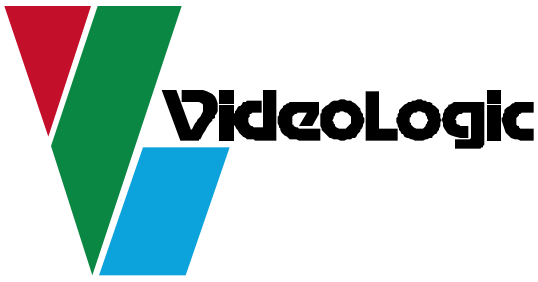
PowerSGL Direct™

PowerVR Graphics Library/API

Programmer's Reference Manual

Edition 2

4 February 1997



COPYRIGHT

DISCLAIMER

TRADEMARKS

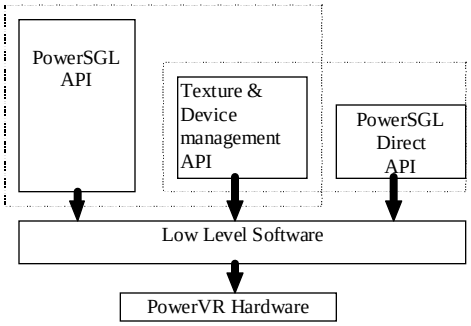
Table of Contents

1 INTRODUCTION

1.1 PowerSGL and PowerSGL Direct

PowerSGL is a high-level graphics library designed to exploit the features of the PowerVR silicon technology. It enables the programmer to define objects and other entities such as cameras and lights in 3D coordinate space, and organize these in a hierarchy. The rendering of these entities can include advanced features such as automatically generated shadows, textures, and depth cueing (fog). PowerSGL is covered in a separate manual.

PowerSGL Direct is a low-level library, in which the user supplies triangles and quadrilaterals in screen coordinates for every rendered frame. All responsibility for transformation, lighting and projection of these polygons falls to the user. Because the polygons have been projected they must be *z*-clipped against the foreground plane, whereas *x* and *y* clipping are optional. The error-checking performed by PowerSGL Direct is minimal.



Use of PowerSGL Direct also requires the programmer to call the texture and device management APIs of PowerSGL.

1.2 Overview of this Guide

This manual is a reference guide for programming in PowerSGL Direct.

1.2.1 Format and Conventions

The manual is divided into chapters covering functional areas of PowerSGL Direct.

Each chapter has an introduction to the subject matter, followed by descriptions of routines, and then functions within those routines.

A function is described in terms of its function header, parameters and error messages. Parameters are identified as input only, output only or input-output via the markers **`I`**, **`O`**, or **`IO`** respectively.

All functions are shown in lower-case courier type, for example, `sgl_delete_device`.

All references to PowerSGL Direct code are shown in courier type, for example, `( int device)`.

All PowerSGL Direct `#defined` system constants are shown in upper-case courier type, for example, `SGL_FIRST_WARNING`.

Function descriptions take the format:

```
TYPE sgl_a_function( TYPE var1,
                LONG_TYPE var2,
                 AAA_TYPE *p_pointer_var3);
```

An example of a function description is given below.

Delete Device

Relinquishes access to a device. The name of the device becomes invalid.

Function Header

```
void sgl_delete_device( int device);
```

Parameters

I	`device`	The device being deleted.

Errors

`sgl_err_bad_name`	There is no device of the given name.

1.3 PowerSGL Direct Features

PowerSGL Direct is a lower-level library than PowerSGL, equivalent to Microsoft D3D and Apple RAVE. It operates at the triangle level, placing triangles into (*x,y*) coordinates, removing them if they are off the screen, calculating screen distances and clipping. (PowerSGL features hidden surface removal rendering, with visual effects including perspective correct anti-aliased texturing, smooth shading with specular highlights, real-time shadows, and fogging.)

For each frame PowerSGL Direct needs to be given the list of triangles or quads for the frame, which must be added again for subsequent frames if they are still present in the scene. The triangles and quads are already transformed into screen coordinates and lit with their on-screen shade, as no camera or light information is required by the API. These processes may be carried out by the Direct3D (Microsoft) and RAVE (Apple) transformation and lighting stages, or directly by the calling application.

