

Name	Numeric	Description
G2D_SCALING	0	Scaling
G2D_ROTATION	1	Rotation
G2D_SRC_YUV	2	Source YUV format
G2D_DST_YUV	3	Destination YUV format
G2D_MULTI_SOURCE_BLT	4	Multisource blit

2.3 G2D function descriptions

2.3.1 g2d_open

Description:

Open a G2D device and return a handle.

Syntax:

```
int g2d_open (void **handle);
```

Parameters:

handle Pointer to receive G2D device handle

Returns:

Success with 0, fail with -1

2.3.2 g2d_close

Description:
Close G2D device with the handle.

Syntax:
int g2d_close (void *handle);

Parameters:
handle G2D device handle

Returns:
Success with 0, fail with -1

2.3.3 g2d_make_current

Description:
Set the specific hardware type for current context, and the default is G2D_HARDWARE_2D.

Syntax:
int g2d_make_current (void *handle, enum g2d_hardware_type type);

Parameters:
handle G2D device handle
type G2D hardware type

Returns:
Success with 0, fail with -1

2.3.4 g2d_clear

Description:
Clear a specific area.

Syntax:
int g2d_clear (void *handle, struct g2d_surface *area);

Parameters:
handle G2D device handle
area The area to be cleared

Returns:
Success with 0, fail with -1

2.3.5 g2d_blit

Description:
G2D blit from source to destination with alternative operation (Blend, Dither, etc.).

Syntax:

```
int g2d_blit (void *handle, struct g2d_surface *src, struct g2d_surface *dst);
```

Parameters:

handle	G2D device handle
src	source surface
dst	destination surface

Returns:

Success with 0, fail with -1

2.3.6 g2d_copy

Description:

G2D copy with specified size.

Syntax:

```
int g2d_copy (void *handle, struct g2d_buf *d, struct g2d_buf* s, int size);
```

Parameters:

handle	G2D device handle
d	destination buffer
s	source buffer
size	copy bytes

Limitations:

If the destination buffer is cacheable, it must be invalidated before g2d_copy due to the alignment limitation of G2D driver.

Returns:

Success with 0, fail with -1

2.3.7 g2d_query_cap

Description:

Query the alternative capability enablement.

Syntax:

```
int g2d_query_cap (void *handle, enum g2d_cap_mode cap, int *enable);
```

Parameters:

handle	G2D device handle
cap	G2D capability to query
enable	Pointer to receive G2D capability enablement

Returns: Success with 0, fail with -1

2.3.8 g2d_enable

Description:

Enable G2D capability with the specific mode.

Syntax:

```
int g2d_enable (void *handle, enum g2d_cap_mode cap);
```

Parameters:

handle	G2D device handle
cap	G2D capability to enable

Returns:

Success with 0, fail with -1

2.3.9 g2d_disable

Description:

Enable G2D capability with the specific mode.

Syntax:

```
int g2d_disable (void *handle, enum g2d_cap_mode cap);
```

Parameters:

handle	G2D device handle
cap	G2D capability to disable

Returns:

Success with 0, fail with -1

2.3.10 g2d_cache_op

Description:

Perform cache operations for the cacheable buffer allocated through the G2D driver.

Syntax:

```
int g2d_cache_op (struct g2d_buf *buf, enum g2d_cache_mode op);
```

Parameters:

buf	the buffer to be handled with cache operations
op	cache operation type

Returns:

Success with 0, fail with -1

2.3.11 g2d_alloc

Description:

Allocate a buffer through G2D device

Syntax:

```
struct g2d_buf *g2d_alloc (int size, int cacheable);
```

Parameters:

size	allocated bytes
cacheable	0, non-cacheable, 1, cacheable attribute defined by system

Returns:

Success with valid G2D buffer pointer, fail with 0

2.3.12 g2d_free

Description:

Free the buffer through G2D device.

Syntax:

```
int g2d_free (struct g2d_buf *buf);
```

Parameters:

buf	G2D buffer to free

Returns:

Success with 0, fail with -1

2.3.13 g2d_flush

Description:

Flush G2D command and return without completing pipeline.

Syntax:

```
int g2d_flush (void *handle);
```

Parameters:

handle	G2D device handle

Returns:

Success with 0, fail with -1

2.3.14 g2d_finish

Description:

Flush G2D command and then return when pipeline is finished.

Syntax:

```
int g2d_finish (void *handle);
```

Parameters:

handle	G2D device handle

Returns:

Success with 0, fail with -1

2.3.15 g2d_multi_blit

Description:

Blit multiple sources to one destination.

Syntax:

```
int g2d_multi_blit (void *handle, struct g2d_surface_pair *sp[], int layers);
```

Parameters:

handle	G2D device handle
sp	array in which elements point to g2d_surface_pair
layers	number of the source layers that need to be blited

Returns:

Success with 0, fail with -1

Note:

There are some restrictions for this API that we should be aware of.

- This API only works on the i.MX 6DualPlus/QuadPlus platform.
- The maximum number of the source layers that can be blited one time is 8.
- Although g2d_surface_pair binds one source g2d_surface and one destination g2d_surface as a pair, it only supports one destination surface. The relationship between the source and destination is many to one, but each source surface can be set separately and differently, and its dimension, stride, rotation, and format can differ with that of the destination surface.
- The rotation of the destination surface is set to 0 degree by defaut, and cannot be changed.
- The key restriction is that the destination rectangle cannot be set, which means that the destination rectangle must be the same as the source rectangle. Therefore, if the source rectangle is set to (l, t, r, b), the destination rectangle should also be set to (l, t, r, b) by hardware. In the chapter on multi source blit (2.4.4), as it makes no sense to set the destination rectangles, we just set all of them to (0, 0, width, height) for future extension.

2.3.16 g2d_query_hardware

Description:

Query whether 2D and VG hardware are available in the current G2D.

Syntax:

```
int g2d_query_hardware (void *handle, enum g2d_hardware_type type, int  *available);
```

Parameters:

handle	G2D device handle
type	G2D hardware type
available	Pointer to receive G2D hardware type availability

Returns:

Success with 0, fail with -1

2.3.17 g2d_query_feature

Description:

Query if the features are available in G2D BLT.

Syntax:

```
int g2d_query_feature (void *handle, enum g2d_feature feature, int *available);
```

Parameters:

handle	G2D device handle

feature	G2D feature in g2d_blit
available	Pointer to receive G2D feature availability

Returns:
Success with 0, fail with -1

2.4 Support of new operating system in G2D

G2D code is independent on operating system (OS) except of buffer allocation. Allocating the memory for buffer is made by mechanism that is offered by each OS differently. The code for allocation is located in [G2D repository copy]/source/os/[OS name]. Therefore, supporting new OS includes the following steps:

1. Create a new folder in **[G2D repository copy]/source/os/** with the name of the new OS and update implementation in the included source code according to the new OS allocation mechanism.
2. When creating new makefiles for the OS, include the files from the new folder.
3. The test named **overlay_test** contains the OS dependent code. For supporting the new OS in this test, create new folder in **[G2D repository copy]/test/overlay_test/os** and update the code according to the new OS mechanism for display initialization. Also update makefiles to include code from the new folder.

2.5 Sample code for G2D API usage

This chapter provides the brief prototype code with G2D API.

2.5.1 Color space conversion from YUV to RGB

```
g2d_open(&handle);

src.planes[0] = buf_y;
src.planes[1] = buf_u;
src.planes[2] = buf_v;
src.left = crop.left;
src.top = crop.top;
src.right = crop.right;
src.bottom = crop.bottom;
src.stride = y_stride;
src.width = y_width;
src.height = y_height;
src.rot    = G2D_ROTATION_0;
src.format = G2D_I420;

dst.planes[0] = buf_rgba;
dst.left = 0;
dst.top = 0;
dst.right = disp_width;
dst.bottom = disp_height;
dst.stride = disp_width;
dst.width = disp_width;
dst.height = disp_height;
dst.rot    = G2D_ROTATION_0;
dst.format = G2D_RGBA8888;
```

```
g2d_blit(handle, &src, &dst);
g2d_finish(handle);

g2d_close(handle);
```

2.5.2 Alpha blend in source over mode

```
g2d_open(&handle);

src.planes[0] = src_buf;
src.left = 0;
src.top = 0;
src.right = test_width;
src.bottom = test_height;
src.stride = test_width;
src.width = test_width;
src.height = test_height;
src.rot    = G2D_ROTATION_0;
src.format = G2D_RGBA8888;
src.blendfunc = G2D_ONE;

dst.planes[0] = dst_buf;
dst.left = 0;
dst.top = 0;
dst.right = test_width;
dst.bottom = test_height;
dst.stride = test_width;
dst.width = test_width;
dst.height = test_height;
dst.format = G2D_RGBA8888;
dst.rot    = G2D_ROTATION_0;
dst.blendfunc = G2D_ONE_MINUS_SRC_ALPHA;

g2d_enable(handle,G2D_BLEND);
g2d_blit(handle, &src, &dst);
g2d_finish(handle);
g2d_disable(handle,G2D_BLEND);

g2d_close(handle);
```

2.5.3 Source cropping and destination rotation

```
g2d_open(&handle);

src.planes[0] = src_buf;
src.left = crop.left;
src.top = crop.left;
src.right = crop.right;
```

```
        src.bottom = crop.bottom;
        src.stride = src_stride;
        src.width = src_width;
        src.height = src_height;
        src.format = G2D_RGBA8888;
        src.rot    = G2D_ROTATION_0;//G2D_FLIP_H or G2D_FLIP_V

        dst.planes[0] = dst_buf;
        dst.left = 0;
        dst.top = 0;
        dst.right = dst_width;
        dst.bottom = dst_height;
        dst.stride = dst_width;
        dst.width = dst_width;
        dst.height = dst_height;
        dst.format = G2D_RGBA8888;
        dst.rot    = G2D_ROTATION_90;

        g2d_blit(handle, &src, &dst);
        g2d_finish(handle);

        g2d_close(handle);
```

2.5.4 Multi source blit

```
const int layers = 8;
struct g2d_buf *d_buf;
struct g2d_buf *mul_s_buf[layers];
struct g2d_surface_pair *sp[layers];

g2d_open(&handle)

for(n = 0; n < layers; n++) {
        sp[n] = (struct g2d_surface_pair *)malloc(sizeof(struct g2d_surface_pair));
}

d_buf = g2d_alloc(test_width * test_height * 4, 0);
for(n = 0; n < layers; n++) {
        mul_s_buf[n] = g2d_alloc(test_width * test_height * 4, 0);
}

for(n = 0; n < layers; n++) {
        sp[n]->s.left = img_info_ptr[n]->img_left;
        sp[n]->s.top = img_info_ptr[n]->img_top;
        sp[n]->s.right =  img_info_ptr[n]->img_right;
        sp[n]->s.bottom = img_info_ptr[n]->img_bottom;

        sp[n]->s.stride = img_info_ptr[n]->img_width;
        sp[n]->s.width =  img_info_ptr[n]->img_width;
        sp[n]->s.height = img_info_ptr[n]->img_height;
```

```
        sp[n]->s.rot =    img_info_ptr[n]->img_rot;
        sp[n]->s.format = img_info_ptr[n]->img_format;
        sp[n]->s.planes[0] = mul_s_buf[n]->buf_paddr;
}

sp[0]->d.left = 0;
sp[0]->d.top  = 0;
sp[0]->d.right = test_width;
sp[0]->d.bottom = test_height;

sp[0]->d.stride = test_width;
sp[0]->d.width  = test_width;
sp[0]->d.height = test_height;
sp[0]->d.format = G2D_RGBA8888;
sp[0]->d.rot    = G2D_ROTATION_0;
sp[0]->d.planes[0] = d_buf->buf_paddr;
for(n = 1; n < layers; n++) {
        sp[n]->d = sp[0]->d;
}

g2d_multi_blit(handle, sp, layers);
g2d_finish(handle);
for(n = 0; n < layers; n++)
        g2d_free(mul_s_buf[n]);
g2d_free(d_buf);
g2d_close(handle);
```

2.6 Feature list on multiple platforms

This user guide is for multiple platforms, such as i.MX 6 and i.MX 8, and the hardwares for the G2D implementation are different on those platforms, so some G2D features are also different.
For example, the G2D_YVYU and G2D_VYUY formats are not supported on the i.MX8, and the g2d_multi_blit function only works on the i.MX 6DualPlus/QuadPlus. Therefore, we list those differences in the following feature table.

Table 11. Feature list on multiple platforms

Feature	i.MX 6		i.MX 7	i.MX 8
	Solo/Dual/Quad	DualPlus/QuadPlus	ULP	QuadMax
G2D_YVYU	Yes	Yes	Yes	No
G2D_VYUY	Yes	Yes	Yes	No
G2D_HARDWARE_VG	Yes	Yes	Yes	No
G2D_MULTI_SOURCE_BLT	No	Yes	Yes	No
g2d_cache_op	Yes	Yes	Yes	No

Chapter 3 i.MX EGL and OGL Extension Support

3.1 Introduction

The following tables list the level of support for EGL and OES extensions available with i.MX hardware and software. Support levels are current as of the date of the document and subject to change.
Two tables are provided. The first table lists the EGL interface extensions. The second table lists extensions for OpenGL ES 1.1, OpenGL ES 2.0, and OpenGL ES 3.0.

Key:
Extension Name and Number: Each listed extension is derived from the relevant khronos.org webpage list and includes the extension number as well as a hyperlink to the khronos description of the extension.
Yes: Support is currently available.
No: Support is not available. (Reasons for lack of support may vary: the extension may be proprietary or obsolete, or not applicable to the specified OES version.)
N/A: Support is not provided as the extension is not applicable in this and subsequent versions of the specification.

3.2 EGL extension support

The following table includes the list of all current EGL Extensions and indicates their support level.
(list from www.khronos.org/registry/egl/ as of 1/24/2013)

Table 12. EGL extension support

EGL Extension Number, Name and hyperlink	Supported
1. EGL_KHR_config_attribs	
2. EGL_KHR_lock_surface	YES
3. EGL_KHR_image	YES
4. EGL_KHR_vg_parent_image	
5. EGL_KHR_gl_texture_2D_image EGL_KHR_gl_texture_cubemap_image EGL_KHR_gl_texture_3D_image EGL_KHR_gl_renderbuffer_image	YES YES no YES
6. EGL_KHR_reusable_sync	YES
8. EGL_KHR_image_base	YES
9. EGL_KHR_image_pixmap	YES
10. EGL_IMG_context_priority	
16. EGL_KHR_lock_surface2	
17. EGL_NV_coverage_sample	
18. EGL_NV_depth_nonlinear	
19. EGL_NV_sync	
20. EGL_KHR_fence_sync	YES
24. EGL_HI_clientpixmap	
25. EGL_HI_colorformats	
26. EGL_MESA_drm_image	
27. EGL_NV_post_sub_buffer	
28. EGL_ANGLE_query_surface_pointer	

29. EGL_ANGLE_surface_d3d_texture_2d_share_handle	
30. EGL_NV_coverage_sample_resolve	
31. EGL_NV_system_time	
32. EGL_KHR_stream	
33. EGL_KHR_stream_consumer_gltexture	
34. EGL_KHR_stream_producer_eglsurface	
35. EGL_KHR_stream_producer_aldatalocator	
36. EGL_KHR_stream_fifo	
37. EGL_EXT_create_context_robustness	YES
38. EGL_ANGLE_d3d_share_handle_client_buffer	
39. EGL_KHR_create_context	YES
40. EGL_KHR_surfaceless_context	
41. EGL_KHR_stream_cross_process_fd	
42. EGL_EXT_multiview_window	
43. EGL_KHR_wait_sync	
44. EGL_NV_post_convert_rounding	
45. EGL_NV_native_query	
46. EGL_NV_3dvision_surface	
47. EGL_ANDROID_framebuffer_target	
48. EGL_ANDROID_blob_cache	
49. EGL_ANDROID_image_native_buffer	YES
50. EGL_ANDROID_native_fence_sync	YES
51. EGL_ANDROID_recordable	
52. EGL_EXT_buffer_age	YES
53. EGL_EXT_image_dma_buf_import	YES
54. EGL_ARM_pixmap_multisample_discard	
55. EGL_EXT_swap_buffers_with_damage	
56. EGL_NV_stream_sync	
57. EGL_EXT_platform_base	
58. EGL_EXT_client_extensions	
59. EGL_EXT_platform_x11	
60. EGL_KHR_cl_event	
61. EGL_KHR_get_all_proc_addresses EGL_KHR_client_get_all_proc_addresses	
62. EGL_MESA_platform_gbm	
63. EGL_EXT_platform_wayland	
64. EGL_KHR_lock_surface3	
65. EGL_KHR_cl_event2	
66. EGL_KHR_gl_colorspace	
67. EGL_EXT_protected_surface	YES
68. EGL_KHR_platform_android	
69. EGL_KHR_platform_gbm	
70. EGL_KHR_platform_wayland	YES
71. EGL_KHR_platform_x11	
72. EGL_EXT_device_base	
73. EGL_EXT_platform_device	
74. EGL_NV_device_cuda	

75. EGL_NV_cuda_event	
76. EGL_TIZEN_image_native_buffer	
77. EGL_TIZEN_image_native_surface	
78. EGL_EXT_output_base	
79. EGL_EXT_device_drm EGL_EXT_output_drm	
80. EGL_EXT_device_openwf EGL_EXT_output_openwf	
81. EGL_EXT_stream_consumer_egloutput	
83. EGL_KHR_partial_update	
84. EGL_KHR_swap_buffers_with_damage	
85. EGL_ANGLE_window_fixed_size	
86. EGL_EXT_yuv_surface	
87. EGL_MESA_image_dma_buf_export	
88. EGL_EXT_device_enumeration	
89. EGL_EXT_device_query	
90. EGL_ANGLE_device_d3d	
91. EGL_KHR_create_context_no_error	
92. EGL_KHR_debug	
93. EGL_NV_stream_metadata	
94. EGL_NV_stream_consumer_gltexture_yuv	
95. EGL_IMG_image_plane_attribs	
96. EGL_KHR_mutable_render_buffer	
97. EGL_EXT_protected_content	
98. EGL_ANDROID_presentation_time	
99. EGL_ANDROID_create_native_client_buffer	
100. EGL_ANDROID_front_buffer_auto_refresh	
101. EGL_KHR_no_config_context	
102. EGL_KHR_context_flush_control	
103. EGL_ARM_implicit_external_sync	
104. EGL_MESA_platform_surfaceless	
105. EGL_EXT_image_dma_buf_import_modifiers	
106. EGL_EXT_pixel_format_float	
107. EGL_EXT_gl_colorspace_bt2020_linear EGL_EXT_gl_colorspace_bt2020_pq	
108. EGL_EXT_gl_colorspace_scrgb_linear	
109. EGL_EXT_surface_SMPTE2086_metadata	
110. EGL_NV_stream_fifo_next	
111. EGL_NV_stream_fifo_synchronous	
112. EGL_NV_stream_reset	
113. EGL_NV_stream_frame_limits	
114. EGL_NV_stream_remote EGL_NV_stream_cross_object EGL_NV_stream_cross_display EGL_NV_stream_cross_process EGL_NV_stream_cross_partition EGL_NV_stream_cross_system	

115. EGL_NV_stream_socket EGL_NV_stream_socket_unix EGL_NV_stream_socket_inet	
EGL_ANDROID_get_render_buffer	YES
EGL_ANDROID_swap_rectangle	YES
EGL_WL_bind_wayland_display	YES

3.3 OpenGL ES extension support

The following table includes the list of all current OpenGL ES Extensions and indicates their support level. (list from www.khronos.org/registry/gles/ as of 9/27/2012)

Table 13. OpenGL ES extension support

Extension Number, Name and hyperlink	ES1.1	ES2.0/3.0/3.1/3.2
1. GL_OES_blend_equation_separate	YES	na
2. GL_OES_blend_func_separate	YES	na
3. GL_OES_blend_subtract	YES	na
4. GL_OES_byte_coordinates	YES	na
5. GL_OES_compressed_ETC1_RGB8_texture	YES	YES
6. GL_OES_compressed_paletted_texture	YES	YES
7. GL_OES_draw_texture	YES	na
8. GL_OES_extended_matrix_palette	YES	
9. GL_OES_fixed_point	YES	
10. GL_OES_framebuffer_object	YES	na
11. GL_OES_matrix_get	YES	na
12. GL_OES_matrix_palette	YES	na
14. GL_OES_point_size_array	YES	
15. GL_OES_point_sprite	YES	
16. GL_OES_query_matrix	YES	na
17. GL_OES_read_format	YES	
18. GL_OES_single_precision	YES	
19. GL_OES_stencil_wrap	YES	
20. GL_OES_texture_cube_map	YES	na
21. GL_OES_texture_env_crossbar		
22. GL_OES_texture_mirrored_repeat	YES	na
23. GL_OES_EGL_image	YES	YES
24. GL_OES_depth24	YES	YES
25. GL_OES_depth32		YES
26. GL_OES_element_index_uint	YES	YES
27. GL_OES_fbo_render_mipmap	YES	YES
28. GL_OES_fragment_precision_high		YES
29. GL_OES_mapbuffer	YES	YES
30. GL_OES_rgb8_rgba8	YES	YES
31. GL_OES_stencil1		
32. GL_OES_stencil4		
33. GL_OES_stencil8	YES	na

Extension Number, Name and hyperlink	ES1.1	ES2.0/3.0/3.1/3.2
34. GL_OES_texture_3D		
35. GL_OES_texture_float_linear		no
GL_OES_texture_half_float_linear		CORE
36. GL_OES_texture_float		CORE
GL_OES_texture_half_float		CORE
37. GL_OES_texture_npot	YES	YES
38. GL_OES_vertex_half_float	YES	YES
39. GL_AMD_compressed_3DC_texture		
40. GL_AMD_compressed_ATC_texture		
41. GL_EXT_texture_filter_anisotropic	CORE	CORE
42. GL_EXT_texture_type_2_10_10_10_REV		CORE
43. GL_OES_depth_texture		YES
44. GL_OES_packed_depth_stencil	YES	YES
45. GL_OES_standard_derivatives		YES
46. GL_OES_vertex_type_10_10_10_2		CORE
47. GL_OES_get_program_binary		YES
48. GL_AMD_program_binary_Z400		
49. GL_EXT_texture_compression_dxt1		YES
50. GL_AMD_performance_monitor		
51. GL_EXT_texture_format_BGRA8888	YES	YES
52. GL_NV_fence		
53. GL_IMG_read_format		
54. GL_IMG_texture_compression_pvrtc		
55. GL_QCOM_driver_control		
56. GL_QCOM_performance_monitor_global_mode		
57. GL_IMG_user_clip_plane		
58. GL_IMG_texture_env_enhanced_fixed_function		
59. GL_APPLE_texture_2D_limited_npot		
60. GL_EXT_texture_lod_bias	YES	N/A
61. GL_QCOM_writeonly_rendering		
62. GL_QCOM_extended_get		
63. GL_QCOM_extended_get2		
64. GL_EXT_discard_framebuffer		YES
65. GL_EXT_blend_minmax	YES	YES
66. GL_EXT_read_format_bgra	YES	YES
67. GL_IMG_program_binary		
68. GL_IMG_shader_binary		
69. GL_EXT_multi_draw_arrays	YES	YES
GL_SUN_multi_draw_arrays	*no*	*no*
70. GL_QCOM_tiled_rendering		
71. GL_OES_vertex_array_object		YES
72. GL_NV_coverage_sample		
73. GL_NV_depth_nonlinear		
74. GL_IMG_multisampled_render_to_texture		
75. GL_OES_EGL_sync	YES	YES
76. GL_APPLE_rgb_422		

Extension Number, Name and hyperlink	ES1.1	ES2.0/3.0/3.1/3.2
77. GL_EXT_shader_texture_lod		
78. GL_APPLE_framebuffer_multisample		
79. GL_APPLE_texture_format_BGRA8888		
80. GL_APPLE_texture_max_level		
81. GL_ARM_mali_shader_binary		
82. GL_ARM_rgba8		
83. GL_ANGLE_framebuffer_blit		
84. GL_ANGLE_framebuffer_multisample		
85. GL_VIV_shader_binary		
86. GL_EXT_frag_depth		YES
87. GL_OES_EGL_image_external	YES	YES
88. GL_DMP_shader_binary		
89. GL_QCOM_alpha_test		
90. GL_EXT_unpack_subimage		N/A
91. GL_NV_draw_buffers		
92. GL_NV_fbo_color_attachments		
93. GL_NV_read_buffer		
94. GL_NV_read_depth_stencil		
95. GL_NV_texture_compression_s3tc_update		
96. GL_NV_texture_npot_2D_mipmap		
97. GL_EXT_color_buffer_half_float		**CORE**
98. GL_EXT_debug_label		
99. GL_EXT_debug_marker		
100. GL_EXT_occlusion_query_boolean		
101. GL_EXT_separate_shader_objects		
102. GL_EXT_shadow_samplers		
103. GL_EXT_texture_rg		**YES**
104. GL_NV_EGL_stream_consumer_external		
105. GL_EXT_sRGB		
106. GL_EXT_multisampled_render_to_texture		YES
107. GL_EXT_robustness		YES
108. GL_EXT_texture_storage		
109. GL_ANGLE_instanced_arrays		
110. GL_ANGLE_pack_reverse_row_order		
111. GL_ANGLE_texture_compression_dxt3		
GL_ANGLE_texture_compression_dxt5		
112. GL_ANGLE_texture_usage		
113. GL_ANGLE_translated_shader_source		
114. GL_FJ_shader_binary_GCCSO		
115. GL_OES_required_internalformat		YES
116. GL_OES_surfaceless_context		YES
117. GL_KHR_texture_compression_astc_hdr		
GL_KHR_texture_compression_astc_ldr		YES
118. GL_KHR_debug		**YES**
119. GL_QCOM_binning_control		
120. GL_ARM_mali_program_binary		

Extension Number, Name and hyperlink	ES1.1	ES2.0/3.0/3.1/3.2
121. GL_EXT_map_buffer_range		
122. GL_EXT_shader_framebuffer_fetch		**CORE**
123. GL_APPLE_copy_texture_levels		
124. GL_APPLE_sync		
125. GL_EXT_multiview_draw_buffers		
126. GL_NV_draw_texture		
127. GL_NV_packed_float		
128. GL_NV_texture_compression_s3tc		
129. GL_NV_3dvision_settings		
130. GL_NV_texture_compression_latc		
131. GL_NV_platform_binary		
132. GL_NV_pack_subimage		
133. GL_NV_texture_array		
134. GL_NV_pixel_buffer_object		
135. GL_NV_bgr		
136. GL_OES_depth_texture_cube_map		YES
137. GL_EXT_color_buffer_float		**CORE**
138. GL_ANGLE_depth_texture		
139. GL_ANGLE_program_binary		
140. GL_IMG_texture_compression_pvrtc2		
141. GL_NV_draw_instanced		
142. GL_NV_framebuffer_blit		
143. GL_NV_framebuffer_multisample		
144. GL_NV_generate_mipmap_sRGB		
145. GL_NV_instanced_arrays		
146. GL_NV_shadow_samplers_array		
147. GL_NV_shadow_samplers_cube		
148. GL_NV_sRGB_formats		
149. GL_NV_texture_border_clamp		
150. GL_EXT_disjoint_timer_query		
151. GL_EXT_draw_buffers		
152. GL_EXT_texture_sRGB_decode		**YES**
153. GL_EXT_sRGB_write_control		
154. GL_EXT_texture_compression_s3tc		YES
155. GL_EXT_pvrtc_sRGB		
156. GL_EXT_instanced_arrays		
157. GL_EXT_draw_instanced		
158. GL_NV_copy_buffer		
159. GL_NV_explicit_attrib_location		
160. GL_NV_non_square_matrices		
161. GL_EXT_shader_integer_mix		
162. GL_OES_texture_compression_astc		
163. GL_NV_blend_equation_advanced		
GL_NV_blend_equation_advanced_coherent		
164. GL_INTEL_performance_query		
165. GL_ARM_shader_framebuffer_fetch		

Extension Number, Name and hyperlink	ES1.1	ES2.0/3.0/3.1/3.2
166. GL_ARM_shader_framebuffer_fetch_depth_stencil		
167. GL_EXT_shader_pixel_local_storage		
168. GL_KHR_blend_equation_advanced		**CORE**
GL_KHR_blend_equation_advanced_coherent		
169. GL_OES_sample_shading		**CORE**
170. GL_OES_sample_variables		**CORE**
171. GL_OES_shader_image_atomic		**CORE**
172. GL_OES_shader_multisample_interpolation		**CORE**
173. GL_OES_texture_stencil8		**CORE**
174. GL_OES_texture_storage_multisample_2d_array		**CORE**
175. GL_EXT_copy_image		**CORE**
176. GL_EXT_draw_buffers_indexed		**CORE**
177. GL_EXT_geometry_shader		**CORE**
GL_EXT_geometry_point_size		**CORE**
178. GL_EXT_gpu_shader5		**CORE**
179. GL_EXT_shader_implicit_conversions		**CORE**
180. GL_EXT_shader_io_blocks		**CORE**
181. GL_EXT_tessellation_shader		**CORE**
GL_EXT_tessellation_point_size		**CORE**
182. GL_EXT_texture_border_clamp		**CORE**
183. GL_EXT_texture_buffer		**CORE**
184. GL_EXT_texture_cube_map_array		**CORE**
185. GL_EXT_texture_view		
186. GL_EXT_primitive_bounding_box		**CORE**
187. GL_ANDROID_extension_pack_es31a		**CORE**
188. GL_EXT_compressed_ETC1_RGB8_sub_texture		
189. GL_KHR_robust_buffer_access_behavior		YES
190. GL_KHR_robustness		YES
191. GL_KHR_context_flush_control		
192. GL_DMP_program_binary		
193. GL_APPLE_clip_distance		
194. GL_APPLE_color_buffer_packed_float		
195. GL_APPLE_texture_packed_float		
196. GL_NV_internalformat_sample_query		
197. GL_NV_bindless_texture		
198. GL_NV_conditional_render		
199. GL_NV_path_rendering		
200. GL_NV_image_formats		
201. GL_NV_shader_noperspective_interpolation		
202. GL_NV_viewport_array		
203. GL_EXT_base_instance		
204. GL_EXT_draw_elements_base_vertex		**CORE**
205. GL_EXT_multi_draw_indirect		**CORE**
206. GL_EXT_render_snorm		
207. GL_EXT_texture_norm16		
208. GL_OES_copy_image		**CORE**

Extension Number, Name and hyperlink	ES1.1	ES2.0/3.0/3.1/3.2
209. GL_OES_draw_buffers_indexed		CORE
210. GL_OES_geometry_shader		CORE
211. GL_OES_gpu_shader5		CORE
212. GL_OES_primitive_bounding_box		CORE
213. GL_OES_shader_io_blocks		CORE
214. GL_OES_tessellation_shader		CORE
215. GL_OES_texture_border_clamp		CORE
216. GL_OES_texture_buffer		CORE
217. GL_OES_texture_cube_map_array		CORE
218. GL_OES_texture_view		
219. GL_OES_draw_elements_base_vertex		CORE
220. GL_OES_copy_image		CORE
221. GL_EXT_texture_sRGB_R8		
222. GL_EXT_yuv_target		
223. GL_EXT_texture_sRGB_RG8		
224. GL_EXT_float_blend		
225. GL_EXT_post_depth_coverage		
226. GL_EXT_raster_multisample		
227. GL_EXT_texture_filter_minmax		
228. GL_NV_conservative_raster		
229. GL_NV_fragment_coverage_to_color		
230. GL_NV_fragment_shader_interlock		
231. GL_NV_framebuffer_mixed_samples		
232. GL_NV_fill_rectangle		
233. GL_NV_geometry_shader_passthrough		
234. GL_NV_path_rendering_shared_edge		
235. GL_NV_sample_locations		
236. GL_NV_sample_mask_override_coverage		
237. GL_NV_viewport_array2		
238. GL_NV_polygon_mode		
239. GL_EXT_buffer_storage		
240. GL_EXT_sparse_texture		
241. GL_OVR_multiview		
242. GL_OVR_multiview2		
243. GL_KHR_no_error		
246. GL_INTEL_framebuffer_CMAA		
247. GL_EXT_blend_func_extended		
248. GL_EXT_multisample_compatibility		
249. GL_KHR_texture_compression_astc_sliced_3d		
250. GL_OVR_multiview_multisampled_render_to_texture		
251. GL_IMG_texture_filter_cubic		
251. GL_IMG_texture_filter_cubic		
252. GL_EXT_polygon_offset_clamp		
253. GL_EXT_shader_pixel_local_storage2		
254. GL_EXT_shader_group_vote		
255. GL_IMG_framebuffer_downsample		

Extension Number, Name and hyperlink	ES1.1	ES2.0/3.0/3.1/3.2
256. GL_EXT_protected_textures		CORE
257. GL_EXT_clip_cull_distance		
258. GL_NV_viewport_swizzle		
259. GL_EXT_sparse_texture2		
260. GL_NV_gpu_shader5		
261. GL_NV_shader_atomic_fp16_vector		
262. GL_NV_conservative_raster_pre_snap_triangles		
263. GL_EXT_window_rectangles		
264. GL_EXT_shader_non_constant_global_initializers		
265. GL_INTEL_conservative_rasterization		
266. GL_NVX_blend_equation_advanced_multi_draw_buffers		
267. GL_OES_viewport_array		
268. GL_EXT_conservative_depth		
269. GL_EXT_clear_texture		
270. GL_IMG_bindless_texture		
271. GL_NV_texture_barrier		
GL_VIV_direct_texture	YES	YES

3.4 Extension GL_VIV_direct_texture

Name

VIV_direct_texture

Name strings

GL_VIV_direct_texture

IPStatus

Contact NXP Semiconductor regarding any intellectual property questions associated with this extension.

Status

Implemented: July, 2011

Version

Last modified: 29 July, 2011
Revision: 2

Number

Unassigned

Dependencies

OpenGL ES 1.1 is required. OpenGL ES 2.0 support is available.

Overview

Create a texture with direct access support. This is useful when an application desires to use the same texture over and over while frequently updating its content. It could also be used for mapping live video to a texture. A video decoder could write its result directly to the texture and then the texture could be directly rendered onto a 3D shape. `glTexDirectVIVMap` is similar

to `glTexDirectVIV`. The only difference is that it has two inputs, “Logical” and “Physical,” which support mapping a user space memory or a physical address into the texture surface.

New Procedures and Functions

glTexDirectVIV

Syntax:

```
GL_API void GL_APIENTRY
glTexDirectVIV (
        GLenum              Target,
        GLsizei             Width,
        GLsizei             Height,
        GLenum              Format,
        GLvoid **           Pixels
);
```

Parameters

Target	Target texture. Must be GL_TEXTURE_2D.
Width **Height**	Size of LOD 0. Width must be 16 pixel aligned. The width and height of LOD 0 of the texture is specified by the Width and Height parameters. The driver may auto-generate the rest of LODs if the hardware supports high quality scaling (for non-power of 2 textures) and LOD generation. If the hardware does not support high quality scaling and LOD generation, the texture remains a single-LOD texture.
Format	Choose the format of the pixel data from the following formats: GL_VIV_YV12, GL_VIV_NV12, GL_VIV_NV21, GL_VIV_YUY2, GL_VIV_UYVY, GL_RGBA, and GL_BGRA_EXT. • If the format is GL_VIV_YV12, glTexDirectVIV creates a planar YV12 4:2:0 texture and the format of the Pixels array is as follows: Yplane, Vplane, Uplane. • If the format is GL_VIV_NV12, glTexDirectVIV creates a planar NV12 4:2:0 texture and the format of the Pixels array is as follows: Yplane, UVplane. • If the format is GL_VIV_NV21, glTexDirectVIV creates a planar NV21 4:2:0 texture and the format of the Pixels array is as follows: Yplane, VUplane. • If the format is GL_VIV_YUY2 or GL_VIV_UYVY, glTexDirectVIV creates a packed 4:2:2 texture and the Pixels array contains only one pointer to the packed YUV texture. • If Format is GL_RGBA, glTexDirectVIV creates a pixel array with four GL_UNSIGNED_BYTE components: the first byte for red pixels, the second byte for green pixels, the third byte for blue, and the fourth byte for alpha. • If Format is GL_BGRA_EXT, glTexDirectVIV creates a pixel array with four GL_UNSIGNED_BYTE components: the first byte for blue pixels, the second byte for green pixels, the third byte for red, and the fourth byte for alpha.

Pixels	Stores the memory pointer created by the driver.

Output

If the function succeeds, it returns a pointer, or, for some YUV formats, it returns a set of pointers that directly point to the texture. The pointer(s) are returned in the user-allocated array pointed to by the Pixels parameter.

GlTexDirectVIVMap

Syntax:

```
GL_API void GL_APIENTRY
glTexDirectVIVMap (
       Glenum              Target,
       Glsizei             Width,
       Glsizei             Height,
       Glenum              Format,
       Glvoid **           Logical,
       const Gluint *      Physical
);
```

Parameters

Target	Target texture. Must be GL_TEXTURE_2D.
Width **Height**	Size of LOD 0. Width must be 16 pixel aligned. See glTexDirectVIV.
Format	Same as `glTexDirectVIV` Format.
Logical	Pointer to the logical address of the application-defined texture buffer. Logical address must be 64 bit (8 byte) aligned.
Physical	Pointer to the physical address of the application-defined buffer to the texture, or ~0 if no physical address has been provided.

GlTexDirectInvalidateVIV

Syntax:

```
GL_API void GL_APIENTRY
glTexDirectInvalidateVIV (
       Glenum              Target
);
```

Parameters

Target	Target texture. Must be GL_TEXTURE_2D.

New Tokens

GL_VIV_YV12	0x8FC0
GL_VIV_NV12	0x8FC1
GL_VIV_YUY2	0x8FC2

GL_VIV_UYVY	0x8FC3
GL_VIV_NV21	0x8FC4

Error codes

GL_INVALID_ENUM	Target is not GL_TEXTURE_2D, or format is not a valid format.
GL_INVALID_VALUE	Width or Height parameter is less than 1.
GL_OUT_OF_MEMORY	A memory allocation error occurred.
GL_INVALID_OPERATION	Specified format is not supported by the hardware, or no texture is bound to the active texture unit, or some other error occurs during the call.

Example 1.

First, call `glTexDirectVIV` to get a pointer.
Second, copy the texture data to this memory address.
Then, call `glTexDirectInvalidateVIV` to apply the texture before drawing something with that texture.

```
… …
glTexDirectVIV(GL_TEXUTURE_2D, 512, 512, GL_VIV_YV12, &texels);
… …
GlTexDirectInvalidateVIV(GL_TEXTURE_2D);
…
glDrawArrays(…);
…
```

Example 2.

First, call `glTexDirectVIVMap` to map Logical and Physical address to the texture.
Second, modify Logical and Physical data.
Then, call `glTexDirectInvalidateVIV` to apply the texture before drawing something with that texture.

```
… …
char *Logical = (char*) malloc (sizeof(char)*size);
Gluint physical = ~0U;
glTexDirectVIVMap(GL_TEXUTURE_2D, 512, 512, GL_VIV_YV12,
    (void**)&Logical, &32hysical);
… …
GlTexDirectInvalidateVIV(GL_TEXTURE_2D);
…
glDrawArrays(…);
```

Issues

None

3.5 Extension GL_VIV_texture_border_clamp

Name

VIV_texture_border_clamp

Name Strings

GL_VIV_texture_border_clamp

Status

Implemented September 2012.

Version

Last modified: 27 September 2012
Vivante revision: 1

Number

Unassigned

Dependencies

This extension is implemented for use with OpenGL ES 1.1 and OpenGL ES 2.0.

This extension is based on OpenGL ARB Extension #13: GL_ARB_texture_border_clamp: www.opengl.org/registry/specs/ARB/texture_border_clamp.txt. See also vendor extension GL_SGIS_texture_border_clamp: www.opengl.org/registry/specs/SGIS/texture_border_clamp.txt.

Overview

This extension was adapted from the OpenGL extension for use with OpenGL ES implementations. The OpenGL ARB Extension 13 description applies here as well:

> "The base OpenGL provides clamping such that the texture coordinates are limited to exactly the range [0,1]. When a texture coordinate is clamped using this algorithm, the texture sampling filter straddles the edge of the texture image, taking 1/2 its sample values from within the texture image, and the other 1/2 from the texture border. It is sometimes desirable for a texture to be clamped to the border color, rather than to an average of the border and edge colors.
>
> This extension defines an additional texture clamping algorithm. CLAMP_TO_BORDER_[VIV] clamps texture coordinates at all mipmap levels such that NEAREST and LINEAR filters return only the color of the border texels."

The color returned is derived only from border texels and cannot be configured.

Issues

None

New Tokens

Accepted by the `<param>` parameter of `TexParameteri` and `TexParameterf`, and by the `<params>` parameter of `TexParameteriv` and `TexParameterfv`, when their `<pname>` parameter is `TEXTURE_WRAP_S`, `TEXTURE_WRAP_T`, or `TEXTURE_WRAP_R`:

CLAMP_TO_BORDER_VIV	0x812D

Errors

None.

New State

Only the type information changes for these parameters.

See OES 2.0 Specification Section 3.7.4, page 75-76, Table 3.10, “Texture parameters and their values.”

Chapter 4 i.MX Framebuffer API

4.1 Overview

The graphics software includes i.MX Framebuffer (FB) API which enables users to easily create and port their graphics applications by using a framebuffer device without the need to expend additional effort handling platform-related tasks. i.MX Framebuffer API focuses on providing mechanisms for controlling display, window, and pixmap render surfaces.

The EGL Native Platform Graphics Interface provides mechanisms for creating rendering surfaces onto which client APIs can draw, creating graphics contexts for client APIs, and synchronizing drawing by client APIs as well as native platform rendering APIs. This enables seamless rendering using Khronos APIs such as OpenGL ES and OpenVG for high-performance, accelerated, mixed-mode 2D, and 3D rendering. For further information on EGL, see www.khronos.org/registry/egl. The API described in this document is compatible with EGL version 1.4 of the specification.

The following platforms are supported:

- Linux® OS/X11
- Android™ platform
- Windows® Embedded Compact OS
- QNX®

4.2 API data types and environment variables

4.2.1 Data types

The GPU software provides platform independent member definitions for the following EGL types:

```
typedef struct _FBDisplay  *  EGLNativeDisplayType;
typedef struct _FBWindow *  EGLNativeWindowType;
typedef struct _FBPixmap  *  EGLNativePixmapType;
```

Types [2.1.1]

unsigned int	EGLBoolean
unsigned int	EGLenum
void	*EGLConfig
void	*EGLContext
void	*EGLDisplay
void	*EGLSurface
void	*EGLClientBuffer

The following types differ based on platform.

Windows platform:

HDC	EGLNativeDisplayType
HBITMAP	EGLNativePixmapType
HWND	EGLNativeWindowType

Linux/X11 platform:

Display	*EGLNativeDisplayType
Pixmap	EGLNativePixmapType
Window	EGLNativeWindowType

Android platform:

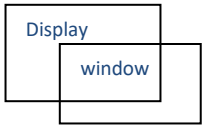
ANativeWindow*	EGLNativeWindowType

Figure 2 Types as listed on EGL 1.4 API Quick Reference Card

(from www.khronos.org/files/egl-1-4-quick-reference-card.pdf)

4.2.2 Environment variables

Table 14. i.MX FB API environment variables

Environment Variables	Description
FB_MULTI_BUFFER	To use multiple-buffer rendering, set the environment variable FB_MULTI_BUFFER to an unsigned integer value, which indicates the number of buffers required. The maximum is 8. **Recommended values: 4.** The FB_MULTI_BUFFER variable can be set to any positive integer value. • If set to 1, the multiple-buffer function is not enabled, and the VSYNC is also disabled, so there may be tearing on screen, but it is good for benchmark test. • If set to 2 or 3, VSYNC is enabled and there are double or trible frame buffer. Because of the hardware limitation of current IPU, there may be tearing on screen. • If set to 4 or more, VSYNC is enabled and no screen tearing appears. • If set to a value more than 8, the driver uses 8 as the buffer count.
FB_FRAMEBUFFER_0, FB_FRAMEBUFFER_1, FB_FRAMEBUFFER_2, FB_FRAMEBUFFER_n	To open a specified framebuffer device, set the environment variable **FB_FRAMEBUFFER_n** to a proper value (for example, FB_FRAMEBUFFER_0 = /dev/fb0). **Allowed values for n: any positive integer.** Note: If there are no environment variables set, the driver tries to use the default framebuffer devices (fb0 for index 0, fb1 for index 1, fb2 for index 2, fb3 for index 3, and so on).
FB_IGNORE_DISPLAY_SIZE	When set to a positive integer and a window's initial size request is greater than the display size, the window size is not reduced to fit within the display. Global. **Allowed values: any positive integer.** Note: The drivers read the value from this environment variable as a Boolean to check if the user wants to ignore the display size when creating a window. • If the variable is set to value, 0, or this environment variable is not set, when creating window, the driver uses display size to cut down the size of the window to ensure that the entire window area is inside the display screen. • If the user sets this variable to 1, or any positive integer value, then the window area can be partly or entirely outside of the display screen area (see the image below in which the ignore display size is equal to 1). 
GPU_VIV_DISABLE_CLEAR_FB	It turns off zero fill memory, so the content of FBDEV buffer is not cleared.
FB_LEGACY	If the board support drm-fb, the gpu will render though drm by default. If the user wants to render to framebuffer directly instead of through drm,

	sets this variable to 1.

