SGX OpenGL ES 1.1 Reference Driver

Software Functional Specification

Filename	:	SGX OpenGL ES 1.1 Reference Driver.Software Functional Specification.doc
Version	:	1.0.98 External Issue
Issue Date	:	12 Aug 2011
Author	:	PowerVR

Contents

List of Figures

Glossary of Terms

Khronos Group	Body that controls the OpenGL ES, OpenVG and OpenMAX specifications
EGL 1.4	The Native Platform Graphics Interface implemented for this project.
OpenGL ES™ 1.1	The fixed function 3D API implemented for this project OpenGL ES is a trademark of Silicon Graphics, Inc.
SGX	IP that provides 2D/3D/Video acceleration

Table 1

1. Introduction

1.1. Scope

This document describes the high level design and functional specification of the OpenGL ES 1.1 reference SGX device driver.

1.2. Related Documents

OpenGL ES 1.1 Full Specification, Khronos Group, Version 1.1.10
EGL 1.4 Specification, Khronos Group, Version 1.0

2. Overview

OpenGL ES 1.1 is an API which, primarily through sub-setting of OpenGL, attempts to provide an API that can be efficiently implemented within the constraints of an embedded platform, but still retain the core functionality and feature set of OpenGL, giving application developers an immediate familiarity with the API.

The SGX OpenGL ES 1.1 reference driver will be designed and constructed based on knowledge developed in the following processes:

- Deep experience of the OpenGL and OpenGL ES standards, including interaction with the ARB, and promoter level membership of the Khronos Group, which controls and extends the OpenGL ES standard.
- Development experience of full featured OpenGL ES 1.1 implementations.
- Porting of these implementations and associated services to multiple disparate operating and windowing systems.
- Development of OpenGL and OpenGL ES implementations for multiple generations of PowerVR hardware.
- Active interaction with the Khronos group's OpenGL ES standards development work.

This experience allows the development of a reference driver with sufficient modularity to ensure rapid portability to different environments, but sufficient internal coherence to ensure that performance is not sacrificed to modularity.

2.1. Functionality Summary

The following table lists the functionality summary of the OpenGL ES 1.1 reference driver with respect to the OpenGL ES 1.1 specification. Functionality beyond the OpenGL ES 1.1 specification is described in the section on Extensions. Functionality that falls short of the OpenGL ES 1.1 specification is described in the section on Conformance.

Functionality	Description
General	
Profiles	Common and Common lite V1.1
EGL	
Render Surfaces	Pbuffers + Pluggable Window System surfaces
Render Formats	Supports 565, 1555, 4444 and 8888 RGB(A) formats
Multiple Context	Support for concurrent 3D applications
Vertex Processing	
Projection Matrix	Yes
Modelview Matrix	Yes
Texture Matrix	Yes
Viewport Clipping	Yes
Polygon offset	Yes
Light Modes	All light modes
Number of Lights	8
Rasterisation	
Multi-Sampling	Support for x4 anti-aliasing
Fogging	Vertex fog support
Hidden Surface Removal	Z buffering with all compare modes and polygon offset

Functionality	Description
Polygon Shading	Flat and Smooth shading of polygons
Polygon culling	Front and Back face polygon culling
Primitives	Native support for indexed triangle and line strips, points, fans, and triangle lists
Multi-Texturing	Supports up to 4 textures (2k x 2k) per object with general blending
Texture Blend modes	All fixed function blend modes
Texture Wrap Modes	All wrap modes
Texture filter modes	All modes
Texture Formats	All formats
Blending	All render target blend modes
Logical Operations	All render target logical operations
Color Mask	Full color mask support
Alpha Testing	All Alpha test modes
Extensions	See Section 4

2.2. Software Module Description

The following diagram ‘Figure 1 OpenGL ES 1.1 Driver Architecture Overview’ details an overview of the architecture of the OpenGL ES 1.1 reference driver.