The polygons (faces) are added by specifying sets of triangles or quads that share a common material, although each vertex has its own color value. Each face indicates the vertices for that face, and each vertex contains a position, shade, and texture coordinate. A single instance of a context structure should be used by the application to indicate global constants and the material state for the current set of triangles or quads.

Multiple `SGLCONTEXT`s must not be used within the same frame because the structure is also used to store values required internally by the system on subsequent function calls. For textured sets of triangles the context contains the PowerSGL Direct name of the texture.

1.4 Returned Values and Errors

PowerSGL Direct avoids returning error or warning values directly from routines. In systems with dedicated co-processing CPUs, this allows the host processor to continue with the main application while secondary CPU(s) handle the function. To avoid losing reported errors, PowerSGL Direct stores two error values which are accessed via a function. The first value is the first error or warning that occurred since the last get errors call, while the second value is the error or warning from the most recent PowerSGL Direct call.

An error status is returned if a routine passes back a name. Valid returned names are positive integers, and error and warning values are negative. If a valid name is returned, then no error has occurred. Warnings are not returned directly from functions since the routine has succeeded. To distinguish between errors and warnings, the most negative warning message value is `SGL_FIRST_WARNING`; error codes are below this value and start at `SGL_FIRST_ERROR`.

The `sgl_err_failed_init` error is generated by a PowerSGL Direct function that fails to initialize the software or hardware (usually the first PowerSGL Direct function called). All other errors are documented with their respective functions. The following function is used to retrieve error values.

1.4.1 Get Errors

Returns the two stored error or warning values, as described above. The values are cleared to `sgl_no_err`.

Function Header

```
void sgl_get_errors( int *earliest_error,
                     int *most_recent_error);
```

Parameters

`earliest_error`	The first error or warning generated since the last time this routine was called. If this is `sgl_no_err`, then no error has occurred in the interim.
`most_recent_error`	This is the error value for the last routine executed. If this is `sgl_no_err`, then there was no error with that routine.

Errors

None

1.5 Implementation Limitations

Although the PowerSGL Direct interface supplies a powerful set of features, because of possible hardware limitations, not all may be available simultaneously. For example, the initial TSP ASIC can only flat-shade translucent objects—smooth shading is not available. In such cases, the system chooses the next best approximation.

1.6 Library Files

The C-based library object code resides in either `sgl.a` for UNIX-based systems, or `sgl.lib` and associated DLLs for Windows systems. The prototypes for all the functions, type definitions and constant definitions are in `sgl.h`. This header file should be `#included` whenever PowerSGL Direct is used.

1.7 Common Types Used Throughout PowerSGL Direct

The following is a list of types defined by PowerSGL Direct. Parameters of these types are used in a number of routines. A description of the use of the type is given, along with the C type definition.

`sgl_vector` a 3D vector used to define direction and position vectors.

```
typedef float sgl_vector[3];
```

`sgl_2d_vec` a 2D vector used to define *u* and *v*, and other 2D points.

```
typedef float sgl_2d_vec[2];
```

`sgl_bool` an `int`, but described as Boolean so that it is easier to understand the PowerSGL Direct function descriptions. FALSE is represented by 0, and TRUE by a non-zero value.

```
typedef int sgl_bool;
```

`sgl_colour` A color in PowerSGL Direct is an array of 3 floats—the red component stored at index 0, green at index 1, and blue at 2. In general, the values lie in the range [0.0, 1.0] with 1.0 representing a fully saturated component.

```
typedef float sgl_colour[3];
```

2 DEVICES

2.1 Introduction

A device is a rectangular grid of pixels with a given color encoding scheme (e.g. 24-bit or 16-bit). A device could be the display screen, a window on the display, or a block of off-screen memory, depending on the operating environment.

2.1.1 Device Creation

To create a device, you must specify the desired dimensions and color encoding scheme. If your choice is not possible due to hardware limitations (e.g., framestore size or rendering hardware), then PowerSGL Direct chooses the closest compatible mode.