Below are some usage syntax examples for environment variables:

To create a window with its size different from the display size, use the environment variable **FB_IGNORE_DISPLAY_SIZE**. Example usage syntax:

```
export FB_IGNORE_DISPLAY_SIZE=1
```

To let the driver use multiple buffers to do swap work, use the environment variable **FB_MULTI_BUFFER.** Example usage syntax:

```
export FB_MULTI_BUFFER=2
```

To specify the display device, use the environment variable **FB_FRAMEBUFFER_n**, where n = any positive integer. Example usage syntax:

```
export FB_FRAMEBUFFER_0=/dev/fb0
export FB_FRAMEBUFFER_1=/dev/fb1
export FB_FRAMEBUFFER_2=/dev/fb2
export FB_FRAMEBUFFER_3=/dev/fb3
```

4.3 API description and syntax

fbGetDisplay

Description:
This function is used to get the default display of the framebuffer device.

To open the framebuffer device, set an environment variable **FB_FRAMEBUFFER_n** to the framebuffer location.

Syntax:

```
EGLNativeDisplayType
fbGetDisplay (
      void *        context
);
```

Parameters:

context	Pointer to the native display instance.

Return Values:
The function returns a pointer to the EGL native display instance if successful; otherwise, it returns a NULL pointer.

fbGetDisplayByIndex

Description:
This function is used to get a specified display within a multiple framebuffer environment by providing an index number.

To use multiple buffers when rendering, set the environment variable **FB_MULTI_BUFFER** to an unsigned integer value, which indicates the number of buffers. Maximum is 3.

To open a specific Framebuffer device, set environment variables to their proper values (e.g., set FB_FRAMEBUFFER_0 = /dev/fb0). If there are no environment variables set, the driver tries to use the default fb devices (fb0 for index 0, fb1 for index 1, fb2 for index 2, fb3 for index 3, and so on).

Syntax:

```
EGLNativeDisplayType
fbGetDisplayByIndex (
        int             DisplayIndex
);
```

Parameters:

DisplayIndex An integer value where the integer is associated with one of the following environment variables for framebuffer devices:
FB_FRAMEBUFFER_0
FB_FRAMEBUFFER_1
FB_FRAMEBUFFER_2
FB_FRAMEBUFFER_n

Return Value:
The function returns a pointer to the EGL native display instance if successful; otherwise, it returns a NULL pointer.

fbGetDisplayGeometry

Description:
This function is used to get display width and height information.

Syntax:

```
void
fbGetDisplayGeometry (
        EGLNativeDisplayType    Display,
        int *                   Width,
        int *                   Height
);
```

Parameters:

Display [in] Pointer to EGL native display instance created by **fbGetDisplay.**
Width [out] Pointer that receives the width of the display.
Height [out] Pointer that receives the height of the display.

fbGetDisplayInfo

Description:
This function is used to get display information.

Syntax:

```
void
fbGetDisplayInfo (
        EGLNativeDisplayType    Display,
```

```
        int *                   Width,
        int *                   Height,
        unsigned long *         Physical,
        int *                   Stride,
        int *                   BitsPerPixel
    );
```

Parameters:

Display	[in] A pointer to the EGL native display instance created by **fbGetDisplay.**
Width	[out] A pointer to the location that contains the width of the display.
Height	[out] A pointer to the location that contains the height of the display.
Physical	[out] A pointer to the location that contains the physical start address of the display.
Stride	[out] A pointer to the location that contains the stride of the display.
BitsPerPixel	[out] A pointer to the location that contains the pixel depth of the display.

fbDestroyDisplay

Description:

This function is used to destroy a display.

Syntax:

```
void
fbDestroyDisplay (
        EGLNativeDisplayType      Display
);
```

Parameters:

Display	[in] Pointer to EGL native display instance created by **fbGetDisplay**.

fbCreateWindow

Description:

This function is used to create a window for the framebuffer platform with the specified position and size. If width/height is 0, it uses the display width/height as its value.

Note: When either window X + width or the Y + height is larger than the display's width or height respectively, the API reduces the window size to force the whole window inside the display screen limits. To avoid reducing the window size in this scenario, users can set a value of "1" to the environment variable **FB_IGNORE_DISPLAY_SIZE**.

Syntax:

```
EGLNativeWindowType
fbCreateWindow (
        EGLNativeDisplayType    Display,
        int                     X,
        int                     Y,
        int                     Width,
        int                     Height
```

```
);
```

Parameters:

Display	[in] Pointer to EGL native display instance created by **fbGetDisplay**.
X	[in] Specifies the initial horizontal position of the window.
Y	[in] Specifies the initial vertical position of the window.
Width	[in] Specifies the width of the window.
Height	[in] Specifies the height of the window in device units.

Return Value:
The function returns a pointer to the EGL native window instance if successful; otherwise, it returns a NULL pointer.

fbGetWindowGeometry

Description:
This function is used to get window position and size information.

Syntax:

```
void
fbGetWindowGeometry (
      EGLNativeWindowType    Window,
      int *                  X,
      int *                  Y,
      int *                  Width,
      int *                  Height
);
```

Parameters:

Window	[in] Pointer to EGL native window instance created by **fbCreateWindow**.
X	[out] Pointer that receives the horizontal position value of the window.
Y	[out] Pointer that receives the vertical position value of the window.
Width	[out] Pointer that receives the width value of the window.
Height	[out] Pointer that receives the height value of the window.

fbGetWindowInfo

Description:
This function is used to get window position and size and address information.

Syntax:

```
void
fbGetWindowInfo (
      EGLNativeWindowType   Window,
      int *                 X,
      int *                 Y,
      int *                 Width,
      int *                 Height
      int *                 BitsPerPixel,
      unsigned int *        Offset
```

```
);
```

Parameters:

Window	[in] A pointer to the EGL native window instance created by **fbCreateWindow**.
X	[out] A pointer to the location that contains the horizontal position value of the window.
Y	[out] A pointer to the location that contains the vertical position value of the window.
Width	[out] A pointer to the location that contains the width of the window.
Height	[out] A pointer to the location that contains the height of the window.
BitsPerPixel	[out] A pointer to the location that contains the pixel depth of the window.
Offset	[out] A pointer to the location that contains the offset of the window.

fbDestroyWindow

Description:

This function is used to destroy a window.

Syntax:

```
void
fbDestroyWindow (
      EGLNativeWindowType    Window
);
```

Parameters:

Window	[in] Pointer to EGL native window instance created by **fbCreateWindow**.

fbCreatePixmap

Description:

This function is used to create a pixmap of a specific size on the specified framebuffer device. If either the width or height is 0, the function fails to create a pixmap and return NULL.

Syntax:

```
EGLNativePixmapType
fbCreatePixmap (
      EGLNativeDisplayType    Display,
      int                     Width,
      int                     Height
);
```

Parameters:

Display	[in] Pointer to the EGL native display instance created by **fbGetDisplay**.
Width	[in] Specifies the width of the pixmap.
Height	[in] Specifies the height of the pixmap.

Return Value:

The function returns a pointer to the EGL native pixmap instance if successful; otherwise, it returns a NULL pointer.

fbCreatePixmapWithBpp

Description:
This function is used to create a pixmap of a specific size and bit depth on the specified framebuffer device. If either the width or height is 0, the function fails to create a pixmap and return NULL.

Syntax:

```
EGLNativePixmapType
fbCreatePixmapWithBpp (
      EGLNativeDisplayType    Display,
      int                     Width,
      int                     Height
      int                     BitsPerPixel
);
```

Parameters:

Display	[in]A pointer to the EGL native display instance created by **fbGetDisplay**.
Width	[in] Specifies the width of the pixmap.
Height	[in] Specifies the height of the pixmap.
BitsPerPixel	[in] Specifies the bit depth of the pixmap.

Return Value:
The function returns a pointer to the EGL native pixmap instance if successful; otherwise, it returns a NULL pointer.

fbGetPixmapGeometry

Description:
This function is used to get pixmap size information.

Syntax:

```
void
fbGetPixmapGeometry (
      EGLNativePixmapType    Pixmap,
      int *                  Width,
      int *                  Height
);
```

Parameters:

Pixmap	[in] Pointer to the EGL native pixmap instance created by **fbCreatePixmap**.
Width	[out] Pointer that receives a width value for pixmap.
Height	[out] Pointer that receives a height value for pixmap.

fbGetPixmapInfo

Description:
This function is used to get pixmap size and depth information.

Syntax:

```
void
fbGetPixmapInfo (
      EGLNativePixmapType    Pixmap,
```

```
        int *                   Width,
        int *                   Height
        int *                   BitsPerPixel
        int *                   Stride,
        void **                 Bits
    );
```

Parameters:

Pixmap	[in] A pointer to the EGL native pixmap instance created by **fbCreatePixmap**.
Width	[out] A pointer to the location that contains a width value for pixmap.
Height	[out] A pointer to the location that contains a height value for pixmap.
BitsPerPixel	[out] A pointer to the location that contains the pixel depth of the pixmap.
Stride	[out] A pointer to the location that contains the stride of the pixmap.
Bits	[out] A pointer to the location that contains the bit address of the pixmap.

fbDestroyPixmap

Description:

This function is used to destroy a pixmap.

Syntax:

```
void
fbDestroyPixmap (
        EGLNativePixmapType     Pixmap
);
```

Parameters:

Pixmap	[in] Pointer to the EGL native pixmap instance created by **fbCreatePixmap**.

Chapter 5 OpenCL

5.1 Overview

5.1.1 General description

OpenCL (Open Computing Language) is an open industry standard application programming interface (API) used to program multiple devices including GPUs, CPUs, as well as other devices organized as part of a single computational platform. The OpenCL standard targets a wide range of devices from mobile phones, tablets, PCs, and consumer electronic (CE) devices, all the way to embedded applications such as automotive and image processing functions. The API takes advantage of all resources in a platform to fully utilize all compute capability and to efficiently process the growing complexity of incoming data streams from multiple I/O (input/output) sources. I/O streams can be camera inputs, images, scientific or mathematical data, and any other form of complex data that can make use of data or task parallelism.

OpenCL uses parallel execution SIMD (single instruction, multiple data) engines found in GPUs to enhance data computational density by performing massively parallel data processing on multiple data items, across multiple compute engines. Each compute unit has its own arithmetic logic units (ALUs), including pipelined floating point (FP), integer (INT) units and a special function unit (SFU) that can perform computations as well as transcendental operations. The parallel computations and associated series of operations are called a kernel, and the GPU cores can execute a kernel on thousands of work-items in parallel at any given time.

At a high level, OpenCL provides both a programming language and a framework to enable parallel programming. OpenCL includes APIs, libraries and a runtime system to assist and support software development. With OpenCL, it is possible to write general purpose programs that can execute directly on GPUs, without needing to know graphics architecture details or using 3D graphics APIs like OpenGL or DirectX. OpenCL also provides a low-level Hardware Abstraction Layer (HAL) as well as a framework that exposes many details of the underlying hardware layer and thus allows the programmer to take full advantage of the hardware.

For more details on all the capabilities of OpenCL, see the following specifications from the Khronos Group:

- OpenCL 1.2 Specification
  www.khronos.org/registry/cl/specs/opencl-1.2.pdf

- OpenCL 1.2 C++ Bindings Specification
  www.khronos.org/registry/cl/specs/opencl-cplusplus-1.2.pdf

5.1.2 OpenCL framework

The OpenCL framework has two principal parts, similar to OpenGL, the host C API and the device C-based language runtime. The host in OpenCL terminology corresponds to the client in OpenGL and the device corresponds to the server. Device programs are called kernels. Execution of an OpenCL program is preceded by a series of API calls that configure the system and GPGPU for execution.

OpenCL abstracts today's heterogeneous architectures using a hierarchical platform model. A host coordinates the execution and data transfers on, to and from one or several compute devices. Compute devices are comprised of compute units and each such unit contains an array of processing elements.

5.1.2.1 OpenCL execution model: kernels and work elements

The OpenCL execution model is defined by how the kernels are executed. When a kernel is submitted for execution by the host, an index space is defined. An instance of the kernel executes for each point in this index space. This kernel instance is called a **work-item**. Work-items are identified by their position in the index space

that provides the global ID for the work-item. Each work-item executes the same code but the specific pathway through the code and the data operated upon varies by work-item.
Work-items are organized into **work-groups**. Work-groups provide a broader decomposition of the index space. Work-groups are each assigned a unique work-group ID with the same dimensionality as the index space used for the work-items. Work-items are assigned a unique local ID within a work-group so that a single work-item can be uniquely identified by its global ID or by a combination of its local ID and work-group ID. The work-items in a given work-group execute concurrently on the same compute device.
The index space supported in OpenCL is called an **NDRange**. An NDRange is an N-dimensional index space, where **N** is one (1), two (2) or three (3). An NDRange is defined by an integer array of length N specifying the extent of the index space in each dimension starting at an offset index **F** (zero by default). Each work-item's global ID and local ID are N-dimensional tuples. The global ID components are values in the range from F, to F plus the number of elements in that dimension minus one.
Work-groups are assigned IDs using a similar approach to that used for work-item global IDs. An array of length N defines the number of work-groups in each dimension. Work-items are assigned to a work-group and given a local ID with components in the range from zero to the size of the work-group in that dimension minus one. Hence, the combination of a work-group ID and the local-ID within a work-group uniquely defines a work-item. Each work-item is identifiable in two ways; in terms of a global index, unique through the whole kernel index space, and in terms of a local index, unique within a work group.

5.1.2.2 OpenCL command queues

OpenCL provides both task and data parallelism. Data movements are coordinated via **command queues** which provide a general means of specifying inter-task relationships and task execution orders that obey the dependencies in the computation. OpenCL may execute several tasks in parallel, if they are not order dependent. Tasks are composed of data-parallel kernels which, similarly to shaders, apply a single function to a range of elements in parallel. Only restricted synchronization and communication is allowed during kernel execution. OpenCL kernels execute over a 1, 2 or 3 dimensional index space. All work-items execute the same program (kernel) but their execution may diverge, with branching dependent on the data or their index. For details regarding how many work groups are allowed within an index space see "Using `clEnqueueNDRangeKernel`".
A kernel or a memory operation is first **enqueued** onto a command queue. Kernels are executed asynchronously and the host application execution may proceed right after the enqueue operation. The application may opt to wait for an operation to complete and an operation (kernel or memory) may be marked with a list of events that must occur before it executes.
Events are kernel completion and memory operations. OpenCL traverses the dependence graph between the kernels and memory transfers in a queue and ensures the correct execution order. Multiple command queues may be constructed, further enhancing parallelism control across platforms and multiple compute devices.

- **Command-queue barriers** are used to control the commands within the command queue. The command-queue barrier indicates which commands must be finished before proceeding. This allows for out-of-order command processing. The command queue barrier ensures that all previously enqueued commands finish execution before any following commands begin execution.

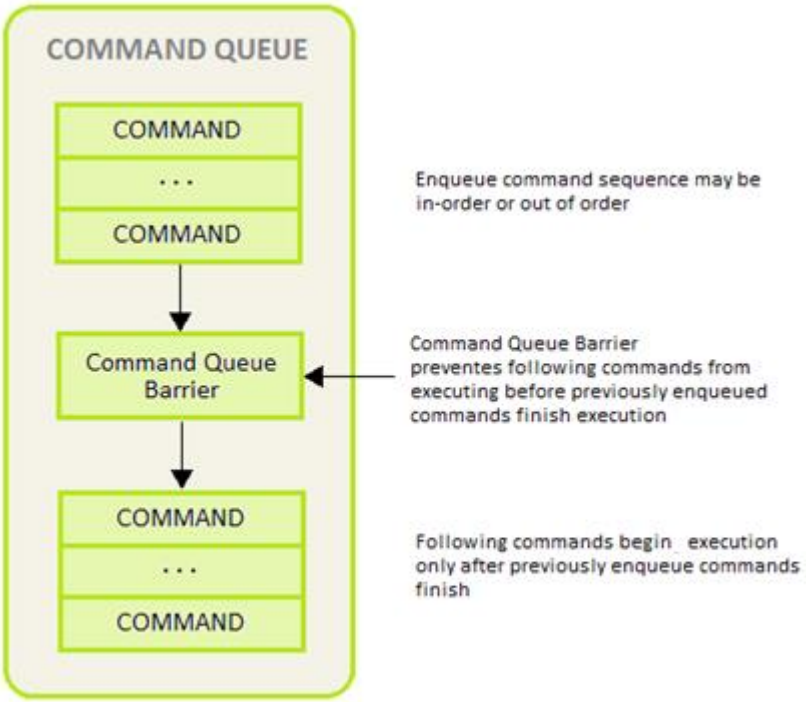


Figure 3 Command queue barrier

The work-group barrier built-in function provides control of the work-item flow within work-groups. All work-items must execute the barrier construct before any can continue execution beyond the barrier.

5.1.2.3 OpenCL memory model

The OpenCL memory model is divided into four different types of memory domains. These are:

- **Global Memory.** Each compute device has global memory space which can reside off-chip in system memory (DRAM) or inside the chip at the L1 or temporary register level. Global memory is accessible to all work-items executing in a context, as well as to the host (read, write, and map commands).
- **Constant Memory** is also global memory, but it is read-only. Constant memory can be placed in any level of memory that the application programmer decides, making it an implementation dependent decision. This is the region for host-allocated and host-initialized objects that are not changed during kernel execution.
- **Local Memory.** Each compute unit has local memory which resides very near the processing elements. Access to local memory is very fast and the size of local memory is much smaller than global memory, making it a scarce resource that needs to be controlled for optimal communication of work-items inside a work-group. Local memory is specific to a work-group, and is accessible only by work-items belonging to that work group.
- **Private Memory.** Each processing element has another level of memory called private memory, which is only accessible to a single work-item. Private memory is specific to a work-item and is not visible to other work-items.

During run-time, each processing element is assigned a set of on-chip registers that are used for data storage of intermediate data. Data that cannot be stored in registers spills over to global memory which can be very costly in

terms of performance and constant data movement to/from temporary registers. Software may emulate local and private memory using global memory. System Memory is often loaded to L1 cache, Temporary or Local Storage Registers and the GPGPU reads from those locations. At every level of the application program, the programmer must be aware of the size and hierarchy of storage elements.

Table 15. Vivante memory structures mapped to Khronos OpenCL memory types

Khronos OpenCL Memory Model Name	Vivante GPGPU OpenCL Memory Structures Utilized	Definition
Private Memory	Registers, System Memory	Accessible only to an individual work-item; not visible to any other work-items
Local Memory	Local Storage Registers, System Memory	Accessible to all work-items within a specific work-group; accessible only by work-items belonging to that work-group
Global Memory	System Memory	Accessible to all-work-items executing in a context, as well as to the host (read, write, and map commands).
Constant Memory	Constant Registers, System Memory	Read only global memory region for host-allocated and initialized objects that are not changed during kernel execution
Host (CPU) Memory	Host Memory	Region for a kernel application's program data and structures

The OpenCL concurrent-read /concurrent-write (CRCW) memory model has so-called relaxed consistency which means that different work-items may see a different view of global memory as the computation proceeds. Within individual work-items reads and writes to all memory spaces are ordered. Synchronization between work-items in a work-group is necessary to ensure consistency. No mechanism for synchronization between work-groups is provided. Such a model assures parallel scalability by requiring explicit synchronization and communication.

For the highest throughput and computational speed, kernels should use high-speed on-chip memories and registers as much as possible. Instruction control flow and memory operations, including data gathering / scattering and direct memory access (DMA) should be automatically reorganized / re-ordered depending on data dependencies detected by the optimized compiler. The Vivante OpenCL compiler automatically maps dependencies and re-orders instructions for the best performance.

5.1.2.4 Host to GPGPU compute device data transfers

The application running on the host uses the OpenCL API to create memory objects in global memory, and to enqueue memory commands that operate on these memory objects. The host and OpenCL device memory models are, for the most part, independent of each other. This is by necessity as the host is defined outside of OpenCL. They do, however, at times need to interact. This interaction occurs in one of two ways: by explicitly copying data from the host to the GPU compute device memory, or implicitly, by mapping and unmapping regions of a memory object.

- **Explicit** using `clEnqueueReadBuffer` and `clEnqueueWriteBuffer` (`clEnqueueReadImage`, `clEnqueueWriteImage`.)

  To copy data explicitly, the host enqueues commands to transfer data between the memory object and host memory. These memory transfer commands may be blocking or non-blocking. The OpenCL function call for a blocking memory transfer returns once the associated memory resources on the host can be

safely reused. For a non-blocking memory transfer, the OpenCL function call returns as soon as the command is enqueued regardless of whether host memory is safe to use.
• **Implicit** using `clEnqueueMapBuffer` and `clEnqueueUnMapMemObject`.

The mapping/unmapping method of interaction between the host and OpenCL memory objects allows the host to map a region from the memory object into its address space. The memory map command may be blocking or non-blocking. Once a region from the memory object has been mapped, the host can read or write to this region. The host unmaps the region when accesses (reads and/or writes) to this mapped region by the host are complete.

The OpenCL specification does not explicitly state where each memory space will be mapped to on individual implementations. This provides great freedom for vendors on the one hand and some uncertainty for programmers on the other. Fortunately, kernels may be compiled just-in-time and possible differences may be tackled during run-time.

When using these interfaces, it is important to consider the amount of copying involved to/from system memory and the various levels within the compute device(s). There is a two-copy process: between host and AXI (or SoC internal bus), and between AXI (or SoC internal bus) and the Vivante GPGPU compute device. Double copying lowers overall system memory bandwidth and lowers performance. Because of variations in system architecture (both internal and external/memory), there is sometimes a large performance delta between the system or calculated GFLOPS and the kernel or GPGPU GFLOPS. GPGPU GFLOPS are based on the theoretical computational capability of the ALUs within the GPGPU, assuming the system architecture can deliver full data to the GPGPU. OpenCL APIs for buffers and images aid in avoiding double copy by allowing the mapping of host memory to device memory. With proper memory transfer management and the use of host/CPU memory remapped to the GPGPU memory space, copying between host memory and GPGPU memory can be skipped so data transfer becomes a one-copy process. The trade-off is that the programmer needs to be mindful of page boundaries and memory alignment issues.

5.1.3 OpenCL profiles

In addition to Full Profile, the OpenCL specification also includes an Embedded Profile, which relaxes the OpenCL compliance requirements for mobile and embedded devices. The main commons and differences between OpenCL 1.1/1.2 EP (Embedded Profile) and FP (Full Profile) come down to:
Commons:

- Both EP and FP significantly offload the CPU of parallel, multi-threaded tasks.
- For both EP and FP double precision and half-precision floating point are optional.

Difference:

- Full Profile is for highly complex, accurate, and real time computations, while Embedded Profile is a small subset targeting smaller devices (handheld, mobile, embedded) that perform GPGPU/OpenCL processing with relaxed data type and precision requirements (image processing, augmented reality, gesture recognition, and more).
- 64-bit integers are required for FP and optional for EP.
- EP requires either RTZ or RTE. FP requires both.
- Computational precision (units in the last place; i.e., ULP) requirements in EP are relaxed.
- Atomic instruction support is not required in EP.
- 3D Image support is not required in EP.

- Minimum requirements for constant buffer size, object allocation size, constant argument counts and local memory sizes are scaled down in EP.
- And more (in general EP is a scaled down version of FP).
- Die size and power increase with FP because of the higher requirements, features and memory sizes.

5.1.4 Vivante OpenCL embedded compatible IP

As of the date of this document, select Vivante GPGPU cores are compatible with OpenCL Embedded Profile version 1.1. Hardware capability deltas include:

Table 16. Vivante OpenCL embedded profile hardware

Hardware and revision	GC2000
Feature	**5.1.0.rc8a**
Compute Devices (GPGPU cores)	1
Compute Units per device (Shader cores)	4
Processing Elements per compute unit	4
Profile	Embedded
Preferred work-group/thread group size	16
Max count global work-items each dim	64K
Max count of work-items each dim per work-group	1K
Local Storage Registers On-chip	64
Instruction Memory	512
Texture Samplers	8 PS + 4 VS
Texture Samplers available to OCL (HW, unlimited via SW)	4
L1 Cache Size	4 KB
L1 Cache Banks	1
L1 Cache Sets/Bank	4
L1 Cache Ways/Set	16
L1 Cache Line Size	64B
L1 Cache MC ports	1

5.1.5 Vivante OpenCL full profile hardware model

As of the date of this document, select Vivante GPGPU cores are compatible with OpenCL Full Profile version 1.2. Hardware capability deltas are subject to change and includes:

Table 17. Vivante OpenCL full profile hardware

Hardware and revision	GC2000+	GC7000XSVX	GC7000L
i.MX SOC	**i.MX 6QuadPlus, i.MX 6DualPlus**	**i.MX 8QuadMax**	**i.MX 8 MQuad**
Compute Devices (GPGPU cores)	1	1	1
Compute Units per device (Shader cores)	4	8	1

Processing Elements per compute unit	4	32	16
Profile	Full-Lite*	Full	Full
Preferred work-group/ thread group size	16	32	8
Max count global work-items each dim (if 3D only 1 dim can be up to 4G, the others 64K)	4 G/64 K	4 G/64 K	4 G/64 K
Max count of work-items each dim per work-group	1 K	1 K	1K
Local Storage Registers On-chip	0	2048 (32 K)	16K
Instruction Memory	I$:512/1 M	I$:512/1 M	I$:512/1 M
Texture Samplers	32	32	32
Texture Samplers available to OCL	32	32	32
L1 Cache Size	4 KB	64 KB	16K
L1 Cache Banks	2	4	2
L1 Cache Sets/Bank	2	8	8
L1 Cache Ways/Set	16	16	8
L1 Cache Line Size	64 B	64 B	64B
L1 Cache MC ports per GPGPU core	2	2	2

5.2 Vivante OpenCL implementation

5.2.1 OpenCL pipeline

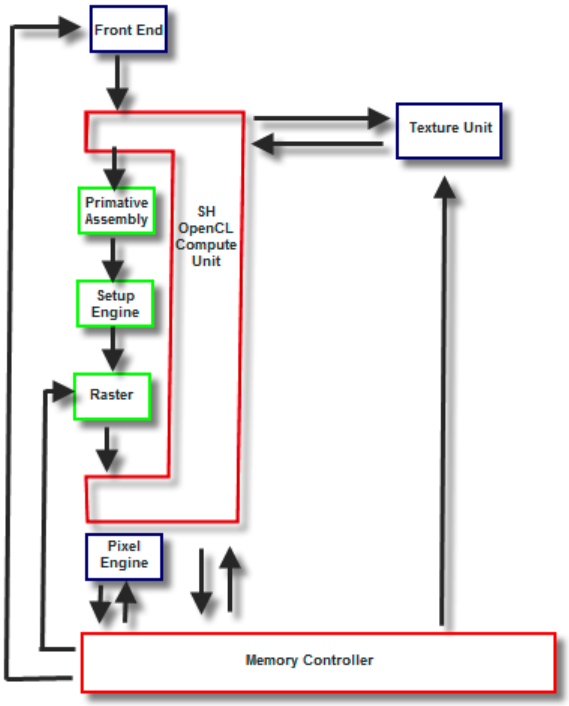


Figure 4 Vivante OpenCL data pipeline for an OpenCL compute device

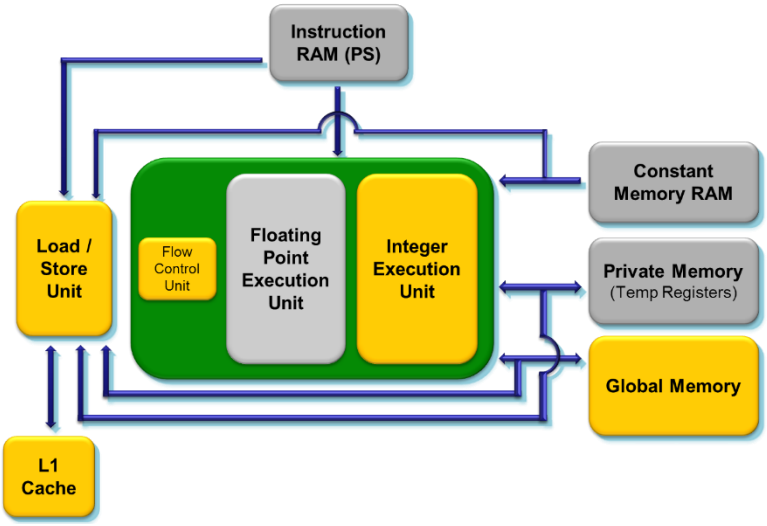


Figure 5 Vivante OpenCL compute device showing memory scheme

5.2.2 Front end

The front end passes the instructions and constant data as State Loads to the OpenCL Compute Unit (Shader) block. State Loads program instructions and constant data and work groups initiate execution on the instructions and the constants loaded.

5.2.3 The OpenCL compute unit

All OpenCL executions occur in this block and all work-groups in a compute unit should belong to the same kernel. Threads from a work-group are grouped into internal “Thread-groups”. All the threads in a thread-group execute in parallel. Barrier instruction is supported to enforce synchronization within a work-group.
The compute unit contains Local Memory and the L1 Cache and is where the Load/Store instruction to access global memory originates. The compute unit can accommodate multiple work-groups (based on the temporary register and local memory usage) simultaneously.

5.2.4 Memory hierarchy

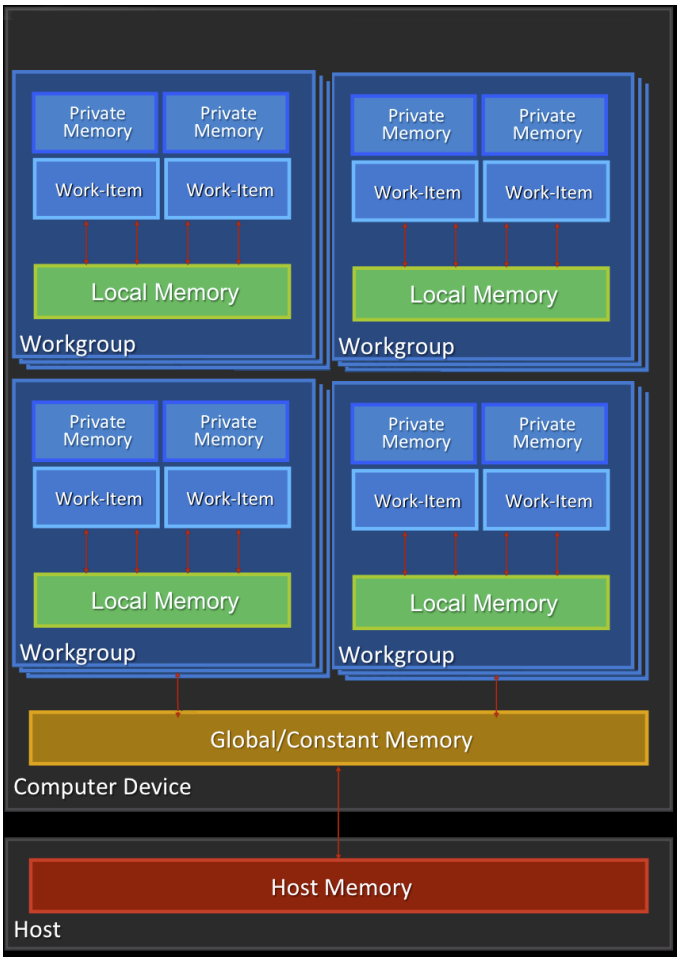


Figure 6 OpenCL memory hierarchy

5.3 Optimization for OpenCL embedded profile

OpenCL EP (Embedded Profile) is basically a scaled down version of OpenCL FP(Full Profile) and thus may require extra optimization. The guidelines below help with the optimization of Vivante OpenCL Embedded Profile GPGPU cores.

When optimizing code on Vivante hardware, it is important to remember a few key points to get the best performance from the hardware:

- Take advantage of algorithm and data parallelism
- Choose the correct execution configuration (more details below)
- Overlap memory transfer from different levels of the OpenCL memory hierarchy with simultaneous thread execution
- Maximize memory bandwidth and minimize data transfers (large transfers are more beneficial than many smaller transfers because of the impact of latency)
- Maximize instruction throughput and minimize instruction count

5.3.1 Using preferred multiple of work-group size

The work-group size should be a multiple of the thread group size, otherwise some threads remain idle and the application does not fully utilize all the compute resources. For example, if the work-group size is 8 and the Vivante core supports 16, only half the compute resources are used. For example, in some early Vivante GPGPU revisions, the work-group size limit is 192 and the thread group size is 16. See the Overview section on OpenCL Compatible IP for IP-specific capabilities.

5.3.2 Using multiple work-groups of reduced size

Multiple work groups need to be set to reduce synchronization penalties. To prevent stalls at barriers, it is recommended to have at least four (4) work-groups to keep the cores busy or as long as the number of work-groups is greater than or equal to two (2). One work-group is very inefficient; four or more is preferred and helps avoid latency.

5.3.3 Packing work-item data

It is important to pack data to extract the optimal performance from the SIMD ALU hardware and align the data into a format supported by the hardware. Efficient use of the Vivante GPGPU core requires that the kernel contains enough parallelism to fill all four vector units. Work-items in the same thread group have the same program counter and execute the same instruction for each cycle. Whenever possible, pack together work-items that follow the same direction (e.g., on branches) since the granularity is very close and there may be less divergence and higher performance. If each work-item handles less than or equal to 8 bytes, it is better to combine two or more work-items into one to improve utilization of the SIMD ALU.

5.3.4 Improving locality

If the input data is an array-of-structs, and each work-item needs to access only a small part of the struct across many array elements at different stages, it may be better to convert and use a struct-of-arrays or several different arrays as input to improve data locality and avoid cache thrashing.
If each work-item needs to process a row of data without sharing any data with other work-items, it is better to check if the algorithm can be converted to make each work-item process a column of data so that data accessed by adjacent work-items can share the same cache lines.

5.3.5 Minimizing use of 1 KB local memory

The OpenCL Embedded Profile specification defines the minimum requirement for local memory to be 1KB to pass conformance testing. Based on algorithm analysis and profiling different image and computer vision algorithms, we found that a 1KB local memory size was too small to benefit those algorithms. In most instances, those algorithms actually slowed down when using 1KB local memory. To increase performance, we recommend not using local memory since it is more efficient to transfer larger chunks of data from system memory to keep the OpenCL pipeline full.
Note: if local memory type is CL_GLOBAL, the local memory is emulated using global memory, and the performance is the same as global memory. There is extra overhead on data copy from global to local, which slows down the performance.

5.3.6 Using 16 byte memory Read/Write size

When accessing memory, it is important to minimize the read/write count and to ensure L1 cache utilization is high to reduce outstanding read/write requests. Since the internal GPGPU read-write-request queue has a limit, if the queue and L1 cache are filled, then the GPGPU remains idle.

5.3.7 Useing _RTZ rounding mode

Wherever possible, use _RTZ (round to zero) since it is natively supported in hardware with one instruction. Support for _RTE (round to nearest even) is optional in OpenCL EP and is only supported in Vivante GPGPU EP hardware from 2013. This function is handled in software for EP cores if necessary.

5.3.8 Using native functions

5.3.8.1 Using native_function() for increased performance

There are two types of runtime math libraries available to developers. Native_function() and regular function().

- Function(): slower, computationally expensive, higher instruction count, and greater accuracy
- Native_function(): faster, computationally inexpensive, lower instruction count (sometimes reduced to one instruction), and lower accuracy.
- If accuracy is not important but speed/performance is, use native math functions that map directly to the Vivante GPGPU hardware.

For image processing computations that do not require high accuracy, use native instructions to significantly lower the instruction count and speed up performance. Based on actual analysis and performance profiling with the Vivante GPGPU, we found that using native_function() instructions such as sin, cos, etc., reduces the instruction count from many instructions to one or two instructions. Use of native functions also sped performance by 3x-10x.

5.3.8.2 Using native_divide and native_reciprocal for faster floating point calculations

There are two use cases for floating point division which a user can select:

- Normal use of the division operator (/) in OpenCL has high precision and covers all corner use cases. This operator generates more instructions and runs slower.
- Native Divide: this use case uses the built-in function native_divide or native_reciprocal, which uses what the hardware supports. The Vivante OpenCL compiler generates one or two instructions for each native_divide or native_reciprocal instruction. If there are no corner use cases in applications, such as NaN, INF, or (2^127) / (2^127), it is better to use native_divide since it is faster.

5.3.8.3 Using compile option for native functions

Both the function() and native_function() methods are supported in the Vivante GPGPUs, so it is up to the developer to use whichever method makes sense for their application. If the OpenCL program uses the standard division operator and a developer wants to use native_divide or native_reciprocal without modifying their program, the Vivante OpenCL compiler has a simple option “-cl-fast-relaxed-math” that uses native built-in functions during compilation.

5.3.9 Using buffers instead of images

For the following image functions, it is better to use buffers instead of images.

- read_image{f/i/ui/h}
- write_image{f/i/ui/h}

Write_image* functions are implemented by software; it is better to use buffers to reduce the additional overhead involved in checking for size, format, etc. Since a few formats are not supported by Vivante GPGPU hardware, some built-in read_image() functions are implemented in software. The software implementation uses more instructions with many steps of “condition” checking. To improve performance, we recommend using buffers since it reduces instruction count.

5.4 OpenCL Debug messages

When writing OpenCL applications, it is important to check the code returned by the API. Since the return codes specified in the OpenCL specification may not be descriptive enough to isolate where the problem is located, the Vivante OpenCL driver provides an environment variable, VIV_DEBUG to help debug problems. When VIV_DEBUG is set to -MSG_LEVEL:ERROR, the Vivante OpenCL driver prints onscreen error messages as well as return the error code to the caller.
The following error code descriptions and suggested workarounds are provided.

5.4.1 OCL-007005: (clCreateKernel) cannot link kernel

One of the following “Not Enough” messages usually precedes this message. Issuer indicates the real reason for the problem which may be:
Not Enough Register Memory (constant or temp)
Not Enough Instruction Memory

5.4.2 Not enough register memory

Local variables, including arrays, are implemented using temp registers. If an array is larger than then number of available temp registers, a link-time failure occurs.
WORKAROUNDS:
1. If the array size is more than 64, use an array address to force the compiler to use private memory instead of temp registers.
2. If there are many variables, use variable addresses to force the compiler to use private memory to reduce register usage.

Note that there is performance degradation when using private memory instead of registers. It is better to change the algorithm to use a smaller array or less variables.

5.4.3 Not enough instruction memory

WORKAROUNDS:
1. Replace sin/cos/tan/divide/powr/exp/exp2/exp10/log/log2/log10/sqrt/rsqrt/recip with native_sin/native_divide, etc.
2. Convert unrolled-loops back to loops.
3. Use buffer instead of image for write, and for reads which are not linear-filtered.
4. If the program is just too long, it should be split into two or more programs with intermediate data saved from one program to next.

5.4.4 GlobalWorkSize over hardware limit

WORKAROUND:
1. Split one clEnqueueNDRangeKernel into several instances. Change the kernel source to compute real global/local/group ID using offset as a parameter.
2. Convert one dimension to two dimensions, or two dimensions to three. For example, one dimension of 1M work-items can be converted to a GlobalWorkSize of 64K x16 work-items. The kernel function needs modification to reflect the change of dimension.

5.5 Instruction Cache Availablity for i.MX Graphics

This section describes the instruction cache (iCache) available in the Vivante graphics IP included in the selected i.MX products.

There is hardware support for iCache available for i.MX 6QuadPlus and all later IP including that used in i.MX 8 products. There is no SH (Shader) instruction limit for these newer chips beyond the ISA limitation of 2*20. Only the older chips have a SH instruction limit.

Table 18 i.MX Products with Graphics IP with iCache

i.MX Product	GPU IP & rev	Instruction Limit	Description
i.MX 8 Series and later	various *(from rev 5450)*	none	HW supports iCache
i.MX 6QuadPlus	GC2000 Plus rev FFFF5450	none	HW supports iCache
S32V234	GC3000 rev 5451	none	HW supports iCache

The SH limitation for i.MX products is listed in the following table.

Table 19 i.MX Products with Instruction Limited Graphics IP

i.MX Product	GPU IP & rev	Instruction Limit	Description
i.MX 6SX	GC400 rev 4645	256 for VS, 256 for PS	Separate Instruction buffers for Vertex Shader and for Pixel Shader
i.MX 7ULP	GCNanoUltra rev 4653a	256 for VS, 256 for PS	Separate Instruction buffers for Vertex Shader and for Pixel Shader
i.MX 6DL	GC880 rev 5106	512	Instruction buffer shared by Vertex and Pixel Shaders
i.MX 6Q	GC2000 rev 5108	512	Instruction buffer shared by Vertex and Pixel Shaders

Chapter 6 OpenVX Introduction

6.1 Overview

OpenVX is a low-level programming framework domain to enable software developers to efficiently access computer vision hardware acceleration with both functional and performance portability. OpenVX has been designed to support modern hardware architectures, such as mobile and embedded SoCs as well as desktop systems. Many of these systems are parallel and heterogeneous: containing multiple processor types including multi-core CPUs, DSP subsystems, GPUs, dedicated vision computing fabrics as well as hardwired functionality. Additionally, vision system memory hierarchies can often be complex, distributed, and not fully coherent. OpenVX is designed to maximize functional and performance portability across these diverse hardware platforms, providing a computer vision framework that efficiently addresses current and future hardware architectures with minimal impact on applications.

OpenVX defines a C Application Programming Interface (API) for building, verifying, and coordinating graph execution, as well as for accessing memory objects. The graph abstraction enables OpenVX implementers to optimize the execution of the graph for the underlying acceleration architecture.

OpenVX also defines the vxu utility library, which exposes each OpenVX predefined function as a directly callable C function, without the need for first creating a graph. Applications built using the vxu library do not benefit from the optimizations enabled by graphs; however, the vxu library can be useful as the simplest way to use OpenVX and as first step in porting existing vision applications.

For more details of programming with OpenVX, see the following specification from Khronos Group,
OpenVX 1.0.1 specification (https://www.khronos.org/registry/vx).

6.2 Designing framework of OpenVX

6.2.1 Software landscape

OpenVX (OVX) is intended to be used either directly by applications or as the acceleration layer for higher-level vision frameworks, engines or platform APIs. Vivante software includes VX (Vision Imaging Accelleration) control mechanisms for hardware accelerated vision imaging, therefy allowing the user to implement customized applications and drivers using the Vivante–specific Vivante VX API (Application Programming Interface). This API provides programmable user kernel extensions for OpenCL 1.2 and provides additional Vision functionality to supplement those currently available with OpenVX 1.0.1 open standard from the Khronos group.

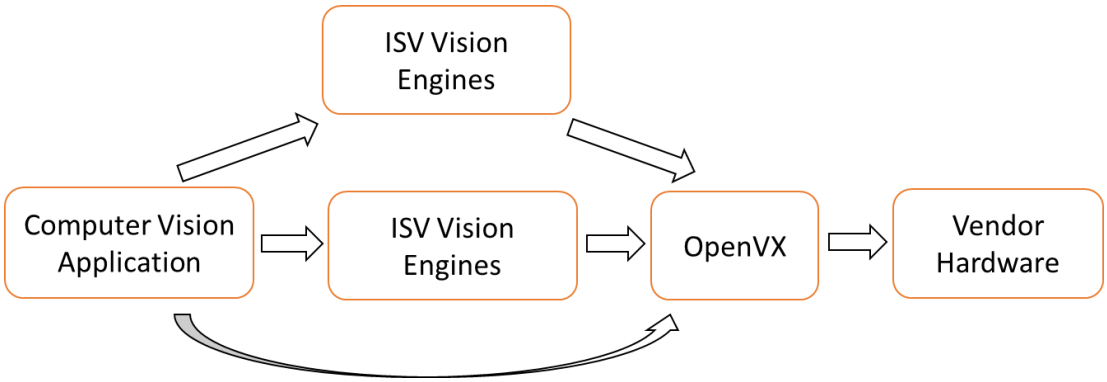


Figure 7 OVX usage overview

6.2.2 Object-oriented behaviors

OpenVX objects are both strongly typed at compile-time for safety critical applications and are strongly typed at

run-time for dynamic applications.

The objects of OVX framework are:

- Context, The OpenVX context is the object domain for all OpenVX objects.
- Kernel, A Kernel in OpenVX is the abstract representation of a computer vision function, such as a “Sobel Gradient” or “Lucas Kanade Feature Tracking”.
- Parameter, an abstract input, output, or bidirectional data object passed to a computer vision function.
- Node, A node is an instance of a kernel that will be paired with a specific set of references (the parameters).
- Graph, A set of nodes connected in a directed (only goes one-way) acyclic (does not loop back) fashion.