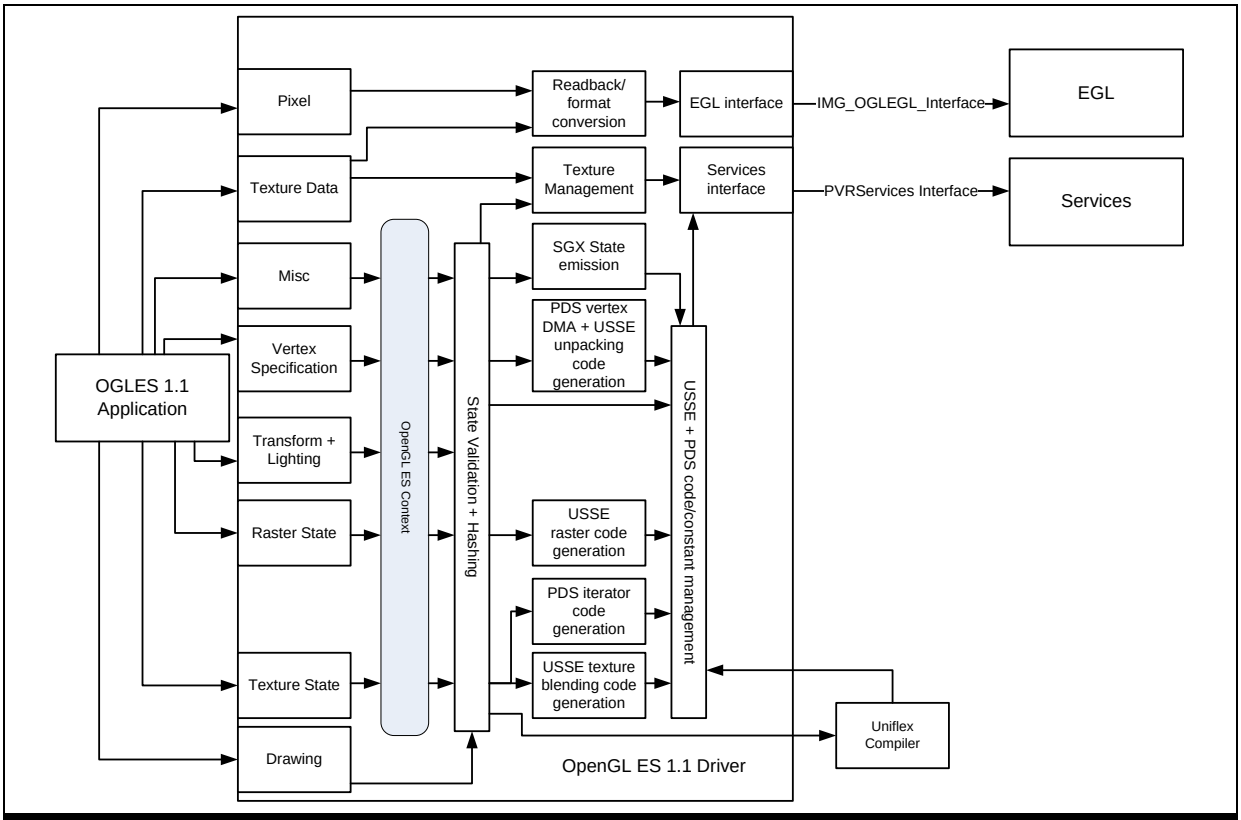

Figure 1 OpenGL ES 1.1 Driver Architecture Overview

2.2.1. OpenGL ES 1.1 Reference Driver

The reference driver contains the following functionality blocks (see SGX OpenGL ES 1.1 Reference Driver Software Functional Specification for details)

- OpenGL ES state machine
- SGX raster code set-up and raster state management
- SGX transform and light code generation and management
- SGX texture blend code generation and management
- SGX constant management
- SGX vertex DMA code generation and management
- Texture management
- Interfaces to TA and Render

The reference implementation contains the OpenGL ES API 1.1, in both *common profile* and *common-lite profile* forms. The selection of which profile will be exposed is a compile time option. This component is where the primary development effort of the driver resides.

2.2.2. Driver Strings on OpenGL ES

OpenGL ES allows an application to query strings to identify the driver that the application is running on. There are 4 standard queryable strings, these are the version string, the vendor string, the renderer string and the extension string. These are obtained by calling glGetString with GL_VERSION, GL_VENDOR, GL_RENDERER and GL_EXTENSIONS respectively.

Version String

The version string returned by OpenGL ES drivers indicates the version of the API and whether the driver is common or common-lite. For version 1.1 these strings are for common and common-lite respectively:

- "OpenGL ES-CM 1.1"
- "OpenGL ES-CL 1.1"

Vendor String

The vendor string is not restricted and may be replaced by the IP customer. The reference driver string will be: "Imagination Technologies"

Renderer String

The renderer string will likely be the primary method by which developers can identify the underlying hardware, and as such is controlled so that IP customers should only modify the string in a restricted manner. It is unlikely that end developers will have any other manner to identify the underlying PowerVR devices (for example the DeviceID register).

The renderer string contains the text “PowerVR”, followed by text to identify the core, its major version and any further variant. The customer may prepend text to the front of the renderer string. The following are examples:

- "ACME OGLES on PowerVR SGX530"
- "Magic3DWidget (TM) PowerVR SGX520"

The reference OpenGL ES driver contains the string:

- "Reference PowerVR SGX530" (for a 2 pipe SGX)

Extension String

The extension string is a space separated alphabetically sorted concatenation of the extension names supported by the driver. See the section on extensions for details of extensions supported.

2.2.3. EGL

EGL is the component that allows the OpenGL ES application to control the configuration attributes with which OpenGL ES rendering will happen, in terms of colour depth, Z-depth, support for different rendering surfaces (window/pixmap/pbuffer), and controls where the render results will appear. See the SGX EGL 1.4 Reference Driver, Software Architecture Specification.doc for details.

2.2.4. PowerVR Services Glue

The PowerVR Services glue layer exists to abstract in a platform independent method access to the PowerVR services. This layer in general should be very thin and may simply abstract the calling method by which the services are obtained, and whether on a given platform the services exist in kernel or user mode, or if they share a common address space with the reference driver. This glue layer is defined in the PowerVR Services Porting Guide for the appropriate OS.

2.3. Development constraints

To ensure the above stated portability the following constraints are applied to the development of the PowerVR OpenGL ES 1.1 reference implementation.

- The library should be able to be compiled to the common and common-lite profiles from the same source base.
- As far as reasonably practical code should be written in a platform neutral style. The Non-windows typedefs and variable naming notation from the PowerVR C Style Guide should be adopted where practical.
- Particular care must be paid to the library entry points to ensure that they are as re-targetable as possible.
- The graphics context (GC) must only ever be retrieved from TLS in the entry point function. There after it must always be passed deeper into the library on the stack. The retrieval from TLS should be macro-ised to simplify porting.
- The code should be extremely well commented specifically in places where care is required to mate OpenGL ES functionality to SGX functionality.
- Code should be written for compactness, but care should be taken that this does not unduly obfuscate the code.
- Floating-point code should where possible be isolated into specific files.

3. Conformance

The Khronos Group publishes a test suite to allow implementers of OpenGL ES to badge their products as conformant. The criteria for OpenGL ES 1.1 is that from the suite of tests (*covegl*, *covgl*, *primtest*, and *conform*), the first three of these test must pass without exception, that last (more complex) test must pass with its *mustpass* criteria, and fail no more than seven other discrete sub-tests.

The reference drivers pass within these criteria (all tests pass).

The following table lists areas where the OpenGL ES driver and SGX combination will deviate from operation as specified in the OpenGL ES 1.1 specification, although the conformance tests do not detect these deviations:

Deviation	Description
Anti-aliased point and lines	Not supported on SGX
Multi-sampling per primitive	This would be prohibitively expensive to support as the Z-buffer would need to be re-sampled in software

No workarounds are provided in the implementation to assist with these deviations to the OpenGL ES specification. Thus if consistent results are desired across platforms, programmers should not attempt to use features that are flagged as deviating from specified behaviour.