The following device types are supported:

Screen	The system renders to the framestore memory. In a windowing environment, a window would be created with the suggested dimensions. On a non-windowing system, the display would be set to the closest compatible mode. (Initially only non-windowing is supported).
Memory	Supported via DirectDraw—see Appendix C.

2.1.2 Device Color Encoding

The following is a list of the types of color encoding schemes that devices can potentially use. If a particular device does not support a given color mode, then the closest matching mode is chosen.

16-bit	The device uses 16 bits for each pixel. The image is not dithered.
24-bit	The device uses 8 bits for each of red, green and blue.

These choices are specified in C as:

```
typedef enum
{
    sgl_device_16bit,
    sgl_device_24bit
}sgl_device_colour_types;
```

2.1.3 Double Buffering

If a device is double buffered, then it has two separate image areas. One is being displayed while PowerSGL Direct renders into the other. You must explicitly instruct PowerSGL Direct to swap the display buffer for the rendering buffer, by setting a flag in the render command. Double buffering can reduce artifacts caused by the display scan interacting with the image changes that occur during rendering. Note that some devices may only support single buffers, while others may only support double buffering—the `sgl_get_device` routine (see page 2-3) can be used to find out what the hardware supports.

2.2 Device Routines

2.2.1 Create Screen Device

Creates a screen rendering device. A device must be created before any rendering can be done. The routine attempts to comply with the requested parameters (choosing the closest available), and either returns a name for that device, or an error value.

If on a system which has multiple output devices (say a 3 screen game) the routine may also specify which physical device corresponds to the logical device created.

Function Header

```
int sgl_create_screen_device( int device_number,
                              int x_dimension,
                              int y_dimension,
          sgl_device_colour_types device_mode,
                         sgl_bool double_buffer);
```

Parameters

I `device_number`	If the hardware has several display devices, this specifies which display to use. These are numbered from 0.
I `x_dimension`	The requested *x* dimension (in pixels) of the device to create. If this size is not available, the closest dimension is chosen.
I `y_dimension`	As above but for *y*.
I `device_mode`	This determines the color format of the pixels. Not all formats are supported by all devices—again the closest match is chosen.
I `double_buffer`	If non-zero, this sets the device to be double buffered if this is an available option for the device, otherwise the device is single buffered. Note that some devices will only support double buffering.
`returned value`	This is either a name for the device, or an error value.

Errors

`sgl_err_bad_device`	The device could not be created.
`sgl_err_no_mem`	There was not enough memory to create the device.
`sgl_err_no_name`	There were no available names so the device information has been ignored.

2.2.2 Get Device

Extracts the parameters of a named device. This is useful when the system has been forced to choose a configuration different from parameters you supplied to `sgl_create_device`, and you want to know what they are.

Function Header

```
int sgl_get_device( int device_name,
                    int *device_number,
                    int *x_dimension,
                    int *y_dimension,
sgl_device_colour_types *device_mode,
                sgl_bool *double_buffer);
```

Parameters

I	`device_name`	The device being queried. If this is not a valid device, then the returned parameters are undefined.
O	`device_number`	The number of the display device. Unless there are multiple framestores, this is normally 0.
O	`x_dimension`	The *x* dimension of the device.
O	`y_dimension`	As above but for *y*.
O	`device_mode`	The device color depth.
O	`double_buffer`	Set to non-zero if the device is set to be double buffered.
	`returned value`	Error status.

Errors

`sgl_err_bad_name`	There is no device of the given name.

2.2.3 Delete Device

Relinquishes access to a device. The name of the device becomes invalid.

Function Header

```
void sgl_delete_device( int device);
```

Parameters

I	`device`	The device being deleted.

Errors

`sgl_err_bad_name`	There is no device of the given name.

3 TEXTURE DEFINITION

3.1 Introduction

This chapter describes how texture maps are defined, and MIP mapping, the technique used for anti-aliasing of textures.

3.2 Texture Definitions and MIP Maps

PowerSGL Direct textures are true-color bitmap images. However, because of the hardware and MIP mapping restrictions, there are a limited set of resolutions and per-pixel color depths for the maps.

PowerSGL Direct also supports a translucent texture type which has a per-pixel translucency value. It is otherwise treated like the translucency factor of a material. The per-pixel translucency can be used to make cloudy objects, or for simulating objects with irregular edges.