OpenVX Data Objects:

- Array, An opaque array object that could be an array of primitive data types or an array of structures.
- Convolution, An opaque object that contains MxN matrix of vx_int16 values. Also contains a scaling factor for normalization.
- Delay, An opaque object that contains a manually controlled, temporally-delayed list of objects.
- Distribution, An opaque object that contains a frequency distribution (e.g., a histogram).
- Image, An opaque image object that may be some format in vx_df_image_e.
- LUT, An opaque lookup table object used with vxTableLookupNode and vxuTableLookup
- Matrix, An opaque object that contains MxN matrix of some scalar values.
- Pyramid, An opaque object that contains multiple levels of scaled vx_image objects.
- Remap, An opaque object that contains the map of source points to destination points used to transform images.
- Scalar, An opaque object that contains a single primitive data type.
- Threshold, An opaque object that contains the thresholding configuration.

Error objects of OVX:

Error objects are specialized objects that may be returned from other object creator functions when serious platform issue occur (i.e., out of memory or out of handles). These can be checked at the time of creation of these objects, but checking also may be put-off until usage in other APIs or verification time, in which case, the implementation must return appropriate errors to indicate that an invalid object type was used.

6.2.3 Graphs concepts

The graph is the central computation concept of OpenVX. The purpose of using graphs to express the Computer Vision problem is to allow for the possibility of any implementation to maximize its optimization potential because all the operations of the graph and its dependencies are known ahead of time, before the graph is processed.

Graphs are composed of one or more nodes that are added to the graph through node creation functions. Graphs in OpenVX must be created ahead of processing time and verified by the implementation, after which they can be processed as many times as needed.

There are several nodes in a graph, which are responsible for independent computation. One node can be linked to another by data dependencies.

6.2.4 User kernels

OpenVX allows users to define new functions that can be excuted as Nodes from inside Graph or are Graph internal. Users will benefit from this mode,

- Exploiting
- Allow componentized functions to be reused elsewhere in OpenVX
- Formalize strict verification requirements (i.e., Contract Programming).

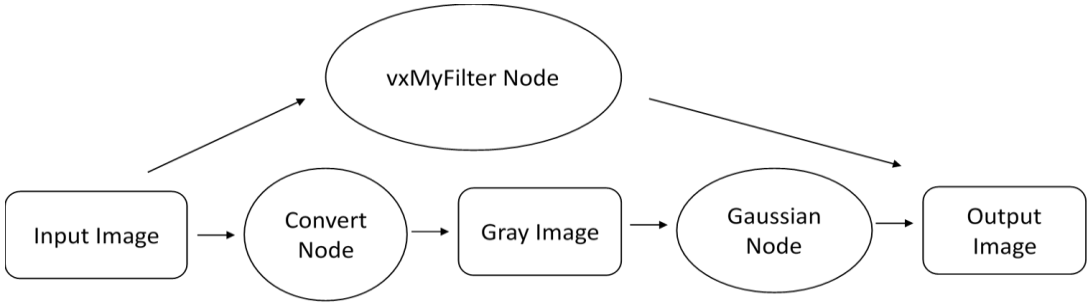


Figure 8 Graph and user kernel usage

6.3 OpenVX extension implementation

VeriSilicon's VX Extensions for Vision Imaging provide additional functionality for Vision Image processing beyond the functions provided through the Khronos Group OpenVX API version 1.0.1. These enhancements take advantage of the enhanced Vision capabilities available in VeriSilicon's Vision-capable hardware. VeriSilicon software provides a set of extensions which interface with OpenCL 1.2 and support higher level C language programming of VeriSilicon's custom EVIS (Enhanced Vision Instruction Set).

The VeriSilicon VX extension and enhancements includes three major components:

- An API level interface to the EVIS (Enhanced Vision Instruction Set),
- Extended C language features for Vision Processing,
- Supported for a subset of Vision-compatible OpenCL built-in functions.

6.3.1 Hardware requirements

Initial VeriSilicon cores with Vision Imaging hardware capabilities include:

- GC7000XSVX (available in i.MX 8QuadMax)

6.3.2 EVIS instruction interface

Vivante's Vision Imaging capable IP have an Enhanced Vision Instruction Set (EVIS), which enhances the ability of the GPU or VIP (Vision Image Processor) to process complex vision operations. A single EVIS instruction can do a task which may require tens or even hundreds of normal ISA instructions to finish.
Table 20 shows the instructions supported as Intrinsic calls.

6.3.3 Extended language features

Vivante's OpenVX C programming Language corresponds closely to the OpenCL C programming language.

- Vivante's C language extensions for OpenVX C share many language facilities with OpenCL C 1.2. However, it can be considered a subset of OpenCL C 1.2, as it does not include OCL features which are useless for OpenVX and other Vision Imaging applications.
- Vivante's OpenVX C includes specific language facilities like Vision built-ins and data types specific for OpenVX.

Table 20. OPCODE EVIS instructions supported as intrinsic calls

EVIS OP_CODE	Description	Supported by Vivante VX
ABS_DIFF	Absolute difference between two values	Y
IADD	Adds two or three integer values	Y
IACC_SQ	Squares a value and adds it to an accumulator	Y
LERP	Linear interpolation between two values	Y
FILTER	Performs a filter on a 3x3 block	Y
MAG_PHASE	Computes magnitude and phase of 2 packed data values	Y
MUL_SHIFT	Multiples two 8-or 16-bit integers and shifts	Y
DP16X1	1 Dot Product from 2 16 component values	Y
DP8X2	2 Dot Products from 2 8 component values	Y
DP4X4	4 Dot Products from 2 4 component values	Y
DP2X8	8 Dot Products from 2 2 component values	Y
CLAMP	Clamps up to 16 values to a max or min value	Y
BI_LINEAR	Computes a bi0linear interpolation of 4 pixel values	Y
SELECT_ADD	Adds a pixel value or increments a counter inside bins	Y
ATOMIC_ADD	Adds a valid atomically to an address	Y
BIT_EXTRACT	Extracts up to 8 bitfields from a packed stream	Y
BIT_REPLACE	Replaces up to 8 bitfields from a packed stream	Y
DP32X1	1 Dot Product from 2 32 component values	Y
DP16X2	2 Dot Products from 2 16 component values	Y
DP8X4	4 Dot Products from 2 8 component values	Y
DP4X8	8 Dot Products from 2 4 component values	Y
DP2X16	16 Dot Products from 2 2 component values	Y

6.3.4 Packed types

Vivante's OpenCL compiler implements OpenCL C signed and unsigned char and short types in an unpacked format, such that a normal char4 occupies 128 bits (4 32-bit registers). This is undesirable for Vision applications, where packed data is the "natural" layout for almost all operations. To fully utilize the computing power of EVIS instructions, Vivante VX includes additional packed types, which can be identified by their **vxc_** prefix.

```
/* packed char2/4/8/16 */
typedef _viv_char2_packed vxc_char2;
typedef _viv_char4_packed vxc_char4;
typedef _viv_char8_packed vxc_char8;
typedef _viv_char16_packed vxc_char16;
/* packed uchar2/4/8/16 */
typedef _viv_uchar2_packed vxc_uchar2;
typedef _viv_uchar4_packed vxc_uchar4;
typedef _viv_uchar8_packed vxc_uchar8;
typedef _viv_uchar16_packed vxc_uchar16;
/* packed short2/4/8 */
typedef _viv_short2_packed vxc_short2;
typedef _viv_short4_packed vxc_short4;
```

```
typedef _viv_short8_packed vxc_short8;
/* packed ushort2/4/8 */
typedef _viv_ushort2_packed vxc_ushort2;
typedef _viv_ushort4_packed vxc_ushort4;
typedef _viv_ushort8_packed vxc_ushort8;
```

6.3.5 Initializing constants on load

Constant data in OpenCL requires compile-time initialization. There is also a need to initialize the data when the kernel is loaded/run, so that the application can control the behavior of a program by changing its constants at load-time. The VeriSilicon VX extended keyword **_viv_uniform** can be used to define load-time initialization constant data,

_viv_uniform vxc_512bits u512;

An application using VeriSilicon VX needs to set the proper values for _viv_uniform before the kernel program is run.

6.3.6 Inline assembly

A packed type cannot be used as an unpacked type in expressions or built-in functions. The programmer needs to convert packed type data to unpacked type data in order to perform these operations. The conversion negatively impacts performance in terms of both instruction count and register usage, so it is desirable to perform operations directly on packed data whenever possible. The Vivante Vision compiler accepts inline assembly for a wide range of operations to speed up packed data calculations.

For example, to add two packed char16 data, the programmer can use following inline assembly:

```
vxc_uchar16 a, b, c;
vxc_short8 b;
_viv_uniform vxc_512bits u512;
...
_viv_asm(ADD, c, a, b); /* c = a + b; */
where the syntax of inline assembly is:
_viv_asm(
OP_CODE,
dest,
source0,
source1
);
```

Table 21 lists the standard shader instructions that operate on packed data and are supported through inline assembly, keyword **_viv_asm**.

Table 21. OPCODES IR instructions supported by inline assembly

IR OP_CODE Instruction	Description	Supported by Vivante VX
ABS	Absolute value	Y
ADD	Add	Y
ADD_SAT	Integer add with saturation	Y
AND_BITWISE	Bitwise AND	Y
BIT_REVERSAL	Integer bit-wise reversal	ES31
BITEXTRACT	Extract Bits from src to dest	ES31

BITINSERT	Bit replacement	ES31
BITSEL	Bitwise Select	Y
BYTE_REVERSAL	Integer byte-wise reversal	ES31
CLAMP0MAX	clamp0max dest, value, max	Y
CMP	Compare each component	Y
CONV	Convert	Y
DIV	Divide	Y
FINDLSB	Find least significant bit	ES31
FINDMSB	Find most significant bit	ES31
LEADZERO	Detect Leading Zero	Y
LSHIFT	Left Shifter	Y
MADSAT	Integer multiple and add with saturation	Y
MOD	Modulus	Y
MOV	Move	Y
MUL	Multiply	Y
MULHI	Integer only	Y
MULSAT	Integer multiply with saturation	Y
NEG	neg(a) is similar to (0 - (a))	Y
NOT_BITWISE	Bitwise NOT	Y
OR_BITWISE	Bitwise OR	Y
POPCOUNT	Population Count	ES31/OCL1.2
ROTATE	Rotate	Y
RSHIFT	Right Shifter	Y
SUB	Substract	Y
SUBSAT	Integer subtraction with saturation	Y
XOR_BITWISE	Bitwise XOR	Y

*ES31 = Supported by VivanteVX, but may not be needed for Vision processing

6.4 OpenCL functions compatible with Vivante vision

Vivante's VX extensions for Vision Image processing support most of the OpenCL 1.2 built-in functions for normal OCL data types. Packed types are not supported in these built-in functions.
For image read/write functions, only sample-less 1D/1D array/2D image read/write functions are supported.

6.4.1 Read_Imagef,i,ui

```
/* OCL image builtins can be used in VX kernel */
float4 read_imagef (image2d_t image, int2 coord);
int4 read_imagei (image2d_t image, int2 coord);
uint4 read_imageui (image2d_t image, int2 coord);
float4 read_imagef (image1d_t image, int coord);
int4 read_imagei (image1d_t image, int coord);
```

```
uint4 read_imageui (image1d_t image, int coord);
float4 read_imagef (image1d_array_t image, int2 coord);
int4 read_imagei (image1d_array_t image, int2 coord);
uint4 read_imageui (image1d_array_t image, int2 coord);
```

6.4.2 Write_Imagef,i,ui

```
void write_imagef (image2d_t image, int2 coord, float4 color);
void write_imagei (image2d_t image, int2 coord, int4 color);
void write_imageui (image2d_t image, int2 coord, uint4 color);
void write_imagef (image1d_t image, int coord, float4 color);
void write_imagei (image1d_t image, int coord, int4 color);
void write_imageui (image1d_t image, int coord, uint4 color);
void write_imagef (image1d_array_t image, int2 coord, float4 color);
void write_imagei (image1d_array_t image, int2 coord, int4 color);
void write_imageui (image1d_array_t image, int2 coord, uint4 color)
```

6.4.3 Query Image Dimensions

```
int2 get_image_dim (image2d_t image);
size_t get_image_array_size(image1d_array_t image);
/* Built-in Image Query Functions */
int get_image_width (image1d_t image);
int get_image_width (image2d_t image);
int get_image_width (image1d_array_t image);
int get_image_height (image2d_t image);
```

6.4.4 Channel Data Types Supported

```
/* Return the channel data type. Valid values are:
* CLK_SNORM_INT8
* CLK_SNORM_INT16
* CLK_UNORM_INT8
* CLK_UNORM_INT16
* CLK_UNORM_SHORT_565
* CLK_UNORM_SHORT_555
* CLK_UNORM_SHORT_101010
* CLK_SIGNED_INT8
* CLK_SIGNED_INT16
* CLK_SIGNED_INT32
* CLK_UNSIGNED_INT8
* CLK_UNSIGNED_INT16
* CLK_UNSIGNED_INT32
* CLK_HALF_FLOAT
* CLK_FLOAT
*/
int get_image_channel_data_type (image1d_t image);
int get_image_channel_data_type (image2d_t image);
int get_image_channel_data_type (image1d_array_t image);
```

6.4.5 Image Channel Orders Supported

```
/* Return the image channel order. Valid values are:
```

```
* CLK_A
* CLK_R
* CLK_Rx
* CLK_RG
* CLK_RGx
* CLK_RA
* CLK_RGB
* CLK_RGBx
* CLK_RGBA
* CLK_ARGB
* CLK_BGRA
* CLK_INTENSITY
* CLK_LUMINANCE
*/
int get_image_channel_order (image1d_t image);
int get_image_channel_order (image2d_t image);
int get_image_channel_order (image1d_array_t image);
```

Chapter 7 Vulkan

7.1 OverView

Vulkan is a new generation graphics and compute API that provides high-efficiency, cross-platform access to modern GPUs used in a wide variety of devices from PCs and consoles to mobile phones and embedded platforms. Vulkan defines as an API (Application Programming Interface) for graphics and compute hardware. The API consists of many commands that allow a programmer to specify shader programs, compute kernels, objects, and operations involved in producing high-quality graphical images, specifically color images of three-dimensional objects.

To the programmer, Vulkan is a set of commands that allow the specification of shader programs or shaders, kernels, data used by kernels or shaders, and state controlling aspects of Vulkan outside the scope of shaders. Typically, the data represents geometry in two or three dimensions and texture images, while the shaders and kernels control the processing of the data, rasterization of the geometry, and the lighting and shading of fragments generated by rasterization, resulting in the rendering of geometry into the framebuffer.

A typical Vulkan program begins with platform-specific calls to open a window or otherwise prepare a display device onto which the program will draw. Then, calls are made to open queues to which command buffers are submitted. The command buffers contain lists of commands which will be executed by the underlying hardware. The application can also allocate device memory, associate resources with memory and refer to these resources from within command buffers. Drawing commands cause application-defined shader programs to be invoked, which can then consume the data in the resources and use them to produce graphical images. To display the resulting images, further platform-specific commands are made to transfer the resulting image to a display device or window.

For more details of programming with Vulkan, refer to the following specification from Khronos Group.
https://www.khronos.org/registry/vulkan/

7.2 Vivante Extension Support for Vulkan

The following table includes a list of all current Vulkan extensions and indicates their support level in Vivante software.

(list from *https://www.khronos.org/registry/vulkan/* as of 5/24/2017)

Note: This list does not include unsupported vendor specific extensions.

Table 22. Vulkan extension

Vulkan Extension Name	SW 6.2.x for Vulkan 1.0
VK_KHR_android_surface	YES
VK_KHR_descriptor_update_template	
VK_KHR_display	YES
VK_KHR_display_swapchain	YES
VK_KHR_get_physical_device_properties2	
VK_KHR_get_surface_capabilities2	
VK_KHR_incremental_present	
VK_KHR_maintenance1	
VK_KHR_mir_surface	
VK_KHR_push_descriptor	
VK_KHR_sampler_mirror_clamp_to_edge	
VK_KHR_shader_draw_parameters	
VK_KHR_shared_presentable_image	
VK_KHR_surface	YES

VK_KHR_swapchain	YES
VK_KHR_wayland_surface	YES
Vulkan Extension Name	**SW 6.2.x for Vulkan 1.0**
VK_KHR_win32_surface	YES
VK_KHR_xcb_surface	
VK_KHR_xlib_surface	
EXT Extensions (Multivendor)	
VK_EXT_acquire_xlib_display	
VK_EXT_debug_marker	
VK_EXT_debug_report	YES
VK_KHR_get_surface_capabilities2	
VK_KHR_incremental_present	
VK_KHR_maintenance1	
VK_EXT_direct_mode_display	
VK_EXT_discard_rectangles	
VK_EXT_display_control	
VK_EXT_display_surface_counter	
VK_EXT_hdr_metadata	
VK_EXT_shader_subgroup_ballot	
VK_EXT_shader_subgroup_vote	
VK_EXT_swapchain_colorspace	
VK_EXT_validation_flags	
GOOGLE Extensions (Google, Inc.)	
VK_GOOGLE_display_timing	
KHX Extensions (full vendor description unavailable)	
VK_KHX_device_group	
VK_KHX_device_group_creation	
VK_KHX_external_memory	
VK_KHX_external_memory_capabilities	
VK_KHX_external_memory_fd	
VK_KHX_external_memory_win32	
VK_KHX_external_semaphore	
VK_KHX_external_semaphore_capabilities	
VK_KHX_external_semaphore_fd	
VK_KHX_external_semaphore_win32	
VK_KHX_multiview	
VK_KHX_win32_keyed_mutex	

Chapter 8 Multiple GPUs and Virtualization

8.1 Overview

Vivante multi-GPU implementations provide a variety of capabilities which can be managed through hardware and software controls. This chapter intends to summarize the software controls used for Vivante multi-GPU IP implementations.

Multi-GPU feature can be enabled with dual GC7000XSVX on i.MX 8QuadMax and the derived devices.

8.2 Multi-GPU configurations

Vivante Multi-GPU IP may be configured into one of the following behavior model through SW:

Combined Mode where two (or more) GPU cores in the multi-GPU design behave in concert. Driver presents multi-GPU to SW application as a single logical GPU. The multiple GPUs work in the same virtual address space and share the same MMU page table. The multiple GPUs fetch and excute a shared Command Buffer.

Independent Mode where each GPU in the multi-GPU design performs independently. The multiple GPUs work in different virtual address spaces but share the same MMU page table. Each GPU core fetches and excutes its own Command Buffer. This enables different SW applications to run simultaneously on different GPU cores.

Note, OpenCL API allows application to handle the multi-GPU Independent Mode directly, as each GPU core in a multi-GPU design represents an independent OpenCL Compute Device. OpenCL driver does not support the multi-GPU combined mode.

8.3 GPU affinity configuration

In the multi-GPU Independent Mode, application can specify to run on a specific GPU among the multiple GPUs through an environment variable VIV_MGPU_AFFINITY. Once an application's GPU affinity is specified, the application will only run on the specified GPU and will not migrate to other GPUs even if those GPUs are idle.

VIV_MGPU_AFFINITY is the environment variable to control the application GPU affinity on multi-GPU platform. The client drivers will assume they are using a standalone GPU through a gcoHARDWARE object no matter how this variable is set. The possible values for the environment variable VIV_MGPU_AFFINITY include:

Not defined or

Defined as "0" gcoHARDWARE objects work in gcvMULTI_GPU_COMBINED mode (default)

"1:0" gcoHARDWARE objects work in gcvMULTI_GPU_INDEPENDENT mode and GPU0 is used

"1:1" `gcoHARDWARE` objects work in `gcvMULTI_GPU_INDEPENDENT` mode and GPU1 is used

On a single GPU device, setting VIV_MGPU_AFFINITY to 0 or 1 does not make any difference as all application processes/threads are bound to GPU0. But the application will fail the GPU context initialization if VIV_MGPU_AFFINITY is set to "1:1" (driver reports error).

8.4 OpenCL on multi-GPU device

OpenCL driver works in multi-GPU Independent Mode only. In this configuration, multiple GPUs in the device operate as individual OpenCL Compute Devices. The OpenCL application is responsible to assign and dispatch the compute tasks to each GPU (Compute Device).

The following OpenCL APIs return the list of compute devices available on a platform, and the device information.

cl_int **clGetDeviceIDs** (cl_platform_id *platform*, cl_device_type *device_type,* cl_uint *num_entries*, cl_device_id **devices*, cl_uint **num_devices*)

cl_int **clGetDeviceInfo** (cl_device_id *device*, cl_device_info *param_name*, size_t *param_value_size*, void **param_value*, size_t **param_value_size_ret*)

8.5 GPU virtualization configuration

Multi-GPU also can be used on different OS systems as independent mode separately, this can be configured by overriding the irq availability n DTS entry for different OS implementation, in arch/arm64/boot/dts/freescale/fsl-imx8qmxxx.dts.

Guest OS 1 (GPU0 only)

```
&gpu_3d1 {
         status = "disable";
};
```

Guest OS 2 (GPU1 only)

```
&gpu_3d0 {
         status = "disable";
};
```

Chapter 9 G2D compositor on Weston

9.1 Overview

Wayland is intended as a simpler replacement for X, easier to develop and maintain. GNOME and KDE are expected to be ported to it.

Wayland is a protocol for a compositor to talk to its clients as well as a C library implementation of that protocol. The compositor can be a standalone display server running on Linux kernel modesetting and evdev input devices, an X application, or a wayland client itself. The clients can be traditional applications, X servers (rootless or fullscreen) or other display servers.

Part of the Wayland project is also the Weston reference implementation of a Wayland compositor. Weston can run as an X client or under Linux KMS and ships with a few demo clients. The Weston compositor is a minimal and fast compositor and is suitable for many embedded and mobile use cases.

This chapter describes how to enable Weston accelerated by G2D APIS. G2D compositor can increase system bandwidth utilization, so the performance was better than GL compositor in the complex environment, but it still doesn't support display rotation and EXT_RESOLVE feature.

9.2 Enabe G2D compositor

9.2.1 Open the file: /etc/default/Weston in the Release image.

```
# cat /etc/default/weston
  #!/bin/sh
  OPTARGS="--xwayland"
```

9.2.2 Add the parameters in the OPTARGS, and disable EXT_RESOLVE feature in compositor.

```
OPTARGS="—xwayland –use-g2d=1"
GPU_VIV_EXT_RESOLVE=0
```

9.2.3 Restart Weston by this command:

```
# systemctl restart weston
```

9.2.4 Disable EXT_RESOLVE feature before running the client application.

```
# export GPU_VIV_EXT_RESOLVE=0
# Weston-simple-egl
```

Chapter 10 XServer Video Driver

10.1 EXA driver

XServer video driver is designed to help XServer to render desktop onto a screen. It manages the display driver, and provides rendering acceleration and other display features, such as rotation and multiple display methods. The video driver implements XServer EXcellent Architecture (EXA).

10.1.1 EXA driver options

These options are used in the configuration file `/etc/X11/xorg.conf`:

```
Section "Device"
    Identifier  "i.MX Accelerated Framebuffer Device"
    Driver      "vivante"
    Option      "fbdev"     "/dev/fb1"
    Option      "vivante_fbdev" "/dev/fb1"
    Option      "SyncDraw"  "false"
EndSection
```

Table 23. EXA driver options

Option	Meaning	Default Value	Comment
ShadowFB	Whether to enable the shadow frame buffer (FB).	False	*Deprecated technology.* It rotates the FB. If it is enabled, acceleration is disabled.
Rotate	Rotation of FB.	<null>	*Deprecated technology.* It can be CW/CCW/UD. If it is set to one of these values, Shadow FB is automatically enabled. Rotation cannot change after XServer is started.
NoAccel	Disables EXA acceleration.	False	If it is set to **True**, the EXA functions are not accelerated by the GPU.
VivCacheMem	Pixmap created by GPU is generally cacheable.	True	Normal Pixmaps are created cacheable. Special Pixmaps used for EGL are still non-cacheable.
SyncDraw	Wait for the GPU to complete for every single drawing.	False	This affects the performance if it is set to **True**.

10.1.2 24 bpp pixmap

The GPU can only accelerate a 16 bpp or 32 bpp pixmap. For a 24 bpp screen, a 32 bpp buffer is actually reserved.

10.1.3 Shared pixmap extension

The Shared Pixmap Extension (SHM) pixmap will be described in next release.

10.1.4 How to disable XRandR

For an embedded device that does not support XRandR (for which the memory can be reduced), set "gEnableXRandR" to **False** in vivante_fbdev_driver.c.

10.1.5 Cursor

Hardware IPU does not provide a hardware cursor.

10.1.6 DRI

DRI is designed to accelerate OpenGL rendering. It enables the GPU direct render to the on-screen buffer. Due to the lack of hard cursor support, and because often the window location is not well aligned, the GPU cannot render to screen directly. Therefore, DRI is not fully used.
DRI is supported in this video driver. DRI2 or DRI3 is not supported.

10.1.7 Tearing

XServer (and early Microsoft Windows OS) does not support double buffering for the screen. There is a copy from off-screen buffer to target on-screen area (or direct rendering to on-screen). The operation cannot be completed in the blank time of the display, and the IPU cannot provide an ideal VSYNC signal. Therefore, there is tearing.
To remove tearing, a GLES compositor is needed. This tearing free feature will be described in next release.

10.2 XRandR

This video driver supports XRandR.
The X Resize, Rotate and Reflect Extension (RandR) is an X Window System extension, which allows clients to dynamically resize, rotate, and reflect the root window of a screen (en.wikipedia.org/wiki/Xrandr).

10.2.1 Useful commands

If the display supports multiple resolution types, use the following commands for a query:

```
root@imx6qsabresd:~# export DISPLAY=:0.0
root@imx6qsabresd:~# xrandr
Screen 0: minimum 240 x 240, current 1920 x 1080, maximum 8192 x 8192
DISP3 BG connected 1920x1080+0+0 (normal left inverted right x axis y axis) 0mm x 0mm
   S:1920x1080p-50   50.0*
   S:1920x1080p-60   60.0
   S:1280x720p-50   50.0
   S:1280x720p-60   60.0
   S:720x576p-50   50.0
   S:720x480p-60   59.9
   V:640x480p-60   60.0
   S:640x480p-60   59.9
```

If using the console serial port for the command line interface, the DISPLAY environment variable is not configured by default and the xrandr command fails. The solution is to set the DISPLAY environment variable. (Reference: see manpage for X)

```
root@imx6qsabresd:~# xrandr
Can't open display
root@imx6qsabresd:~# echo $DISPLAY
```

```
root@imx6qsabresd:~# export DISPLAY=:0.0
root@imx6qsabresd:~# xrandr
Screen 0: minimum 240 x 240, current 1024 x 768, maximum 8192 x 8192
DISP4 BG - DI1 connected 1024x768+0+0 (normal left inverted right x axis y axis) 0mm x
0mm
   U:1024x768p-60   60.0*+
```

- Change the resolution:
  root@imx6qsabresd:~# xrandr -s 1920x1080

Figure 9 Changing the resolution

- Rotate the screen:
  root@imx6qsabresd:~# xrandr -o left:

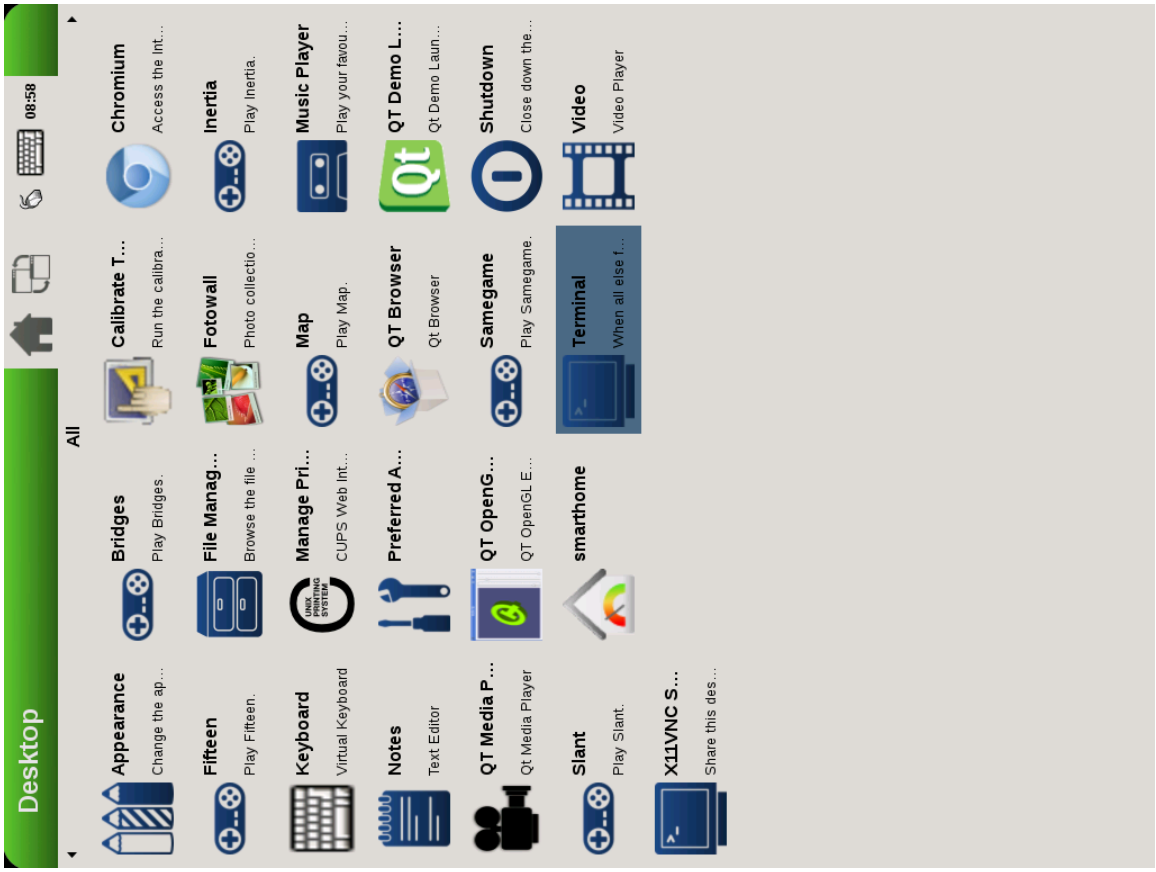


Figure 10 Rotating the screen

root@imx6qsabresd:~# xrandr -o right:

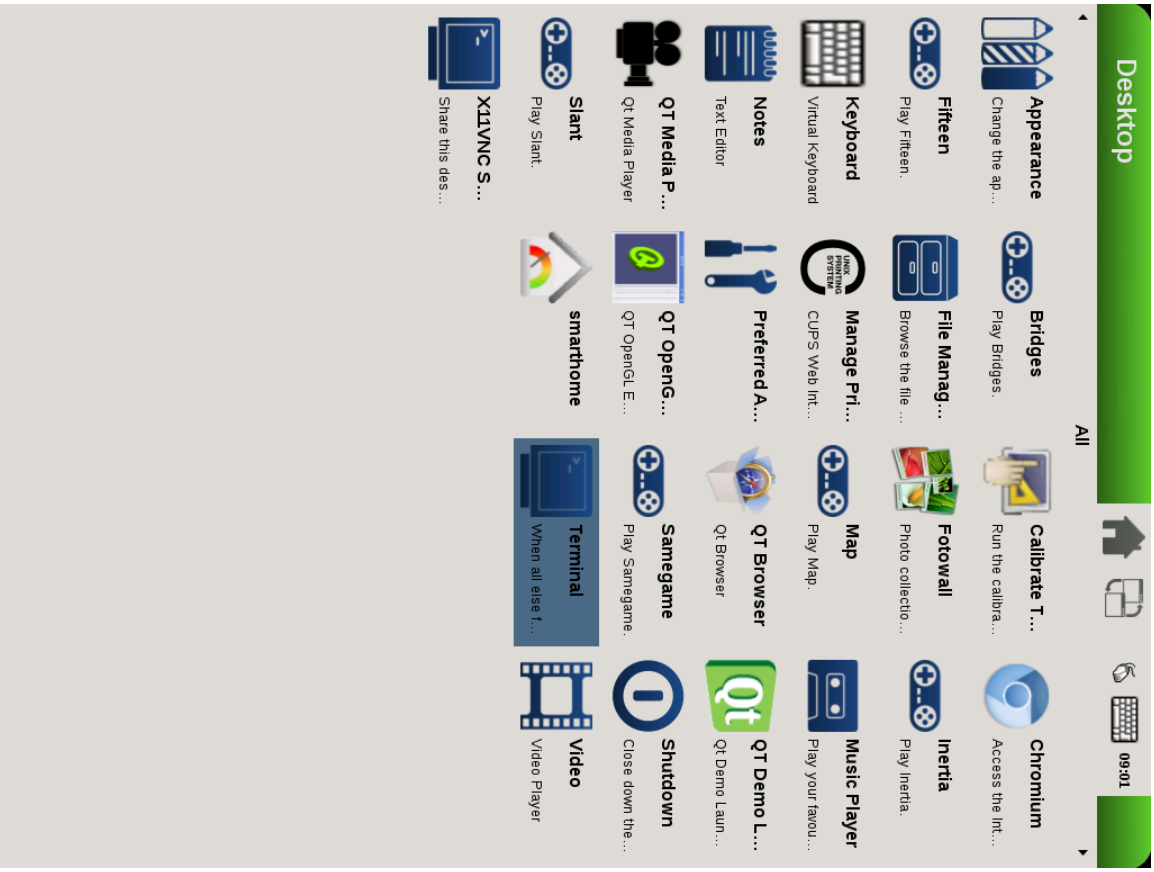


Figure 11 Rotating the screen

root@imx6qsabresd:~# xrandr -o inverted:

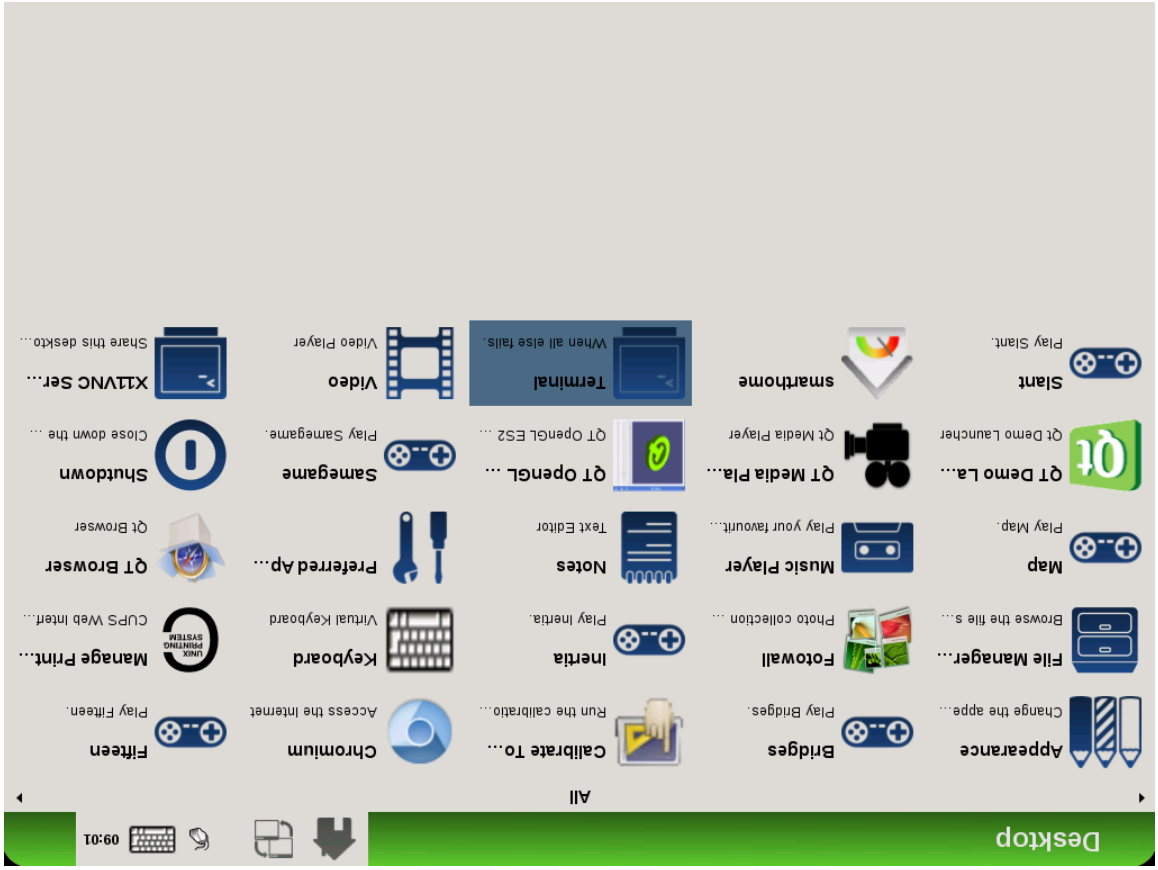


Figure 12 Rotating the screen

- Reflect the screen:
  root@imx6qsabresd:~# xrandr -x

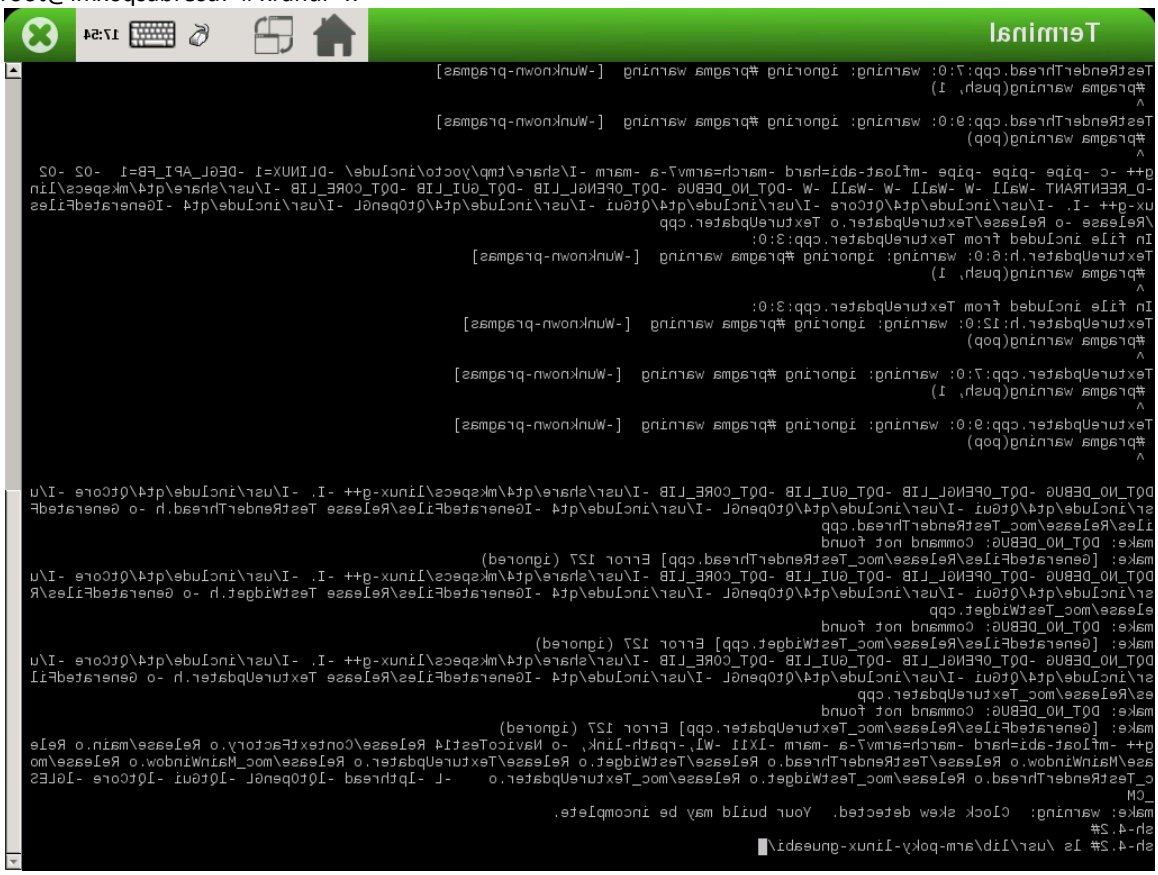


Figure 13 Reflecting the screen

```
root@imx6qsabresd:~# xrandr -y
```

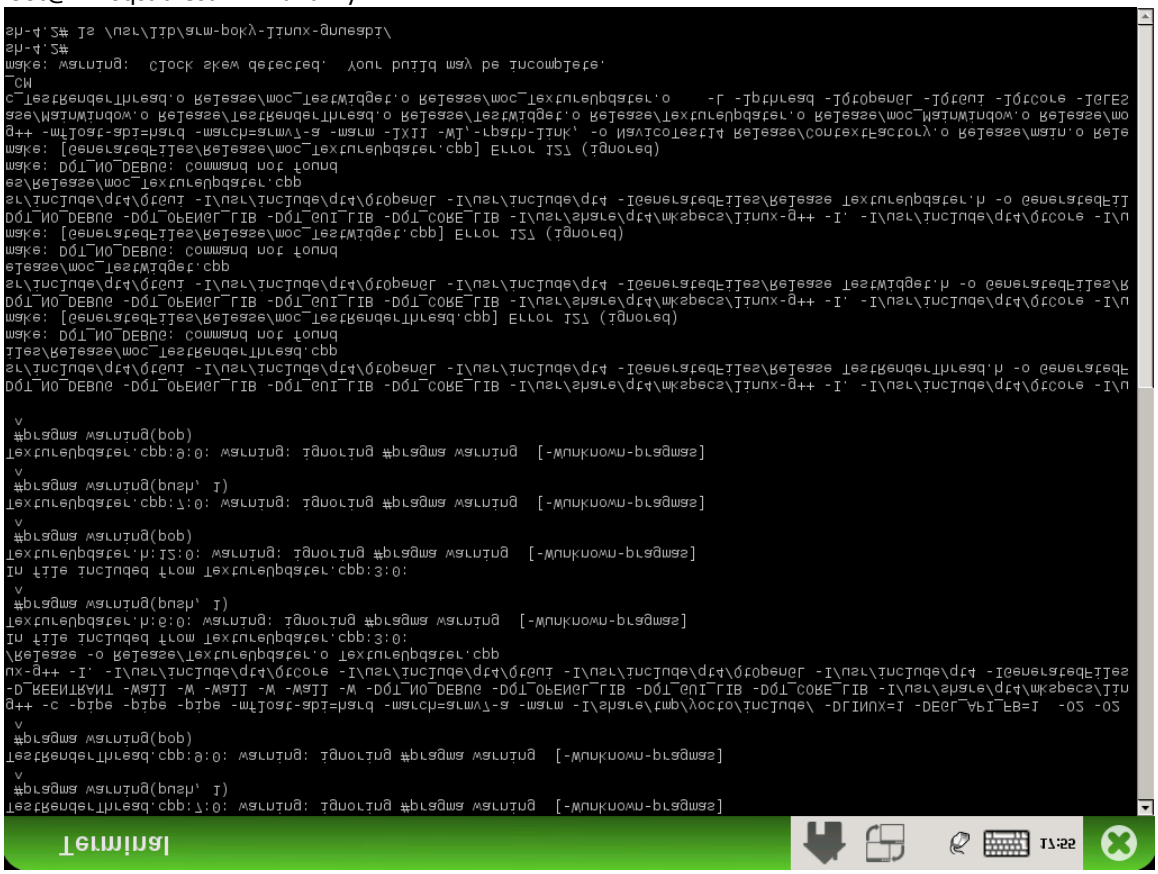


Figure 14 Reflecting the screen

- Restore to normal state:
  root@imx6qsabresd:~# xrandr -o normal:

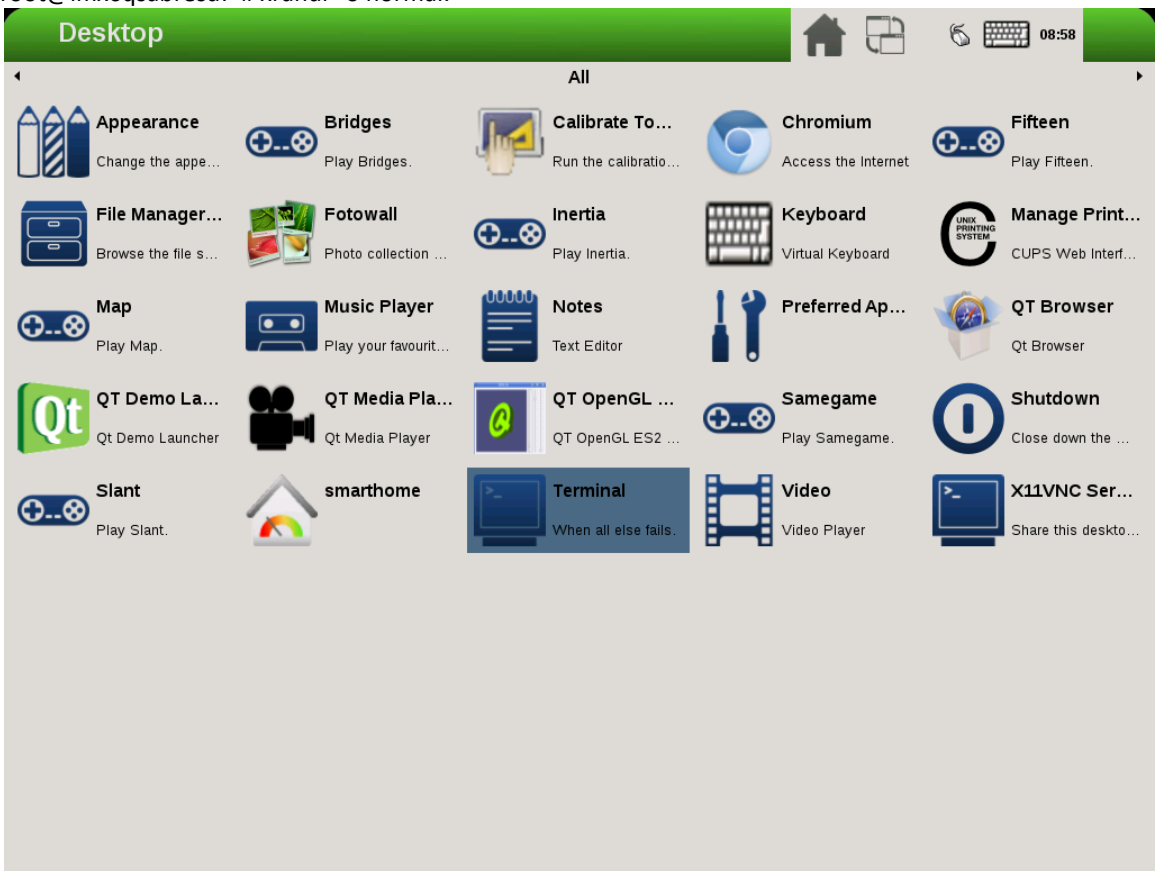


Figure 15 Restoring to normal state

10.2.2 Rendering the desktop on overlay

`/dev/fb1` is the overlay device on the same screen as `/dev/fb0`; and `/dev/fb3` is the overlay of `/dev/fb2`. Use xorg.conf to specify fb1 or fb3:

```
Section "Device"
    Identifier  "i.MX Accelerated Framebuffer Device"
    Driver      "vivante"
    Option      "fbdev"     "/dev/fb1"
    Option      "vivante_fbdev" "/dev/fb1"
EndSection
```

After rebooting the system, the desktop is rendered on the overlay:

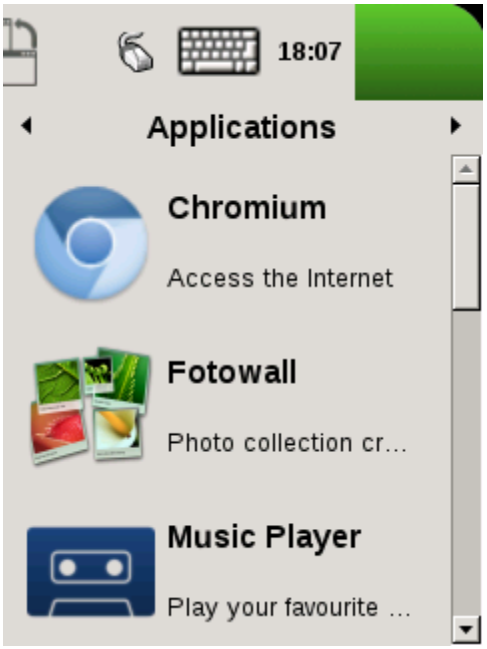


Figure 16 Rendering the desktop on overlay

If the size is too small (240x240), XRandR can be used to define a new mode.