4. Extensions

The following table illustrates the intended extension support. Core and mandatory extensions are required for conformance, while optional and vendor extensions are shown for completeness.

Extension type	Extension name	Functionality
Core	OES_byte_coordinates Available from 1.0 DDK	Allows bytes to be supplied for vertex position and texture coordinates.
	OES_fixed_point Available from 1.0 DDK	Allows fixed point data to be supplied for vertex components and certain state values.
	OES_single_precision Available from 1.0 DDK	Allows single precision data to be supplied for certain state values.
	OES_matrix_get Available from 1.0 DDK	Allows current matrices to be retrieved from the GL.
Mandatory	OES_read_format Available from 1.0 DDK	Allows implementation specific read formats to be used for framebuffer readback.
	OES_compressed_paletted_texture Available from 1.0 DDK	Allows paletted textures to be supplied to the GL.
	OES_point_sprite Available from 1.0 DDK	Allows sized points to be turned into sprites.
	OES_point_size_array Available from 1.0 DDK	Allows a point size per point to be supplied efficiently
Optional	OES_matrix_palette Available from 1.0 DDK	Allows vertex blending (skinning) to be performed
	OES_draw_texture Available from 1.0 DDK	Allows a rectangle with a texture applied to be drawn.
Other	OES_query_matrix Available from 1.0 DDK	Allows the querying of the components and state of the current matrix
	OES_texture_env_crossbar Available from 1.0 DDK	Allows other texture units as the source for combine texture operations
	OES_textured_mirrored_repeat Available from 1.0 DDK	Adds mirrored-repeat texture wrapping mode
	OES_texture_cube_map Available from 1.0 DDK	Allows the use of cubic environment maps
	OES_blend_subtract Available from 1.0 DDK	Adds two blend equations
	OES_blend_func_separate Available from 1.0 DDK	Allows the use of separate blend factors for RGB and Alpha
	OES_blend_equation_separate Available from 1.0 DDK	Allows the use of separate blend equations for RGB and Alpha
	OES_stencil_wrap Available from 1.0 DDK	Adds two stencil wrapping operations
	OES_extended_matrix_palette Available from 1.0 DDK	Increases the number of palette matrices and vertex units
	OES_framebuffer_object Available from 1.0 DDK	Allows rendering to destinations other than the GL window system buffers
	OES_rgb8_rgba8 Available from 1.0 DDK	Enables rendering to RGB8 and RGBA8 renderbuffers

Extension type	Extension name	Functionality
	OES_depth24 Available from 1.0 DDK	Enables rendering to 24 bit depth buffers attached to a framebuffer object
	OES_stencil8 Available from 1.0 DDK	Enables rendering to 8 bit stencil buffers attached to a framebuffer object
	OES_compressed_ETC1_RGB8_texture Available from 1.0 DDK	Allows ETC1 precompressed data to be supplied to the GL.
	OES_map_buffer Available from 1.0 DDK	Allows vertex buffer objects to be mapped.
	OES_EGL_image Available from 1.2 DDK	Allows creation of texture and render buffers targets from EGL images.
	OES_required_internal_format Available from 1.4 DDK	Guarantees the internal format of textures.
	OES_vertex_array_object Available from 1.6 DDK	Allows vertex array state to be grouped in named objects
	OES_EGL_sync Available from 1.6 DDK	Allows EGL to insert fences and query completion in the GL command stream.
	EXT_multi_draw_arrays Available from 1.2 DDK	Allows multiple draw calls to be issued in one GL call.
	IMG_read_format Available from 1.0 DDK	Increases the number of format/type combinations for glReadPixels
	IMG_texture_compression_pvrtc Available from 1.0 DDK	Allows PVRTC precompressed data to be supplied to the GL.
	IMG_vertex_program Available from 1.0 DDK	Adds vertex programmability (Note 1)
	IMG_texture_stream Available from 1.0 DDK	Allows a hardware source containing image data to be used as a texture
	IMG_texture_format_BGRA8888 Available from 1.0 DDK EXT_texture_format_BGRA8888 Available from 1.4 DDK	Allows the GL to accept textures with a BGRA format.

Table 2

Note 1: This extension was only enabled for MBX support and is deprecated (should not be used) in 1.5 DDK and will be removed in 1.6 DDK and beyond. It is expected that the programmable vertex usage cases will have moved to OpenGL ES 2.0 in this time frame .

4.1. PowerVR Texture Compression

PowerVR Texture Compression (PVR-TC) will be exposed as enumerated formats through the OpenGL ES compressed texture interface. The presence of the formats will be detectable through the vendor extension “IMG_texture_compression_pvrtc”. Because PVR-TC is non-symmetric between encoding and decoding the driver will not provide compression of supplied data through the *glTexImage2D* call with the PVR-TC internal format token. All compression must be done offline via the tools provided by PowerVR Developer Relations. In addition, the PVR-TC format cannot efficiently

support loading of non-complete levels, so *glCompressedTexSubImage2D* will only support the case where the sub-image equals the level size.

5. Implementation Details

5.1. Implementation Dependant Values

The following table enumerates a selecation of important data that is queryable through the glGet* interface:

State	Value	Comment
SUBPIXEL_BITS	4	
MAX_TEXTURE_SIZE	2048 / 4096	4096 for SGX XT cores and SGX DX10 cores (which means SGX543/544/545/554)"
MAX_CUBE_MAP_TEXTURE_SIZE	2048 / 4096	4096 for SGX XT cores and SGX DX10 cores (which means SGX543/544/545/554)"
MAX_VIEWPORT_DIMS	2048 / 4096	4096 for SGX XT cores and SGX DX10 cores (which means SGX543/544/545/554)"
ALIASED_POINT_SIZE_RANGE	1	
	32	
ALIASED_LINE_WIDTH_RANGE	1	
	16	
SMOOTH_POINT_SIZE_RANGE	1	
	1	
SMOOTH_LINE_SIZE_RANGE	1	
	1	
MAX_ELEMENTS_INDICES	65536	
MAX_ELEMENTS_VERTICES	65536	
SAMPLE_BUFFERS	0	
	1	
SAMPLES	0	
	4	2x2 anti aliasing
NUM_COMPRESSED_TEXTURE_FORMATS	15	
COMPRESSED_TEXTURE_FORMATS	GL_PALETTE4_RGB8_OES	
	GL_PALETTE4_RGBA8_OES	
	GL_PALETTE4_R5_G6_B5_OES	
	GL_PALETTE4_RGBA4_OES	
	GL_PALETTE4_RGB5_A1_OES	
	GL_PALETTE8_RGB8_OES	
	GL_PALETTE8_RGBA8_OES	