3.2.1 MIP Mapping

MIP mapping[1] [2] is a technique used to overcome aliasing as textures recede into the distance. With each square texture map, a series of filtered copies of the map are stored, each half the (*x,y*) resolution of the previous one, down to a 1x1 texture map. For example, a 32x32 MIP map contains 32x32, 16x16, 8x8, 4x4, 2x2 and 1x1 texture maps.

When a pixel needs to be textured using a MIP map, PowerSGL Direct estimates how many texture pixels are involved. It then chooses which two maps in the MIP map are of the right resolution, and interpolates between the pixels in those two maps.

When defining a MIP map, the lower resolution maps can either be user-defined or automatically computed by PowerSGL Direct.

3.3 Texture Memory

The texturing process requires considerable memory bandwidth. Because of this, PowerVR, and some other rendering systems, have a dedicated texture memory. This memory is typically in the range of 1 to 16 MB, depending on system budget. The normal way to define a texture is to put it into the texture memory (using `sgl_create_texture` or `sgl_set_texture`, see pages 3-4 and 3-8), and then reference its name in the display list.

---

[1] Foley, van Dam, Feiner and Hughes, “Computer Graphics, Principles and Practice”, Second Edition, Addison and Wesley, 1990, page 826

[2] Watt and Watt, “Advanced Animation and Rendering Techniques. Theory and Practice.”, First Edition, Addison and Wesley, 1992, page 140

3.4 Intermediate Texture Structure

A texture map is defined by using an intermediate texture data structure. There are two basic formats, which both have a header containing an identifying tag and the texture dimensions, and a pointer to the pixel data.

The first, a user-accessible format, is very simple and is intended for you to initially specify textures. The pixels are listed in row-by-row order. Each pixel has 8 bits for each component, with a further 8 bits for alpha. The alpha bits are ignored for non-translucent textures. When creating a translucent texture, a zero alpha value represents a fully opaque pixel, and 255 is fully transparent.

The second is a pre-processed format, which has been prepared for direct loading into texture memory. This intermediate texture format can be generated from the user-accessible format, by using `sgl_preprocess_texture` (see page 3-6).It can then be loaded, say, directly from disk.

3.4.1 C Definition of Intermediate Texture

The C definition for the intermediate texture map is as follows:

```
/*
// Pixel structure for the user-accessible intermediate texture map
*/
typedef struct
{
  unsigned char red;
  unsigned char green;
  unsigned char blue;
  unsigned char alpha;    /*  0=fully opaque, 255=fully transparent  /*
} sgl_map_pixel;
/*
// Structure for the intermediate map
*/
typedef struct
{
  /*
  // ID for type of map.
  // For the user-accessible version, this must be set to IMAP.
  // The pre-processed version sets this to PTxy  where xy depend
  // on the type of the texture map. (x is set to a sgl_map_types value,
  // and y to an sgl_map_sizes value)
  // All other values are reserved.
  */
  char id[4];
  /*
  // X and Y dimensions of texture map
  */
  int x_dim;
  int y_dim;
  /*
  // Array of pixels. The sgl_map_pixels format is only
  // valid for the user-accessible version of the intermediate
  // texture map.
  */
  sgl_map_pixel *pixels;
} sgl_intermediate_map;
```

3.4.2 Size of Pixel Data

To determine the size of the pixel data, use the function `sgl_texture_size`. This takes an `sgl_intermediate_map` structure as a parameter, reads the fields, and returns the size, in bytes, of the pixel data. The function can be used for malloc, copying, and file reading purposes.

Earlier versions of PowerSGL Direct specified the macro `SGL_INTERMEDIATE_MAP_SIZE( x_dim,y_dim)`for determining the size of the pixel data. This macro is being phased out in preference to the above function.