1. Get the output name:

```
root@imx6qsabresd:~# xrandr
Screen 0: minimum 240 x 240, current 240 x 320, maximum 8192 x 8192
DISP4 FG connected 240x320+0+0 (normal left inverted right x axis y axis) 0mm x 0mm
   U:240x320p-60   60.0*
```

2. Define a new mode:

   root@imx6qsabresd:~# xrandr --newmode "640x480R" 23.50 640 688 720 800 480 483 487 494 +hsync -vsync

3. Add the newly created mode:

   root@imx6qsabresd:~# xrandr --addmode "DISP4 FG" 640x480R

4. Check the modes:

   root@imx6qsabresd:~# xrandr

```
Screen 0: minimum 240 x 240, current 240 x 320, maximum 8192 x 8192
DISP4 FG connected 240x320+0+0 (normal left inverted right x axis y axis) 0mm x 0mm
   U:240x320p-60   60.0*
   640x480R       59.5
```

5. Switch to a new mode:

   root@imx6qsabresd:~# xrandr -s 640x480

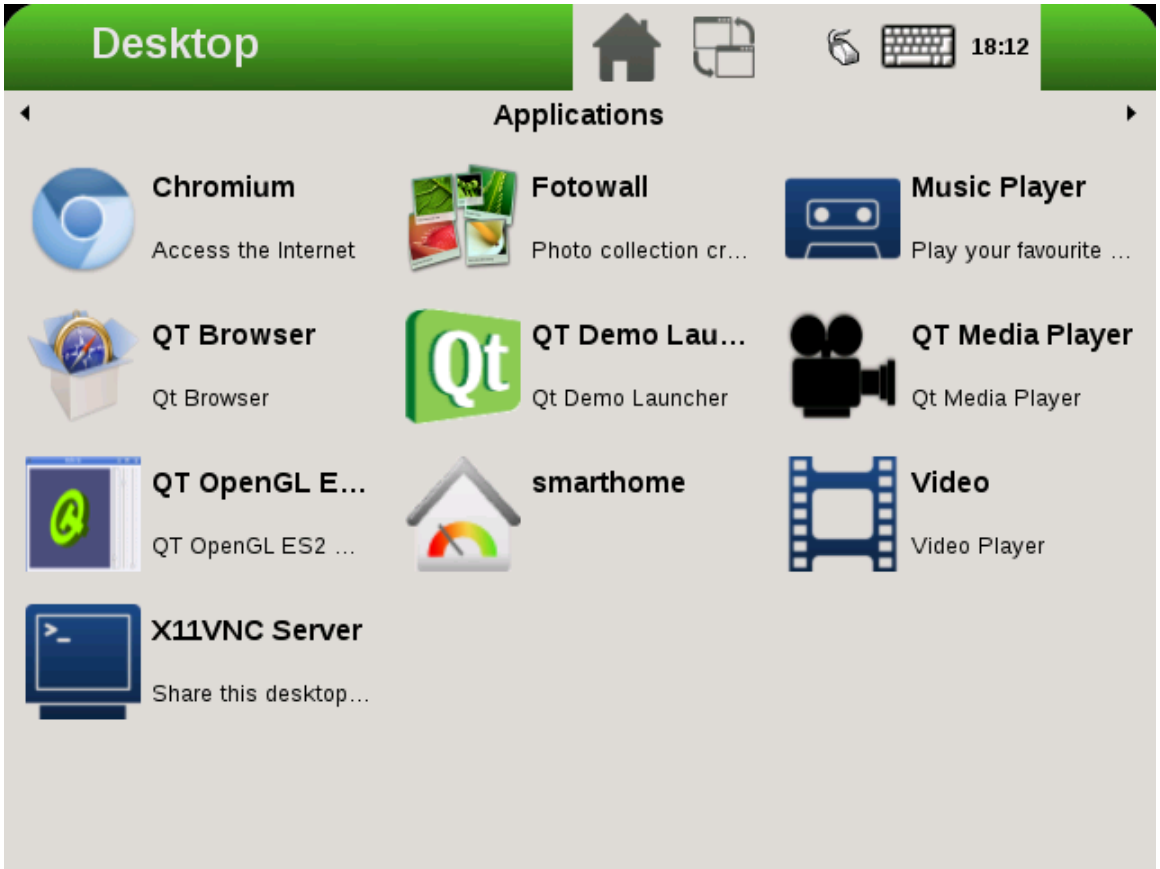


Figure 17 Switching to a new mode

Note:

- The overlay size cannot exceed the display size. For example, if LVDS is 1024x768, the overlay size cannot be larger than this.
- Timings for overlay are meaningless, but wrong timings may damage the display, so be careful when creating a new display mode for the display.
- If fb3 is used, fb2 must be enabled. Otherwise, fb3 is invisible.

10.2.3 Process of selecting the HDMI default resolution

The process of selecting the HDMI default resolution is as follows:

1. Set the user preferred mode (must be within the initial size).
2. Set the display preferred mode (must be within the initial size).
3. Check the aspect (if not found, use 4:3. Find the biggest resolution within the initial size for the aspect ratio).
4. Check the first mode.

Initial size: initial FB virtual size or configured maximum size.
To specify the user preferred mode, add the option “PreferredMode” or “modes”.

10.2.4 Performance

The performance is decreased during screen rotation or mirroring.

10.2.5 Memory consumption

The video driver supports a maximum of 1920x1080@32bpp. To support rotation, a shadow buffer is reserved, so the total memory consumption is 16 MB (1920x1080x4x2).

Chapter 11 Advanced GPU Configuration

11.1 GPU Scaling Governor

i.MX 8QuadMax GPU DVFS design supports different running modes: overdrive, nominal, and underdrive. Nominal is the default, the overdrive is supposed to be performance/benchmark mode, and underdrive mode is expected as energy saving mode.
Try to switch among the 3 modes, just using command line after boot without recompile the gpu driver.

```
$ echo "overdrive"   >  /sys/bus/platform/drivers/galcore/gpu_mode
$ echo "nominal"     >  /sys/bus/platform/drivers/galcore/gpu_mode
$ echo "underdrive" >  /sys/bus/platform/drivers/galcore/gpu_mode
```

Try to check which mode is running on now, using command line as below:

```
$ cat /sys/bus/platform/drivers/galcore/gpu_mode
```

11.2 GPU Device Cooling

i.MX device support the thermal driver, which could signal the overheat event to GPU driver, once GPU driver receive the event, it can enable GPU DFS feature to reduce GPU frequency as N/64 of the original designated clock. The default N factor is 1 in the original BSP release, the end-user can reconfigure it through below command:

```
echo N >/sys/bus/platform/drivers/galcore/gpu3DMinClock
```

The user also can check the existing config as below

```
cat  /sys/bus/platform/drivers/galcore/gpu3DMinClock
```

Chapter 12 Vivante Software Tool Kit

This chapter contains copyright material disclosed with permission of Vivante Corporation.

12.1 Vivante Tool Kit overview

The Vivante Tool Kit (VTK) is a set of applications designed to be used by graphics application developers to rapidly develop and port graphics applications either stand alone, or as part of an IDE targeting a system-on-chip (SoC) platform containing an embedded GPU.

12.1.1 VTK component overview

The VTK includes a graphics and OpenCL emulator (vEmulator) to enable embedded graphics and compute application development on a PC platform, a driver and hardware performance profiling utility (vProfiler), and a visual analyzer (vAnalyzer) for graphing the performance metrics. Also provided are pre-processing utilities for stand-alone development of optimized shader programs (vShader) and for compiling shader code (vCompiler) into binary executables targeting Vivante accelerated hardware platforms. An image transfer utility (vTexture) provides compression and decompression options.

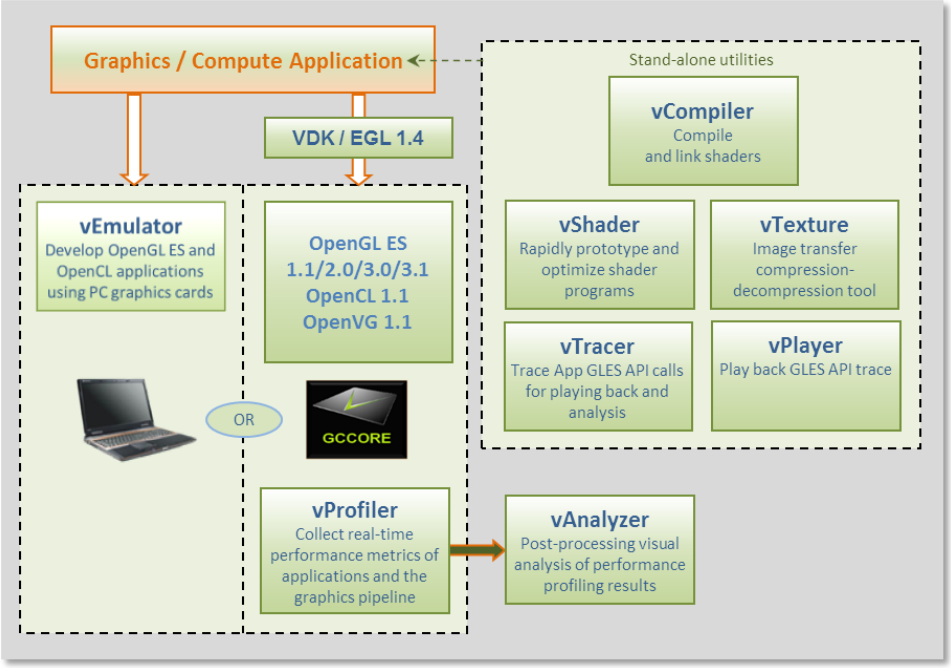


Figure 18 Vivante Tool Kit vTools components

12.1.2 VTK operating system requirements

Most VTK vTools applications are designed to run on Microsoft Windows operating systems. The following systems are compatible with current releases of vTools:

- Microsoft Windows® XP Professional, with Service Pack 2 or later
- Microsoft Windows® Vista with Service Pack 2 or later
- Microsoft Windows® 7 Professional

Some components, such as the vProfiler, are run on other platforms. See the individual vTools component detail description.

12.1.3 VTK installation

The vProfiler tool is not included in the VTK. This tool can be built by setting a build command option when making the Vivante Graphics Drivers.

The VTK package contains a **`vtools`** folder. Inside this folder are six .zip packages which can be individually extracted. As an example, for a WinRAR system, right-click and select Extract Here. A folder is created with the same name as the .zip file.

- **vAnalyze.zip**
- **vCompiler.zip**
- **vEmulator.zip**
- **vShader.zip**
- **vTexture.zip**
- **vTracer.zip**

Each vTools extracted folder contains a **`SETUP.exe`** and a ***`vToolName.msi`*** file. The tool can be installed independently by running the **`SETUP.exe`** located in the tool folder. Typical licensing and folder placement options may appear as part of the installation prompts.

vAnalyzer and vShader have a Windows GUI. vEmulator is a library. vCompiler and vTexture are utilities run from the command line.
NOTES:
• The default installation location for the VTK is usually a folder named something like **C:\Program Files\Vivante*vToolName*,** where *vToolName* is the name of the tool being installed. Some systems may install to a Program Files (x86) folder.
• Windows OS navigation instructions such as Control Panel navigation vary with the different Windows operating systems.
• Administrator rights may be required to install the tool.
• When installing an updated version, use Windows OS Add/Remove programs to remove the installed version of the tool, before installing the update version.

12.1.4 Software release compatibility

- SW release 5.0.11.p7 - VTK v1.6.2.p1
- SW release 5.1.1 - VTK v1.6.3
- SW release 5.0.11_p6 - VTK v1.6.2
- SW release 5.0.11_p5 - VTK v1.6.1
- SW release 5.0.11_p4 - VTK v1.6.0
- SW release 5.0.11_p3 - VTK v1.5.9 and v1.5.8
- SW release 5.0.11_p2 - VTK v1.5.7
- SW release 4.6.9.p13, 5.0.9 and 5.0.9.1 - VTK v1.5.3
- SW release 4.6.9.p13 and 5.0.7 - VTK v1.5
- SW release 4.6.9.p9 - VTK v1.4.2
- SW release 4.6.9 - VTK v1.4

12.2 vEmulator

Vivante's vEmulator duplicates the graphics and compute functionality of the Khronos APIs—namely, OpenGL ES 3.0, 2.0, 1.1 and OpenCL 1.1—in a desktop PC environment. This enables developers to write and test applications for Vivante embedded GPU cores prior to their availability, using the graphics cards on Windows® XP or Windows® Vista or Windows® 7 PC platforms.

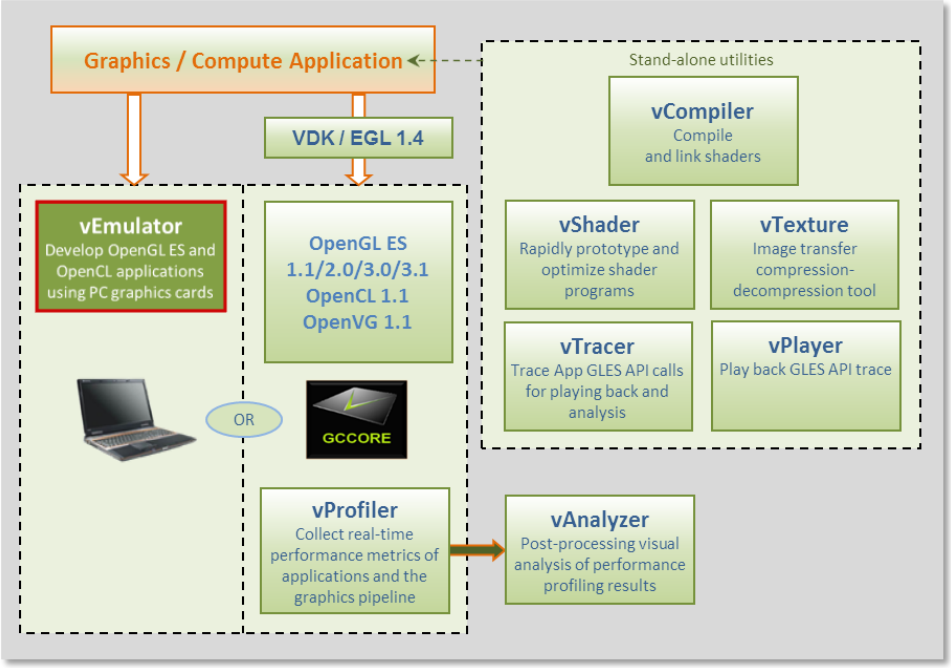


Figure 19 vEmulator embedded graphics emulator

vEmulator is not an application, but rather a set of libraries that convert Khronos mobile API function calls into OpenGL desktop or OpenCL function calls. These libraries can be accessed directly by the graphics / compute application.

12.2.1 Supported operating systems and graphics hardware

vEmulator libraries are available for Microsoft Windows XP, Windows Vista and Windows 7 operating systems:

- Microsoft Windows XP Professional, with Service Pack 2 or later
- Microsoft Windows Vista with Service Pack 2 or later
- Microsoft Windows 7 Professional

vEmulator has been tested on popular graphics cards, including:

- NVIDIA GeForce GTX 200 series with driver version 182.05 or later
- NVIDIA GeForce 9000 and 8000 series with driver version 182.05 or later
- NVIDIA GeForce 8400 GSwithForceWare driver version 176.44 or later
- ATI Radeon HD 3000 and 4000 series with driver version Catalyst 9.1 or later

vEmulator for OpenGL ES 3 has been tested on the nVidia GeForce GT430 card with driver version 310.90. Additional graphics cards to be added as testing is confirmed.

12.2.1.1 Specifying platform mode for Windows OS

vEmulator supports both 32-bit and 64-bit operation on the same host (*from VTK 1.61*). The installation uses the following locations for vEmulator files on Windows platforms:

- C:\Program Files\vivante\vEmulator\x86 (for 32-bit emulation)
- C:\Program Files\vivante\vEmulator\x64 (for 64-bit emulation)
- Start Menu location: All Programs\Vivante\vEmulator\x86 (for 32-bit)
- Start Menu location: All Programs\Vivante\vEmulator\x64 (for 64-bit)

To run samples for 32-bit emulation in the x86 folder, select the platform option **Win32** from the dropdown list box in the toolbar area:

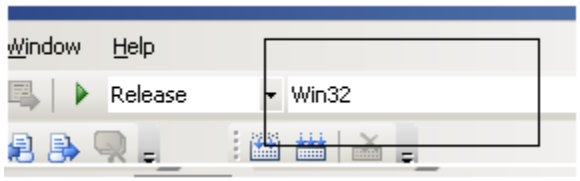


Figure 20 Selecting Win32

To run samples for 64-bit emulation in the x64 folder, select the platform option **x64** from the dropdown list box in the toolbar area:

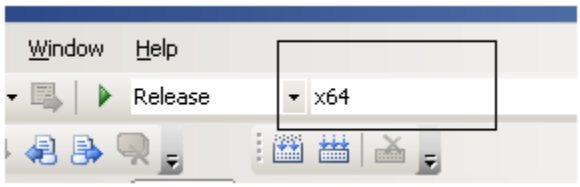


Figure 21 Selecting x64

12.2.2 vEmulator components

vEmulator libraries are packaged with the Vivante VTK installer. Once installed, the libraries resides in a folder vEmulator in the VTK installation path, which can be specified by the user at time of installation. The default location of the Vivante VTK is:

`C:\Program Files\Vivante`

The vEmulator folder contains everything that is needed for emulation. The vEmulator directory structure and its files are described in the following table.

Table 24. vEmulator Directory Contents

vEmulator subdirectory	Filename	Description
bin	libEGL.dll	Dynamic library for invoking EGL at runtime
	libGLESv1_CM.dll	Dynamic library for OpenGL ES 1.1 emulation
	libGLESv2x.dll	Dynamic library for OpenGL ES 2.0 emulation
	libGLESv3.dll	Dynamic library for OpenGL ES 3.0 emulation
	libOpenCL.dll	Dynamic library for OpenCL 1.1 emulation
	libVEmulatorVDK.dll	Dynamic library for vEmulator VDK functions
inc	gc_vdk.h	Vivante VDK declarations
	gc_vdk_types.h	Vivante VDK type declarations
inc/EGL	gc_sdk.h	Vivante SDK declarations and definitions
	egl.h	EGL declarations
	eglext.h	EGL extension declarations
	eglplatform.h	Platform-specific EGL declarations
	eglrename.h	Rename for building static link driver
	eglunname.h	For mixed usage of ES11, ES20
inc/GLES	eglvivante.h	Vivante EGL declarations
	egl.h	EGL declarations
	gl.h	OpenGL 1.1 declarations
	glext.h	OpenGL1.1 extension declarations

	glplatform.h	Platform-specific OpenGL 1.1 declarations
	glrename.h	Rename for building static link driver
inc/GLES2	glunname.h	For mixed usage of ES11, ES20
	gl2.h	OpenGL 2.0 declarations
	gl2ext.h	OpenGL 2.0 extension declarations
	gl2platform.h	Platform-specific OpenGL 2.0 declarations
	gl2rename.h	Rename for building static link driver
inc/GLES3	gl2unname.h	Unified name definitions
	gl3.h	OpenGL 3.0 declarations
	gl3ext.h	OpenGL 3.0 extension declarations
	gl3platform.h	Platform-specific OpenGL 3.0 declarations
inc/hal	gc_hal_eglplatform_type.h	Vivante HAL Platform-specific struct declarations
inc/KHR	khrplatform.h	Platform-specific Khronos declarations
lib samples/es11, /es20	libEGL.lib	Static library for linking EGL functions
	libGLESv1_CM.lib	Static library for linking OpenGL ES 1.1 functions
	libGLESv2x.lib	Static library for linking OpenGL ES 2.0 functions
	libGLESv3x.lib	Static library for linking OpenGL ES 3.0 functions
	libVEmulatorVDK.lib	Static library for linking vEmulator VDK functions
	tutorials.sln	Microsoft Visual Studio® project solution file for samples
samples/es11/tutorial*N*	-- Varies with *N* --	Sample OpenGL ES 1.1 applications
samples/es20/tutorial*N*	-- Varies with *N* --	Sample OpenGL ES 2.0 applications
bin	libEGL.dll	Dynamic library for invoking EGL at runtime

12.2.3 vEmulator for OpenCL

If vEmulator includes support for OpenCL, additional files may be present. For OpenCL emulation using vEmulator on the PC, see the OpenCL emulator readme file (OCL_Readme.txt) in the vEmulator folder for additional installation instruction.

Note: An additional environment variable **CL_ON_GC2100** needs to be set for simulation for GC2100. The value can be any characters, as long as it is not null. This variable does not need to be set for other OCL cores.

Table 25. vEmulator Files for OpenCL 1.1

vEmulator subdirectory	Filename	Description
	OCL_Readme.txt	Readme file for OpenCL 1.1
bin	libOpenCL.dll	Dynamic library for invoking OCL at runtime
inc/CL	cl.h	OpenCL 1.1 core API header file
	cl.hpp	OpenCL 1.1 C++ binding header file
	cl_d3d10.h	OpenCL 1.1KhronosOCL/Direct3D extensions header file
	cl_ext.h	OpenCL 1.1 extensions header file
	cl_gl.h	OpenCL 1.1Khronos OCL/OpenGL extensions header file
	cl_gl_ext.h	OpenCL 1.1Vivante OCL/OpenGL extensions header file
	cl_platform.h	Platform-specific OCL declarations
	opencl.h	Vivante HAL version
lib	libOpenCL.lib	Dynamic library for linking OpenCL functions

samples/cl11	cl_sample.cpp	Sample OpenCL 1.1 source code
samples/cl11	cl_sample.sln	Sample OpenCL 1.1 Visual Studio solution file
samples/cl11	cl_sample.vcproj	Sample OpenCL 1.1 Visual Studio solution project file
samples/cl11	square.cl	Sample OpenCL 1.1 kernel file

12.2.4 Supported extensions

See Section "EGL and OES Extensions Support" for a list of supported and custom extensions available for EGL and OpenGL ES.
Software extensions have not been added to vEmulator for OpenGL ES 2.0. vEmulator relies on the extensions available with the installed version of native OpenGL.

12.2.5 vEmulator environment variable setup

There are two steps to running an OpenGL ES or OpenCL application with vEmulator:

Step 1. Link to the vEmulator **`*.lib`** static libraries at build time when creating an application executable image.

Step 2. Provide a path to the vEmulator **`*.dll`** dynamic libraries during run-time.

These steps require a one-time setup in which the location of the vEmulator libraries is added to the Microsoft Windows system environment variable named "Path." In our example, the following string would be added to the system "Path" variable: C:\Program Files\vivante\vEmulator\lib.

To add vEmulator DLL files to the Windows XP system path:

a. Click **Start** then click **Control Panel** then double-click **System**
   - Vista: then click **Advanced system settings** from the Tasks list in the upper-leftcorner of the window.
   - Windows 7: in the System and Security window, click System, then on the left menu column click **Advanced system settings**.

b. Select the **Advanced** tab, then click on the **Environment Variables…** button.
   - An Environment Variables dialogue box is displayed, with two panes for variables.

c. Select **Path**, and then click on the **Edit…** button.

d. In the **Variable value:** field type the following environment variables in the order they should be found. For instance:

   `C:\Program Files\vivante\vEmulator\lib;<current path>`

   Note: The system parses a path string in left-to-right order when looking for a file. Whatever it finds first is what is used.

e. If the Vivante Core is GC2100, an additional variable **`CL_ON_GC2100`** should be set to any non-null value.

f. Click **OK**.
   - Click **OK** to close the Environment Variables dialogue window.
   - Click **OK** to close the System Properties dialogue window.

- Close the Control Panel > System window.

12.2.6 Sample code overview

In the discussions that follow about the various sample programs included with the vEmulator distribution, we assume that vEmulator has been installed in the default location within the vivante/VTK folder:

C:\Program Files\vivante\vEmulator

Relative to this path:

- run-time dlls are located at **…\bin**
- include-files are found at **…\inc**
- library files are located at **…\lib\<API>**
- examples are located at **…\samples\<API>\tutorial***
  where API is one of: **es11**or **es20**

The code examples are distributed with working *.exe executable images so that the VTK user can see how the results should look.

They are presented in a tutorial fashion, progressing from simpler programs to more complex as the tutorial number increases.

12.2.7 Building and running the code examples

The steps to build and run are identical for all code examples, regardless of the API (es11 or es20). There are two general guidelines to keep in mind.

1. A Visual Studio project has environment variables that allow the specification of additional paths to "include" and "library" files when a source module from that project is being built. The Visual Studio projects that are part of the vEmulator distribution package are configured out-of-the-box for building all of the sample code executables, relative to the location where vEmulator is installed. Specifically the additional paths are set as "$(SolutionDir)..\..\inc" and "$(SolutionDir)..\..\lib".

   If \samples is moved, or if the VTK user begins with the provided projects as templates for developing applications in a directory that is not directly under the \vEmulator installation, then the project path variables must be adjusted accordingly. For example:

   To access these path variables for tutorial1, first launch the tutorials.sln

   - Right-click on **tutorial1**, then select **Properties** (at the bottom of the pop-up menu)
   - Under "Configuration Properties" > "C/C++" > "General", edit the **Additional Include Directories** entry
     - For example, change **..\..\..\inc** to **C:\Program Files\vivante\vEmulator\inc**
   - Under "Configuration Properties" >"Linker" > "General", edit the **Additional Library Directories** entry
     - For example, change **..\..\..\lib** to **C:\Program Files\vivante\vEmulator\lib**
2. Make sure that the system environment variable **PATH** contains a path to the vEmulator DLL files. (See above section on vEmulatorEnvironment Variable Setup, above.) Remember that the path is order-dependent;

whatever the system finds first is used. If there is more than one DLL with the same name, ensure that the path to the desired one is listed first in the **PATH** string.

12.2.8 OpenGL ES 1.1 examples

12.2.8.1 Tutorial1: rotating three-color triangle

Renders a cube centered at the origin with a different color on each face. Flat shading is used. The cube rotates about the vertical axis. The default projection is ORTHO, which can be toggled between ORTHO and PERSPECTIVE by left-clicking in the display window with the mouse or pressing Enter.

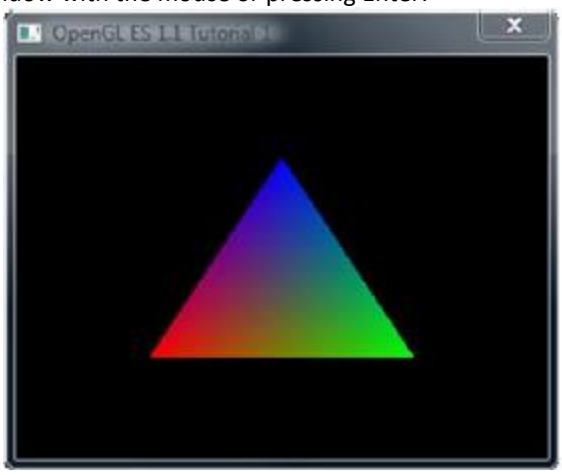


Figure 22 Rotating three-color triangle

12.2.8.2 Tutorial2: rotating six-color cube

Renders a cube centered at the origin with a different color on each face. Flat shading is used. The cube rotates about the vertical axis. The default projection is ORTHO, which can be toggled between ORTHO and PERSPECTIVE by left-clicking in the display window with the mouse or pressing Enter.

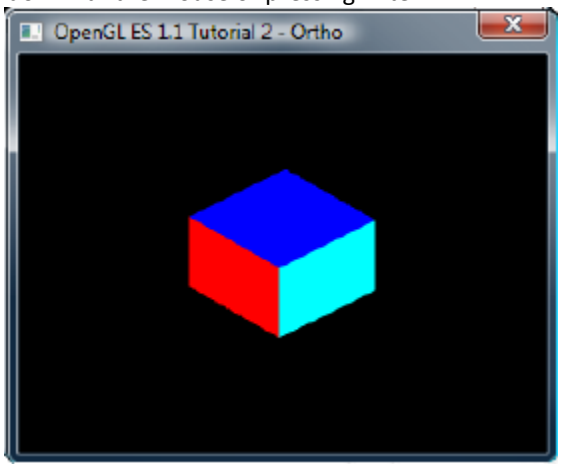


Figure 23 Rotating six-color cube

12.2.8.3 Tutorial3: rotating multi-textured cube

This example takes the cube of the previous example with PERSPECTIVE projection, loads two textures from file and combines them using GL_ADD blending mode, and applies the resulting texture to the cube faces.

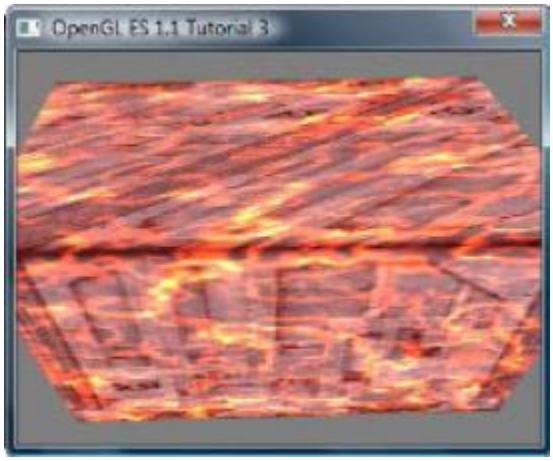


Figure 24 Rotating multi-textured cube

12.2.8.4 Tutorial4: lighting and fog

What appears to be a torus, a cone, and an oblate spheroid orbiting about the center of a plane is actually a single mesh being lit by a single rotating, diffuse light source. Green fog is added to the scene by left-clicking on the display window with the mouse or pressing Enter.

Figure 25 Lighting and fog

12.2.8.5 Tutorial5: blending and bit-mapped fonts

This example makes use of alpha blending to animate sprites across the display, and it also instructs how to create a bit-mapped font from a texture. Jumbled letters iteratively print and move across the display as they unscramble into a text message.

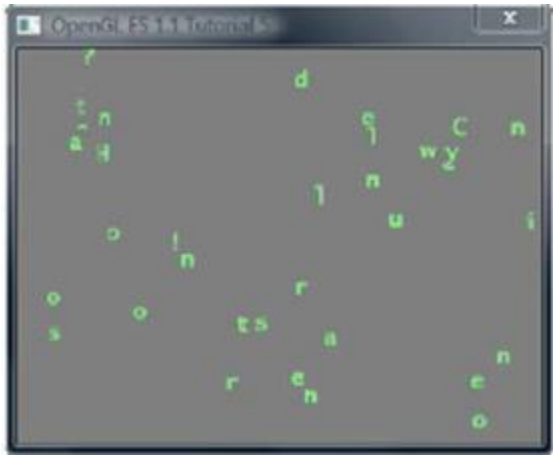


Figure 26 Blending and bit-mapped fonts

12.2.8.6 Tutorial6: particles using point sprites

This example reuses the bit-mapped font technique from the previous tutorial, but it adds a particle generator to simulate and animate particles being emitted from the textured plane. All computation is performed in fixed-point arithmetic.

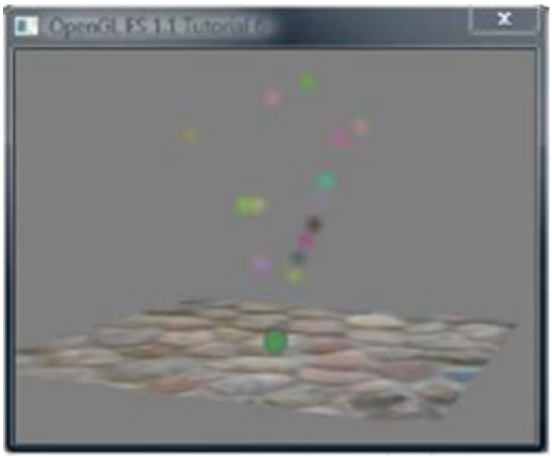


Figure 27 Particles using point sprites

12.2.8.7 Tutorial7: vertex buffer objects

Using Vertex Buffer Objects (VBO) can substantially increase performance by reducing the bandwidth required to transmit geometry data. Information such vertex, normal vector, color, and so on is sent once to locate device video memory and then bound and used as needed, rather than being read from system memory every time. This example illustrates how to create and use vertex buffer objects.

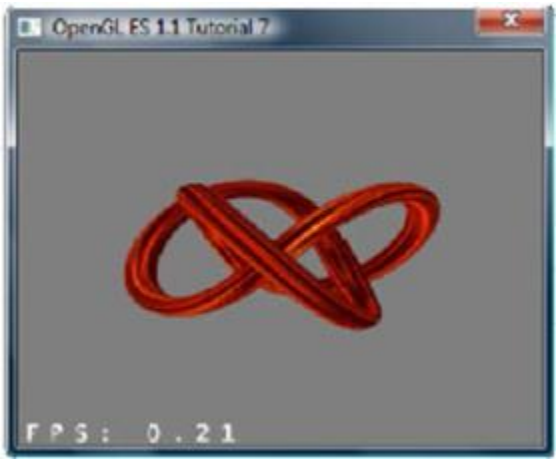


Figure 28 Vertex buffer objects

12.2.9 OpenGL ES 2.0 examples

12.2.9.1 Tutorial1: rotating three-color triangle

A single triangle is rendered with a different color at each vertex, Gouraud shading for blending, rotational animation in the final display. This is the same example as es11/tutorial1, only implemented in OpenGL ES 2.0.

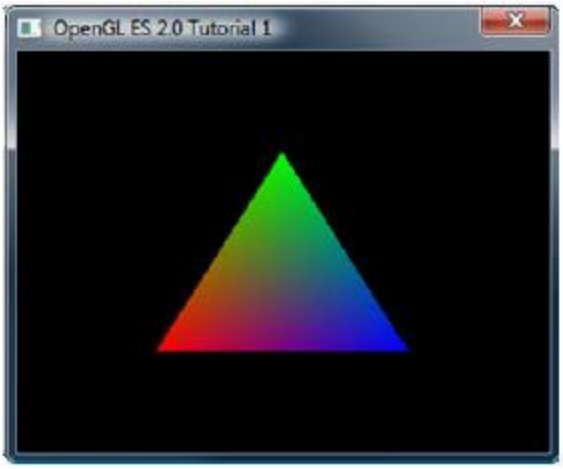


Figure 29 Rotating three-color triangle

12.2.9.2 Tutorial2: rotating six-color cube

Renders a cube centered at the origin with a different color on each face, and rotates it about the vertical axis. Similar to the es11/tutorial2 example, the default projection is ORTHO. But there is no toggle for PERSPECTIVE.

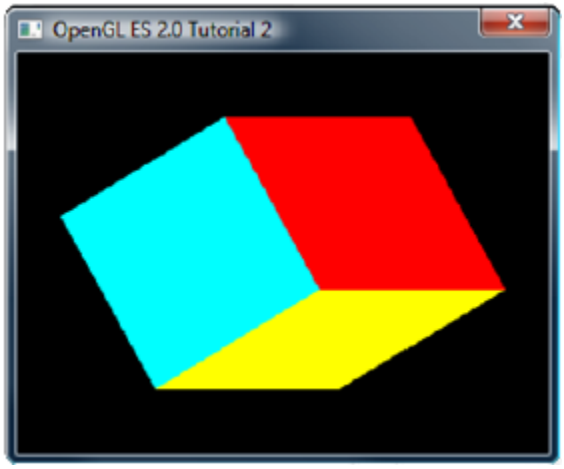


Figure 30 Rotating six-color cube

12.2.9.3 Tutorial3: rotating reflecting ball

A ball made of a mirroring material and centered at the origin spins about its Y-axis and reflects the scene surrounding it.

Note: if the program cannot be executed and print “GL error” in the console, remove the line “return” before the line of “DeleteCubeTexture(cubeTexData);”

Figure 31 Rotating reflecting ball

12.2.9.4 Tutorial4: rotating refracting ball

This example is the same as the previous one, except that the ball is made of clear glass which refracts the surrounding environment.

Note: if the program cannot be executed and print “GL error” in the console, remove the line “return” before the line of “DeleteCubeTexture(cubeTexData);”

Figure 32 Rotating refracting ball

12.3 vShader

vShader is a complete off-line environment for editing, previewing, analyzing, and optimizing shader programs.

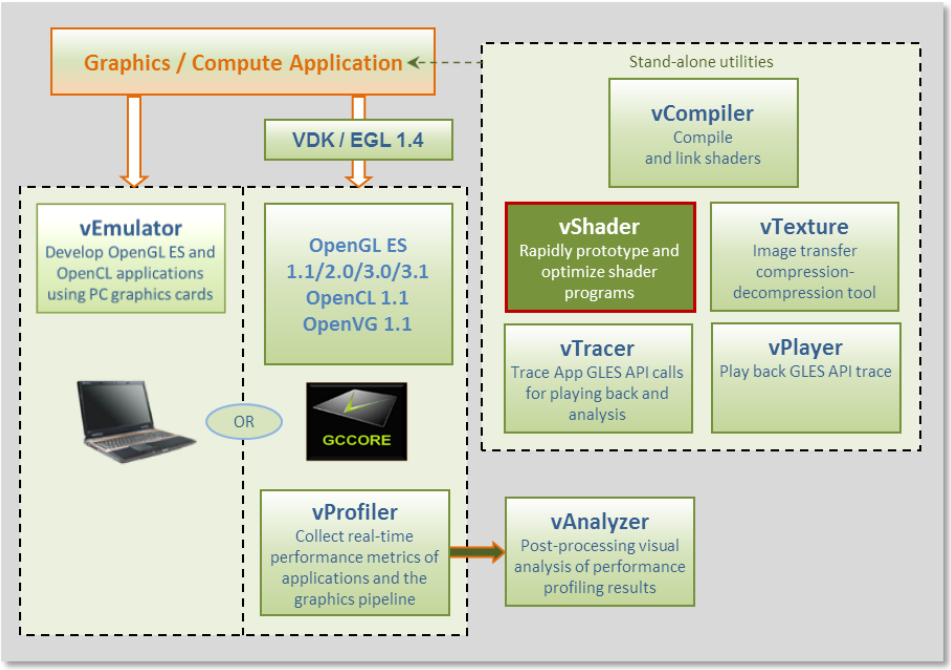


Figure 33 vShader shader editor

vShader allows users to:

- Map any texture onto shaders
- Import user-defined meshes
- Bind mesh attributes to shaders
- Set uniforms in shaders
- View shader compiler output for optimization hints
- Predict hardware performance

12.3.1 vShader components

By default, the vShader executable installs in the following location within the Vivante Toolkit directories:
`C:\Program Files\Vivante\vShade.`
The vShader package includes samples of shader programs, a number of standard meshes (sphere, cube, tea pot, pyramid, etc.) and a text editor. These extra features help programmers get a quick start on creating their shader programs.
By combining vertex shaders and fragment shaders into a single shader program, an application can produce a shader effect. A project can make use of many shader effects, which can share vertex and fragment shaders, mixing and matching to achieve the desired results.
The scope of this guide is to cover the vShader user interface. The tutorials provided with the vShader package are there to help the reader learn about shaders, if needed.

12.3.2 Getting started with vShader

Once the vShader utility is launched by clicking on a shortcut or directly on the executable vShader.exe projects can be created, developed and saved. Project files have an extension **`.vsp`**.

12.3.2.1 Creating a new project

To create a new project, locate the main menu bar: Select **File** then **New Project…**

Depending on the current project status, one of three things happen:

1. If this is the first time vShader is launched, there is no project already open and selecting "File > New Project…" has no effect.
2. If there have been no changes to the current project since the last save, then the current project closes and a new and empty project is opened.
3. If the current project has been modified, then a dialog box appears to ask to save the changes. Choosing **Yes** commits the changes to the current project, which is then closed, and a new, empty project is opened.

12.3.2.2 Opening an existing project

To open an existing project, locate the main menu bar:
To open an existing project, locate the main menu bar:

1. Select **File** then **Open Project…**
2. Double-click on the desired project from the list that pops up, or single-click on the project name and click **OK**.

The project loads into vShader and appear in the state it is last saved.

12.3.2.3 Saving a project

To save a project, locate the main menu bar:

1. Select **File** then **Save Project…**
2. In the resulting dialog box indicate where to save the project, then click **OK.**

12.3.3 vShaderNavigation

The vShader application runs on the Windows XP, Windows Vista and Windows 7 platforms and is driven from a graphical user interface as shown in the figure below.
Main components of the GUI include:

- on upper portion of window: a Menu Bar, Menu Icons,
- on left: Preview pane, Project Explorer pane

- on right: Shader Editor pane
- on lower portion of window: InfoLog pane.

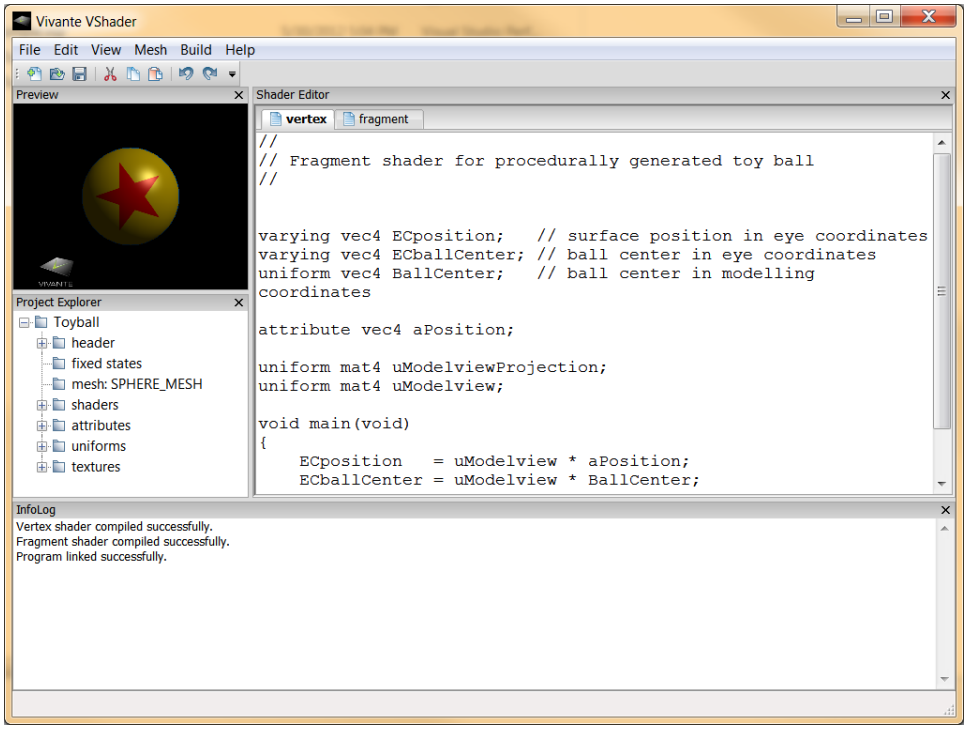


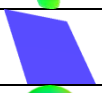
Figure 34 vShader GUI main window

12.3.3.1 vShader menu bar

The main window opens when a user launches vShader. The main menu bar contains drop-down menus for File, Edit, View, Mesh, Build, and Help.