State	Value	Comment
	GL_PALETTE8_R5_G6_B5_OES	
	GL_PALETTE8_RGBA4_OES	
	GL_PALETTE8_RGB5_A1_OES	
	GL_COMPRESSED_RGB_PVRTC_2BPPV1_IMG	
	GL_COMPRESSED_RGBA_PVRTC_2BPPV1_IMG	
	GL_COMPRESSED_RGB_PVRTC_4BPPV1_IMG	
	GL_COMPRESSED_RGBA_PVRTC_4BPPV1_IMG	
	GL_ETC1_RGB8_OES	
GLES1_MAX_TEXTURE_UNITS	4	
MAX_CLIP_PLANES	6	
MAX_LIGHTS	8	
MAX_MODELVIEW_STACK_DEPTH	16	
MAX_PROJECTION_STACK_DEPTH	2	
MAX_TEXTURE_STACK_DEPTH	4	
MAX_PROGRAM_MATRIX_STACK_DEPTH_ARB	2	
MAX_PROGRAM_MATRICES_ARB	8	
MAX_VERTEX_ATTRIBS_ARB	8	
MAX_PROGRAM_ENV_PARAMETERS_ARB	16	
MAX_PROGRAM_LOCAL_PARAMETERS_ARB	16	
MAX_PALETTE_MATRICES_OES	32	
MAX_VERTEX_UNITS_OES	4	
RED_BITS, GREEN_BITS, BLUE_BITS, ALPHA_BITS	8,8,8,8	
	4,4,4,4	
	5,5,5,1	
	5,6,5,0	
DEPTH_BITS	0, 16, 24	
STENCIL_BITS	0, 8	
RENDERER	PowerVR SGX 535	
VENDOR	Imagination Technologies	
VERSION	OpenGL ES-CM 1.1	
	OpenGL ES-CL 1.1	
IMPLEMENTATION_COLOR_READ_TYPE_OES	GL_UNSIGNED_BYTE	for ARGB8888 surface

State	Value	Comment
	GL_UNSIGNED_SHORT_4_4_4_4	for ARGB4444 surface
	GL_UNSIGNED_SHORT_5_6_5	for RGB565 surface
IMPLEMENTATION_COLOR_READ_FORMAT_OES	GL_BGRA	for ARGB8888 surface
	GL_BGRA	for ARGB4444 surface
	GL_RGB	for RGB565 surface

5.2. Texture Management

PowerVR SGX is a Tile based deferred renderer which captures a scene in one pass (and tiles the data), and renders that scene in a second pass, allowing it to reject pixels which would not be visible in the final scene. The second pass resolves which fragments are visible, and textures and shades them.

One consequence of this is that the driver must ensure that textures required for the render pass must remain available to the hardware until that pass has completed, even if an application has changed the texture immediately after, or even during, submission of a scene. This is referred to as texture management.

OpenGL-ES texturing works on a separate object data and image data model. Thus, the texture has state relating to the object that must be reconciled with the data in the texture images. This reconciliation is referred to as texture completeness and can only occur at the time the texture is used to draw.

Information that is included in the completeness check includes the format and size of every texture level in the object, as well as the texture filter applied to that object at the point of use. Only when the completeness check has passed, can the layout of the texture in device memory be determined. Once this has taken place, the texture memory can be “uploaded” from the host copy to the device memory. During this upload process, the texture data may be “twiddled” to fit the optimum layout for hardware access. (Twiddling is a reorganisation of the texture data to minimise page breaks when accessed by the hardware).

The SGX OpenGLES1 driver has a texture management model that aims to minimise texture memory overhead, and the size of the texture management code.

In line with the first aim of minimising the memory used for textures, the driver only allocates a single copy of any texture level at any given time, except during the upload process. This may be either a CPU only mapped copy of the texture level or a device mapped (and formatted) copy of the level.

The device memory version of the level is only present if the texture object is “complete”. The sequence of events is

1. When texture level data is supplied through glTexImage2D, a copy of the data is taken and converted from the incoming application specified format, into a native hardware format - this is known as “format swizzling”.
2. When a texture is referenced in a draw call the completeness check is applied, and if the texture object passes, its levels are uploaded to device memory - texture twiddling takes

place at this point for a whole texture level at time. As each level is uploaded, the strided host copy is freed to ensure only one coherent copy of the level data persists.

Due to the deferred nature of the hardware, it is possible for a new device memory allocation to take place prior to the upload procedure, even though there may already be a device memory allocation for that texture object. This case would happen if a single texture object had multiple glTexImage2D, glDrawArrays/Elements pairs within a scene applied to it. In that case, a new device memory allocation is created if the texture object has already been referenced, either in the current scene, or in a previous scene which the hardware has yet to complete rendering (an 'in-flight' render). The new texture data is uploaded to this new device memory, while the previous texture data which is already in device memory is retained in a separate list of active texture data. This is referred to as "Texture Ghosting".

If a texture is found to be incomplete during the validation phase of a draw call, a single, strided, host copy of all texture levels is gathered together. This may involve readback (and detwiddling) of some texture levels, and may involve discarding other texture levels from device memory.