Table 26. vShader menu commands

Menu Name	Menu Command	Description
File		
	New Project…	Create a new project file; if a project is currently open, then the user is prompted to choose whether to save it first.
	Open Project…	Browse for and load a .vsp VShader project.
	Save Project…	Save the current project; if this is the first time saving this project, then the user is prompted to choose where to save it.
	Load Vertex…	Browse for and load a vertex shader from an existing text file.
	Load Fragment…	Browse for and load a fragment shader from an existing text file.
	Save VertexShader As…	Prompts for filename and location to save the active vertex shader.
	Save FragmentShader As…	Prompts for filename and location to save the active fragment shader.
	Exit	Close all open files and exit VShader.

Edit			
	Undo [Ctrl-z]	Revert to a previous edit state (Note: Undo is only 1-level deep)	
	Redo [Ctrl-y]	Re-apply the last “undone” edit command (Note: Redo is only 1-level deep)	
	Cut [Ctrl-x]	Delete the selected item(s) and save a copy in the paste buffer	
	Copy [Ctrl-c]	Save a copy of the selected item(s) item in the paste buffer	
	Paste [Ctrl-v]	Insert the contents of the paste buffer	
	Delete [Del or Bkspc]	Remove the selected item(s)	
	Select All [Ctrl-a]	Highlight all items in the current view	
View			
	Reset Preview	Reset Preview window.	
	Snapshot	Save current preview image to bitmap bmp file. A dialog box is displayed to let user choose where to save the bmp.	
	Perspective	Use perspective projection in the Shader Preview pane	
	Ortho	Use orthographic projection in the Shader Preview pane	
	Tool Bar	Show or hide toolbar icons	
	Preview Window	Show or hide Preview window	
	Project Explorer	Show or hide Project Explorer window	
	Shader Editor	Show or hide Shader Editor window	
	InfoLog	Show or hide InfoLog window	
Mesh			
	Conic	Looks like a spiral horn.	
	Cube	A 3D cube.	
	Klein	The Klein bottle.	
	Plane	A 2D square.	
	Sphere	A ball.	
	Teapot	The Utah teapot.	
	Torus	Looks like a donut.	
	Trefoil	A trefoil knot.	
	Custom Mesh…	Browse for and open a 3DS mesh file.	
Build			
	Compile	Compile the active shader.	
	Link	Link the vertex and fragment shaders into a shader program, and apply it to the mesh showing in the Shader Preview window pane.	
	Clear InfoLog	Remove all text currently showing in the InfoLog window pane.	
Help			

	About	Information about the version of VShader being used.

12.3.3.2 vShader Window OS panes

There are four window panes in the vShader GUI: Preview, Project Explorer, Shader Editor, and InfoLog. Each pane can be resized by left-mouse-dragging the pane edge. A pane can be hidden by clicking the **X** in the upper-right corner of the pane, or by un-checking the box next to its name in the View pull-down of the main menu. Restoring a hidden window pane is done by checking the appropriate box in the View pull-down menu.
Individual panes in the vShader application can be resized, relocated or converted to detached windows, as in the example to the right.
Note: Changes made to pane arrangement are not restored on application or project relaunch.

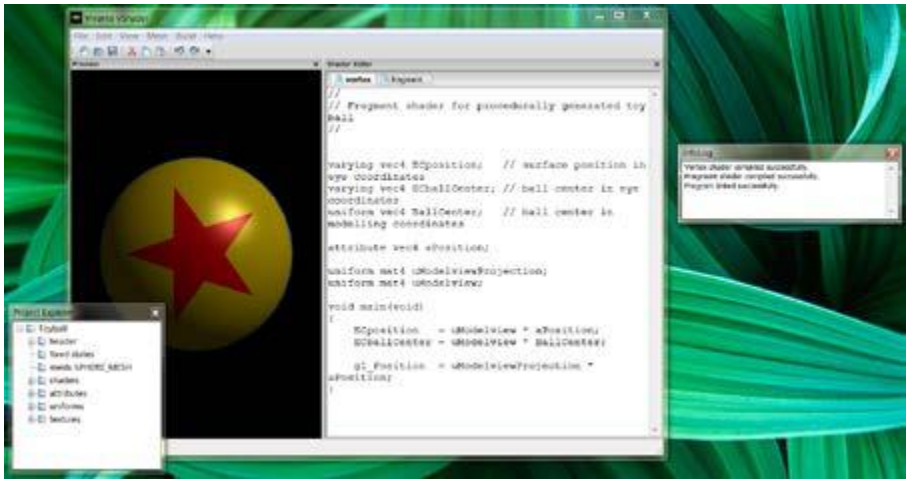


Figure 35 vShader moveable panes

12.3.3.2.1 Preview

The shader Preview pane shows the current effect of the shaders on the chosen mesh geometry. A different mesh may be chosen either via the Mesh pull-down menu in the menu bar near the top of the vShader main window or by right-mouse clicking in the Preview pane.
When using the right-click method, the user also can choose between perspective and orthographic views of the mesh, can reset the view orientation to the default, or can save the current view in the Preview window as a bitmap file by selecting **Snapshot**.
The object in the Preview window can be rotated, translated, and scaled. Rotation is controlled by left-mouse-drag; translation is done by holding the Ctrl key plus left-mouse-drag; scaling the image is seen by holding the Alt key while applying left-mouse-drag.
When shader variables are changed, the shader preview updates automatically. When shader programs are changed they must be recompiled and relinked by the user, through the Build menu. The Preview display is automatically updated to reflect the new Build.

12.3.3.3 Project explorer

The Project Explorer displays all of the project resources in a familiar tree structure. The root of the tree is the project name, and the branches and leaves classify the resources. Folders can be expanded by clicking on the plus sign next to them, and they can be collapsed by choosing the minus sign. By right-mouse clicking on any resource name, the user can view and usually edit that resource.

12.3.3.3.1 Shader editor

The Shader Editor is a work area for entering and modifying shader programs. There are two tabs: one for vertex shader, and one for fragment shader. Changes made to a shader must be compiled and linked in order for their effect to appear in the Shader Preview.
Compiling can be done by selecting **Build** then **Compile** from the main menu bar. Likewise, linking and applying the shaders is performed by choosing **Build** then **Link**.

12.3.3.3.2 Info log

The Info Log window pane receives diagnostic messages from the compiler and linker, so that the user can see if the current shaders have built without errors. This pane can be cleared of text by selecting the **Build** then **Clear InfoLog** entry in the main menu.

12.3.4 vShader project resources

Project resources are accessible from the Project Explorer pane. Click on the item and an Editor pop-up dialog box appears where the user can enter alternate values. Resources include: header, fixed states, mesh, shaders, attributes, uniforms, and textures.

12.3.4.1 Header

Some project identifying information, namely version, author, and company. Expand the folder to see the settings, or right-click (or double-click) the folder to edit them.

Figure 36 Header editor

12.3.4.2 Fixed states

The Fixed State Editor is a list of OpenGL ES 2.0 fixed states settings, such as depth test enable/disable, etc. It allows the user to set all fixed states manually. Right-click or double click to display an edit dialog.

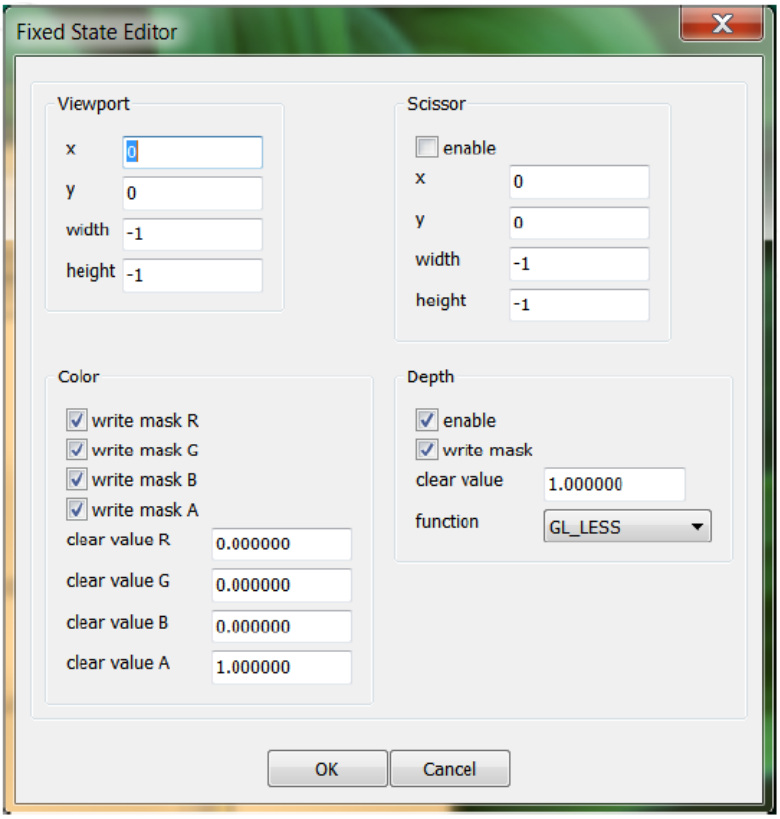


Figure 37 Fixed states

12.3.4.3 Mesh

This resource shows the name of the mesh which is currently being displayed in the Preview pane. It does not have a pop-up window. Right-click on the mesh name to select a different mesh can be selected from the resulting pull-down menu.

12.3.4.4 Shaders

Left-click on the plus sign next to the “shaders” folder to reveal the two sub nodes in this section, which are vertex and fragment. Double-click (or right-click and then choose **Active**) on either shader to bring it forward in the Shader Editor for editing.

12.3.4.5 Attributes

The Attribute Editor dialog displays all attributes bound to the current project. It allows the user to add new attributes, and edit or remove existing attributes. Right-click **Attributes** to add a new one. Click on the plus sign to expand the attributes list, and then double-click to edit a particular attribute. Also, by right-clicking on an attribute, the user can edit or remove that attribute or add a new one. Up to 12 attributes are allowed.

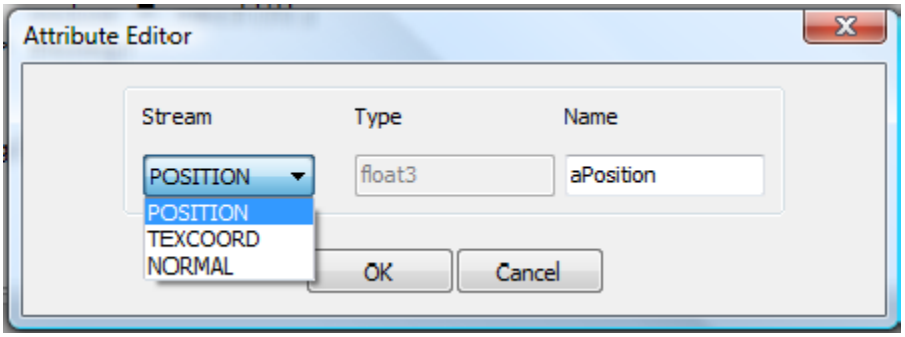


Figure 38 Attributes

12.3.4.6 Uniforms

This displays all uniforms bound to the current project. Right click on **Uniforms** to add a new one, or expand the list and double-click on a given uniform to bring up the Uniform Editor dialog. When a uniform is right-clicked, the user can add new uniforms, or edit or remove existing uniforms. Up to 160 uniforms are allowed.

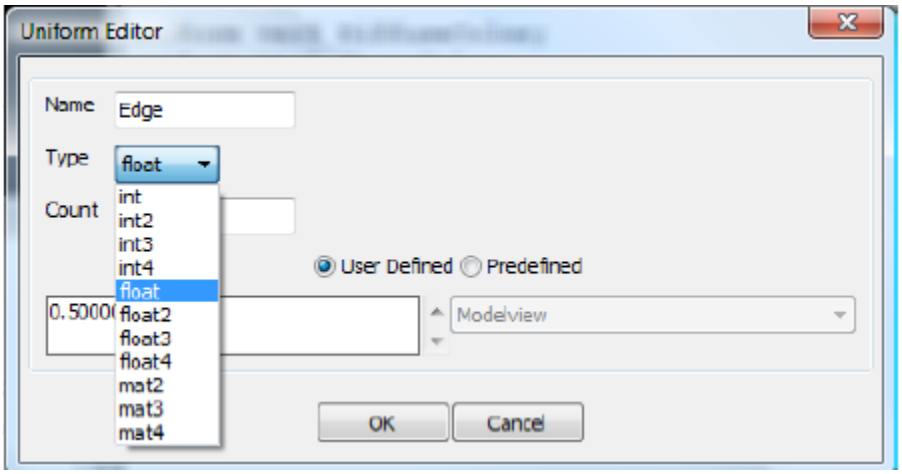


Figure 39 Uniforms

12.3.4.7 Textures

The Texture Editor dialog allows the user to select a texture for each of up to 8 texture units. The effect of applying each texture is shown immediately in the Shader Preview pane.
The texture selection option list is created from the texture files located in the “textures” subfolder of the project. The list can be expanded by adding textures to the textures folder, formatted as bitmap files.

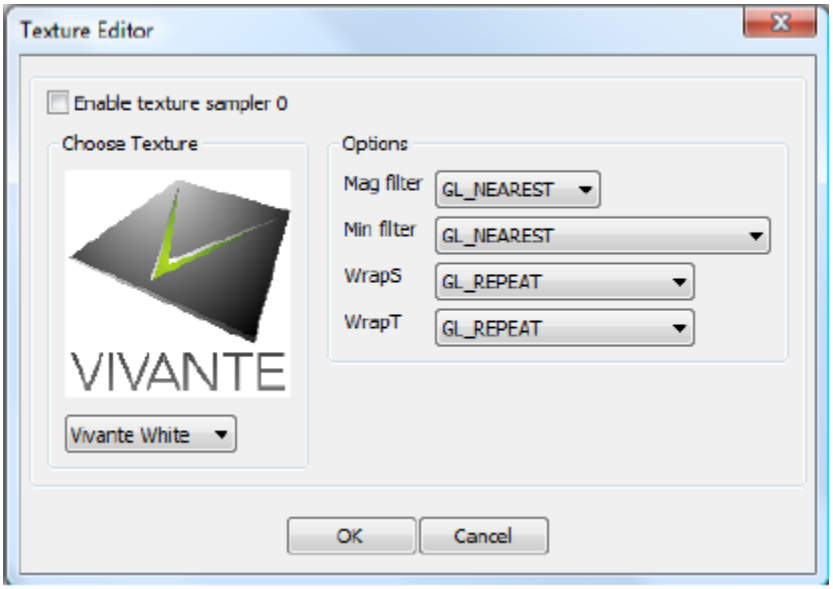


Figure 40 Textures

12.4 vCompiler

vCompiler is an off-line compiler and linker for translating vertex and fragment shaders written in OpenGL ES Shading Language (ESSL) into binary executables targeting Vivante accelerated hardware platforms. vCompiler is driven by a simple command-line interface.

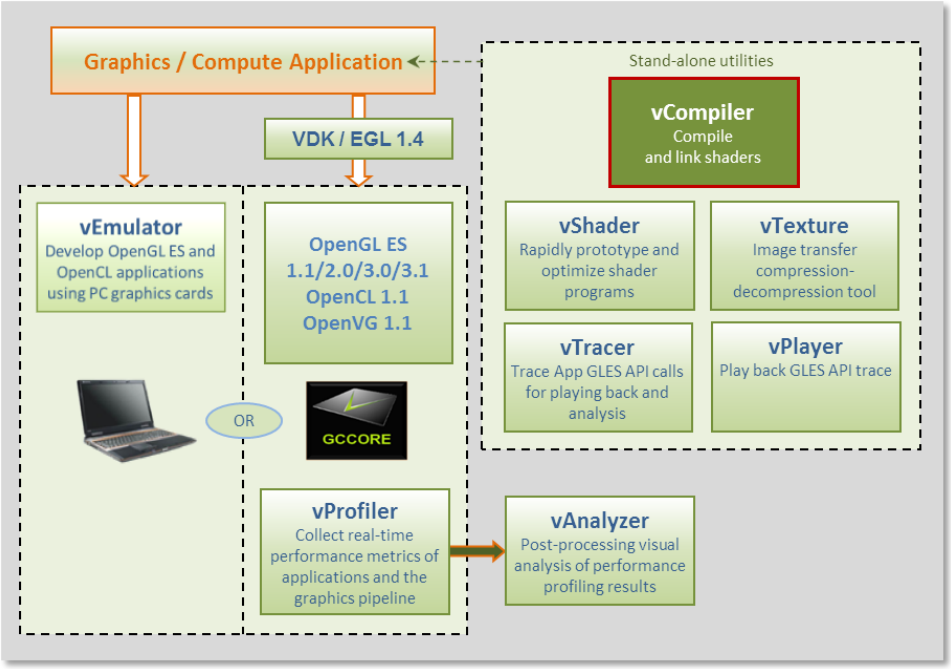


Figure 41 vCompiler compiler/linker

12.4.1 vCompiler command line syntax

12.4.1.1 Syntax:

Optional inputs are indicated by italic font.

```
vCompiler [-c] [-h] [-l] [-On] [-v] [-x <shaderType>] [-o <outputFileName>]
                <shaderInputFileName> <shaderInputFileName_2>
```

12.4.1.2 Input parameters (required):

shaderInputFileName shader input file name, which **must** contain one of the following file extensions:

vert	vertex shader source file
frag	fragment shader source file
vgcSL	previously compiled vertex shader input/output file
pgcSL	previously compiled pixel shader input/output file

12.4.1.3 Input parameters (optional):

shaderInputFileName_2 up to two shader files can be specified. The second shader file is optional but must have one of the file extensions described above for shader InputFileName. If the first shader is a vertex shader, this second shader should be a fragment shader; conversely if the first shader is a fragment shader, the second should be a pixel shader.

Note: pre-compiled and compiled shaders may be mixed, as long as one is a vertex shader and the other a fragment shader.

-c Compile each vertex .vert file into a **vgcSL** file and/or fragment shader .frag file into a **pgcSL** only, with no merged result file of type **.gcPGM**.
If the **–c** option is not specified:
a)When only one shader is specified, that shader is compiled into a .[v/p]gcSL file.
b) When two shaders are specified, one is assumed to be a vertex shader and the other a fragment shader. Each shader can be either a previously compiled .vgcSL or .pgcSL. file or a .vert or .frag still to be compiled. The two are merged into a .gcPGM file after successful compilation.

-f <gpuConfigurationFile> Specifies a configuration file (*from VTK 1.6.2*). If **–f** is not specified, the file **viv_gpu.config** in the vCompiler working directory is used as the default configuration file. Example syntax:

```
vCompiler –f viv_gpu_880.config foo.vert bar.frag
```

Note: vCompiler does not work correctly if the GPU configuration file cannot be found or contains incorrect content. See Section on vCompiler Core-specific configuration **for .config file content organization.**

-h Shows a help message on all the command options.

-l

Create a log file. The log file name is created by taking the first input file name, then replacing its file extension with ".log". If the input file name does not have a file extension, .log is appended, e.g.,

```
myvert.vert        => myvert.log
inputfrag          => inputfrag.log
```

-o <outputFileName>

Specify the output file name. If the path is other than the current directory, it must also be specified. Any extension can be specified. If the extension is not specified, the following are **outputFileName** supported default types:

- vgcSL — compiled vertex shader output file, usually compiled from a .vert input source file (default result for single file compile)
- pgcSL — compiled pixel shader output file, usually compiled from a .frag source input file.
- gcPGM — compiled file merging vertex shader and fragment/pixel shader into a single output file

-O*n*

Optimization level. Default is **–O1:**

- **-O0** — Disable optimizations
- **-O1- -O9** — Indicates on which level optimization should be done. The default is level 1. Note: Optimization is actually implemented in the compiler, not vCompiler.

-s

Deprecated from 5.0.11_p5; instead, use file **viv_gpu.config** in the vCompiler work directory contains GPU core-specific configuration detail.

-v

Verbose; prints compiler version and diagnostic messages to STDOUT.

-x<shaderType>

Explicitly specifies the type of shader instead of relying on the file extension. This option applies to all following input files until the next **-x** option.

ShaderType: supported values for Shader type include:

- **vert** — vertex shader source file
- **frag** — fragment shader source file
- **vgcSL** — compiled vertex shader input/output file
- **pgcSL** — compiled pixel shader input/output file

-x none

revert back to recognizing shader type according to the file name extension.

12.4.1.4 vCompiler output

Output files are placed in the current directory, unless another directory is specified with the –o option. The files can be of the three types described above under outputFileName value of the –o option.

12.4.1.5 vCompiler syntax examples

vCompiler foo.vert	produces foo.vgcSL
vCompiler bar.frag	produces bar.pgcSL
vCompiler foo.vert bar.frag	produces foo.gcPGM
vCompiler -v -l -O1 foo.vert bar.frag	produces foo.gcPMG and foo.log
vCompiler -v -l -O1 -o foo_bar foo.vert bar.frag	produces foo_bar.gcPGM and foo_bar.log

12.4.2 vCompiler core-specific configuration

To ensure the shader binaries generated by vCompiler work correctly and optimally on the specified GPU, specify the GPU before starting to run vCompiler.

There are two or more configuration files *(available in VTK 1.6.1)* in the vCompiler installation directory. For example:

`viv_gpu.config`	configuration file for GC2000-5108a (default)
viv_gpu_880.config	configuration file for GC880-5106

To change the GPU configuration, rename the GPU file to **`viv_gpu.config`**. For example, on a Linux OS platform, use the following commands:

```
mv viv_gpu.config viv_gpu_2100.config
mv viv_gpu_880.config viv_gpu.config
```

Keep in mind that the content of these files should not be modified, and the **`viv_gpu.config`** file must be in the vCompiler work directory. If customization is required, note that the format for the file contents is fixed and only the value for each parameter may be changed.

Here is the default **viv_gpu.config** file:

```
chipModel              = 0x2000;
chipRevision           = 0x5108;
chipFeatures           = 0xE0296CAD;
chipMinorFeatures      = 0xC9799EFF;
chipMinorFeatures1     = 0x2EFBF2D9;
chipMinorFeatures2     = 0x00000000;
chipMinorFeatures3     = 0x00000000;
chipMinorFeatures4     = 0x00000000;
chipMinorFeatures5     = 0x00000000;
chipMinorFeatures6     = 0x00000000;
pixelPipes             = 2;
streamCount            = 8;
registerMax            = 64;
threadCount            = 1024;
shaderCoreCount        = 4;
vertexCacheSize        = 16;
vertexOutputBufferSize = 512;
instructionCount       = 512;
numConstants           = 168;
bufferSize             = 0;
varyingsCount          = 11;
superTileMode          = 1;
```

12.5 vTexture

The Vivante vTexture tool is a command line tool which provides compression and decompression functions to help developers transfer image formats.

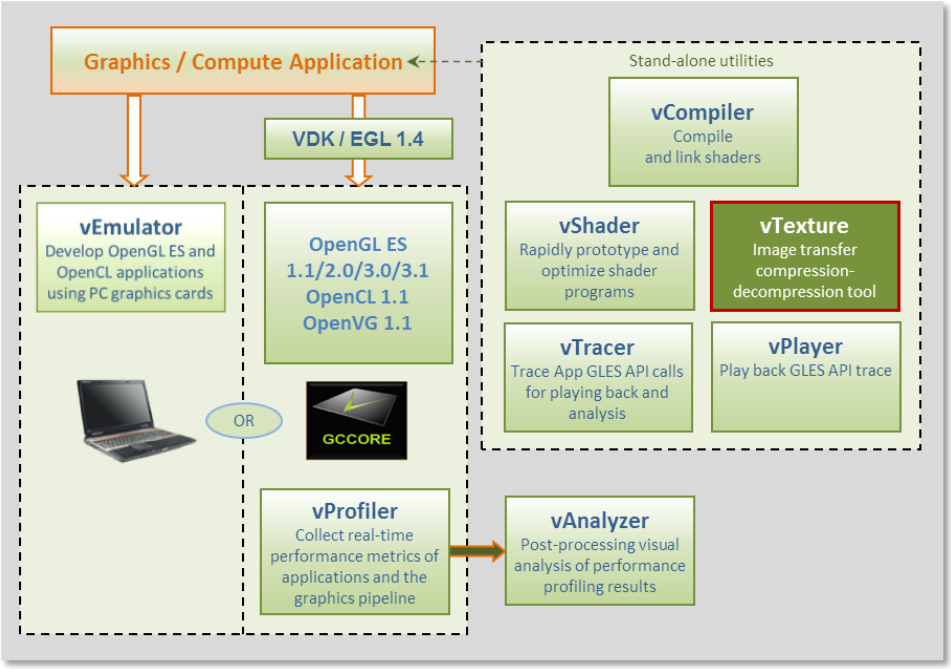


Figure 42 vTexture Image Transfer Tool

12.5.1 Formats

12.5.1.1 Supported formats

The vTexture tool supports:

- compression of uncompressed TGA format files to any of the following formats:
  - DXT1
  - DXT3
  - DXT5
  - ETC1
  - ETC2
- decompression to uncompressed TGA format of the following compressed format file types:
  - DXT1
  - DXT3
  - DXT5
  - ETC1
  - ETC2

The compressed DXTn format image file is stored as a DDS file, and the ETCn format image is stored as a PKM or KTX file.

The TGA format either the RGBA or RGB color model and ETCn format provides an image following the RGB color model RGB888. Note that compressing a TGA image of RGBA format to an ETCn format results in a loss of alpha values.

12.5.1.2 Supported formats for tile and de-tile conversions

vTexture supports conversions between linear textures and the tile configurations supported in Vivante hardware:

- Linear — no tiling
- Tile — 4x4 tile
- Supertile — 64x64 tile

The following two tile configurations are supported by some hardware, but not routinely utilized in Vivante software:

- Multi-tile — A split-tile (possible, but rarely used).
- Multi-supertile — A split or multi-supertile surface can occur with GC2000 and above, where, each pixel engine of the multi-pipe renders into a different render buffer and each render buffer is supertiled.

Formats supported for tile format conversions include the following:

- source data
  - BMP
  - TGA
- output data
  - BMP
  - raw data of a specified type. Supported formats are: RGBA8888 BGRA8888 RGB888 BGR888 RGB565 BGR565 ARGB1555

12.5.1.3 vTexture output formats

Output from the compress option:

- DXTn format image file is stored as a DDS file,
- ETC1 and ETC2 format images is stored as a PKM or KTM file.

Output from the decompress option:

- all supported formats are decompressed to an uncompressed TGA file.

Output from tile / de-tile options:

- BMP if `-r` not specified
- RAW if `-r` specified.

12.5.1.4 vTexture RAW output file format definition

The Vivante vTexture Tools RAW file is a Vivante-defined file. The file extension is .RAW.
The format consists of the following:

Table 27. Vivante RAW file header and pixel data definition

Vivante RAW File Header and Pixel Data Definition	Size 16 bytes	Data type	Detail
Width in pixels	4 bytes	INT	Number of pixels
Height in pixels	4 bytes	INT	Number of pixels
Pixel format	4 bytes	INT	Integer value of numeric for a supported format, as defined in gceSURF_FORMAT enumeration: **Supported Format** / **Numeric** ARGB_1555 / 208 RGB_565 / 209 BGR_888 / 210

			<table><tr><td>BGRX_8888</td><td>211</td></tr><tr><td>BGRA_8888</td><td>212</td></tr><tr><td>BGR_565</td><td>302</td></tr><tr><td>RGB_888</td><td>303</td></tr><tr><td>RGBX_8888</td><td>305</td></tr><tr><td>RGBA_8888</td><td>306</td></tr></table>These value can also be found in samples(Named TiledTexture).
Tile format	1 byte	BOOL	bit 0: tile bit 1 supertile bit 5: flag for multi- other bits reserved
Supertile format	1 byte	INT	Integer value: 0 = supertile layout mode 0 1 = supertile layout mode 1 2 = supertile layout mode 2
Reserved	2 bytes		not used

12.5.2 Set vTexture environment variable

The following table summarizes the only environment variable that vTexture currently expects.

Table 28. vTexture Environment Variables

Environment Variable	Description
PATH	set PATH=%PATH%;"C:\Program Files\Vivante\vTexture\"

12.5.3 Command line syntax

Open a Command prompt.
Navigate to the folder which contains the vTexture files (for example, **C:\Program Files (x86)\Vivante\vTexture)**.
Launch the **vTexture** or **vTextureTools** application using the command line syntax described below.

12.5.4 Syntax

The usage of the command line tool is as follows for compression/decompression:

```
vTextureTools -c TYPE [-s SPEED] –src FILE [–dest FILE]
```

or

```
vTextureTools -d TYPE –src FILE [–dest FILE]
```

The usage of the command line tool is as follows for tiling/de-tiling:

```
vTextureTools -t|-st [-2 [–r|--raw=FORMAT] –m LAYOUT] –src FILE [–dest FILE]
```

or

```
vTextureTools -dt -t|-st [-2 [–r|--raw=FORMAT] –m LAYOUT] –src FILE [–dest FILE]
```

12.5.4.1 General Parameters

General parameters:

-h show help

-src [FILE] source file - input image path and filename.
Note:
for option **–c** compress, the application expects an input filename with a .TGA extension;
for **–d** decompression the application expects .DDS, .KTX or .PKM ;

for **-t** tile the application expects .BMP or .TGA;
for **-dt** detile the application expects .BMP or .TGA

-dest [FILE] destination file - image path and filename.
Note: the application expects a filename with a .TGA, .DDS, .KTX or .PKM extension for compress/uncompress or .BMP or .RAW for tile/detile.
If the **-dest** parameter is not set, vTexture automatically generates a name for the newly generated file, using the source file name as the prefix appending critical parameters and file type information.

12.5.4.2 Compression/Decompression parameters

These parameters are used for compression and decompression:

-c compress a source image of format uncompressed TGA
[TYPE] specify the target output compression format:
- **-DXT1** compress image to DXT1 format (default format).
- **-DXT3** compress image to DXT3 format.
- **-DXT5** compress image to DXT5 format.
- **-ETC1** compress image to ETC1 format.
- **-ETC2** compress image to ETC2 format .

-d decompress a source image of format specified by the value **[TYPE]**.
The resulting file type is uncompressed TGA.
This option decompresses DXT1, DXT3, DXT5, ECT1 or ETC2 format image to TGA format.
Note: [TYPE] supported tga. namely, we can only use -d tga

-s compression **[SPEED]** mode for ETCn images:
slow
medium
fast (default)

12.5.4.3 Tile/De-Tile parameters

These parameters are used for tiling and de-tiling between linear and tiled formats:

-t Convert linear data to tiled texture output

-st Enable supertile format. This option is an alternate to **-t**. If **-st** and **-t** are used together, -st is set.

-dt De-tile: Convert tiled texture to linear texture output

-2 Tile/de-tile in multi- format. Tile format is multi-tiled (when used with **-t**) or multi-supertiled (with **-st**).

-m **[LAYOUT]**: layout mode for supertiled or multi-supertiled textures:
0: Legacy supertile mode (default).
1: Supertile mode when hardware has HZ.
2: Supertile mode when hardware has NEW_HZ or FAST_MSAA.

-r Specify output data as raw pixel output instead of BMP.
Use: **--raw=rgb565** to specify raw pixel **[FORMAT]**. Supported raw formats (7) are:

rgba8888, bgra8888, rgb888, bgr888, rgb565, bgr565, argb1555.

12.5.4.4 vTexture syntax examples

COMPRESS:

```
vTextureTools -c dxt1 -src d:\myfile.tga -dest c:\compress.dds
vTextureTools -c etc2 -src d:\myfile.tga -dest c:\compress.pkm
vTextureTools -c etc2 -src d:\myfile.tga -dest c:\compress.ktx
vTextureTools -c etc1 -s slow -src d:\myfile.tga -dest c:\compress.pkm
vTextureTools -c etc2 -s slow -src d:\myfile.tga -dest c:\compress.ktx
```

DECOMPRESS:

```
vTextureTools -d etc1–srcC:/vtexin/myfile2.pkm –dest C:/vtextout/myfile2.tga
vTextureTools -d –srcC:/vtexin/myfile3.dds –dest C:/vtextout/myfile3.tga (assumes DXT1)
vTextureTools -d tga -src d:\myfile.dds -dest c:\decompress.tga
vTextureTools–dtga -src d:\myfile.ktx -dest c:\decompress.tga
```

TILE: LINEAR TO TILE CONVERSION:

Tile linear texture to standard tile texture

```
vTextureTools.exe -t -src 123.bmp
```

Tile linear texture to multi-tiled texture

```
vTextureTools.exe -t -2 -src 123.bmp
```

Tile linear texture to supertiled texture

```
vTextureTools.exe -st -src 123.bmp
```

Tile linear texture to multi-supertiled texture

```
vTextureTools.exe -2 –st-src 123.bmp
```

Tile linear texture to multi-supertiled texture and output rgb565

```
vTextureTools.exe -2 --raw=rgb565 -src 123.bmp
```

Tile linear texture to multi-supertiled texture with layout mode 2

```
vTextureTools.exe -st -2 -m 2 -src 123.bmp
```

DE-TILE: TILED TO LINEAR CONVERSION:

De-tile tiled texture to linear texture

```
vTextureTools.exe –dt -t -src 123-tiled.bmp
```

De-tile supertiled texture to linear texture

```
vTextureTools.exe -dt -st -src 123-supertiled.bmp
```

De-tile multi-supertiled texture to linear texture

```
vTextureTools.exe –dt -t -2 -src 123-tiled-multi-tiled.bmp
```

De-tile multi-Super-tiled texture with layout mode 2 to linear texture

```
vTextureTools.exe -dt -st -2 -m 2 -src 123-multi-supertiled-2.bmp
```

12.6 vProfiler and vAnalyzer

vProfiler is a run-time environment for collecting performance statistics of an application and the graphics pipeline. vAnalyzer is a utility for graphically displaying the data gathered by vProfiler and aiding in visual analysis of graphics performance. Used together, these tools can assist software developers in optimizing application performance on Vivante enabled platforms. The GPU includes performance counters that track a variety of GPU functions. vProfiler gathers data from these counters during runtime and can track data for a range of frames or a

single frame from any application. Appendix A contains a partial list of the data gathered by the hardware performance counters. Additional counters are present in the software drivers and hardware access layer.

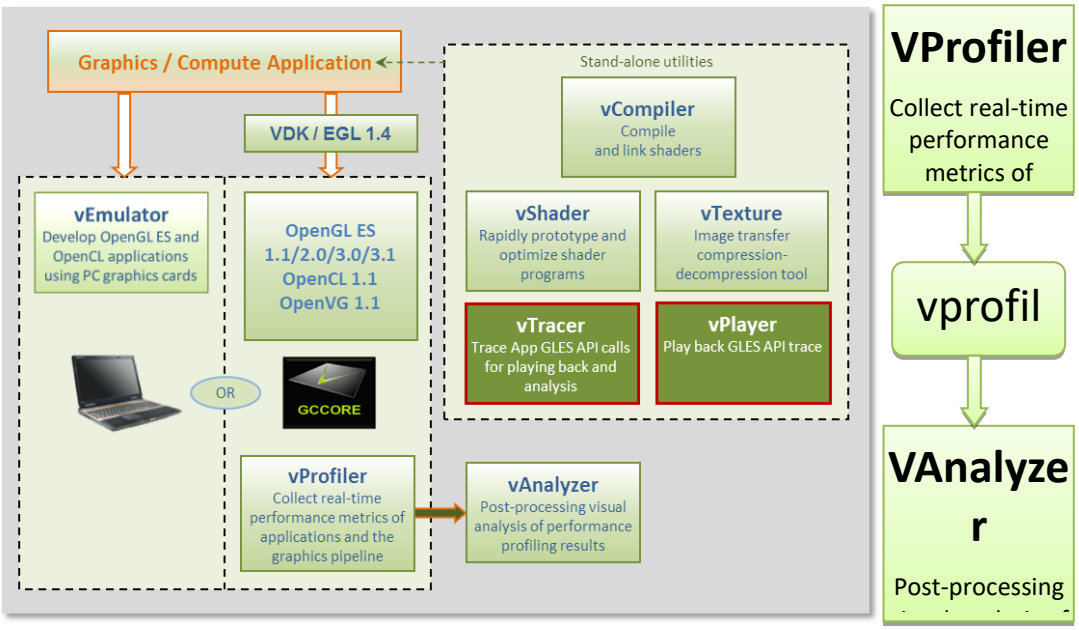


Figure 43 vProfiler performance profiling save data for review in the vAnalyzer visual analyzer

12.6.1 Fundamentals of performance optimization

Whenever an application runs on a computer, it makes use of one or more of the available resources. These compute resources include the CPU, the graphics processor, caches and memory, hard disks, and possibly even the network. Viewed simplistically, it is always true that one of these resources is the limiting factor in how quickly the application can finish its tasks. This limiting resource is the performance bottleneck. Remove this bottleneck, and application performance should be improved. Note, however, that removing one limiting factor always promotes something else to become the new performance bottleneck.
The goal of optimizing, or tuning, application performance is to balance the use of resources so that none of them holds back the application more than any of the others. In practice there is no single, simple way to tune an application. The whole system needs to be considered, including the size and speed of individual components as well as interactions and dependencies among components.

vProfiler collects information on GPU usage and on calls to Vivante functions within the graphics pipeline. As such it provides an excellent view into what is happening on the GCCORE graphics processor at any point in time, down to the individual frame. When the application performance is GPU-bound, vProfiler and vAnalyzer are the right tools to help determine why.

Note that the initial determination regarding which component of the computer system is the performance bottleneck—CPU, GPU, memory, etc.—is the domain of system performance analyzers and is outside the scope of the GPU tools. A list of such performance analysis tools can be found at Wikipedia: en.wikipedia.org/wiki/List_of_performance_analysis_tools.

12.6.2 vProfiler setup for the Linux OS

The VTK Windows OS package includes vAnalyzer for the Windows OS environment. The vProfiler tool can be compiled for the Linux OS, as per the instructions below.
vProfiler stores software and hardware counters captured per frame in the **`vprofiler.vpd`** file. vAnalyzer reads the .vpd file and allows the user to browse all counters, visualize application performance bottlenecks, and measure system utilization of that application run. Presently, vProfiler does not store frame buffer images due to excessive overhead that changes the behavior of applications.

12.6.2.1 Enable vProfiler option in kernel

When building Vivante Graphics Drivers in a Linux OS environment, the driver is built with vProfiler capability.

To activate vProfiler functionality, build the drivers per the instructions in Section "How to Build the GCCORE Drivers for the Linux OS" in the Vivante Driver Development Guide.In Step 3 of the subsection "Run on the target board" where **`insmod`** is used to insert the GAL kernel driver, use the command line to add the **`gpuProfiler=1`** option, or add the option into an existing **`.sh`** script similar to the following:

```
#!/system/bin/sh
#
insmod /system/lib/modules/galcore.ko gpuProfiler=1 [OPTIONS]
chmod 777 /dev/graphics/*
```

12.6.2.2 Enable vProfiler option in U-Boot

vProfiler can also be enabled from U-Boot with kernel command parameters. Minimum Linux kernel version 3.10.y needs to support this galcore.powerManagement=0 galcore.showArgs=1 galcore.gpuProfiler=1.

12.6.2.3 Set vProfiler environment variables

The following table summarizes the environment variables that vProfiler supports. (Note that environment variable names for the Linux OS were changed from driver releases 4.6.9.p13 and 5.0.7 and toolkit release 1.5.)

Table 29. vProfiler Environment Variables for the Linux OS

Environment Variable	Description
VIV_PROFILE	[0] Disable vProfiler (default), [1] Enable vProfiler,[2] Control via application call, [3] Allows control over which frames to profile with vProfiler
VP_OUTPUT	Specify the output file name of vProfiler (default is vprofiler.vpd)
VP_FRAME_NUM	When VIV_PROFILE=1, specify the number of frames dumped by vProfiler.
VP_FRAME_START	When VIV_PROFILE=3, specify the frame to start profiling with vProfiler.
VP_FRAME_END	When VIV_PROFILE=3, specify the frame to end profiling with vProfiler.
VP_SYNC_MODE	Enable [1] or disable [0] the synchronous mode of vProfiler (default is synchronous enabled)

12.6.2.3.1 VIV_PROFILE

The environment variable **VIV_PROFILE** can be used to control enable /disable and set profiling modes for vProfiler.

VIV_PROFILE=0:

By default, vProfiler is disabled in the driver. If vProfiler has been enabled and to disable it,set **VIV_PROFILE** equal to 0:

```
export VIV_PROFILE=0
```

VIV_PROFILE=1:

To enable vProfiler, set **VIV_PROFILE** to 1:

```
export VIV_PROFILE=1
```

To limit the number of frames to analyze, use the environment variable **VP_FRAME_NUM**. (This option is available only when **VIV_PROFILE=1**.) For example, this example setting makes vProfiler dump performance data for the first 100 frames.

```
export VP_FRAME_NUM=100
```

VIV_PROFILE=2:

Mode **VIV_PROFILE=2** *(available from VTK 1.5.7)* provides support for `glEnable(GL_PROFILE_VIV)` and `glDisable(GL_PROFILE_VIV)`, which are used to choose which frames are to be profiled. In this mode, vProfiler is disabled by default. It begins to do profiling only after a glEnable(GL_PROFILE_VIV) call from the application. And it stops profiling when glDisable(GL_PROFILE_VIV) is called. Note that the flag is only checked at every frame end, i.e., in eglSwapBuffers. To use this mode, set **VIV_PROFILE** to 2:

```
export VIV_PROFILE=2
```

VIV_PROFILE=3:

Setting **VIV_PROFILE** to 3 *(available from VTK 1.5.8)* provides support for two environment variables **VP_FRAME_START** and **VP_FRAME_END**, which are used to choose which frames are to be profiled. In this mode, vProfiler is disabled by default. It begins to do profiling starting at the frame number specified by **VP_FRAME_START**, and it ends the profiling after the frame number specified by **VP_FRAME_END**. For example to use this mode, set **VIV_PROFILE** to 3:

export VIV_PROFILE=3
export VP_FRAME_START=10
export VP_FRAME_END=90

NOTE: The GPU profiling mode requires the GPU Power Management (PM) functions to be disabled to get the precise profiling data. When kernel module “galcore” is inserted with **gpuProfiler=1**, the PM functions in the driver are not disabled. The PM functions are disabled when **VIV_PROFILE** is set to 1, 2, or 3, and the application starts. The PM functions are enabled when **VIV_PROFILE** is set to 0, and the application starts again.

12.6.2.3.2 VP_OUTPUT

The output file of vProfiler is **`vprofiler.vpd`** by default. To specify an alternate filename use the environment variable **VP_OUTPUT**. For example,

```
exportVP_OUTPUT =sample.vpd
```

12.6.2.3.3 VP_SYNC_MODE

To get accurate values from the GPU counters, vProfiler needs to commit the GPU commands at the end of every frame; this is so-called synchronous mode. The environment variable **VP_SYNC_MODE** can be used to enable or disable synchronous mode. By default, vProfiler works in synchronous mode. The command below makes vProfiler work in asynchronous mode.

```
export VP_SYNC_MODE=0
```

12.6.3 vProfiler setup for the Android platform

The vProfiler tool can be set up for use with the Android platform, as per the instructions below.

12.6.3.1 Enable vProfiler option in kernel

When building Vivante Graphics Drivers in an Android environment, build the drivers per the instructions in the Vivante Driver Development Guide section entitled "How to Build the GCCORE Drivers for Android Platform." In Step 2 of the subsection "Run on the Target board", use the provided **install-recovery.sh** script or add the **gpuProfiler=1** option into the existing **.sh** script similar to the following:

```
#!/system/bin/sh
#
insmod /system/lib/modules/galcore.ko gpuProfiler=1 [OPTIONS]
chmod 777 /dev/graphics/*
```

Put the **install-recovery.sh** file in the target Android system's **/system/etc/** folder. Continue following the instructions in the Vivante Driver Development Guide or the readme guide in the driver source package.

Use **adb push** to migrate the drivers to the target system, and then reboot the target Android system.

> NOTE: If using an **install-recovery.sh** script as described above, and cannot reboot the Android platform successfully, there may be a problem with file access permissions. Workaround: run **adb shell.** Go to **/system/etc/**, then run the command **chmod 777 install-recovery.sh**.

12.6.3.2 Setting property options for vProfiler

The following table summarizes the property options that vProfiler supports through running the commands **adb shell setprop [OPTIONS]**. These options are similar to the environment variables available for the Linux OS.

Table 30. vProfiler Set Property Options for Android Platform

adb shell setprop OPTIONS	Description
setprop VIV_PROFILE 0	Run this command in **adb shell** to disable vProfiler in the drivers
setprop VIV_PROFILE 1	Run this command in **adb shell** to enable vProfiler in the drivers
setprop VIV_PROFILE 2	Run this command in **adb shell** to have vProfiler enable/disable controlled in the application by glEnable(GL_PROFILE_VIV) and glDisable(GL_PROFILE_VIV calls. *(available from VTK 1.5.7)*
setprop VIV_PROFILE 3 **setprop VP_FRAME_START** **setprop VP_FRAME_END**	Run these commands in **adb shell** to have vProfiler start-stop at frames specified in VP_FRAME_START and VP_FRAME_END. *(available from VTK 1.5.8)*
setprop VP_PROCESS_NAME appname	Run this command in **adb shell** to specify the application to profile. Change the app name as needed to profile another application. NOTES: There may be different sub use case names used by an app. Be sure to accurately specify a use case name to match the name on the command line when using **ps** command. *This option is only available for the Android platform, not available for the Linux OS.*
setprop VP_OUTPUT newpath	Run this command in **adb shell** to specify a new location for vProfiler output. By default, the **vpd** file is created under **/sdcard/**. If an application has no access to the SD card, specify another path where the application does have write permission. NOTE: For applications which initialize during Android system boot startup, such as launcher, kill the process after you change to a new

	path. When the application automatically restarts, then the vpd is accessible in a desired location.
setprop VP_FRAME_NUM xxx	Run this command in **`adb shell`** to limit the number of frames to analyze. For example, to make vProfiler dump performance data for the first 100 frames: **`setprop VP_FRAME_NUM 100`** NOTES: Only use when VIV_PROFILER is set to 1. When this option is not used, the profile file generated when running an application for a long time can be very large. This takes up a large amount of disk space and also makes it hard to view the data in vAnalyzer.
setprop VP_SYNC_MODE 0 **setprop VP_SYNC_MODE 1**	Run this command in **`adb shell`** to enable or disable synchronous mode. By default, vProfiler works in synchronous mode (=1). To get accurate values from the GPU counters, vProfiler needs to commit the GPU commands at the end of every frame; this is so-called synchronous mode. This example command makes vProfiler work in asynchronous mode: **`setpropVP_SYNC_MODE0`**

12.6.4 vProfiler setup for the QNX OS

The vProfiler tool can be set up to use with the QNX platform according to the instructions below.

12.6.4.1 Enable vProfiler option

When building the Vivante Graphics Drivers for QNX environment, build the driver with the vProfiler capability.

The graphics.conf file contains the configuration information for Screen and is found under the following directory:

SCREEN-DIR/usr/lib/graphics/TARGET-SPECIFIC

To activate the vProfiler functionality, add the **`gpu-gpuProfiler=1`** option into the *khronos* section of the corresponding *graphics.conf* file:

```
begin khronos
 ...
 begin wfd device 1
 ...
 gpu-gpuProfiler=1
 ...
 end wfd device
 ...
end khronos
```

When the QNX Screen graphic subsystem is started, it reads this option from the config file and enables the vProfiler function.

12.6.4.2 Setting property options for vProfiler

The following table summarizes the property options that vProfiler supports by setting environment variables. These options are similar to the environment variables available for the Linux OS.

Table 31. vProfiler Set Property Options for Android Platform

Environment Variable	Description
export VIV_PROFILE=0	Set VIV_PROFILE to 0 to disable vProfiler in the drivers
export VIV_PROFILE=1	Set VIV_PROFILE to 1 to enable vProfiler in the drivers. Optionally set

export VP_FRAME_NUM=100	the VP_FRAME_NUM to specify the number of frames dumped by vProfiler.
export VIV_PROFILE=2	Set VIV_PROFILE to 2 to control the vProfiler in the application by glEnable(GL_PROFILE_VIV) and glDisable(GL_PROFILE_VIV) calls. *(available from VTK 1.5.7)*
export VIV_PROFILE=3 **export VP_FRAME_START=**10 **export VP_FRAME_END=**90	Set VIV_PROFILE to 3 to have vProfiler start-stop at frames specified in VP_FRAME_START and VP_FRAME_END. *(available from VTK 1.5.8)*
export VP_OUTPUT=newpath	The output file of vProfiler is vprofiler.vpd by default. To specify an alternate filename use the environment variable VP_OUTPUT. For example, export VP_OUTPUT=/tmp/sample.vpd. Make sure the directory specified has correct access rights.
export VP_SYNC_MODE=0 **export VP_SYNC_MODE=1**	Set VP_SYNC_MODE to enable [1] or disable [0] synchronous mode. By default, vProfiler works in synchronous mode (=1).
export VP_USE_GLFINISH=0 **export VP_USE_GLFINISH=1**	Set the VP_USE_GLFINISH to enable [1] or disable [0] the use of glFinish() instead of eglSwapBuffers() as the frame delimiter (default is disabled).

12.6.5 vProfiler collecting performance data

vProfiler is implemented by using hardware counters and a group of instrumentations inserted into drivers that are controlled by compilation flags.

12.6.5.1 Performance counters

vProfiler counters are divided into five sets: HAL (Vivante Graphics driver), (shader) program, OpenGL and OpenVG. The counters provide detailed per frame runtime information about the application that can help the developer monitor and tune an application's resource usage. The following table briefly lists the various profile counter sets. For further information, see Appendix A at the end of this document.

Table 32. Performance Counter Types

Counter Type	Description
HALCounters	Driver memory usage
Program	Statistics of the shaders loaded in the GPU (Note: Available only for OpenGL ES 2.0 applications.)
OGLCounters	Various OpenGL (OpenGL ES 20 or 11) counters, such as API usage and primitives drawn.
OVGCounters	Various OpenVG counters, such as API usage and primitives drawn.

12.6.6 vAnalyzer viewing and analyzing a run-time profile

vAnalyzer is a GUI-based tool whose purpose is to help the user view and analyze GPU performance data that was collected using counters during an application run. The performance data from a binary file (*.vpd) written by vProfiler is displayed by vAnalyzer both in text lists and as line graphs. vAnalyzer features a multi-tab, multi-pane, graphical user interface that gives the user several ways to inspect any frame in a captured animation sequence.

12.6.6.1 Loading profile files

vAnalyzer accepts a profile for input, which is a **.vpd** file of performance data created by the Vivante vProfiler during a run. For example, the saved file may have a name such as **sample.vpd**.

A **.vpd** file can be selected using the **File/Load Profile Data** menu option.

When a performance profile is loaded, vAnalyzer populates the title bar with information about the GPU and the CPU.

The vAnalyzer screen shot below shows the vAnalyzer GUI immediately after loading a.vpd performance file, and moving the frame number slider to frame 700. By default, the main pane of the vAnalyzer window displays the Charts tab which provides charts for frame time, driver time and GPU idle cycles. Additional charts can be added in the graph window by selecting from the list of variables on the right. Different combinations of counters can be displayed in graphical and list form to illustrate resource utilization for any portion of the profiled application. A second tab contains system information.

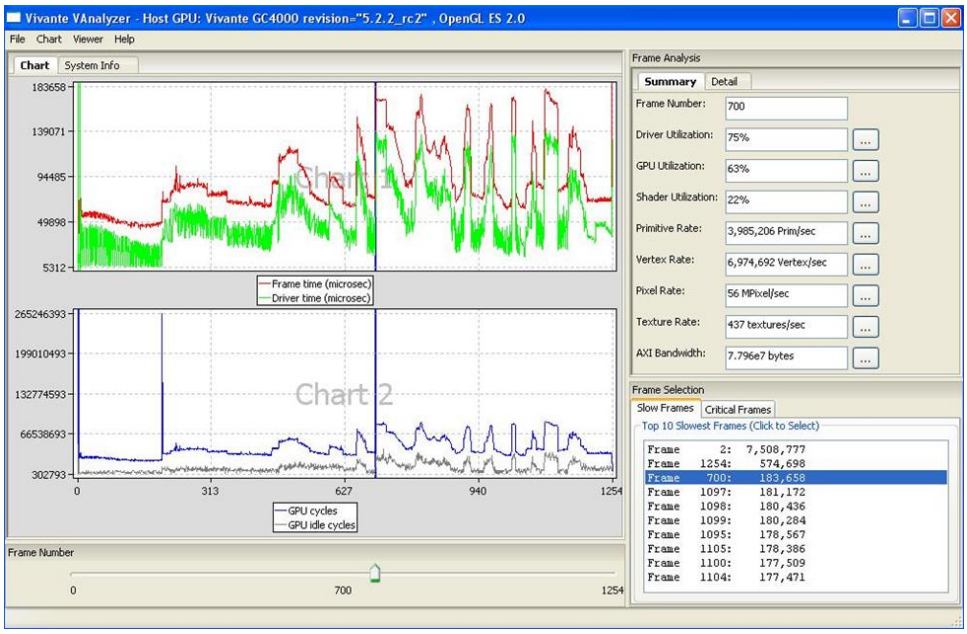


Figure 44 vAnalyzer GUI Main Window

12.6.6.2 vAnalyzer menu bar

The vAnalyzer main window opens when a user launches vAnalyzer. The main menu bar contains drop-down menus for **File**,**Chart**, **Viewer** and **Help**. Menu options include the following:

File

- **Load Profile Data**: load a .vpd profile file
- **Export Current Frame Data**: dump all the counters for the frame being viewed to a .cvs file
- **Exit**: exit vAnalyzer

Chart

- **Create chart**: create a new chart
- **Customize chart**: add or delete counters in an existing chart
- **Remove chart**: delete a chart
- **Export data from chart**: dump the counters in a chart to a .csv file
- **Save chart to png**: dump the chart to a .png file
- **View**: zoom in, zoom out or fit the chart

Viewer

- **OpenGL function call viewer**: display the OpenGL function call statistics
- **Program viewer**: display the shader program statistics

Help

- **About**: gives version information for vAnalyzer

12.6.7 vAnalyzer charts

12.6.7.1 vAnalyzer upper left pane: chart tab and menu options

On the Chart tab in the vAnalyzer main window two default line graphs are displayed.

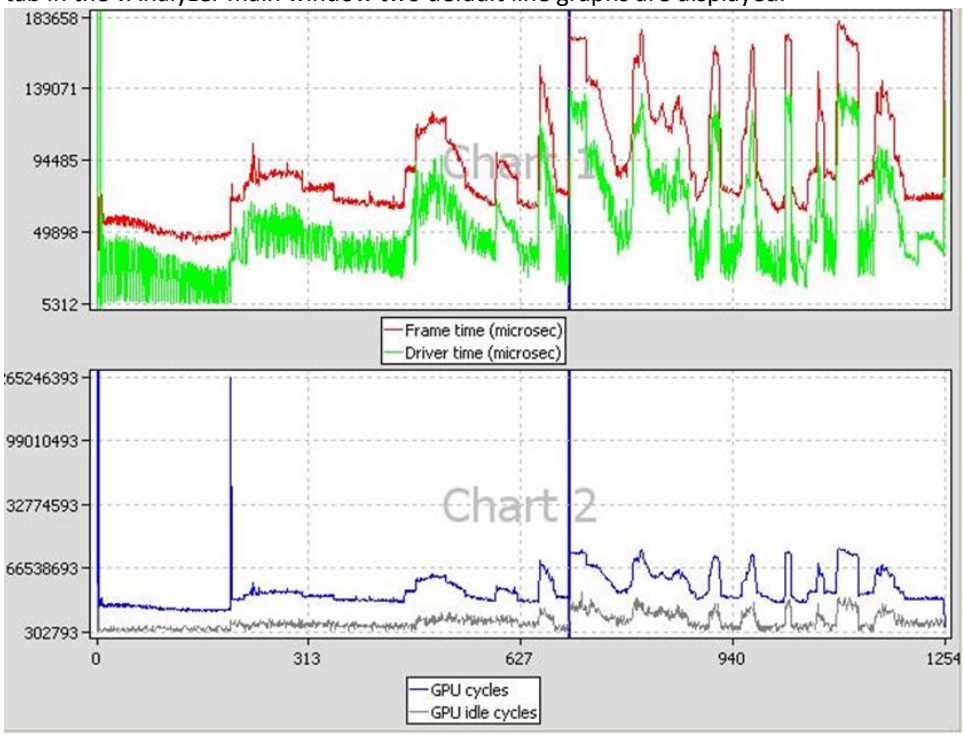


Figure 45 vAnalyzer Performance Counter Charts

12.6.7.2 Chart customization

Chart/Customize: Additional performance counters can be added to existing chart using the **Customize Chart** dialog window, which can be invoked from the drop menu **Chart/Customize**, or from a pop-up menu, which can be invoked by right clicking in the Chart tab area.

Create New Chart: A new chart can be added in a similar way. A single chart can display up to four (4) counters, and the Chart pane can hold up to eight (8) charts. Thus a maximum of thirty-two (32) counters can be graphed at the same time.

Remove Chart: Any chart can be removed from the display using the drop menu **Chart/Remove Chart**.

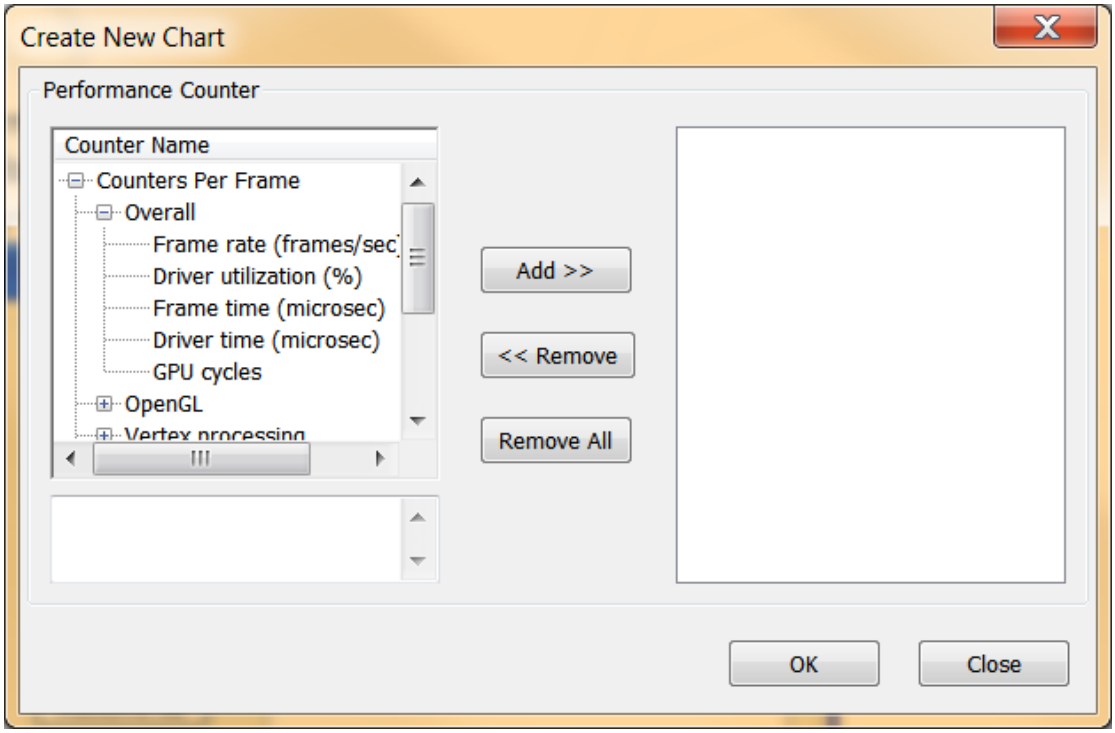


Figure 46 vAnalyzer Create New Chart Dialog

12.6.7.2.1 Chart components and navigation

Frame Marker: On the plots displayed in the chart example above there is a blue, vertical frame marker. This marks the current frame position in the timeline.

Zoom:

Zooming in on a set of frames can be achieved in one of two ways.

- One method is to hold down the left mouse button and then sweep a selection box across a range of frame numbers, either on a plot itself or in the common X-axis (frame numbers) in the "Chart" pane, before releasing the mouse button. All charts in the "Chart" pane zooms in to the same range of frames.
- Alternatively, if the mouse has a scroll wheel, zoom in by rolling the wheel forward--toward the screen.

To zoom out move the scroll wheel backward.

To reset zoom to the default, which shows the entire timeline, press the escape key (ESC) on the keyboard. The chart view changes to include all frames, from start to end.

12.6.7.2.2 Data export

The performance counters in a chart can be dumped to a `.csv` file by selecting from the dropdown menu **Chart / Export Data From**. The `.csv` file can be viewed using Excel or another text viewer.

The chart can also be dumped to a **.png** file by selecting from the main menu **Chart / Save chart to PNG**.

12.6.7.3 vAnalyzer lower left pane: frame number slider bar

In the lower left pane of the vAnalyzer window, there is a Frame Number gauge in the form of a slider bar. Numbers at each end of the bar indicate the initial frame (0) and the last frame available in the loaded sample. By left-clicking and holding the slider, the user can change which frame is selected for analysis. When the frame

number is changed, the blue vertical line which indicates the current frame is moved, and the reported Frame Number changes in the upper right pane Frame Analysis Summary.

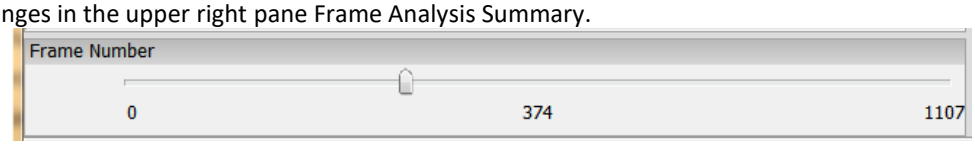


Figure 47 vAnalyzer Frame Number Slider Bar

12.6.7.4 vAnalyzer left pane: System Info tab

When a **.vpd** profile is loaded, system information about the profiled machine populates the fields on the System Info pane. Some information is repeated in the title bar of the main GUI for quick reference.

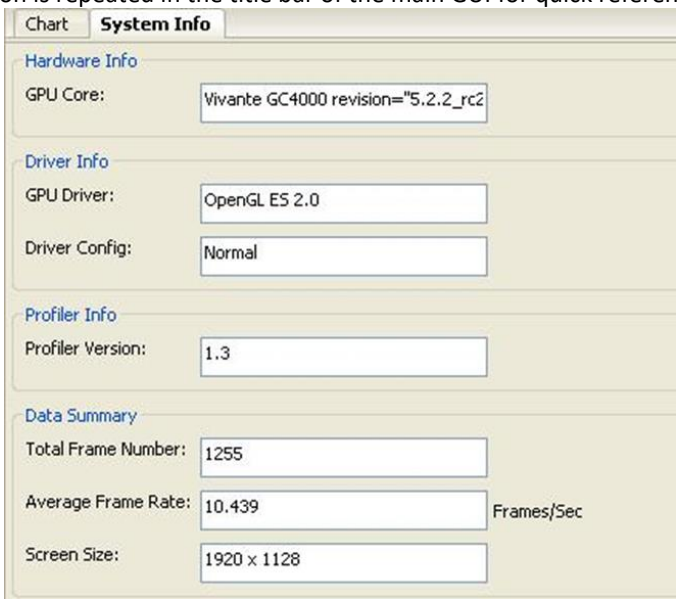


Figure 48 vAnalyzer System Info Tab

12.6.7.5 vAnalyzer upper right pane: Frame Analysis

A selection of performance counters for the frame being viewed are displayed on the right side of the vAnalyzer main GUI. The user can convert this pane to a pop-up window by dragging the pane outside the application window. Drag it back to the right pane area of the application window to reintegrate the pane.

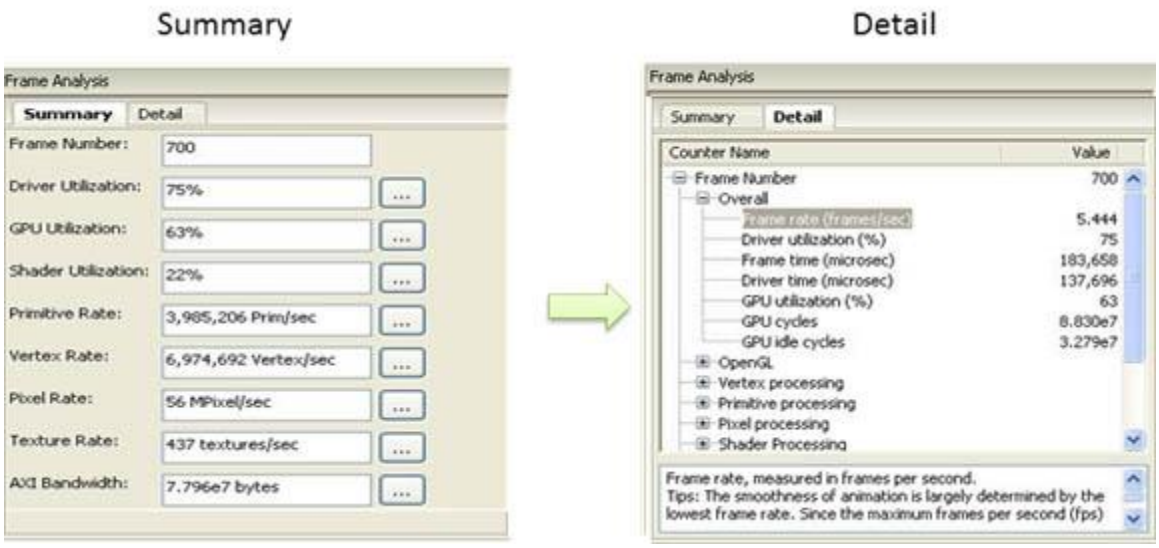


Figure 49 vAnalyzer Frame Analysis Summary and Detail Tabs

12.6.7.5.1 Summary tab

The **Summary** tab displays summary information for the frame being viewed.

The Selected Frame Number can be changed by entering a new frame number in the text box at the top of the list. The user must press Enter after the input to activate the change. Then Summary values, sliders, and charts all change to reflect the newly entered frame number.

The Summary values below frame number are not directly changeable. They change only when the frame number is changed, either in the Summary tab, by moving the Frame Number slider, or by selecting a frame from the Frame Selection pane. Clicking the "**…**" button to the right of a **Summary** item brings up the corresponding counters in the **Detail**tab. For example, clicking the "**…**" button to the right of**Primitive Rate:** switches the view to the **Detail** tab and expands the **Primitive processing**catogory. Clicking the "**…**" button for**Driver Utilization:** brings up the pop-p window **OpenGL function call viewer**.

12.6.7.5.2 Detail tab

The **Detail** tab reports values for overall performance evaluation, such as Frame Rate, Driver Utilization, and GPU cycles. Additionally counter detail is accessible on this tab. The categories of available counters in the **Detail** tabare: Overall, OpenGL, Vertex processing, Primitive processing, Pixel processing, Shader Processing, Texturing and AXI Bandwidth. Appendix A lists performance as well as hardware counters.

12.6.7.6 vAnalyzer lower right pane: Frame Selection

As with the Frame Analysis pane, this pane can be dragged to display as an independent popup window.

12.6.7.6.1 Slow Frames tab

The "Slow Frames" tab lists the ten (10) slowest frames in the animation sequence, by time in ascending order from slowest to tenth slowest.

The user can left click on any entry, or can use the arrow keys to move up and down the list, and the display in each of the other GUI panes changes to match that frame.

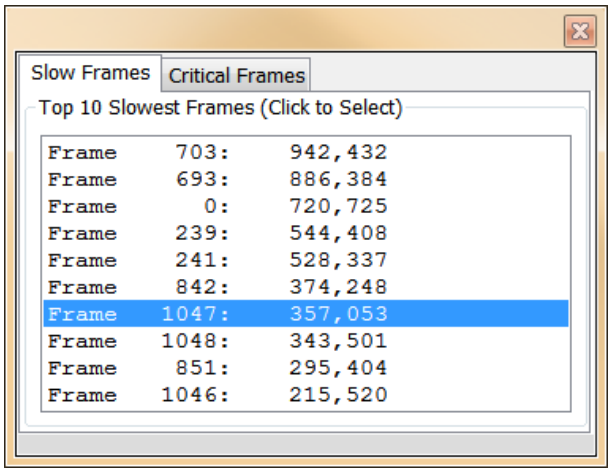


Figure 50 vAnalyzer Frame Selection Slow Frames Tab

12.6.7.6.2 Critical Frames tab

Select the "Critical Frames" tab to customize the criteria by which a frame is chosen for inspection. One or more of the performance counters can be specified in building the query, which also allows for AND and OR logic. Queries should follow a pattern such as:

`"counter name" condition('<','>','==') values.`

Users can identify counter names from those in the Frame Analysis pane Detail tab. An example is provided just below the Query input text box.

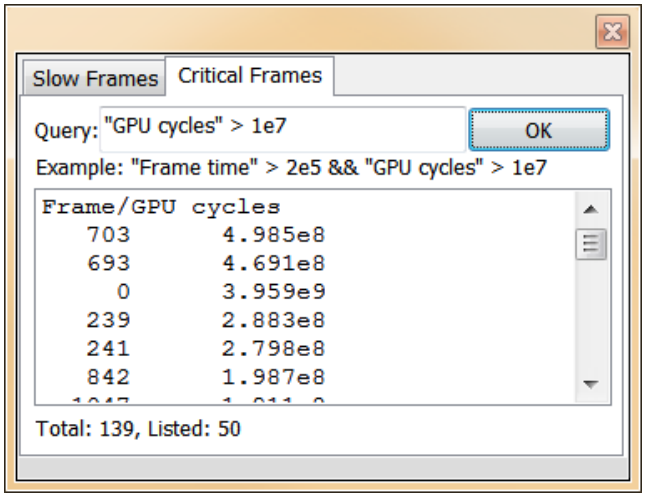


Figure 51 vAnalyzer Frame Selection Critical Frames Tab

12.6.8 vAnalyzer viewers

The Viewer information pop-up window can be launched by selecting **Viewer/Function Call Viewer** or **Viewer/Program Viewer** from the Main menu. The selected Viewer appears in a pop-up window.

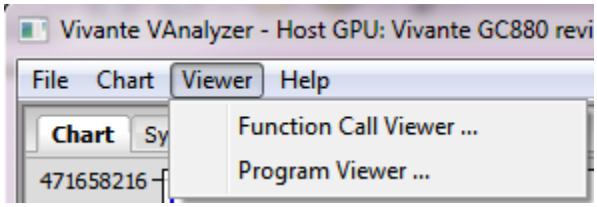


Figure 52 vAnalyzer viewers

12.6.8.1 OpenGL Function Call viewer

The OpenGL function call viewer includes three information areas.

- The **OGL Function Name** area contains a table which lists the available OpenGL ES functions by Function Name and Function Type, the run time and the number of times each has been called for this frame. Functions can be sorted by clicking in the column heading area. For example, sort the functions by call count or run time by clicking the title bar of "# of Call" or "Time (ms)".
- The **Top 5 Functions** area contains a histogram which shows the top 5 call count of the listed OpenGL functions.
- The **Property view** area shows the summary when no function is selected; while it shows performance hints for the function when one is selected.

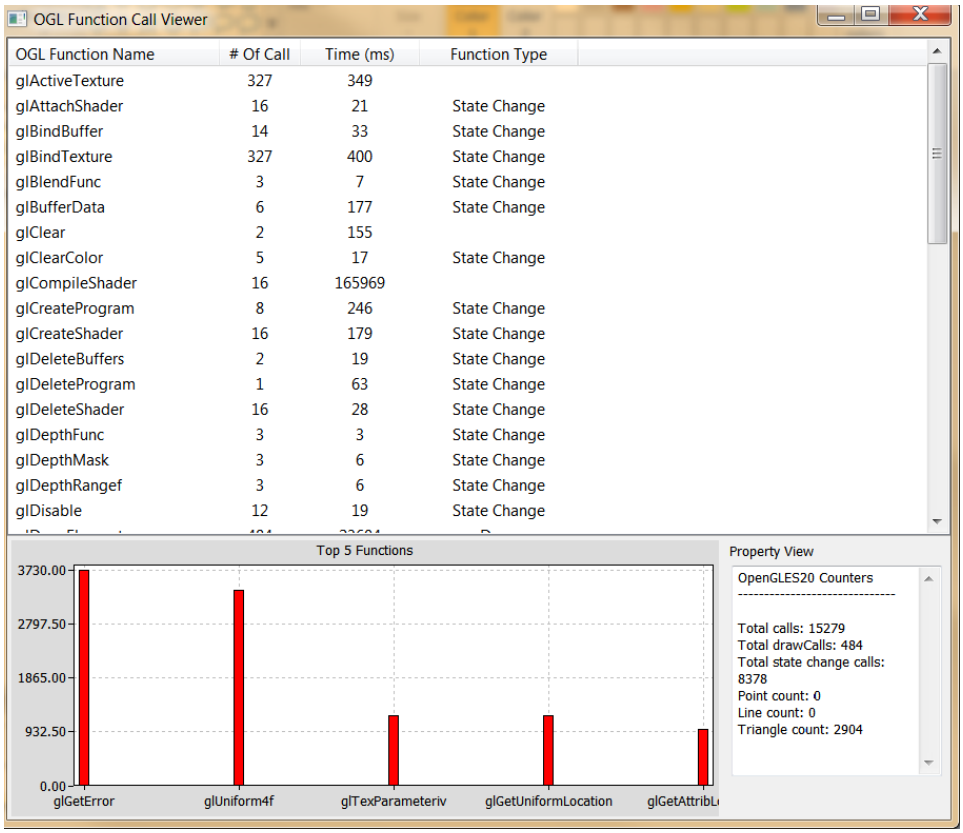


Figure 53 vAnalyzer OpenGL function call viewer window

12.6.8.2 Program viewer

For a given Frame Number, the Program Viewer gives the statistics for shader programs: uniforms, attributes, and the number of instructions in the shader. This is only for OpenGLES2, ES3 profile data. The description of the item

is displayed in the lower text window when selecting the item. Expand by clicking on **VS** or **PS** submenu to expand the detail for that shader's source code.

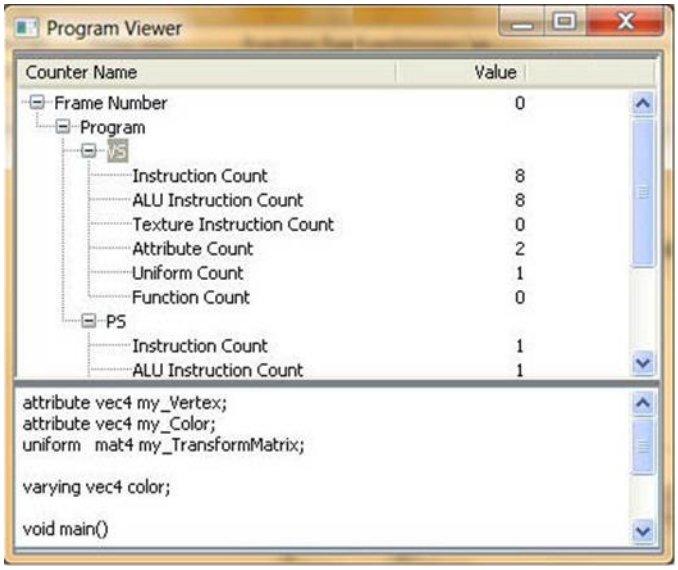


Figure 54 vAnalyzer Program Viewer

12.7 Debug and performance counters

Availability of some counters varies depending on core capabilities and software source tree.

12.7.1 AXI bandwidth

- Read bandwidth (byte)
- Write bandwidth (byte)
- Total bandwidth (byte)
- AXI cycles when read request stalled
- AXI cycles when write request stalled
- AXI cycles when write data stalled

12.7.2 Overall

- Frame rate (frames/sec)
- Driver utilization (%)
- Frame time (microsec)
- Driver time (microsec)
- GPU utilization (%)
- GPU cycles
- GPU idle cycles

12.7.3 OpenGL

- Total calls
- Total draw calls
- Total state change calls
- Point count
- Line count
- Triangle count

12.7.4 Pixel processing

- Valid pixel count
- % alpha test fail
- % depth&stencil test fail
- Overdraw

12.7.5 Shader processing

- VS instruction count
- VS branch instruction count
- VS texture fetch count
- Rendered vertex count
- PS instruction count
- PS branch instruction count
- PS texture fetch count
- Rendered pixel count

12.7.6 Texturing

- Total bilinear requests
- Total trilinear requests
- Total texture requests
- Total discarded texture requests

12.7.7 Vertex processing

- Input vertex count
- Vertics per batch
- Vertics per primitive

12.7.8 Vertex shader and fragment shader

(per shader, for ES20 and ES30 applications only)

- Total instruction count
- ALU instruction count
- Texture instruction count
- Function calls
- Attribute count

Chapter 13 GPU Tools

13.1 gpuinfo tool

13.1.1 Introduction

gpuinfo is a script to gather GPU runtime status through debugfs interface. It exports below information:

- GPU hardware information.
- GPU total memory usage.
- GPU memory usage of certain process or all processes (user space only).
- GPU idle percentage.

13.1.2 Usage

The script is located at Yocto rootfs /unit_tests/. There are three ways to run it.

1. Normal run to get all GPU-related processes information:

```
>/unit_tests/gpuinfo.sh
```

2. Get GPU information for certain process by clarifying the process id.
   The process id (pid) can be got by command ps or top. Take the process 1035 as example.

```
>/unit_tests/gpuinfo.sh 1035
```

3. Get the GPU information for certain process by clarifying part of process name.
   Take the process sample_test_fbo as an example.

```
>/unit_tests/gpuinfo.sh sample_test_fbo
```

or

```
>/unit_tests/gpuinfo.sh sample
```

or

```
>/unit_tests/gpuinfo.sh test
```

13.1.3 Sample log information

13.1.3.1 GPU hardware information

This section shows all GPU cores model name and revision information with index in the SoC.

The sample information:

```
GPU Info
gpu      : 0
model    : 2000
revision : 5108

gpu      : 1
model    :  320
revision : 5007

gpu      : 2
model    :  355
revision : 1215
```

13.1.3.2 Total memory information

This part shows total GPU memory information.

Table 33. Total memory information

gcvPOOL_SYSTEM:	GPU reserved system memory.
gcvPOOL_CONTIGUOUS:	contiguous memory allocated from CMA pool, low memory zone and high memory zone.
gcvPOOL_VIRTUAL:	non-contigous memory allocated from low memory zone and high memory zone.
NON PAGED MEMORY:	Allocated from CMA pool(mainly for command buffer)

The sample information:

```
    VIDEO MEMORY:
        gcvPOOL_SYSTEM:
            Free  :  124170474 B
            Used  :   10047254 B
            Total :  134217728 B
        gcvPOOL_CONTIGUOUS:
            Used  :          0 B
        gcvPOOL_VIRTUAL:
            Used  :          0 B

    NON PAGED MEMORY:
        Used  :          0 B
    Paged memory Info
    low:  892928 bytes
    high: 0 bytes
    CMA memory info
    cma:  0 bytes
```

13.1.3.3 Process user space GPU memory usage information

This part shows detail user space GPU memory usage per process.

Table 34. User space GPU memory usage

Index	memory for index buffer.
Vertex	memory for vertex data buffer.
Texture	memory for texture buffer.
RT	memory for render target buffer.
Depth	memory for depth buffer.
Bitmap	memory for bitmap buffer.
TS	memory for tile status buffer.
Image	memory for vg image buffer.
Mask	memory for vg mask buffer.
Scissor	memory for vg scissor buffer.
HZDepth	memory for hierarchical Z depth buffer.

The sample information:

```
VidMem Usage (Process 1106):
Counter: vidMem (for each surface type)
All     Index    Vertex   Texture        RT     Depth    Bitmap        TS     Image      Mask   Scissor
HZDepth
Current    10047254    489362   1213248    435200   3866624   3727360         0     36352         0
0         0    245760
Maximum    10047254    489362   1213248    435200   3866624   3727360         0     36352         0
0         0    245760
Total      10047254    489362   1213248    435200   3866624   3727360         0     36352         0
0         0    245760
Counter: vidMem (for each pool)All         1         2         3         4         5         6         7
8         9
    Current    10047254         0         0         0         0         0  10047254         0         0
0
    Maximum    10047254         0         0         0         0         0  10047254         0         0
0
    Total      10047254         0         0         0         0         0  10047254         0          0
0
    Counter: nonPaged
                       All
    Current              0
    Maximum              0
    Total                0
    Counter: contiguous
                       All
    Current              0
    Maximum              0
    Total                0
    Counter: mapUserMemory
                       All
    Current              0
    Maximum              0
    Total                0
    Counter: mapMemory
                       All
    Current    134217728
    Maximum    134217728
    Total      134217728
```

13.1.3.4 GPU idle percentage

This part shows GPU idle percentage in past 1s.

The sample information:

```
    >>>>
```

```
root@sabresd_6dq:/ # gmem_info
 Pid          Total      Reserved    Contiguous       Virtual      Nonpaged    Name
  151    28,283,476    19,239,508     5,898,240     3,145,728             0    /system/bin/surfaceflinger
  667    25,160,836    21,212,292     3,162,112       786,432             0    com.android.launcher
  638    14,714,500    13,928,068             0       786,432             0    com.android.inputmethod.latin
  873    11,589,380    10,802,948             0       786,432             0    com.android.email
 1039    11,074,692    10,288,260             0       786,432             0    com.fsl.ethernet
  519     6,438,532     5,652,100             0       786,432             0    com.android.systemui
 1091     5,419,268     4,632,836             0       786,432             0    com.android.calculator2
  908     5,306,500     4,520,068             0       786,432             0    com.android.calendar
  986     5,265,668     4,479,236             0       786,432             0    com.android.contacts
  966     5,236,932     4,450,500             0       786,432             0    com.android.gallery3d
 1015     4,710,660     3,924,228             0       786,432             0    com.android.quicksearchbox
  436     1,247,232       198,656             0     1,048,576             0    system_server
 ---------------------------------------------------------------------------
   12   124,447,676   103,328,700     9,060,352    12,058,624             0    Summary
    -             -    30,889,028             -             -             -    Available
GPU Idle time:  83562.359375 ms
```

Figure 55 gmem_info tool

13.3 Apitrace user guide

13.3.1 Introduction

Apitrace is a set of tools enhanced from open source project apitrace, supported by i.MX 6, i.MX 7, and i.MX 8 with Vivante GPU IP. This tool can dump OpenGL/GLES1.1/GLES2.0/GLES3.0 API calls and replay on a wide range of other devices.
For more information, see apitrace.github.io/.

13.3.2 Install

13.3.2.1 Yocto

APITrace source code release part of the i.MX Yocto Project Linux BSP release. The source code have more patches added on top of official API Trace release. The Yocto Project recipes pull the apitrace source package and install as needed for X11, Framebuffer or Wayland backend.Yocto FB/DFB/Wayland

13.3.2.2 Android Platform

It will be preinstalled in next release. Currently have to install them by hand:
Mount release package to Android system:

```
mkdir /data/share; busybo mount -t nfs -o nolock <host> /data/share
cp -r apitrace/android/apitrace /data/
```

A convenient alternative:

```
adb push apitrace/android/ /data/local/tmp/
```

Note 1: If install to a directory other than /data/apitrace, update apitrace/bin/apitrace_dalvik.sh to use the new path.
Note 2: Pay attention to file attributes. You need to **grant access to the whole file path of eglretrace for normal user**, because Java applications are running as normal user even if it is started by root user.

13.3.2.3 PC

APITrace have set of PC tools. Prebuilt binary packages can be directly downloaded from APITrace website. Currently supports Ubuntu 14.04 LTS, 64-bit.

```
sudo apt-get install libgles1-mesa libgles2-mesa libqt4-dev
```

13.3.3 Usage

13.3.3.1 Trace OpenGL ES1.1/2.0/3.0 application

```
apitrace trace --api=egl <app name and arguments>
```

e.g., apitrace trace --api=egl es2gears_x11
It generates trace file (.trace) under the current directory. To specify a new path, use --output=<path_name>

13.3.3.2 Trace OpenGL ES 1.1/2.0/3.0 Java application on the Android platform

On the Android platform, a GLES application can be native (e.g., frameworks/native/opengl/angeles). This type of application can be traced as normal Linux application. Some other applications involving the Java virtual machine cannot run in this way. A script apitrace_dalvik.sh is provided to run this type of application. This is an example to trace com.android.settings:

```
sh /data/apitrace/bin/apitrace_dalvik.sh com.android.settings start
```

To stop tracing, run:

```
sh /data/apitrace/bin/apitrace_dalvik.sh com.android.settings stop
```

Because there is no “current” directory for a Java application, the trace file is stored to under /sdcard/
If apitrace is installed in a different directory, you need to update apitrace_dalvik.sh by hand

13.3.3.3 Trace OpenGL application

```
apitrace trace --api=glx <app name and arguments>
```

Only the X11 backend supports this feature

13.3.3.4 Replay

This utility is also called retrace. It reads in the trace file and executes OpenGL(ES) APIs one by one. Each OpenGL(ES) API call is processed by a callback function. In that callback function, a hook can be inserted for debug or analysis purposes.

Figure 56 Replay

OpenGL ES 1.1/2.0/3.0 applications can be replayed with eglretrace; Open GL applications can be replayed with **glretrace**:

```
eglretrace <trace file>
glretrace <trace file>
```

Supported platforms:

	eglretrace	Glretrace
Yocto-X11	X	X
Yocto-FB/DFB/Wayland	X	
Android		
PC	X	X

For ES 3.0 replay, only i.MX supports this feature. It is not available on PC.