When a glTexSubImage2D call is made, the driver attempts to guarantee coherency of texture level data. This is achieved by continuing to ensure there is only one copy of texture level data. In the case where a texture object has not yet been referenced in a draw call, there will be no device memory allocated at that time, and the subtexture data is simply loaded into the host memory copy of the texture level data. If a device memory version of the texture level is already present, there are two options. The first one is to employ HWTQ (Hardware Transfer Queue) to directly upload subtexture data into the device memory. If this fails, then the second approach is adopted – a similar process to dealing with an incomplete texture is instigated. The texture level is read back (and detwiddled) into strided, host memory. The subtexture data is then loaded into this host copy and the level is marked for future upload. If and when this texture is referenced in a subsequent draw call, the level data is uploaded to device memory again. This upload may need to take place to a new device memory allocation, depending on which scene(s) the old device memory is referenced in, as documented above.

When glDeleteTexture is called the texture data will be deleted immediately if the texture has never been made resident (i.e never been used to draw with). If the texture has been used to draw with, but is no longer referenced by an 'in-flight' render it will be deleted immediately. If the texture is referenced by an 'in-flight' render then the texture will be added to the ghost list. Items on the ghost list are freed at the start of the next subsequent frame when they are no longer referenced by an 'in-flight' render.

5.2.1. Application Considerations

Applications should avoid using the default texture, and always use a different named texture for each real texture that they have.

If an application knows that they will reload entire textures using glTexSubImage2D, then the application should rotate through a working set of different named textures that is large enough to avoid modified texture being active.

Applications should avoid non-complete texture level usage of glTexSubImage2D if at all possible.

5.3. Fixed Function Transformation & Lighting Code Generation

From 1.6 DDK onwards, the USSE code for Transformation and Lighting (TnL) is generated using a optimising compiler (USC). In previous DDK versions this USSE code was created by stitching together blocks of pre-generated code. This was quicker to generate, but had no global optimisations. The use of the USC allows better HW code to be generated for complex shaders (including lighting and matrix palette), at the cost of higher one-time code generation cost. All USSE code is cached in the driver, so this extra time to compile is only experienced once per TnL state combination.

6. Debug, Metrics and Profiling

6.1. Debug

The driver will implement synchronous debug messages out in debug builds where the underlying operating system allows, for instance in a desktop Windows environment the *OutputDebugString* function would be used. If a synchronous method is not available a non-synchronous method will be used instead (i.e. *printf*). The messages will be in one of five categories summarised as follows:

Message Level	Usage
Verbose	High frequency message, used for specific debugging
Message	Informational message that may be of use to developer
Warning	Recoverable error, or warning of incorrect behaviour
Error	Unrecoverable error, not fatal
Fatal	Unrecoverable error, leading to a fault

By default the driver will not emit messages of less severity than warning, but this default level is controllable.

6.2. Metrics

Metrics describes the collection of data that is of direct interest for assessing performance, both hardware and software, as a function of the data collectable from the OpenGL ES driver. Metrics data is output in a textual form at driver termination. The data by default is collected for all frames, but on systems that support registries or initialisation files the data can be collected over a single range of frames. Metric data will be available in debug and timing builds. The metric collection relies on the presence of a high-resolution system timer. Sample data from an existing PowerVR OpenGL ES driver is presented in Appendix B to illustrate the information that may be presented.

6.3. Profiling

Profiling describes the collection of data indicating how the OpenGL ES API is being used with emphasis on data that is useful to both driver developers and application developers to efficiently use the API on SGX hardware. Profile data is output in a textual form at driver termination. The data by default is collected for all frames, but on systems that support registries or initialisation files the data can be collected over a single range of frames. Profiling is available in debug and metrics builds. Some of the profiling data relies on the presence of a high-resolution system timer. Sample data from an existing PowerVR OpenGL ES driver is presented in Appendix C to illustrate the information that may be presented.

7. Configurations

7.1. Build time configurations

There are three build targets for any given SGX HW configuration and environment configuration these are:

Target	Description
Debug	Non-optimised build, debug statements compiled in, profiling and metric collection present
Timing	Optimised build, profiling and metric collection present, no debug statements
Release	Optimised build, no profiling or metric collection present, no debug statements

In addition to the overall build targets listed above the following compile time configurations exist, these should be set as C preprocessor defines from the target environment makefile:

Configuration	Description
TBD	
…	

7.2. Runtime configurations

The OpenGL ES driver will read run time configuration data from the registry or initialisation file on systems that support these during driver initialisation. Systems that don't support such capabilities may not be runtime configurable.

There are several different classes of use for runtime configuration parameters. These include performance tuning, data collection, debugging, and application compatibility. Of these only performance tuning and application compatibility will be available in the *Release* build target.

Runtime configuration options and their function and usage are listed below:

Option	Default Value	Function	Usage
DumpProfileData	0	Emit profile data at profile end frame or application normal termination	DataColl
ProfileStartFrame	0	Frame to start collecting profile and metric data	DataColl
ProfileEndFrame	0xffffffff	Frame to stop collecting profile and metric data	DataColl
DefaultVertexBufferSize	200*1024	Defines the default size of the internal vertex buffer. This affects the number of vertices for non VBO draw calls which can fit in the buffer before a TA kick is required.[1]	PerfTune

Option	Default Value	Function	Usage
MaxVertexBufferSize	800*1024	Defines the maximum size of the internal vertex buffer. This can grow from the default size if a slow “batching” path is detected.	PerfTune
DefaultIndexBufferSize	200*1024	Defines the default size of the internal index buffer. This affects the number of indices for non VBO draw calls which can fit in the buffer before a TA kick is required. [1]	PerfTune
DefaultPregenPDSVertBufferSize	80*1024	Defines the default size of the internal vertex PDS program buffer. This affects the number of draw calls which can fit in the buffer before a TA kick is required. [1]	PerfTune
DefaultPregenMTECopyBufferSize	50*1024	Defines the default size of the internal MTE copy state program buffer. This affects the number of state changes which can fit in the buffer before a TA kick is required. [1]	PerfTune
DefaultPDSVertBufferSize	50*1024	Defines the default size of the internal vertex state/constant buffer. This affects the number of draw calls which can fit in the buffer before a TA kick is required. [1]	PerfTune
DefaultVDMBufferSize	20*1024	Defines the default size of the internal vertex control buffer. This affects the number of draw calls which can fit in the buffer before a TA kick is required. [1]	PerfTune
PDSFragBufferSize*	50*1024	Defines the default size of the internal fragment state/constant buffer. This affects the number of draw calls with different state which can fit in the buffer before a 3D kick is required. This buffer is only used for special objects.	PerfTune
ParamBufferSize **	1.6 DDK onwards: 1*1024*1024 From 1.0 DDK -> 1.4 DDK: 4*1024*1024	Defines the default size of the internal parameter buffer	PerfTune
MaxParamBufferSize**	4*1024*1024	Defines the maximum size the internal parameter buffer. 0 allows growth until memory is exhausted.	PerfTune 1.6 DDK onwards
FlushBehaviour	0	Allows glFlush/glFinish behaviour to be changed to kick TA (FlushBehaviour=1) or kick 3D (FlushBehaviour=2). Note this only affect EGL Window or PBuffer surfaces which are not being used by multiple client APIs.	PerfTune

Option	Default Value	Function	Usage
KickTAMode ***	4	Allows TA to be kicked more frequently to enable low latency scheduling.	PerfTune
KickTAThreshold ***	3	Threshold to use for TA kicking	PerfTune
DisableHWTextureUpload	0	Allows disabling of the use of the HW for texture uploads	Debug
DisableHWMipGen	0	Allows disabling of the use of the HW for mipmap generation	Debug
ForceExternalZBuffer	0	Force upfront creation of an external Z-buffer and use it for SPM events or fragment buffer overflows (pre 1.5 DDK only)	1.0 DDK -> 1.4 DDK
ForceNoExternalZBuffer	0	Do not allow the on-demand creation of an external Z-buffer (pre 1.5 DDK only)	1.0 DDK -> 1.4 DDK
ExternalZBufferMode****	1	Controls the creation and use of an external Z-buffer (post 1.4 Beta DDK only, overrides settings for ForceExternalZBuffer and ForceNoExternalZBuffer in 1.4 RC DDK)	1.4 RC DDK onwards
EmulateVGP	0	Controls adding VGP to the GL_RENDER string	

* Defined per surface in the EGL module.

** Defined in the EGL module.

*** See KickTAMode Table Below

**** See ExternalZBufferMode Table Below

[1] There is no direct mapping between the number of draw calls and the number of TA kicks. It will differ for all applications.

KickTAMode	Meaning	KickTAThreshold Interpretation
0	Don't submit extra TA kicks	None
1	Kick TA every N primitives (draw calls)	Number of primitives before next kick
2	Kick TA every N indices	Number of indices before next kick
3	Kick TA heuristically based on primitive count of previous frame to try to generate N TA kicks per frame	Number of kicks to try to generate per frame
4	Kick TA heuristically based on index count of previous frame to try to generate N TA kicks per frame	Number of kicks to try to generate per frame

ExternalZBufferMode	Allocation	Use	Meaning

ExternalZBufferMode	Allocation	Use	Meaning
0	On demand	Always	External Z-buffer is allocated after an SPM event or before a fragment buffer overflow, and then always used
1	On demand	As needed	External Z-buffer is allocated after an SPM event or before a fragment buffer overflow, and then only used for future SPM events or fragment buffer overflows
2	Up front	Always	External Z-buffer is allocated when EGL surface is created, and then always used
3	Up front	As needed	External Z-buffer is allocated when EGL surface is created, and then only used for SPM events or fragment buffer overflows
4	Never	Never	External Z-buffer is never created or used

8. Memory allocations

This section describes some of the memory allocations made by the SGX driver. These allocations are in two main forms: device memory and host memory.

8.1. Device memory allocations for internal driver use

The following table details some of the significant device memory allocations made by the driver for its own use:

8.1.1. Allocations in 1.2 DDK

Allocation	Storage for	Default size	Controlled by
Parameter buffer	SGX internal parameter data	4 MB	`#define EGL_DEFAULT_PARAMETER_BUFFER_SIZE in include\srvcontext.h Apphint: "ParamBufferSize" (EGL module)`
Vertex buffer	Dynamic vertex data	200 KB	`ui32Default = 200*1024; in opengles1\misc.c Apphint: "DefaultVIBufferSize"`
Index buffer	Dynamic index data	200 KB	`ui32Default = 200*1024; in opengles1\misc.c Apphint: "DefaultVIBufferSize"`

Allocation	Storage for	Default size	Controlled by
VDM buffer	VDM input control stream	10 KB	ui32Default = 10*1024; in opengles1\misc.c Apphint: "DefaultVDMBufferSize"
Vertex PDS buffer	Vertex state and constants	50 KB	ui32Default = 50*1024; in opengles1\misc.c Apphint: "DefaultPDSVertBufferSize"
Pixel PDS buffer	Dynamic PDS pixel programs	50 KB	#define EGL_DEFAULT_PDS_FRAG_BUFFER_SIZE in include\srvcontext.h Apphint: "PDSFragBufferSize" (EGL module)
Vertex USSE code heap	Static USSE vertex programs	32 KB, grows to maximum of 2 MB	#define SGX_VERTEXSHADER_HEAP_SIZE in services4\srvkm\devices\sgx\sgxconfig.h
Pixel USSE code heap	Static USSE pixel programs	32 KB, grows to maximum of 5 MB	#define SGX_PIXELSHADER_HEAP_SIZE in services4\srvkm\devices\sgx\sgxconfig.h
Vertex PDS code heap	Static PDS vertex programs	32 KB, grows to maximum of 32 MB	#define SGX_PDSVERTEX_CODEDATA_HEAP_SIZE in services4\srvkm\devices\sgx\sgxconfig.h
Pixel PDS code heap	Static PDS pixel programs	32 KB, grows to maximum of 32 MB	#define SGX_PDSPIXEL_CODEDATA_HEAP_SIZE in services4\srvkm\devices\sgx\sgxconfig.h