13.3.3.5 Analysis

qapitrace provides a detailed look at the trace file. It can only run on a PC. Verified on Ubuntu 14.04 LTS 64-bit. The command is:

```
qapitrace <trace file name>
```

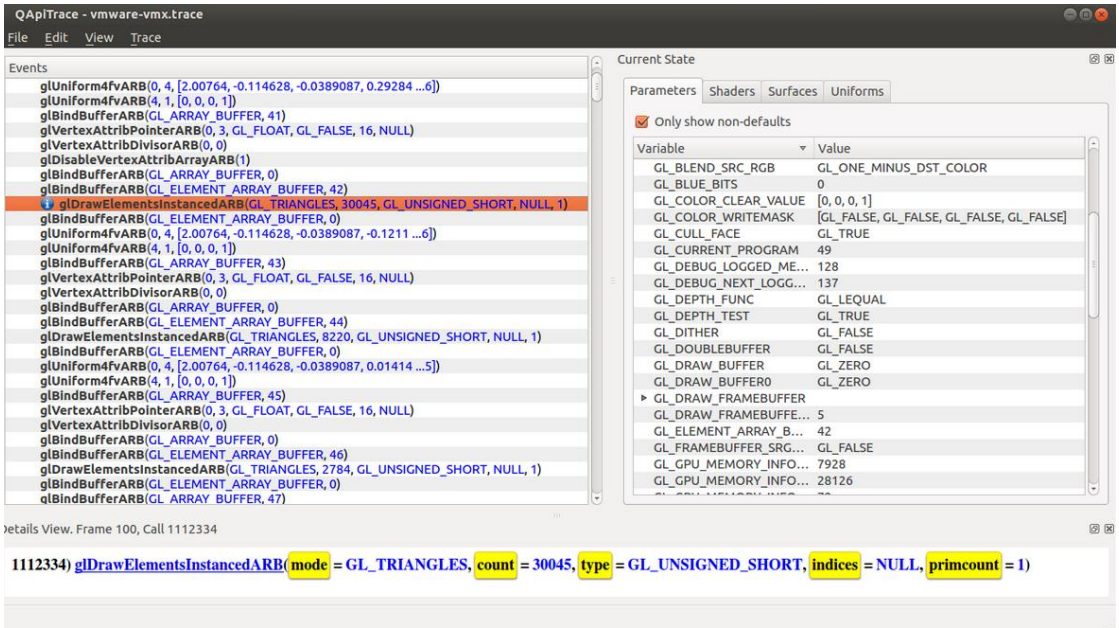


Figure 57 Checking state of every API call

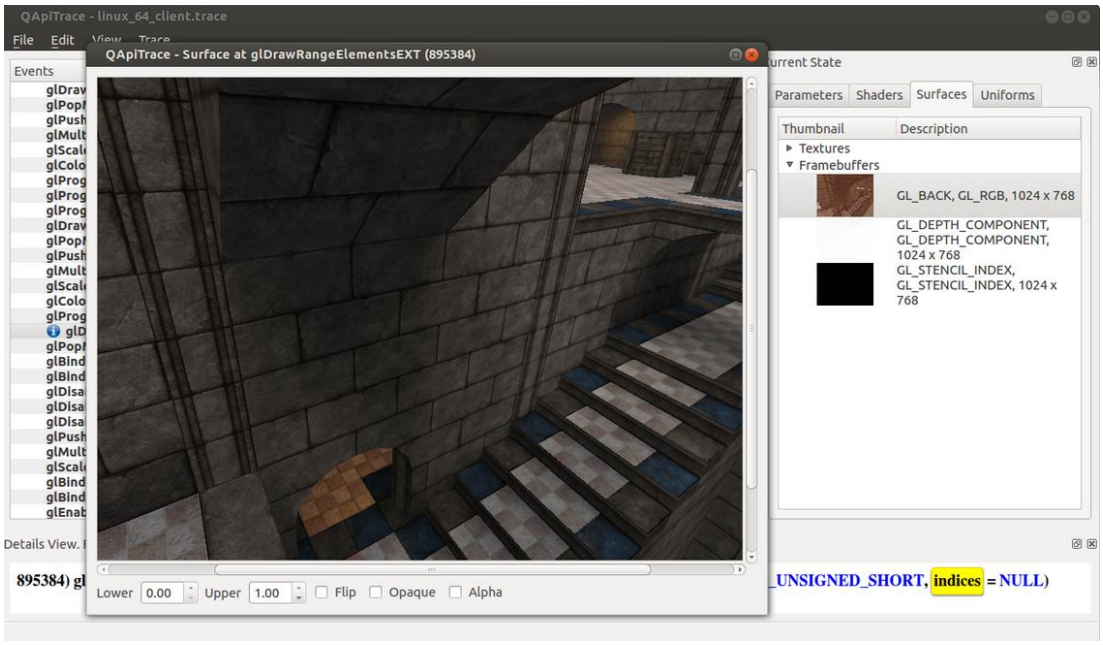


Figure 58 Checking Framebuffer

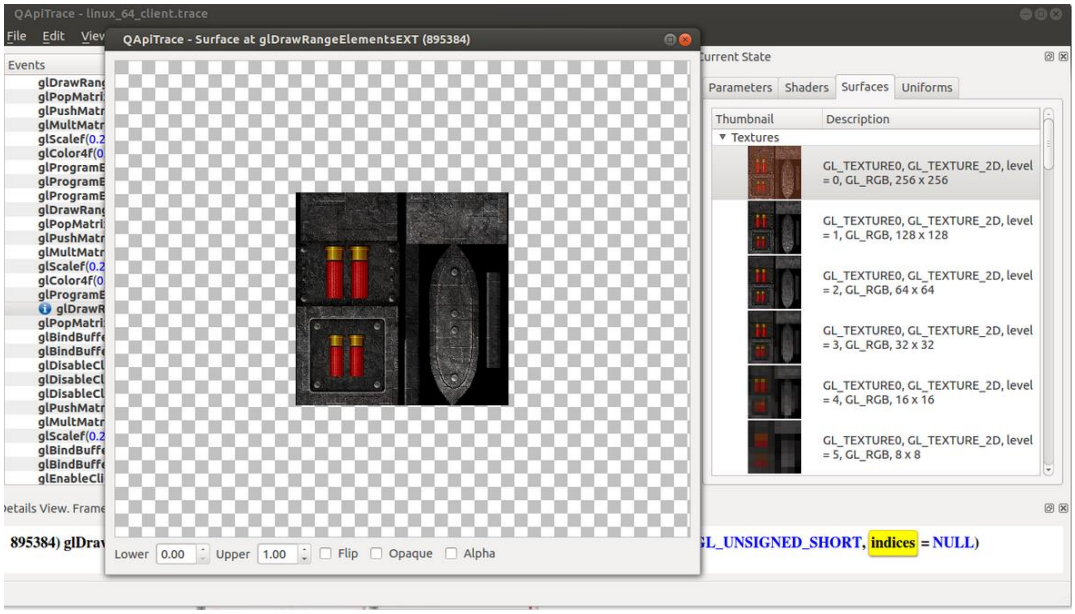
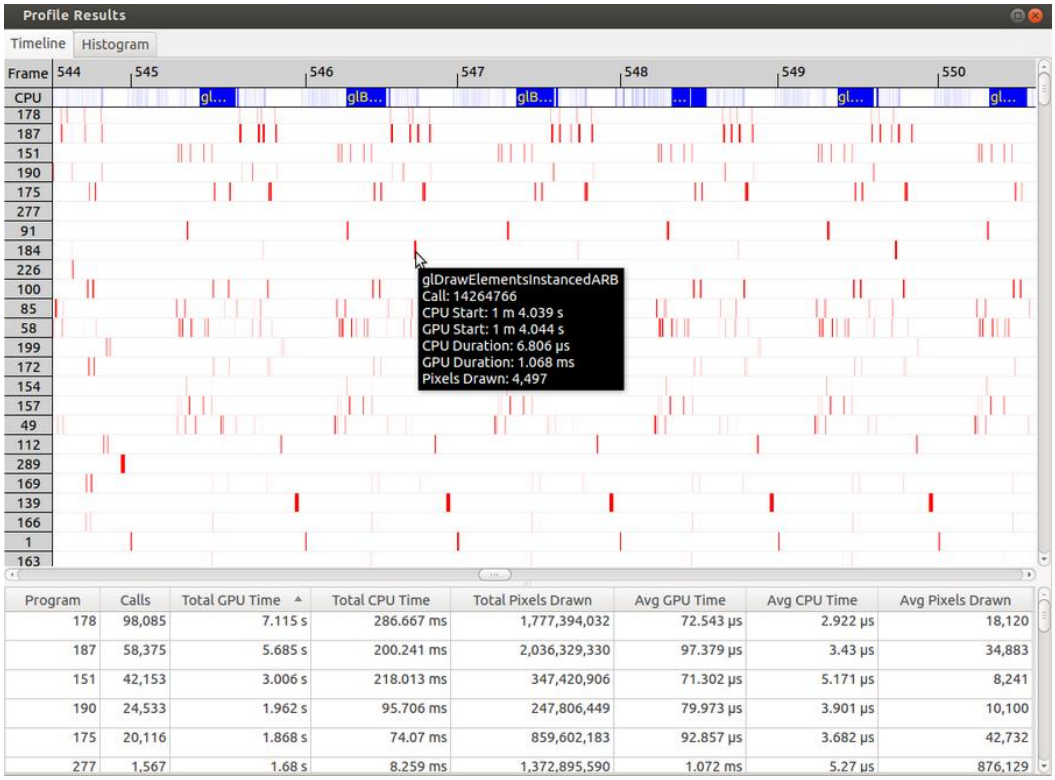


Figure 59 Checking Texture



Program	Calls	Total GPU Time	Total CPU Time	Total Pixels Drawn	Avg GPU Time	Avg CPU Time	Avg Pixels Drawn
178	98,085	7.115 s	286.667 ms	1,777,394,032	72.543 µs	2.922 µs	18,120
187	58,375	5.685 s	200.241 ms	2,036,329,330	97.379 µs	3.43 µs	34,883
151	42,153	3.006 s	218.013 ms	347,420,906	71.302 µs	5.171 µs	8,241
190	24,533	1.962 s	95.706 ms	247,806,449	79.973 µs	3.901 µs	10,100
175	20,116	1.868 s	74.07 ms	859,602,183	92.857 µs	3.682 µs	42,732
277	1,567	1.68 s	8.259 ms	1,372,895,590	1.072 ms	5.27 µs	876,129

Figure 60 Checking performance

13.3.4 Reference

1. Apitrace introduction: apitrace.github.io/
2. More uses: github.com/apitrace/apitrace/blob/master/README.markdown

Chapter 14 GPU Memory Introduction

14.1 GPU memory overview

- OpenGL-ES
  - Texture buffer
  - Vertex buffer
  - Index buffer
  - PBuffer surface
  - Color buffer
  - Z/Stencil buffer
  - HZ depth buffer
  - Tiled status buffer
  - 3D Command buffer
  - 3D Context buffer
- OpenVG
  - Image buffer
  - Tessellation buffer
  - VG command buffer
  - VG context buffer
- 2D buffers
  - 2D command buffer
  - 2D temporary buffer

14.2 GPU memory pools

- Reserved memory
  In the Linux 3.10.y kernel, the memory is reserved from CMA implemented in the GPU kernel driver, the size can be changed through U-Boot args with “galcore.contiguoussize =xxx”
  The memory allocation and lock very fast, but cannot support cacheable attribute.
- Contiguous memory
  The contiguous memory is from CMA or Normal or Highmem with alloc_pages_exact.
  The GPU driver tries the CMA allocator for non-cacheable request first. If CMA memory is used up, it goes to system allocator.
  The CMA allocator does not support the cacheable attribute, the system allocator supports cacheable attribute, but the memory performance is slow with the additional cache flush operations.
- Virtual memory pool
  The virtual memory is from Normal or Highmem with multiple page_alloc.
  The memory support cacheable attribute, but slow with GPU MMU and cache flush.
  The GPU virtual command buffer is allocated from virtual memory pool directly.
- Nonpaged memory pool
  In the 5.x GPU driver, this pool is not used any more

14.3 GPU memory allocators

Two kinds of allocators are implemented in i.MX GPU kernel driver, see drivers/mxc/gpu-viv/

- The video memory allocator implementation is very complicated. The memory is from the reserved pool, system contiguous pool (supports CMA), or system virtual pool (enables GPU MMU).
- The CMA allocator supports non-cacheable contiguous memory. It is implemented as a part of contiguous pool. When the system requests contiguous memory, the allocator tries CMA first. If CMA is used up, it goes to allocate the system contiguous pages.

- GPU memory-killer is implemented for special requirement of force contiguous GPU memory.

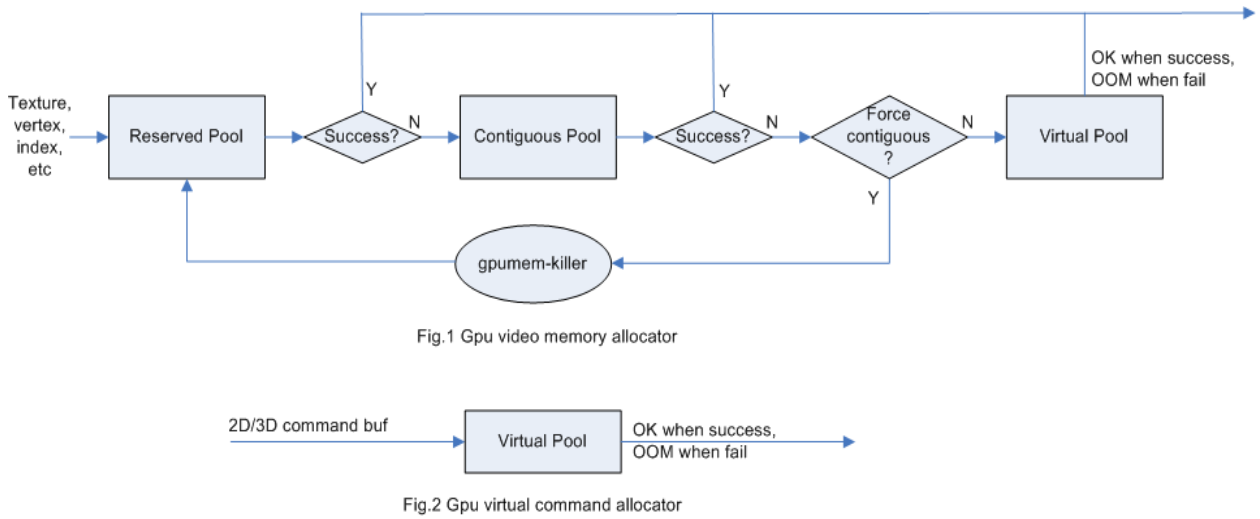


Figure 61 GPU memory allocators

14.4 GPU reserved memory

- The reserved memory is managed by two dual linked lists, one is free list, and another is node list.
- When allocate the reserved memory, the free list is scanned from head to tail until a available node is selected, it is very fast but makes more memory fragments, under test, 10~20M of 128M is not available to use after a lot of allocate/free operations.
- When the available node is selected, it is removed from the free list, but it always keeps the dual linked nodes to merge the conjoint available memory when freed.
- The reserved memory is mapped once when application process is attached, during 3D application running, the memory map/un-map operations are very fast, the virtual address is just calculated with logical base and offset.

14.5 GPU memory base address

- GPU support contiguous physical memory within (0~2G) address directly:
  - GPU address = CPU Physical address – GPU BaseAddress
- GPU MMU is enabled for two kinds of memory type as below:
  - Separated page memory from Virtual memory pool
  - Contiguous page memory with address out of (0~2G)
- BaseAddress should be set to RAM start address to achieve the better performance by reducing GPU MMU mapping.

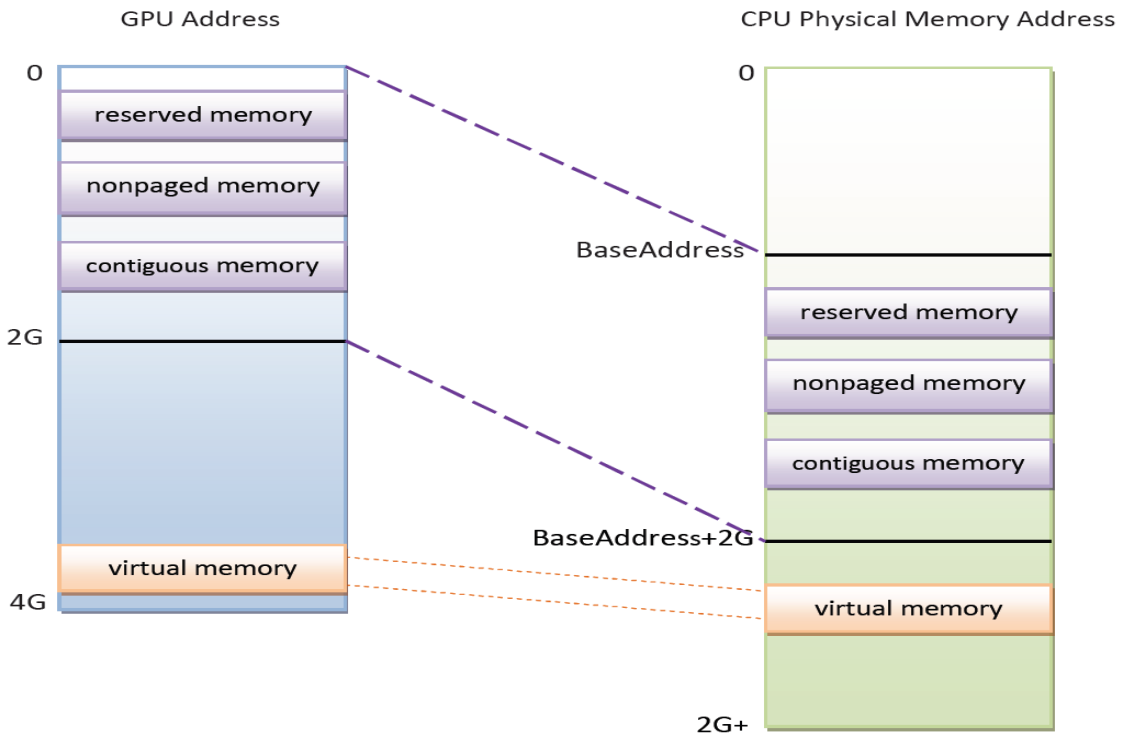


Figure 62 GPU memory base address

Chapter 15 Application Programming Recommendations

The recommendations listed below take a holistic approach centered on overall system level optimizations that balance graphics and system resources.

15.1 Understand the system configuration and target application

Knowing details about the application and use case allows developers to correctly utilize the hardware resources in an ideal access pattern. For example, an implementation for a 2D or 3D GUI could be rendered in a single pass instead of multiple passes if the draw call sequence is correctly ordered. In addition, knowing the most common graphics function calls allow developers to parallelize rendering to maximize performance.
Using Vivante and vendor-specific SoC profiling tools, you can determine bottlenecks in the GPU and CPU and make changes as needed. For example, in a 3D game, most CPU cycles may be spent on audio processing, AI, and physics and less on rendering or scene setup for the GPU. In this instance, the application is CPU-bound and configurations dealing with non-graphics tasks need to be reviewed and modified. If the system is GPU-bound, the profiler can point out where the GPU programming code bottlenecks are located and which sections to optimize to remove restrictions.

15.2 Optimize off chip data transfer such as accessing off-chip DDR memory/mobile DDR memory

Any data transfer off-chip takes bandwidth and resources from other functional blocks in the SoC, increases power, and causes additional cycles of latency and delay as the GPU pipeline needs to wait for data to return from memory. Using on-chip cache and writing the application to better take advantage of cache locality and coherency increase performance. In addition, accessing the GPU frame buffer from the CPU (not recommended) cause the driver to flush all queued render commands in the command buffer, slowing down performance as the GPU has to wait since the command queue is partially empty (inefficient use of resources) and CPU-GPU synchronization is not parallelized.

15.3 Avoid W-Clipping issue in the Application Program

The w-clipping overflow issue typically occurs with these three factors:

- **Objects with very large primitives.** In a 3D scene, this is usually the sky, the outer world or a long road that expands far behind the camera and far in front of the camera. At the same time, the object may also expand far in either the x or y direction.
- **Near-plane with a very small value.** Usually this value is very close to zero. An example would be 10^{-4}
- **Large screen resolution.**

These three factors can cause the final window coordinate to overflow the 24-bit mantissa precision in IEEE single precision floating point format.

The following are suggested ways to modify an application to avoid overflow:

1. For draw calls with very large primitives such as sky or world, set the near-plane to 0.99 as an initial value.

2. If this removes the rendering error and the entire scene is rendered correctly, the issue can be considered resolved.
3. If the rendering error is still there and no desired objects are being culled (or there are no missing objects), increase the near-plane value until the rendering error disappears.
4. If the near-plane value is large (>10.0) already, the issue persists and some desired objects are being culled, reduce the near-plane value until the desired objects appear again then go to the next step.
5. Tessellate the large objects into smaller primitives until the rendering error disappears.

Please note that the suggested near plane adjustment can be done on a per draw call basis, and only needs to be modified for objects with very large primitives. Some applications scale the object by reduce the w value in vertex shader, as change w value will finally affect the near plane, this is not recommended, a better way to scale the object is scale the x, y, z coordinate, not w.

15.4 Avoid GPU hang and data corruption when use occlusion query

Description:
On i.MX6D/Q GPU IP, both Hierarchical Depth (HZ) write and Occlusion Query (OQ) write share the same port. If HZ Fast Clear(FC) is enabled, and OQ uses the HZ port to perform a write, the HZ FC data may become corrupted, even lead to GPU hang unexpectedly.

Software Workaround:
A software workaround is recommended for this issue and is available from L4.9 bsp release. Because the issue occurs very infrequently, a per-application work around is most efficient. Software will disable HZ with a per-app detection and also provide a new environment variable control (VIV_DISABLE_HZ).

15.5 Avoid random cache or memory accesses

Cache thrashing, misses, and the need to access data in external memory causes performance hits. An example would be random texture cache access since it is expensive when performing per-pixel texture reads if the texture units need to access the cache randomly and go off-chip if there is a cache miss.

15.6 Optimize your use of system memory

Memory is a valuable resource that needs to be shared between the GPU (frame buffer), CPU, system, and other applications. If you allocate too much memory for your OpenGL ES application, less memory is available for the rest of the system, which may impact system performance. Claim enough memory as needed for your application then deallocate it as soon as your application no longer needs it. For example, you can allocate a depth buffer only when needed or if your application only needs partial resources, load the necessary items initially and load the rest later.

15.7 Target a fixed frame rate that is visibly smooth

Smooth frame rate is achieved from a combination of a constant FPS and the lowest FPS (frames per second) that is visually acceptable. There is a trade-off between power and frame rates since the graphics engine loading increases with higher FPS. If the application is smooth at 30 FPS and no visual differences for the application are perceived at 50 FPS, then the developer should cap the FPS at 30 since the extra 20 FPS do not make a visual

difference. The FPS limit also guarantees an achievable frame rate at all times. The savings in FPS help lower GPU and system power consumption.

15.8 Minimize GL state changes

Setting up state values between draw calls adds significant overhead to application performance so they must be minimized. Most of these call setups are redundant since you are saving / restoring states prior to drawing. Try to avoid setting up multiple state calls between draw calls or setting the same values for multiple calls. Sometimes when a specific texture is used, it is better to sort draw calls around that texture to avoid texture thrashing which inhibits performance. Application developers should also try to group state changes.

15.9 Batch primitives to minimize the number of draw calls

When your application submits primitives to be processed by OpenGL ES, the CPU spends time preparing commands for the GPU hardware to execute. If you batch your draw calls into fewer calls, you reduce the CPU overhead and increase draw call efficiency. Batch processing allows a group of draw calls to be quickly executed without any intervention from the CPU (driver or application) in a fire-and-forget method.
Some examples of batching primitives are:

- Branching in shaders may allow better batching since each branch can be grouped together for execution.
- For primitives like triangle strips, the developer can combine multiple strips that share the same state to save successive draw calls (and state changes) into a single batch call that uses the same state (single setup) for many triangles.
- Developers can also consolidate primitives that are drawn in close proximity to take advantage of spatial relationships. If the batched primitives are too far apart, it is more difficult for the application to effectively cull if they are not visible in the frame.

15.10 Perform calculations per vertex instead of per fragment/pixel

Since the number of vertices is usually much less than the number of fragments/pixels, it is cheaper to do per vertex calculations to save processing power.

15.11 Enable early-Z, hierarchical-Z and back face culling

Hardware support of depth testing to determine if objects are in the user's field of view are used to save workload and processing on vertex and pixel processing. If the object is in view, then the vertices are sent down the pipeline for processing. If the object is hidden or not viewable, the triangles are culled and not sent to the pipeline. This improves graphics performance since computations are only spent on visible objects. If the application already knows details about the contents and relative position of objects in the scene or screen, the developer can use that information to automatically bound areas that never need to be touched (for example an automotive application that has multiple layers of dials where parts of the underlying dials are occluded can have the application avoid occluded areas from the beginning). Another optimization is to perform basic culling on the CPU since the CPU has first-hand information about the scene details and object positions so it knows what scene data to send to the GPU.

15.12 Use branching carefully

Static branches perform well since states are known but they tend to use many general purpose registers. An example is a long shader that combines multiple shaders into a single, large shader that reduces state changes and batch draw calls. Dynamic branching has non-constant overhead since it processes multiple pixels as one and everything executes whether a branch is taken or not. In other words, dynamic branching goes through different permutations/branches in parallel to reach the correct results. If all pixels take the same path, then performance is good. The more pixels processed translates to higher overhead and lower performance. For dynamic branching, smaller pixel sizes/groups are optimal for throughput. Developers need to be aware of branching in their code to make sure excessive calculations and branches are efficient. Profiling tools can help determine if certain parts of code are optimized or not.

15.13 Do not use static or stack data as vertex data - use VBOs instead

A vertex buffer object (VBO) is a buffer object that provides the benefits of vertex array and display list and allows a substantial performance gain for uploading data (vertex position, color, normals, and texture coordinates) to the GPU. VBOs create buffer objects in memory and allow the GPU to directly access memory without CPU intervention (DMA). The memory manager can optimize buffer placement using feedback from the application. VBOs can also handle static and dynamic data sets and are managed by the Vivante driver. The benefits of each are:

- A vertex array reduces the number of function calls and allows redundant data to be shared between related vertices, instead of re-sending all the data each time. Access to data can be referenced by the array index.

- The display list allows commands to be stored for later execution and can be used repeatedly over multiple frames without re-transmitting data, thus minimizing CPU cycles to transfer data. The display list can also be shared by multiple OpenGL / OpenGL ES clients so they can access the same buffer with the corresponding identifier. If you put computationally expensive operations (ex. lighting or material calculations) inside display lists, then these computations are processed once when the list is created and the final result can be re-used multiple times without needing to re-calculate again.

If you combine the benefits of both by using VBO, the performance is enhanced over static or stack data sets.

15.14 Use dynamic VBO if data is changing frame by frame

Locking a static vertex buffer while the GPU is using it can create a performance penalty since the GPU needs to finish reading the vertex data from the buffer before it can return to the calling application. Locking and rendering from a static buffer many times per frame also prevents the GPU buffering render commands since it mush finish commands before returning the lock pointer. Without buffered commands the GPU remains idle until the application finishes filling the vertex buffer and issues the draw commands.
If the scene data never changes from frame to frame then a static buffer may be sufficient. With newer applications (ex. games, maps) that have dynamic viewports where vertex data changes multiple times per frame or frame-to-frame, then a dynamic VBO is required to ensure performance is still met. If the *current* buffer is being used by the GPU when a lock is called, a pointer to a *new* buffer location is returned to the application to ensure updated data is written to the *new* buffer. The GPU can still access the old data (current buffer) while the application puts updated data into the new buffer. The Vivante memory management unit and driver automatically take care of allocating, re-allocating, or destroying buffers.

You can implement dynamic VBO depending on your preference, but one recommendation is to allocate a 1 MB dynamic VBO block and upload data to using different offsets for each dynamic buffer. If the buffer overflows you can loop back and use location offset 0 again.

15.15 Tessellate your data so that Hierarchical Z (HZ) can do its job

We can break this into how OpenGL and OpenGL ES handle this use case.
OpenGL only renders simple convex polygons (edges only intersect at vertices with no duplicate vertices and only two edges meet at any vertex), in addition to points, lines, and triangles. If the application requires concave polygons (polygons with holes or intersecting edges), those polygons need to be subdivided into simple convex polygons, which is called tessellation (subdividing a polygon mesh into a bunch of smaller meshes). Once you have all the meshes in place our HZ hardware can automatically cull hidden polygons to efficiently process the frame, effectively breaking the frame into smaller chunks that can be processed very fast.
OpenGL ES only renders triangles, lines, and points. The same concepts apply as in OpenGL, which is to avoid very large polygons by breaking them down into smaller polygons where our internal GPU scheduler can distribute them into multiple threads to fully parallelize the process and remove hidden polygons.

15.16 Use dynamic textures as a texture cache (texture atlas)

The main reason for using dynamic textures as a cache is the application developer can create one larger texture that is subdivided into different regions (texture atlas). The application can upload data into each region and use an application side texture atlas to access the data. Each dynamic texture and sub-region can be locked, written to, and unlocked each frame, as needed. This method of allocating once is more efficient than using multiple smaller textures that need to be allocated, generated, and then destroyed each time.

15.17 If you use many small triangle strips, stitch them together

It is better to combine several small, spatially related triangle strips together into a larger triangle stip to minimize overhead and increase performance. For each triangle strip, there are overhead and start up costs that are required by the CPU and GPU, including state loads. If there are too many small triangle strips that need to be loaded, this impacts performance. An application developer can combine multiple triangle strips by adding a degenerate triangle to join the strips together. The overhead to restart multiple new strips is much higher than adding the degenerate triangle.

15.18 Specify EGL configuration attributes precisely

To obtain a 16 bit/pixel window buffer for rendering, the EGL config attributes need to be specified precisely according to the EGL spec. Specifying inaccurate EGL attributes may result in getting a 32-bit bit/pixel window buffer which doubles the bandwidth requirement for rendering which in turn leads to lower performance.

15.19 Use aligned texture/render buffers

The GPUs work on buffers with hardware-specific width/height alignment for better efficiency. Use the available API to query the GPU buffer alignment and allocate the texture / render buffers to satisfy these requirements, to avoid the cost of copies to aligned shadow memory.

15.20 Disable MSAA rendering unless high quality is needed

Although MSAA rendering can achieve higher image quality with smoother lines and triangle edges, it requires much higher (4x, 8x) bandwidth because it has to rendering a single pixel 4x/8x times. So, if high rendering quality is not required, MSAA should be disabled.

15.21 Avoid partial clears

Most GPUs have special hardware logic to do a fast clear of an entire buffer. So it is better to utilize the fast clear function to clear the entire buffer then render graphics again, instead of doing a partial clear to preserve a graphics region. If a partial clear is required by the application, make sure the clear area is aligned according to the GPU-specific requirements. Unaligned partial clears are expensive and should be avoided.

15.22 Avoid mask operations

Do not use mask unless the mask is 0 (other than when you need a specific render quality). Clearing a surface with mask (color /depth stencil mask) could have a performance penalty.Pixel mask operations are normally pretty expensive on some GPUs as the mask operation has to be done on every single pixel.

15.23 Use MIPMAP textures

MIPMAP textures enable the application to sample a lower resolution texture image (1/2, 1/4, 1/8, 1/16, ... size of the original texture image) when the triangle is rendering further away from the view point. Thus, the bandwidth required to read the texture image is reduced which leads to better performance.

15.24 Use compressed textures if constricted by RAM/ROM budget

Compressed textures are normally only a fraction (up to 1/8) of the original texture size. Using compressed textures reduces the storage requirements in memory and can also reduce the required texture upload bandwidth, when using a format that is supported natively by the hardware.
Compressed textures should not be chosen, if only for the purposes of reducing the memory bandwidth required for sampling of the texture during rendering. This is because due to a fixed read request size from the GPU, the memory controller load is the same as for an uncompressed texture.

15.25 Draw objects from near to far if possible

Drawing objects from near to far normally has better performance because the objects in the near foreground can block entire or partial objects in the background. Most GPUs have early Z rejection logic to reject the pixels that fail a Z compare. The GPU can skip fragment shader computations on these rejected pixels.

15.26 Avoid indexed triangle strips.

Index triangle strips can usually maximize the vertex cache utilization as each set of vertex data can be used in two triangles. There is however an errata in the GC2000 and GC880 GPUs which requires a SW conversion of indexed triangle strips to triangle lists in the driver. For small strips the conversion overhead is neglible, but for large geometries a different primitive type should be used.

15.27 Vertex attribute stride should not be larger than 256 bytes

Most Vivante GPUs provide native support for a 256 byte vertex attribute stride. If the vertex attribute stride is larger than 256 bytes, then the driver has to copy the vertex data around. Hardware versions v55 and higher (such as the GC7000L v55) support a 2048 byte vertex attribute stride as required in the OES3.1 spec.

15.28 Avoid binding buffers to mixed index/vertex array

Most of Vivante GPUs do not natively support mixed index/vertex arrays. So the Vivante driver must copy the index and vertex data around to form separate vertex data streams for the GPU. Avoid mixing index and vertex data so the driver does not have to incur a performance hit while performing this task.

15.29 Avoid using CPU to update texture/buffer contexts during render

Do not use the CPU to update texture/buffer contexts in the middle of rendering. Using the CPU to update texture/buffer causes the rendering pipeline to flush and stall, so that CPU can safely update the buffer contents. The pipeline flush/stall/resume causes significant performance impact.

15.30 Avoid frequent context switching

Context switch is an inherently expensive operation as many GPU states need to be reset to start a new rendering context. Thus, frequent context switching has a negative impact on application performance.

15.31 Optimize resources within a shader

Most GPUs have optimal support for a limited amount of resources (uniforms, varying, etc.). Using resources beyond the optimal working set causes the GPU to fetch/store resources from a lower performance memory pool and shader performance is negatively impacted.

15.32 Avoid using glScissor Clear for small regions

glScissor Clear for small regions (less than 16x8 aligned window) fall back to CPU so the performance is not optimal.

15.33 Use PRE to accelerate data transfer

PRE is an optimized hardware that can transform tiled format image to linear framebuffer. With PRE, GPU can onlyoutput tiled render target and has no need to resolve it. To enable the PRE feature, set the environment GPU_VIV_EXT_RESOLVE variable to 1; otherwise set it to 0. Its default value on the FB backend is 1, which means PRE is enabled by default on FB.

Warning: VG use cases can only output the linear format image. It is impossible to render linear and tiled format target to the same framebuffer at the same time. Therefor, when running 3D use cases with PRE and VG use cases together, there is garbage on the display. Besides, when running 3D use cases with PRE, the framebuffer format is changed from linear to tiled. It is the user's responsibility to convert the format back after the use cases end, or the display is abnormal when showing the FB console.

15.34 i.MX 8QuadMax dual-GPU performance

For some legacy applications with small texture/rendering size and less shader complex, dual-GPU performance may become worse than single GPU mode, because the driver needs to take more CPU effort for dual-GPU programming, and the driver overhead is more significant than GPU load in the hardware pipeline.
For such a kind of legacy case, the users can single-GPU to achieve better performance on the i.MX 8QuadMax.

Chapter 16 Demo Framework

16.1 Summaries

This document describes the Freescale Demo Framework, targeted at platform agnostic development of graphical demos. It covers the goals, architecture and instructions of how to use it across platforms, examples and best practices.

16.1.1 Executive summary

- Write a demo application once.
- Run it on Android, Yocto Linux, Ubuntu and MS Windows.[1]
- Easily portable to additional platforms.
- Supports: OpenGL ES2, OpenGL ES3, OpenVG and experimental G2D support.

16.1.2 Technical overview

- Written in a limited subset of C++11 and uses RAII to manage resources.
- Uses a limited subset of STL to make it easier to port.
- No copyleft restrictions from GPL / L-GPL licenses[2]
- Allows for direct access to the expected API's (EGL,ES2, ES3, VG)
- Provides optional helper classes for commonly used tasks
  - Matrix, Vector3, GLShader, GLTexture, etc
- Services
  - Keyboard & mouse
  - Persistent data manager
  - Assets management (models, textures)
- Defines a standard way for handling
  - Init, shutdown & window resize.
  - Program input arguments.
  - Input events like keyboard, mouse and touch.
  - Fixed time-step and variable time-step demo implementations.

[1] "Yocto Linux" or Yocto in the demo framework section means the BSP release.
[2] We don't use GPL or LGPL.

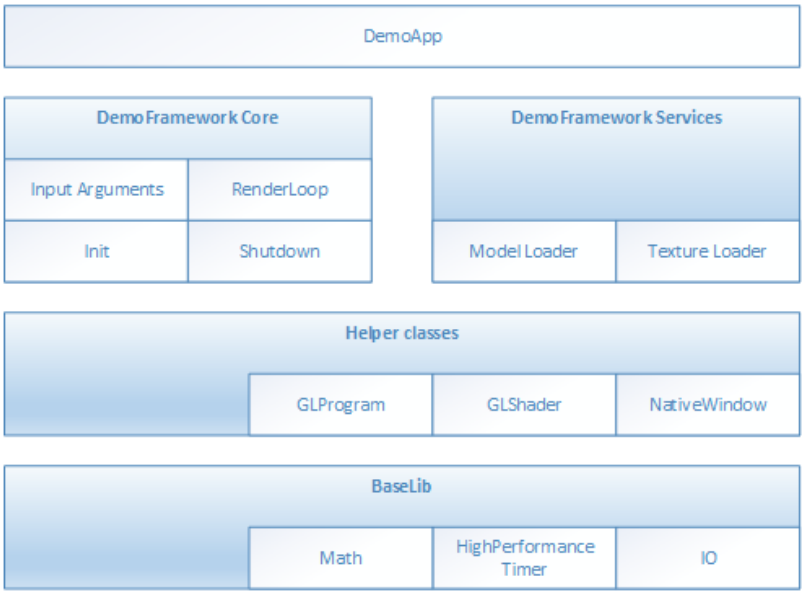
- Logging functionality.

16.2 Introduction

The Demo Framework is a multi-platform framework that enables demos to run on various platforms without any changes. The framework abstracts away all the boilerplate & OS specific code of allocating surfaces, creating the context, model loading, texture loading, shader compilation, render loop, animation ticks, benchmarking graph overlays etc. This allows the demo/benchmark developer to focus on writing rendering code. It also enables them to develop demos on PC or Android where the tool chain and debug facilities allows for faster turnaround time and then take the working code and deploy without code changes to the supported platforms. The platforms we currently support are Windows (for development via emulated backends), Android NDK and Linux with various windowing systems. The framework allows us to provide 'real' comparative benchmarks between the different OS and windowing systems we support, since we can run the exact same demo/benchmark code on them all.
The long term plans for the framework include extending it with support for other relevant API's.

16.3 Design overview

The framework is written in C++ and uses RAII[3] to manage resources. The resource management code focuses on 'ease of use' over raw performance, since it's mainly run on construction and destruction of the demo.



To allow the demo framework to be easily portable to new platforms its functionality is split into two parts: 'core' and 'services'. The core framework depends on a limited subset of STL to make it easier to port. Framework services come with their own set of library requirements. The model importer Assimp[4] requires boost to be available on the platform. Beside the demo framework core and demo framework services there is a set of helper classes for commonly used functionality, which makes it easier to write demo's for the API's we support. The helper classes do not depend on the demo framework and can be used in any program for the given API. For example, for OpenGL ES, there is a GLShader and GLProgram class which hides away the complexities of compiling the shader object and linking the program object and since they are RAII objects, they also clean up after themselves once you are done with them.

Since our primarily supported BSPs are linux based, we decided to utilize an input argument framework that is compatible with the standard Unix parameter format, like the one exposed by getopt[5].

[3] http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization

[4] http://assimp.sourceforge.net/

[5] We do however not utilize getopt to remain GPL free across platforms.

16.4 High level overview

The framework consists of three high level domains.

16.4.1 DemoMain

All the code that binds everything together and it is platform independent.

1. It gets the current demo setup
    a. Which demo host to utilize for the demo.
    b. Which demo app that needs to be run.
2. It parses the input arguments
3. It launches the demo host.
4. It logs any errors that might occur.

16.4.2 DemoHost

The demo-host is responsible for init & shutdown of the host environment and running the main loop.
The main loop utilizes the DemoAppManager to control the life of the DemoApp.
In other words, the DemoHost is the graphics API specific code needed to initialize and shutdown a given API and some code to run a render loop. All the API and platform independent code of the render loop resides inside the DemoAppManager class.
The exact capabilities of a DemoHost are also platform dependent. For example, some EGL implementations support running OpenVG and OpenGL ES, allowing a demo app to utilize both API's at once. This is not something that is supported by most windows emulation layers.

16.4.3 DemoApp

A demo application written for one or more specific APIs which are supported by a specific DemoHost. The demo is usually platform independent – the exception to the rule is if it depends on specific features that only exist on certain platforms.

16.5 Demo application details

The following description of the demo application details uses a GLES2 demo named 'S01_SimpleTriangle' as example. It lists the default methods that a demo should implement, the way it can provide customized parameters to the windowing system and how asset management is made platform agnostic.

16.5.1 Demo method overview

This is a list of the methods that every Demo App is most likely to override[6].

```
// Init
S01_SimpleTriangle(const DemoAppConfig& config)
// Shutdown
~S01_SimpleTriangle()
// OPTIONAL: Custom resize logic (if the app requested it). The default logic is to
// restart the app.
void Resized(const Point2& size)
// OPTIONAL: Fixed time step update method that will be called the set number of times
// per second. The fixed time step update is often used for physics.
void FixedUpdate(const DemoTime& demoTime)
// OPTIONAL: Variable time step update method.
void Update(const DemoTime& demoTime)
// Put the rendering calls here
void Draw(const DemoTime& demoTime)
```

When the constructor is invoked, the Demo Host API will already be setup and ready for use, the demo framework will use EGL to configure things as requested by your EGL config and API version.
It is recommended that you do all your setup in the constructor.
This also means that you should never try to shutdown EGL in the destructor since the framework will do it at the appropriate time. The destructor should only worry about resources that your demo app actually allocated by itself.

16.5.1.1 Resized

The resized method will be called if the screen resolution changes (if your app never changes resolution this will never be called)[7].

16.5.1.2 FixedUpdate

Is a fixed time-step update method that will be called the set number of times per second. The fixed time step update is often used for physics[8].

[6] See DemoFramework\FslDemoApp\include\FslDemoApp\ADemoApp.hpp for a complete list.
[7] This version of the framework always restart the app, so this will never be called.
[8] This version uses a fixed update frequency of 60 ticks per second. This will be configurable in the future.

16.5.1.3 Update

Will be called once before every draw call and you will normally update your animation using delta time.
For example if you need to move your object 10 units horizontally per second you would do something like

```
m_positionX += 10 * demoTime.DeltaTime;
```

16.5.1.4 Draw

Should be used to render graphics.

16.5.2 Fixed or variable timestep update

Depending on what your demo is doing, you might use one or the other - or both. It's actually a very complex topic once you start to dig into it, but in general anything that need precision and predictable/repeatable calculations, like for example physics, often benefits from using fixed time steps. It really depends on your algorithm and it's recommended to do a couple of google searches on fixed vs variable, since there are lots of arguments for both. It's also worth noting that game engines like Unity3D[9] support both methods.

16.5.3 Execution order of methods during a frame

The methods will be called in this order

- Events (if any occurred)[10]
- Resized[11]
- FixedUpdate (0-N calls. The first frame will always have a FixedUpdate call)
- Update
- Draw

After the draw call, a swap will occur.

16.5.4 Exit

The demo app can request an exit to occur, or it can be terminated via an external request.
In both cases one of the following things occur.

1. If the app has been constructed and has received a FixedUpdate, then it will finish its FixedUpdate, Update, Draw, swap sequence before its shutdown.
2. If the app requests a shutdown during construction, the app will be destroyed before calling any other method on the object (and no swap will occur).

The app can request an exit to occur by calling:

```
GetDemoAppControl()->RequestExit(1);
```

[9] http://unity3d.com/
[10] For an example of event handling see the "DemoApps\GLES2\InputEvents" sample.
[11] In this version of the framework this is never called as the app will be recreated on screen size changes (future versions will allow demo apps to handle resize events if they so desire)

16.5.5 Dealing with screen resolution changes

Per default the app is destroyed and recreated when a resolution change occurs[12].
It is left up to the DemoApp to save and restore demo specific state.

16.5.6 Content loading

The framework supports loading files from the Content folder on all platforms.

Given a content folder like this:

```
Content/Texture1.bmp
Content/Stuff/Readme.txt
```

You can load the files via the *IContentManager* service that can be accessed by calling

```
std::shared_ptr<IContentManager> contentManager = GetContentManager();
```

You can then load files like this:

Binary file:

```
std::vector<uint8_t> content;
contentManager->ReadAllBytes(content, "MyData.bin");
```

Text file:

```
const std::string content = contentManager-
>ReadAllText("Stuff/Readme.txt");
```

Bitmap file[13]:

```
Bitmap bitmap;
contentManager->Read(bitmap, "Texture1.bmp", PixelFormat::R8G8B8_UINT);
```

[12] Future versions will allow demo apps to handle resize events if they so desire.

[13] The current framework only png, bmp and jpeg images on all platforms but a few platforms has access to all formats supported by the DevIL library.

If you prefer to control the loading yourself, you can retrieve the path to the files like this:

```
IO::Path contentPath = contentManager->GetContentPath();
IO::Path myData = IO::Path::Combine(contentPath, "MyData.bin");
IO::Path readmePath = IO::Path::Combine(contentPath, "Stuff/Readme.txt");
IO::Path texture1Path = IO::Path::Combine(contentPath, "Texture1.bmp");
```

You can then open the files with any method you prefer.
Both methods work for all supported platforms.

For detailed information about how the content is handled on each platform, see the build guide appendixes.
The details of the available helper classes for a Demo Application are described in **Error! Reference source not found.**.

16.5.7 Demo registration

This is done in the S01_SimpleTriangle_Register.cpp file.

```
namespace Fsl
{
  namespace
  {
     // Custom EGL config (these will per default overwrite the custom settings.
     // However, an exact EGL config can be used)
     static const EGLint g_eglConfigAttribs[] =
     {
        EGL_SAMPLES, 0,
        EGL_RED_SIZE, 8,
        EGL_GREEN_SIZE, 8,
        EGL_BLUE_SIZE, 8,
        EGL_ALPHA_SIZE, 0, // buffers with the smallest alpha component size are preferred
        EGL_DEPTH_SIZE, 24,
        EGL_SURFACE_TYPE, EGL_WINDOW_BIT,
        EGL_NONE,
     };
  }

  // Configure the demo environment to run this demo app in a OpenGLES2 host environment
  void ConfigureDemoAppEnvironment(HostDemoAppSetup& rSetup)
  {
    DemoAppHostConfigEGL config(g_eglConfigAttribs);

    DemoAppRegister::GLES2::Register<S01_SimpleTriangle>(rSetup, "GLES2.S01_SimpleTriangle", config);
  }
}
```

Since the demo framework is controlling the main method, you need to register your application with the Demo Host specific registration call (in this case the OpenGL ES2 host), for the framework to register your demo class.

16.5.7.1 OpenGLES 3.X registration

To register a demo for OpenGLES 3.X you would use the GLES3 register method:

```
DemoAppRegister::GLES3::Register<S01_SimpleTriangle>(rSetup, "GLES3.S01_SimpleTriangle", config);
```

16.6 Demo playback

16.6.1 Command line arguments

All demos support various command line arguments.

Key	Function
-h	Show the command line argument help.
--Stats	Show a performance graph.
--LogStats	Log various stats to the console.
--ScreenshotFrequency	Create a screenshot at the given frame frequency.
--ExitAfterFrame	Exit after the given number of frames has been rendered
--ContentMonitor	Monitor the Content directory for changes and restart the app on changes. WARNING: Might not work on all platforms and it might impact app performance (experimental)

Use –h on a demo for a complete list

16.6.2 Demo single stepping / pause.

Under windows all samples support time stepping which can be useful for debugging. It might also be available on under platforms that support the given keys.

Key	Function
Pause	Pause the sample.
PageDown	Move forward one timestep.
Delete	Toggle between normal and Slow 2x playback
End	Toggle between normal and Slow 4x playback
Insert	Toggle between normal and fast 2x playback.
Home	Toggle between normal and fast 4x playback.

16.7 Helper Class Overview

16.7.1 FslBase

Provides basic functionality missing from C++ standard libraries.

16.7.1.1 Bits

BitsUtil	Utility methods for working with bits
ByteArrayUtil	Utility methods for reading and writing values from byte arrays in a specific endian format. This functionality is useful when working on platform independent load and save methods.