8.1.2. Allocations in 1.3+ DDK (including performance improvements)

Allocation	Storage for	Default size	Controlled by
Parameter buffer	SGX internal parameter data	4 MB	#define EGL_DEFAULT_PARAMETER_BUFFER_SIZE in include\srvcontext.h Apphint: "ParamBufferSize" (EGL module)
Vertex buffer	Dynamic vertex data	200 KB	ui32Default = 200*1024; in opengles1\misc.c Apphint: "DefaultVertexBufferSize"

Allocation	Storage for	Default size	Controlled by
Index buffer	Dynamic index data	200 KB	ui32Default = 200*1024; in opengles1\misc.c Apphint: "DefaultIndexBufferSize"
VDM buffer	VDM input control stream	20 KB	ui32Default = 20*1024; in opengles1\misc.c Apphint: "DefaultVDMBufferSize"
Vertex PDS buffer	Vertex state and constants	50 KB	ui32Default = 50*1024; in opengles1\misc.c Apphint: "DefaultPDSVertBufferSize"
Pregen Vertex PDS buffer	Static PDS vertex programs	80 KB	ui32Default = 80*1024; in opengles1\misc.c Apphint: "DefaultPregenPDSVertBufferSize"
Pregen MTE Copy Buffer	Static MTE copy state programs	50KB	ui32Default = 50*1024; in opengles1\misc.c Apphint: "DefaultPregenMTECopyBufferSize"
Pixel PDS buffer	Dynamic PDS pixel programs	50 KB	#define EGL_DEFAULT_PDS_FRAG_BUFFER_SIZE in include\srvcontext.h Apphint: "PDSFragBufferSize" (EGL module)
Vertex USSE code heap	Static USSE vertex programs	32 KB, grows to maximum of 2 MB	#define SGX_VERTEXSHADER_HEAP_SIZE in services4\srvkm\devices\sgx\sgxconfig.h
Pixel USSE code heap	Static USSE pixel programs	32 KB, grows to maximum of 5 MB	#define SGX_PIXELSHADER_HEAP_SIZE in services4\srvkm\devices\sgx\sgxconfig.h
Vertex PDS code heap	Static PDS vertex programs	32 KB, grows to maximum of 32 MB	#define SGX_PDSVERTEX_CODEDATA_HEAP_SIZE in services4\srvkm\devices\sgx\sgxconfig.h
Pixel PDS code heap	Static PDS pixel programs	32 KB, grows to maximum of 32 MB	#define SGX_PDSPIXEL_CODEDATA_HEAP_SIZE in services4\srvkm\devices\sgx\sgxconfig.h

8.2. Device memory allocations for application supplied data

The OpenGL ES driver will allocate device memory to store application supplied data such as textures and vertex buffer objects.

The size of these allocations depends on the size of the application supplied data and on associated OpenGL ES state (e.g. internal texture format).

8.3. Host memory allocations for internal driver use

The following table details some of the significant host memory allocations made by the driver for its own use:

Allocation	Storage for	Size	Controlled by
OpenGL ES context	OpenGL ES state	Approximately 18 KB	C-structure

8.4. Host memory allocations for application supplied data

The OpenGL ES driver will allocate host memory to store application supplied texture data. After the texture data is copied to device memory, the host copy is freed. The size of these allocations depends on the size of the application supplied data and on associated OpenGL ES state (e.g. internal texture format).

9. Review of this Document

9.1. Signatories

Signed off for document version:	

9.2. Change Control

Any changes to this document will result in a new version with appropriate sign off.

To do this the Signatories Section will need to be duplicated and the dates and version removed. Please leave the old details also present in the document so that the original signed version can be found.

Appendix A. OGL ES Reference Driver Code Modules

The following table lists all the source files compiled to construct the OpenGL ES 1.1 reference driver. Each entry is accompanied by a short description:

Source File	Description
TBD	

Appendix B. Sample Metric Data

```
PVR:  Total Frames                                              1
PVR:  Profiled Frames                              (0-1)        1
PVR:  Average Elapsed Time per Profiled Frame (ms)         0.1072
PVR:  Average Driver Time per Profiled Frame (ms)          7.8895
PVR:  Estimated FPS                                     9332.6525
PVR:
PVR:            Statistics per profiled frame            [Calls/Time(ms)]
PVR:  Total Prepare to draw                                3/     0.0400
PVR:  Total SGXKickTA                                      1/     0.0582
PVR:  Total Renders                                   1.0000/        -
PVR:  Total Wait for 3D                                    1/     0.0008
PVR:  Total Wait for TA                                    1/     0.0003
PVR:  Total ValidateState()                                2/     2.3838
PVR:   - Setup_TextureState()                              2/     0.5173
PVR:   - Setup_Streams()                                   2/     0.0015
PVR:   - UpdateMVP()                                       2/     0.0016
PVR:   - Setup_VertexShader()                              2/     1.1101
PVR:   - Setup_VertexConstants()                           2/     0.7010
PVR:   - Setup_Outputs()                                   2/     0.0006
PVR:   - Setup_VertexVariant()                             2/     0.0113
PVR:   - Setup_RenderState()                               2/     0.0030
PVR:   - Setup_FragmentShader()                            2/     0.0339
PVR:  Total EmitState()                                    2/     0.2249
PVR:   - WritePDSPixelShaderProgram()                      2/     0.0730
PVR:   - WritePDSVertexShaderProgram()                     2/     0.0100
PVR:   - WritePDSUSEShaderSAProgram()                      4/     0.0464
PVR:   - WriteMTEState()                                   2/     0.0046
PVR:   - WriteMTEStateCopyPrograms()                       2/     0.0811
PVR:   - SetupStateUpdate()                                4/     0.0035
PVR:   - WriteVDMControlStream()                           2/     0.0025
PVR:  USE Codeheap Alloc - Fragment                        5/     0.0042
PVR:  USE Codeheap Alloc - Vertex                          8/     0.0049
PVR:  Total Primitive Draw                                 2/     5.4112
PVR:  Total Vertex Data Copy                               2/     0.0034
PVR:  Total Index Data Generate/Copy                       2/     0.0002
PVR:  Array Triangles                                      2/     5.4108
PVR:
PVR:  PDS Variant hit/miss totals
PVR:  PDSPixelShaderProgram - variant hit                  0
PVR:  PDSPixelShaderProgram - variant miss                 2
PVR:
PVR:          Statistics per call            [Maximum time (ms) in a single
call]
PVR:  Max Prepare to draw                     0.039326
PVR:  Max SGXKickTA                           0.058221
PVR:  Max Wait for 3D                         0.000812
PVR:  Max Wait for TA                         0.000297
PVR:  Max ValidateState()                     1.928312
PVR:  Max EmitState()                         0.177015
PVR:  Max USE Codeheap Alloc - Fragment       0.001821
PVR:  Max USE Codeheap Alloc - Vertex         0.000968
PVR:  Max NamesArray operation                0.004396
PVR:  Total Code Heap operations                     31/     0.2905
PVR:
PVR:  Buffer Statistics                          [  kB/kB per Frame  ]
PVR:   VDM Control                            0.1211/     0.1211
PVR:   MTE State                              0.1602/     0.1602
PVR:   Vertex data                            0.2109/     0.2109
```

```
PVR:    Index data                                  0.0234/     0.0234
PVR:    PDS vertex data                             0.8906/     0.8906
PVR:    PDS fragment data                           0.3750/     0.3750
PVR:    USE Codeheap - Fragment                     0.0703/     0.0703
PVR:    USE Codeheap - Vertex                       0.3047/     0.3047
PVR:
PVR:  Average TexImage Load MTexels/s                      46.5728
PVR:  Texture Statistics          [  Calls/perCall-Time|Texels  / perFrame-
Time/Texels  ]
PVR:      TexImage                      1/      0.0879/    4096/
0.0879/ 4096.00
PVR:      TexSubImage                   0/      0.0000/       0/
0.0000/    0.00
PVR:  Texture Statistics          [  Calls/Time per Call/Time per Frame]
PVR:      CopyImage                     0/       0.0000/         0.0000
PVR:      AllocateTexture               1/       0.1109/         0.1109
PVR:      TranslateLoadTexture          1/       0.4026/         0.4026
PVR:      Load Ghost Texture            0/       0.0000/         0.0000
PVR:      Texture Readback              0/       0.0000/         0.0000
PVR:
PVR:  Texture allocation HWM = 32769 bytes
PVR:
PVR: Memory Stats
PVR: ------------
PVR:
PVR: High Water Mark = 311212 bytes
PVR:
```

Appendix C. Sample Profile Data

```
Profiling (per context)
=======================

Draw calls profiling

DrawArrays          Vertices      Calls Vertices/Frame Calls/Frame
GL_TRIANGLES               6          2              6           2

Entry point profiling (Times in uSx10)

Call                                                        Number Calls
Time/Call  Calls/Frame  Time/Frame
glClear                                                              1
0.087650             1  0.087650
glClearColor                                                         1
0.002261             1  0.002261
glColorPointer                                                       1
0.001081             1  0.001081
glDisable                                                            1
0.002380             1  0.002380
glDrawArrays                                                         2
0.852086             2  1.704172
glEnable                                                             1
0.000795             1  0.000795
glEnableClientState                                                  3
0.000151             3  0.000452
glLoadIdentity                                                       2
0.000567             2  0.001133
glMatrixMode                                                         1
0.000047             1  0.000047
glRotatef                                                            2
0.004787             2  0.009574
glShadeModel                                                         1
0.000130             1  0.000130
glTexCoordPointer                                                    1
0.000676             1  0.000676
glTexImage2D                                                         1
0.047817             1  0.047817
glTexParameterf                                                      1
0.000795             1  0.000795
glTranslatef                                                         2
0.000741             2  0.001481
glVertexPointer                                                      1
0.000826             1  0.000826

            Dirty flag statistics        [Total Hits/Avg Hits per Frame]
 GLES1_DIRTYFLAG_ISP                                2/         2
 GLES1_DIRTYFLAG_MTE_CONTROL                        1/         1
 GLES1_DIRTYFLAG_ATTRIB_STREAM                      1/         1
 GLES1_DIRTYFLAG_ATTRIB_POINTER                     2/         2
 GLES1_DIRTYFLAG_VIEWPORT                           1/         1
 GLES1_DIRTYFLAG_TAMISC_STATE                       1/         1
 GLES1_DIRTYFLAG_PDS_OUTPUT_STATE                   1/         1
 GLES1_DIRTYFLAG_PDS_VP_SA_STATE                    1/         1
 GLES1_DIRTYFLAG_VERTPROG_CONSTANTS                 2/         2
 GLES1_DIRTYFLAG_FRAGPROG_CONSTANTS                 2/         2
 GLES1_DIRTYFLAG_TEXTURE_STATE                      2/         2
 GLES1_DIRTYFLAG_VP_STATE                           2/         2
```

```
 GLES1_DIRTYFLAG_FP_STATE                          2/         2
 GLES1_DIRTYFLAG_PDS_FP_SA_STATE                   1/         1
 GLES1_DIRTYFLAG_PDS_VP_STATE                      1/         1
 GLES1_DIRTYFLAG_PDS_FP_STATE                      1/         1
 GLES1_DIRTYFLAG_VERTEX_PROGRAM                    2/         2
 GLES1_DIRTYFLAG_FRAGMENT_PROGRAM                  2/         2
 GLES1_DIRTYFLAG_OUTPUTSELECTS                     2/         2


State combinations used in rendering

State        0  1
TEXTURE0     x
DITHER       x  x
MULTISAMPLE  x  x

State             0       1
Usage Count       1       1

Enable/Disable Call Table

State          ValidCalls RedundantCalls PerFrame-Valid-Redundant
TEXTURE0                2              0           2.00      0.00
```