16.7.1.2 IO

Platform independent IO.

Directory	Helper methods for working on directories. • GetCurrentWorkingDirectory.
File	Helper methods for working with files • Checking if file exists. • File length. • Read all content from a file.
Path	A UTF8 path class and helper methods for working on it. • Combing paths. • Extracting directory or filename. • Getting the full path from a relative path.

16.7.1.3 Log

Platform independent logging.
Instead of using printf or std::cout to log information it's better to utilize the provided logging macro's since work across all supported platforms.

Log	Various logging macros • FSLLOG • FSLLOG_IF • FSLLOG_WARNING • FSLLOG_WARNING_IF • FSLLOG_ERROR • FSLLOG_ERROR_IF

16.7.1.4 Math

Mainly focused on math functionality useful for working with graphics. It focuses on ease of use instead of raw performance.

MathHelper	Various commonly used helper methods and constants like • PI • Clamping • Lerp • Conversions between radians and angles • PowerOfTwo
Matrix	Matrix helper methods like • Perspective • Rotate • Translate • Scale • Multiply
Point2	A 2D integer point.
Rectangle	A integer based rectangle with helper methods like • Union • Intersection
Vector2	A 2d float point with helper methods like • Dot • Length • Lerp • Min, max • Normalize • Reflect
Vector3	A 3d float point with helper methods like • Cross • Dot • Length • Lerp • Min, max • Normalize • Reflect • Transform by matrix
Vector4	A 4d float point with helper methods like • Dot • Length • Lerp • Min, max • Normalize • Reflect • Transform by matrix
Quaternion	Basic Quaternion operations.

16.7.1.5 String

Various string functionality

StringParseUtil	Various utility method for converting a string to a number.
UTF8String	A UTF8 string representation.

16.7.1.6 System

HighResolutionTimer	A platform independent high resolution timer.

16.7.2 FslGraphics

Bitmap	A RAII class to manage bitmap data.
BitmapUtil	Contains various helper methods that works on the bitmap class. • Horizontal flip • Pixel format conversion
Color	RGBA color utility class.
PixelFormat	Various standardized pixel formats supported by the bitmap classes.
RawBitmap	Read only bitmap information.
RawBitmapEx	Writeable access to bitmap information
RawBitmapUtil	Low level helper methods that work on RawBitmap's • Horizontal flip • Padding clear • Swizzle

16.7.2.1 Font

BasicFontKerning	Contains basic kerning information for a font.
BinaryFontBasicKerningLoader	Load basic kerning information from "fbk" files.
FontDesc	A very basic font description.
FontGlyphBasicKerning	Basic kerning for one glyph.
FontGlyphPosition	Position information for one glyph
FontGlyphRange	Font glyph range information.
IFontBasicKerning	Interface for extracting basic font kerning information.
TextureAtlasBitmapFont	Describes a bitmap font stored in a texture atlas.
TextureAtlasGlyphInfo	Texture atlas glyph information.

16.7.2.2 IO

BMPUtil	A simple helper class for loading and saving BMP images. It's not recommended to utilize it directly. Instead utilize the framework for loading images[14]. See **Error! Reference source not found.** for more details.

16.7.2.3 Render

AtlasFont	An atlas based bitmap font using a API independent texture.
AtlasTexture2D	An atlas based API independent texture.
BlendState	API independent blend states.
GenericBatch2D	A API independent 2D quad batcher.
Texture2D	A API independent texture representation.

16.7.2.4 TextureAtlas

AtlasTextureInfo	Represents information about one texture that is stored in a texture atlas.
BasicTextureAtlas	A simple manager for looking up AtlasTextureInfo.
BinaryTextureAtlasLoader	A "BTA" basic texture atlas loader.
ITextureAtlas	Simple interface for accessing texture information.
NamedAtlasTexture	A named atlas texture.
TextureAtlasHelper	A simple way to extract AtlasTextureInfo from a texture atlas.
TextureAtlasMap	A more performance efficient way to extract AtlasTextureInfo from a texture atlas.

[14] A future version will also add saving to the ContentManager.

16.7.2.5 Vertices

API independent vertex helper classes.

IndexConverter	Simple utility class to convert between index formats. It might not be efficient but it gets the job done.
VertexConverter	Simple utility class to convert between vertex formats. It might not be efficient but it gets the job done.
VertexDeclaration	Defines how a vertex is constructed in an API independent way.
VertexElementEx	Defines a vertex element
VertexPositionColor	A vertex comprised of • position • color.
VertexPositionColorNormalTexture	A vertex comprised of • position • color • normal • texture coordinates
VertexPositionColorTexture	A vertex comprised of • position • color • texture coordinates
VertexPositionNormalTexture	A vertex comprised of • position • normal • texture coordinates
VertexPositionTexture	A vertex comprised of • position • texture coordinates

16.7.2.6 Window

INativeWindow	An abstract from native windows.

16.7.3 FslUtil.OpenGLES2

RAII based helper classes for common GLES2 operations.

GLBatch2D	A specialization of GenericBatch2D GLES2.
GLBatch2DQuadRenderer	The GenericBatch2D backend for rendering quads.
GLCheck	Various helper macro's for checking and transforming OpenGL ES errors to exception.
GLFrameBuffer	A RAII based frame buffer encapsulation.
GLIndexBuffer	A RAII based index buffer. • uint8_t & uint16_t based index buffers. • Easy creation and update.
GlIndexBufferArray	A RAII based index buffer array. • Improved efficiency when allocating many index buffers of the same format.
GLProgram	A RAII based GL program encapsulation. • Vertex and fragment shader combination.
GLRenderBuffer	A RAII based GL render buffer encapsulation.
GLShader	A RAII based GL shader encapsulation. • Compilation and logging.
GLTexture	A RAII based GL texture encapsulation. • Can be created from either FslGraphics RawBitmap's or Bitmaps. • Easy content update. • Supports both normal and cubemap textures.
GLUtil	Contains various utility methods for OpenGL ES2 • Capture screenshots
GLVertexBuffer	A RAII based vertex buffer. • Easy creation and updating from Custom or FslGraphics.Vertices. • Helper methods for quickly enabling/disabling Attribs
GLVertexBufferArray	A RAII based vertex buffer array. • Improved efficiency when allocating many vertex buffers of the same vertex format.
NativeBatch2D	Extends GenericBatch2D with direct support for GLES2 native textures.
NativeTexture2D	Implements the INativeTexture2D for GLES2. This is used by the Batch2D system.

16.7.4 FslUtil.OpenGLES3

RAII based helper classes for common GLES3 operations.
GLES3 has the exact same helper classes as GLES2 and the following additions:

GLVertexArray	A RAII based vertex array. • Easy creation

16.7.5 FslUtil.OpenGLES3v1

RAII based helper classes for common GLES3.1 operation's.

GLProgramPipeline	A RAII based program pipeline encapsulation.
GLShaderProgram	A RAII based shader program encapsulation.

16.7.6 FslUtil.OpenVG

RAII based helper classes for common OpenVG operations.

VGPathBuffer	A RAII based path buffer • Easy creation
VGUtil	Contains various utility methods for OpenVG • Capture screenshots
VGCheck	Various helper macro's for checking and transforming OpenVG errors to exception.

16.7.7 FslGraphics3D

API independent descriptions of common 3D classes. This library is in development.
See the ModelLoaderBasics and ModelViewer samples for examples of how to use it.

Mesh	A basic mesh
Scene	A basic scene
SceneNode	A basic node in the scene

16.7.8 FslAssimp

The demo framework's Assimp integration. Provides various helper classes that make it easier to work with assimp in the framework.

MeshHelper	Helps to extract information from some assimp structures.
MeshImporter	Helps convert Assimp mesh structures to the FslGraphics3D ones.
SceneHelper	Extract basic information from a assimp scene.
SceneImporter	Helps convert Assimp scene structures to the FslGraphics3D ones.

16.7.9 FslGraphics3D.SceneFormat

Code to load and save a very basic portable scene format.

BasicSceneFormat	Load/save scene functionality.

16.7.10 FslSimpleUI

A new experimental UI framework that makes it easy to get a basic UI up and running. The main code is API independent. It is not a show case of how to render a UI fast but only intended to allow you to quickly get a UI ready that is good enough for a demo.
You can look at:

- DFSimpleUI100
- DFSimpleUI101
- TessellationSample

To see how it's used.
The next release of the framework should make it even easier to work with.

When working with the UI system its recommended to store all or at least the most used bitmaps in the same texture atlas. One commercially available texture packer is Texture Packer which can output a json file that we can convert to a binary format that can be loaded by the demo framework.
If you look at the DFSimpleUI100 sample, there is "OriginalContent/TextureAtlas" directory which contain a "MainAtlas.tps" file that can be loaded into texture packer. Pressing publish in texture packer produces a "MainAtlas.png" and "MainAtlas.json" file based on the files under "Main". The "MainAtlas.png" can be copied directly to the samples "Content" directory but the json file needs to be converted to a binary file. For this we included the TPConvert python script that can be run like this:

```
TPConvert MainAtlas.json -f bta1
```

This will then produce a "MainAtlas.bta" file that can be copied to the "Content" directory which contains all the needed atlas meta data.
Please beware that the default atlas is required to contain the default font as well. The documentation for creating the "MainAtlas.fbk" file has not been completed yet. The fbk file contains some basic font kerning information.

16.8 FslBuild scripts

For an easy to read text version of this document look in the demo framework "Doc/FslBuild_toolchain_readme.md" it contains more detailed information.

16.8.1 FslBuildGen.py

Is a cross-platform build-file generator. Which main purpose is to keep all build files consistent, in sync and up to date. See **FslBuildGen.docx** for details.

16.8.2 FslBuild.py

Extends the technology behind FslBuildGen with additional knowledge about how to execute the build system for a given platform.
So basically, FslBuild works like this

1. Invoke the build-file generator that updates all build files if necessary.
2. Filter the builds request based on the provided feature list.
3. Build all necessary build files in the correct order.

16.8.2.1 Useful arguments

FslBuild comes with a few useful arguments

--ListFeatures	List all features required by the build
--UseFeatures	Allows you to limit what's build based on a provided feature list. For example [EGL,OpenGLES2]. This parameter defaults to all features.
-t 'sdk'	Build all demo framework projects
-v	Set verbosity level
--	All arguments written after this is send directly to the native build system.

16.8.2.2 Important notes

- Don't modify the auto-generated files.
  The FslBuild scripts are responsible for creating all the build files for a platform and verifying dependencies. Since all build files are auto generated you can never modify them directly as the next build will overwrite your changes.
  Instead add your changes to the 'Fsl.gen' files as they control the build file generation!
- The 'Fsl.gen' file is the real build file.
- All include and source files in the respective folders are automatically added to the build files.

16.8.3 Build system per platform:

Android	gradle
Qnx	Make
Ubuntu	Make
Windows	Visual studio (IDE or nmake)
Yocto	make

16.9 Android SDK+NDK on windows build guide

For an easy to read text version of this document look in the demo framework “Doc/Setup_guide_android.md” it contains more detailed information.

16.9.1 Prerequisites:

- Read 16.8 so you know about the custom build system
- **IMPORTANT:** The way Gradle currently handles CMake builds on windows place some serious limits on the path length, so its recommended to either place the DemoFramework folder close to the root of the drive or to set the environment variable FSL_GRAPHICS_SDK_ANDROID_PROJECT_DIR to a directory close to the root of the drive.
- JDK (64 bit)
  IMPORTANT: Make sure to configure JAVA_HOME to point to the JDK directory
- Android SDK
  Once it’s installed it’s a good idea to run "SDK Manager.exe" and make sure everything is up to date.
  IMPORTANT: Android studio must be at least 3.1
  IMPORTANT: Get the android studio full package and enable the default packages.
  Configure the SDK manager
    - "SDK Platforms" add if necessary
      - Android 7.0 (Nougat)
    - "SDK Tools" add if necessary
      - CMake, LLDB, NDK, Android Support Repository

  IMPORTANT: Make sure to configure ANDROID_HOME to point to the android sdk directory
  IMPORTANT: Make sure to configure ANDROID_NDK to point to the android ndk directory
  IMPORTANT: Make sure you have at least android-ndk-r16b
- Python 3.4.x or better. We highly recommend at least 3.5+
  - For 64bit windows

16.9.2 Environment setup:

Android projects are generated to the path specified in the environment variable FSL_GRAPHICS_SDK_ANDROID_PROJECT_DIR. If it's not defined the 'prepare' script sets it to a default location.

1. Start a windows console (cmd.exe) in the DemoFramework folder.
2. Run the 'prepare.bat' file located in the root of the framework folder to configure the necessary environment variables and paths. Please beware that the prepare.bat file requires the current working directory to be the root of your demoframework folder to function (which is also the folder it resides in).

16.9.3 To Compile and run an existing sample application.

In this example we will utilize the GLES2 S06_Texturing app.

1. Make sure that you performed the environment setup.
2. Change directory to the sample directory:

```
cd DemoApps\GLES2\S06_Texturing
```

3. Build a app for Android using gradle + cmake

```
FslBuild.py -p android
```

If you just want to regenerate the cmake build files then you can just run

```
FslBuildGen.py -p android
```

If you want to save a bit of compilation time you can build for the ANDROID ABI you need by adding

```
FslBuildGen.py --Variants [ANDROID_ABI=armeabi-v7a]
```

or

```
FslBuild.py --Variants [ANDROID_ABI=armeabi-v7a]
```

16.9.4 To create a new GLES2 demo project named 'CoolNewDemo'

1. Make sure that you performed the environment setup.
2. Change directory to the GLES2 sample directory:

```
cd DemoApps/GLES2
```

3. Create the project template using the FslBuildNew.py script

```
FslBuildNew.py GLES2 CoolNewDemo
```

4. Change directory to the newly created project folder 'CoolNewDemo'

```
cd CoolNewDemo
```

5. Build a app for Android using gradle + cmake

```
FslBuild.py -p android
```

If you just want to regenerate the cmake build files then you can just run

```
FslBuildGen.py -p android
```

If you want to save a bit of compilation time you can build for the ANDROID ABI you need by adding

```
FslBuildGen.py --Variants [ANDROID_ABI=armeabi-v7a]
```

or

```
FslBuild.py --Variants [ANDROID_ABI=armeabi-v7a]
```

16.9.5 Using android studio.

1. Follow the instructions for "creating a new project" or "building an existing project".
2. As projects are generated to the path specified by the FSL_GRAPHICS_SDK_ANDROID_PROJECT_DIR environment variable you can locate the project there and open it with android studio. Be sure to open Android studio in a correctly configured environment. Here it could be a good idea to create a script for launching android studio with the right environment.

16.9.6 Linux notes:

- Install for private user and unzip android studio like this:

```
sudo unzip android-studio-ide_FILENAME.zip -d ~/sdk
cd ~/sdk/android-studio/bin
./studio.sh
```

- In the ui make sure to install the sdk in a directory you have access to for example

  ~/sdk/android-sdk-linux

16.9.7 Notes

16.9.7.1 Content

As long as you utilize one of the methods above to load the resources, you don’t really need to know the following. However if you experience problems it might be useful for you to know.
Under android builds we package all content using the Android 'assets' system. Since the system requires that the asset files are located under it's 'assets' folder (located at Android/assets in our samples) we utilize a one way folder synchronization utility called 'FslContentSync.py' to ensure that all files and directories under Content exist inside the asset folder as well. The synchronization script is automatically invoked during the android build process. To complicate things further the Android assets cannot normally be accessed via filenames using standard C/C++ methods. Because of this the assets are 'unpacked' on target to either the external or internal file system which allows us to open the files any way we like. Unfortunately this means that there will be a slight unpacking delay the first time a sample is executed.

16.9.7.2 Command line app building via Ant

http://developer.android.com/tools/building/building-cmdline.html

16.10 Ubuntu build guide

For an easy to read text version of this document look in the demo framework “Doc/Setup_guide_ubuntu16.04.md” it contains more detailed information.

16.10.1 Prerequisites:

- Read 16.8 so you know about the custom build system
- Ubuntu16.04 64 bit
- Build tools and xrand
  `sudo apt-get install build-essential libxrandr-dev`
- Python 3.4+
  It should be part of the default Ubuntu16.04 install.
- An OpenGL ES 2+ emulator
  - Mesa OpenGL ES 2
    `sudo apt-get install libgles2-mesa-dev`
  - Arm Mali OpenGL ES 3.0 Emulator V3.0.2 (64 bit)
    ```
    wget
    https://armkeil.blob.core.windows.net/developer/Files/downloads/open-gl-
    es-emulator/3.0.2/Mali_OpenGL_ES_Emulator-v3.0.2.g694a9-Linux-64bit.deb
    sudo dpkg -i Mali_OpenGL_ES_Emulator-v3.0.2.g694a9-Linux-64bit.deb
    ```
- DevIL
  - Developer's Image Library (DevIL)
    `sudo apt-get install libdevil-dev`
- Assimp
  Is now downloaded and build from source when needed. So its no longer necessary to run "sudo apt-get install libassimp-dev".

16.10.2 Environment setup:

1. Start a terminal (ctrl+alt t) in the DemoFramework folder
2. Run the 'prepare.sh' file located in the root of the framework folder to configure the necessary environment variables and paths. Please beware that the prepare.sh file requires the current working directory to be the root of your demoframework folder to function (which is also the folder it resides in).
   `source prepare.sh`

16.10.3 To Compile all samples

1. Make sure that you performed the environment setup

2. Compile

```
FslBuild.py -t sdk --BuildThreads 2
```

16.10.4 To Compile and run an existing sample application.

In this example we will utilize the GLES2 S06_Texturing app.

1. Make sure that you performed the environment setup
2. Change directory to the sample directory:

```
cd DemoApps/GLES2/S06_Texturing
```

3. Compile the project (a good rule of thumb for '--BuildThreads N' is number of cpu cores * 2)
   If you FslBuild without the --BuildThreads argument it will be set to 'auto' which uses your cpu core count.

```
FslBuild.py --BuildThreads 2
```

16.10.5 To create a new GLES2 demo project named 'CoolNewDemo'

1. Make sure that you performed the environment setup
2. Change directory to the GLES2 sample directory:

```
cd DemoApps/GLES2
```

3. Create the project template using the FslBuildNew.py.py script

```
FslBuildNew.py GLES2 CoolNewDemo
```

4. Change directory to the newly created project folder 'CoolNewDemo'

```
cd CoolNewDemo
```

5. Compile the project

```
FslBuild.py
```

Note:
Once a build has been done once you can just invoke the make file directly. However, this requires that you didn't change any dependencies or add files.
To do this run

```
make -j 2
```

If you add source files to a project or change the Fsl.gen file then run the FslBuildGen.py script in the project root folder to regenerate the various build files or just make sure you always use the FslBuild.py script as it automatically adds files and regenerate build files as needed.

16.10.6 NOTES:

16.10.6.1 Content

As long as you utilize one of the methods above to load the resources, you don't really need to know the following. However if you experience problems it might be useful for you to know.
The ubuntu build expects the content folder to be located at "<executable directory>/content". Since the binary is put in the sample root directory where the content folder is located, there should be no problem loading the resources.

16.10.6.2 Manual environment setup:

1. Configure your FSL_GRAPHICS_SDK to point to the downloaded sdk without the ending backslash:
```
export FSL_GRAPHICS_SDK=~/fsl/YourDemoFrameworkFolder
```
2. For easy access to the python scripts (not required for building)
```
PATH=$PATH:$FSL_GRAPHICS_SDK/.Config
```

16.10.6.3 Override platform auto-detection

To override the platform auto detection code set the following variable

```
export FSL_PLATFORM_NAME=Ubuntu
```

16.10.6.4 Executable location

The final executable will be placed in the root of the demo application folder. If it is moved the content folder (if it exist) needs to be copied to the same location.

16.11 Windows build guide

For an easy to read text version of this document look in the demo framework “Doc\Setup_guide_windows.md” it contains more detailed information.

16.11.1 Prerequisites:

- Read 16.8 so you know about the custom build system
- Visual Studio 2017 (community edition or better)
- Python 3.5.x or newer
    - For 64bit windows
- An OpenGL ES 2+ emulator
    - Arm Mali OpenGL ES Emulator 3.0.2.g694a9 (64 bit)
        - Please use the exact version (64bit) and use the installer to install it to the default location!
    - Vivante OpenGL ES Emulator

To get started its recommended to utilize the Arm Mali OpenGL ES 3.0.2 emulator (64 bit) which this guide will assume you are using.

16.11.2 Environment setup:

1. Start a windows console (cmd.exe) in the DemoFramework folder
2. Run the visual studio ```vcvarsall.bat x64``` to prepare your command line compiler environment for x64 compilation.
   - For VS2017 its often located here:
   "C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\VC\Auxiliary\Build\vcvarsall.bat" x64
3. Run the 'prepare.bat' file located in the root of the framework folder to configure the necessary environment variables and paths. Please beware that the prepare.bat file requires the current working directory to be the root of your demoframework folder to function (which is also the folder it resides in).

16.11.3 Compiling and running an existing sample application

In this example we will utilize the GLES2 S06_Texturing app.

1. Make sure that you performed the environment setup
2. Change directory to the sample directory:
   ```
   cd DemoApps\GLES2\S06_Texturing
   ```
3. Generate the build files
   ```
   FslBuildGen.py
   ```
4. Launch visual studio using the Arm Mali Emulator:
   ```
   .StartProject.bat arm
   ```
5. Compile and run the project (The default is to press F5)

To utilize a different emulator the .StartProject.bat file can be launched with the following arguments

arm	the arm mali emulator
powervr	the powervr emulator
qualcomm	the qualcomm andreno adreno emulator (expects its installed in "c:\AdrenoSDK”)
vivante	the vivante emulator

If it is launched without an argument it defaults to the arm emulator.

16.11.4 To create a new GLES2 demo project named 'CoolNewDemo'

1. Make sure that you performed the environment setup
2. Change directory to the GLES2 sample directory:
   ```
   cd DemoApps/GLES2
   ```
3. Create the project template using the FslBuildNew.py script
   ```
   FslBuildNew.py GLES2 CoolNewDemo
   ```
4. Change directory to the newly created project folder 'CoolNewDemo'
   ```
   cd CoolNewDemo
   ```
5. Generate build files for Android, Ubuntu and Yocto (this step will be simplified soon)
   ```
   FslBuildGen.py
   ```
6. Launch visual studio using the Arm Mali Emulator:
   ```
   .StartProject.bat arm
   ```
7. Compile and run the project (The default is to press F5) or start creating your new demo.

If you add source files to a project or change the Fsl.gen file then run the FslBuildGen.py script in the project root folder to regenerate the various build files.

16.11.5 Notes

16.11.5.1 Content

As long as you utilize one of the methods above to load the resources, you don’t really need to know the following. However, if you experience problems it might be useful for you to know.
The windows build expects the content folder to be located at "<current working directory>/content". When you launch the sample via the visual studio project the current working directory will be equal to the sample root directory where the content folder is located, so there should be no problem loading the resources.

16.11.5.2 Switching between emulators

The visual studio projects have been configured so that emulator builds can co-exist without interfering with each other. Furthermore, the only the emulator dependent parts will be rebuild when changing emulator.
So all in all it ought to be very fast to switch between emulators.

16.11.5.3 Executable location

The executable location is based upon the build type release/debug and which emulator you are using and
So the executable for a demo called S06_Texturing build as debug and using the arm emulator will be located under

bin\S06_Texturing\Debug_ARM\

The content folder is located at

Content

If you want to move them then make sure that both the S06_Texturing.exe and Content folder is moved to the same location like this:

S06_Texturing.exe
Content

16.12 Yocto build guide

First you need to decide how you are going to be building for Yocto.

- Building using a prebuild Yocto SDK
- Building using a full Yocto build

For an easy to read text version of this document look in the demo framework “Doc/Setup_guide_yocto.md” it also more detailed information.

16.12.1 Building using a prebuild Yocto SDK

Building using a prebuild Yocto SDK and a prebuild sd-card image.
This tend to be the fastest way to get started.

16.12.1.1 Prerequisites

- Read Appendix2 so you know about the custom build system.
- Ubuntu 16.04
- Python 3.5 (this is standard in Ubuntu 16.04)
- A prebuild sdk for your board typically called something like ‘toolchain.sh’
- A prebuild sd-card image for your board typically called ‘BoardName.rootfs.sdcard.bz2’
- Git: sudo apt-get install git

For this guide we will assume you are using a FB image.

- Download the DemoFramework source using git.

It's also a good idea to read the introduction to the FslBuild toolchain in “Doc/FslBuild_toolchain_readme.md”

16.12.1.2 Preparing a Yocto SDK build

1. Start a terminal (ctrl+alt t)
2. Install the sdk:
```
./fsl-imx-internal-xwayland-glibc-x86_64-fsl-image-gui-aarch64-toolchain-4.9.51-mx8-beta.sh
```
Chose where to install it, you can use the default location or a location of your choice.
For this example, we use "`~/sdk/4.9.51-mx8-beta`".
When the setup is complete it will list the configuration script you need to run to configure the sdk environment.
Something like this
```
$ . ~/sdk/4.9.51-mx8-beta/environment-setup-aarch64-poky-linux
```
Each time you wish to use the SDK in a new shell session, you need to source the environment setup script e.g.
3. Your SDK is now installed.

16.12.1.3 Yocto SDK environment setup

1. Start a terminal (ctrl+alt t)
2. Prepare the yocto build environment by running the config command you got during the sdk install
```
. ~/sdk/4.9.51-mx8-beta/environment-setup-aarch64-poky-linux
```
3. You should now be ready to build using the demo framework. However, if you experience issues with the ‘`prepare.sh`’ script you can help it out by defining the platform name and the location of the root fs
```
export FSL_PLATFORM_NAME=Yocto
export ROOTFS=~/sdk/4.9.51-mx8-beta/sysroots/aarch64-poky-linux
```
Another possible error you can encounter is that the FslBuild.py scripts fail to include the 'typing' library.

This can happen because the SDK comes with a too old Python3 version or a incomplete Python3.5 version.
As a workaround for that you could delete the Python3 binaries from the SDK which will cause it to use the system Python3 version instead.

16.12.1.4 Ready to build

You are now ready to start building Yocto apps using the demo framework.
Please continue the guide at “Using the demo framework”.

16.12.2 Building using a full Yocto build

Building using a full manually build Yocto build.
This process provides the most flexible solution but it also takes significantly longer to build the initial Yocto sdcard and toolchain.

16.12.2.1 Prerequisites:

- Read 16.8 so you know about the custom build system.
- The Ubuntu version required by the BSP release.
- Python 3.4 or newer
  It should be part of the default Ubuntu install.
  If you use 3.4 you need to install the 'typing' library manually so we highly recommended using 3.5 or newer.
  To install the typing library in Python **3.4** run:
  ```
  sudo apt-get install python3-pip
  sudo pip3 install typing
  ```
- A working yocto build
  For example, follow one of these:
  - http://git.freescale.com/git/cgit.cgi/imx/fsl-arm-yocto-bsp.git/
  - https://community.freescale.com/docs/DOC-94866

For this guide we will assume you are using a FB image.

- Download the DemoFramework source using git.
- It's also a good idea to read the introduction to the FslBuild toolchain at Doc/FslBuild_toolchain_readme.md

16.12.2.2 Preparing a Yocto build

Before you build one of these yocto images you need to

1. Run the yocto build setup (X11 example).
   ```
   MACHINE=imx6qpsabresd source fsl-setup-release.sh -b build-x11 -e x11
   ```
3. Bake
   bitbake fsl-image-gui
   bitbake meta-toolchain
   bitbake meta-ide-support

You can now build one of the images below (or a custom one)

x11 yocto image

Example:

```
MACHINE=imx6qpsabresd source fsl-setup-release.sh -b build-x11 -e x11
bitbake fsl-image-gui
bitbake meta-toolchain
bbitbake meta-ide-support
```

Extracted rootfs

We assume your yocto build dir is located at `~/fsl-release-bsp/build-x11` and
that the rootfs will be unpacked to `~/unpacked-rootfs/build-x11` and

the image is called `fsl-image-gui-imx6qpsabresd.rootfs.tar.bz2` (you will need to locate your image name) `runqemu-extract-sdk ~/fsl-release-bsp/build-x11/tmp/deploy/images/imx6qpsabresd/fsl-image-gui-imx6qpsabresd.rootfs.tar.bz2 ~/unpacked-rootfs/build-x11`

FB yocto image
 	
Example:
 	
`ACHINE=imx6qpsabresd source fsl-setup-release.sh -b build-fb -e fb`
`bitbake fsl-image-gui`
`bitbake meta-toolchain`
`bitbake meta-ide-support`
 	
Extracted rootfs
 	
We assume your yocto build dir is located at `~/fsl-release-bsp/build-fb` and
that the rootfs will be unpacked to `~/unpacked-rootfs/build-fb` and
the image is called `fsl-image-gui-imx6qpsabresd.rootfs.tar.bz2` (you will need to locate your image name)
 	
`runqemu-extract-sdk ~/fsl-release-bsp/build-fb/tmp/deploy/images/imx6qpsabresd/fsl-image-gui-imx6qpsabresd.rootfs.tar.bz2 ~/unpacked-rootfs/build-fb`

Wayland yocto image

Example:

```
MACHINE=imx6qpsabresd source fsl-setup-release.sh -b build-wayland -e wayland
bitbake fsl-image-gui
bitbake meta-toolchain
bitbake meta-ide-support
```

Extracted rootfs

We assume your yocto build dir is located at `~/fsl-release-bsp/build-wayland` and
that the rootfs will be unpacked to `~/unpacked-rootfs/build-wayland` and
the image is called `fsl-image-gui-imx6qpsabresd.rootfs.tar.bz2` (you will need to locate your image name)

```
runqemu-extract-sdk ~/fsl-release-bsp/build-
wayland/tmp/deploy/images/imx6qpsabresd/fsl-image-gui-
imx6qpsabresd.rootfs.tar.bz2  ~/unpacked-rootfs/build-wayland
```

For this guide we will assume you are using an FB image.

16.12.2.3 Yocto environment setup:

Prepare the yocto build environment

```
pushd ~/fsl-release-bsp/build-fb/tmp
source environment-setup-cortexa9hf-neon-poky-linux-gnueabi
export ROOTFS=~/unpacked-rootfs/build-fb
export FSL_PLATFORM_NAME=Yocto
popd
```

16.12.2.4 Ready to build

You are now ready to start building Yocto apps using the demo framework.
Please continue the guide at “Using the demo framework”.

16.12.3 Using the demo framework

1. Make sure that you performed the Yocto environment setup for your chosen Yocto environment.
   - SDK build [Yocto SDK environment setup]
   - Custom build [Yocto environment setup].cd to the demoframework folder
2. cd to the demoframework folder
3. Run the 'prepare.sh' file located in the root of the framework folder to configure the necessary environment variables and paths. Please beware that the prepare.sh file requires the current working directory to be the root of your demoframework folder to function (which is also the folder it resides in).

   ```
   source prepare.sh
   ```

   also verify that the script detect that you are doing a Yocto build by outputting

   ```
   PlatformName: Yocto
   ```

   If it doesn't you can override the platform auto detection by setting the environment variable

   ```
   export FSL_PLATFORM_NAME=Yocto
   ```

   Before running the prepare.sh script.

16.12.4 To Compile all samples

1. Make sure that you performed the demo framework environment setup
2. Compile everything

   ```
   FslBuild.py --Variants [WindowSystem=FB] -t sdk
   ```

   WindowSystem can be set to either: FB, Wayland or X11

16.12.5 To Compile and run an existing sample application.

In this example we will utilize the GLES2 S06_Texturing app.

1. Make sure that you performed the demo framework environment setup.
2. Change directory to the sample directory:

   ```
   cd DemoApps/GLES2/S06_Texturing
   ```

3. Compile the project

   ```
   FslBuild.py --Variants [WindowSystem=FB]
   ```

   WindowSystem can be set to either: FB, Wayland or X11

16.12.6 To create a new GLES2 demo project named 'CoolNewDemo'

1. Make sure that you performed the demo framework environment setup.
2. Change directory to the GLES2 sample directory:

```
cd DemoApps/GLES2
```

3. Create the project template using the FslBuildNew.py script

```
FslBuildNew.py GLES2 CoolNewDemo
```

4. Change directory to the newly created project folder 'CoolNewDemo'

```
cd CoolNewDemo
```

5. Compile the project

```
FslBuild.py --Variants [WindowSystem=FB]
```

   WindowSystem can be set to either: FB, Wayland or X11

Note:
Once a build has been done once you can just invoke the make file directly. However, this requires that you didn't change any dependencies or add files. To do this run

```
make -f GNUmakefile_Yocto -j 2 WindowSystem=FB
```

If you add source files to a project or change the Fsl.gen file then run the FslBuildGen.py script in the project root folder to regenerate the various build files or just make sure you always use the FslBuild.py script as it automatically adds files and regenerate build files as needed.

16.12.7 NOTES:

16.12.7.1 Content

As long as you utilize one of the methods above to load the resources, you don't really need to know the following. However, if you experience problems it might be useful for you to know.
The Yocto build expects the content folder to be located at "<executable directory>/content".

16.12.7.2 Manual environment setup:

Configure your FSL_GRAPHICS_SDK to point to the downloaded sdk without the ending backslash:

```
export FSL_GRAPHICS_SDK=~/fsl/YourDemoFrameworkFolder
```

1. For easy access to the python scripts

```
PATH=$PATH:$FSL_GRAPHICS_SDK/.Config
```

16.12.7.3 Override platform auto-detection

To override the platform auto detection code set the following variable

```
export FSL_PLATFORM_NAME=Yocto
```

16.12.7.4 Building for multiple backends

The makefiles have been configured so that the builds for all backends can co-exist without interfering with each other. Furthermore the only the backend dependent parts will be rebuild when changing backend.
So all in all it ought to be very fast to switch between backends.
The demo app executables will be post fixed with the backend its build for to ensure no conflicts occurs.

A1.1.1. Executable location

The final executable will be placed in the root of the demo application folder. If it is moved the content folder (if it exist) needs to be copied to the same location.
The executables follows this naming scheme:

<DemoAppName>_<BackendName>[<TargetPostFix>]

So a debug build of S06_Texturing for the FB backend will be called

S06_Texturing_FB_d

A release build of S06_Texturing for the X11 backend will be called

S06_Texturing_X11

16.13 FslContentSync.py notes

- Does not copy files that start with a '.' in its file or directory name.
- Does not allow files to contain ".." in its name.
- Do **not** utilize file names that only differ by casing like this:
    - Shader.txt
    - shader.txt
- Due to the android asset packer it's not recommended to use Unicode file names as they are unsupported by the android tool at the moment.

16.14 Known limitations

16.14.1 General

- Android, Ubuntu and Windows OpenVG support is considered experimental for this release.
- G2D support is experimental and it’s not recommended to use it yet.

16.14.2 Android

- Android does not handle Unicode file names inside the 'content' folder. So do not utilize Unicode for filenames stored in Content. The culprit is the android assets folder which we utilize for content files.

16.14.3 Ubuntu

- OpenGLES3 is currently unsupported on Ubuntu, as we rely on the Mesa 3D graphics library for OpenGLES emulation.
- OpenVG is emulated via the Mesa 3D graphics library and it might contain unsupported features.

16.14.4 Windows

- OpenVG is emulated via the Mesa 3D graphics library and it might contain unsupported features.

16.15 Upgrading samples from earlier SDKs

To convert a sample to the newest sdk start at the SDK version you are using and upgrade the app one step at a time. So a 2.0 app needs to be updated to 2.1 before it can be updated to 2.2.

16.15.1 From 2.0 to 2.1

Since version 2.1 contains minor incompatibilities with 2.0, any existing application will have to be upgraded. The easiest way to upgrade a sample is to rename the old directory, then run

- `FslNewDemoProject.py all -t <type> <name>`
- `cd <name>`
- `FslBuildGen.py`

Then do a two way merge of the old source directory and the new one. If any dependencies were manually added to Fsl.gen in the sample, they will have to be re-added to the new one.
Then run

- `FslBuildGen.py`

The project should now be converted.

16.15.2 From 2.1 to 2.2

V2.1 can easily be upgraded to 2.2, just run FslBuildGen.py to update it.

16.15.3 From 2.2 to 2.3

V2.2 can easily be upgraded to 2.3, just run FslBuildGen.py to update it.

16.16 What's new

Version 5.1

- All ThirdParty code is now downloaded as needed instead of being included in the repo.
- Windows builds now default to Visual Studio 2017 instead of 2015.
- Basic support for changing the color-space via EGL.
- Examples of how to setup SRGB and HDR framebuffers.
- HDR to LDR display rendering examples with various basic tone-mapping algorithms.
- Vulkan enabled for the Yocto Wayland backend.
- Assimp upgraded to 4.1 on most platforms.
- GLES3.ColorspaceInfo
- GLES3.EquirectangularToCubemap
- GLES3.GammaCorrection demo.
- GLES3.HDR01_BasicToneMapping
- GLES3.HDR02_FBBasicToneMapping
- GLES3.HDR03_SkyboxTonemapping
- GLES3.HDR04_HDRFramebuffer
- GLES3.MultipleViewportsFractalShader demo.
- GLES3.Scissor101
- GLES3.Skybox
- GLES3.SRGBFramebuffer
- GLES3.TextureCompression demo.
- Vulkan.VulkanInfo demo.
- Android build now requires Android Studio 3.1 and the Android NDK16b or newer.

Version 5.0.1

- OpenVX.SoftISP demo.
- OpenCL.SoftISP demo.

Version 5.0

- Tools now require Python 3.4+ instead of python 2.7
- FslBuildNew script that can help you create a new project fast.
- Vulkan support is much closer to its final state.
- The application registration method has been changed so it's more future proof and allow for greater customization.
- Prebuild binaries have been removed.
    - FslImageConvert.exe was removed as we now support saving screenshots directly in jpg.
    - Prebuild windows libraries removed as we now download and build them on demand instead.
- The directory structure was updated to make it simpler.
- Some tags in Fsl.gen xml files were deprecated.
- Gamepad support.
- New libraries
    - Stb, xinput, perfcounters.

Version 4.0

- First public release on github.

- Early access support for Vulkan, OpenCL, OpenCV and OpenVX.
    - Vulkan samples.
    - OpenVX samples.
    - OpenCL samples.
    - OpenCV samples.
- New libraries
    - GLI 0.8.10, GLM 0.9.7.6
- PixelFormats are now compatible with the vulkan pixel formats.
- FslBuild.py script introduced as a simple unified way to build on all platforms if so desired. It's still possible to build using the native platform method.
- FslBuild scripts now support limited feature based filtering.
- Introduced a content pipeline to help build vulkan shaders.
- Windows builds
    - Visual Studio 2015 is now the default environment instead of 2013
    - We now use the OpenVG reference implementation to emulate OpenVG.

Version 2.3

- OpenGLES 3.1 support.
- A new ContentMonitor can reload your sample when it detects changes to the content folder (this does not work on Android). This allows for rapid prototyping on most platforms.
- New samples:
    - DFSimpleUI101, ModelLoaderBasics, ModelLoaderViewer, Tessellation101, TessellationSample.
- New libraries:
    - FslAssimp, FslGraphics3D, FslSceneFormat, FslSimpleUI, FslGraphicsGLES3v1
- New experimental UI framework intended to quickly create a UI for your sample app.
- Assimp support on most platforms. It is not supported on Android here we recommend using the FslSceneFormat instead. In general, it will be much more efficient to preprocess your model on a fast platform like a PC and save it in the FslSceneFormat instead of doing it on relatively slow target platform.
- Experimental support for generating Visual Studio 2015 projects (see the FslBuildgen documentation for details).
- Content loader for Binary texture and basic font kerning information.
- Windows PowerVR OpenGLES emulation support.

Version 2.2

- Demo content can now be stored in bmp, png and jpeg format on all platforms.
    - Some platforms support extra formats via the DevIL image library.
- Onscreen performance graph support that can be augmented with custom data.
- Pause and single stepping during demo playback.
- Added infrastructure that allows samples to share a library. See DemoApps/Shared for example libraries.
- Lots of new samples.
    - The Blur, FractalShader, FurShellRendering and DirectMultiSamplingVideoYUV are functional but experimental.
- Experimental G2D support.
- Experimental NativeBatch2D support under 3D api's. See the DFNativeBatch2D samples for an example of how it works.

- Experimental –mmdc parameter for Yocto builds. If it shows the wrong info then run mmdc2 before running the sample as it will reset things correctly.

Version 2.1

- OpenVG support.
- OpenVG examples
- Examples: T3DstressTest for GLES2 + GLES3
- Most samples were upgraded to use the Content system to load their shaders and graphics.
- All samples now support the following arguments
    - –LogStats = Log basic rendering stats
    - –ScreenshotFrequency <frequency> = Create a screenshot at the given frame frequency (Not supported for OpenVG).

Chapter 17 Environment Variables Summary

The table below lists the environment variables (ENV) available in the GPU drivers.
The use of most environment variables remains static from driver version to driver version, but sometimes these variables need refinements to meet new, advanced conditions not present with the ENV initially introduced.

17.1 Environment variable for drivers and HAL

Table 35. Environment variables for drivers and HAL

ENV name	Backends suported	Note
FB_IGNORE_DISPLAY_SIZE	FB/WLD	0: Clip window to device display size. 1: Do not clip window to the device limits for width and height.
FB_MULTI_BUFFER	FB/WLD	Number of backend buffers of the framebuffer device. For WLD, define the multibuffer number of Weston.
FB_FRAMEBUFFER_N	FB/WLD	Define the Nth framebuffer device.
FB_LEGACY	FB	If board doesn't support drm-fb, ignore this variable. 0: GPU render through drm 1: GPU directly render to framebuffer.
VG_APITIME	FB/WLD/X11	Enable VG API function execution time print.
VIV_MGPU_AFFINITY	FB/WLD/X11	Control the multiple GPUs affinity configuration. Possible value: • Not defined or defined as "0" GPUs work in GPU_COMBINED mode. • 1:0 GPUs work in GPU_INDEPEDNENT mode, GPU0 is used. • 1:1 GPUs work in GPU_INDEPEDNENT mode, GPU1 is used.
VIV_DEBUG	FB/WLD/X11	Define the user debug message level (-MSG_LEVEL: ERROR/WARNING).
VIV_FBO_PREFER_MEM	FB/WLD/X11	Renderbuffer is not freed after colorbuffer detaches from FBO (GL ES 2.0)
VIV_DISABLE_HZ	FB/WLD/X11	This variable can be specifically enabled for i.mx6d/q to avoid gpu hang with occlusion query in ES30, because of gpu hardware problem HBN1246
GPU_VIV_EXT_RESOLVE	FB/WLD/X11	Enable the external resolve mode (1 by default for FB).
GPU_VIV_DISABLE_SUPERTILED_TEXTURE	FB/WLD/X11	Disable supertiled texture (64x64 tiled texture is not used).
GPU_VIV_DISABLE_CLEAR_FB	FB/WLD/X11	Enable clear buffer when a new Window surface is created.
GPU_VIV_WL_MULTI_BUFFER	WLD	Define the client multibuffer number.
DRI_IGNORE_DISPLAY_SIZE/ X_IGNORE_DISPLAY_SIZE	X11	0: Clip window to device display size. 1: Do not clip window to the device limits for width and height.
__GL_DEV_FB	X11	Set the path for framebuffer device like /dev/fb0.
LIBGL_ALWAYS_INDIRECT	X11	Make OGL go into indirect mode. All rendering is done by XserverSet.
LIBGL_DEBUG	X11	Print error message to stderr if LIBGL_DEBUG env var is set. Print info message to stderr if LIBGL_DEBUG env var is set to "verbose".
VIV_PROFILE	vProfiler	Enable profiler. Different level results generate different results.
VP_COUNTER_FILTER	vProfiler	Used to control profile different system resource like memory/CPU time usage.
VP_FRAME_END	vProfiler	When VIV_PROFILE=3, specify the frame to end profiling with vProfiler.
VP_FRAME_NUM	vProfiler	When VIV_PROFILE=1, used to specify the number of frames dumped by vProfiler.

VP_FRAME_START	vProfiler	When VIV_PROFILE=3, specify the frame to start profiling with vProfiler.
VP_OUTPUT	vProfiler	Specify the output file name of vProfiler (default is vprofiler.vpd).
VP_PROCESS_NAME	vProfiler	Choose profiler enable process (This option is only available for Android platform, not available for Linux OS).
VP_SYNC_MODE	vProfiler	Enable [1] or disable [0] the synchronous mode of vProfiler (default is synchronous enabled).
VP_USE_GLFINISH	vProfiler	Use glFinish as the frameEnd.
VIV_TRACE	vTracer	Enable tracer. Different levels could generate different logs.

17.2 Environment variable for compiler

Table 36. Environment variables for compiler

ENV NAME	Compiler	Note
VC_DUMP_SHADER_SOURCE	GLSLC/VSC	Enable dumping the shader source code.

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.

NXP, the NXP logo, Freescale, and the Freescale logo are trademarks of NXP B.V. All other product or service names are the property of their respective owners.

Arm, the Arm logo, and Cortex are registered trademarks of Arm Limited (or its subsidiaries) in the EU and/or elsewhere. All rights reserved.
© 2018 NXP B.V.

Document Number: IMXGRAPHICUG
Rev. 0
06/2018