3.4.3 Texture Types

PowerSGL Direct defines four sizes and five types of texture map. The sizes of MIP maps are restricted to 32x32, 64x64, 128x128, and 256x256, while the types are:

16-bit	This has 16 bits per pixel (5 bits per red,green and blue, and one reserved bit).
16-bit MIP Mapped	As above, but includes the lower resolutions for anti-aliasing.
8-bit	This has 3 bits for red and green, and 2 bits for blue. This texture type can be used for very simple images, e.g. text, when texture memory space is critical. Because of the limited color resolution, there is no MIP map version of the 8-bit textures.
Translucent 16-bit	In this format, each pixel has an opacity factor. Four bits are allotted to each of red, green, blue and opacity.
Translucent 16 MIP Map	As above, but MIP mapped.

The C definitions for these are given by:

```
typedef enum
{
  sgl_map_16bit,
  sgl_map_16bit_mm,
  sgl_map_8bit,
  sgl_map_trans16,
  sgl_map_trans16_mm
} sgl_map_types;

typedef enum
{
  sgl_map_32x32,
  sgl_map_64x64,
  sgl_map_128x128,
  sgl_map_256x256
} sgl_map_sizes;
```

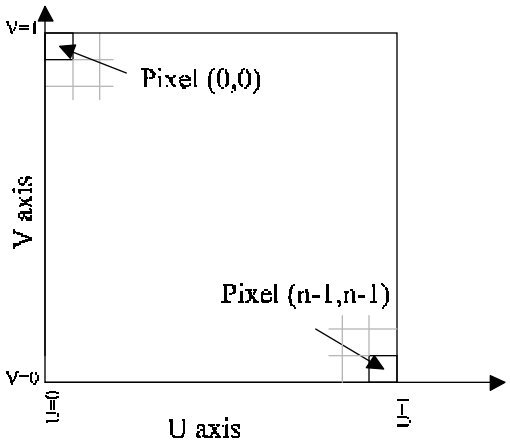
3.4.4 Conversion Between Color Formats

Because the intermediate format is 24-bit, whereas the internal maps have fewer bits, the library converts between the formats. PowerSGL Direct converts these back to 24-bit color when calculating shading.

To help alleviate the loss of color resolution when creating texture maps, you can allow PowerSGL Direct to dither when going from the intermediate map to the internal format.

3.5 Texture Coordinate System

The coordinate system for texture mapping is commonly expressed as a *uv* coordinate system, the *u* corresponding to *x* and the *v* to *y*. This is illustrated in the following diagram:



The *uv* coordinates are scaled such that the whole texture bitmap is mapped to the unit square (0,0)-(1,1). This means that the resolution of a texture map can be changed without having to modify the *uv* coordinates. The texture automatically repeats outside of this unit square.

3.6 Texture Definition Routines

3.6.1 Create Texture

Takes the intermediate texture map and the type of texture map to create, both of which you supply, formats it, and stores it in PowerSGL Direct's texture memory, returning a name for the texture.

The supplied intermediate map can either be in the user-accessible or the pre-processed format. In the case of the pre-processed format, the type, size, etc., of the map are already determined, that is, all parameters except `pixel_data` are ignored.

For MIP mapped textures, you must also state whether the smaller maps are supplied, or if the system should automatically generate them. Note that PowerSGL Direct uses a generic filter to produce the lower resolution MIP map levels.

Because there is a reduction in color resolution when moving from this format of the intermediate map to the internal format, you can specify if the system should dither the texture.

The memory allocated for the `pixel_data` and `filtered_maps` parameters can be released or reused after the call, as the system has its own internal storage.

Function Header

```
int sgl_create_texture( sgl_map_types  map_type,
                        sgl_map_sizes  map_size,
                             sgl_bool  generate_mipmap,
                             sgl_bool  dither,
                sgl_intermediate_map *pixel_data,
                sgl_intermediate_map *filtered_maps[]);
```

Parameters

I `map_type`	The type of texture map to create. This is ignored if a pre-processed intermediate map is supplied.
I `map_size`	If the texture is a MIP map, this specifies the dimensions of the highest resolution map. This is ignored if a pre-processed intermediate map is supplied.
I `generate_mipmap`	If the type of texture is a MIP map, then this specifies if PowerSGL Direct should automatically generate the lower resolution maps. If non-zero, the filtered textures are generated, otherwise, the lower resolution maps are read from the `filtered_maps` parameter. This is ignored if a pre-processed intermediate map is supplied.
I `dither`	If non-zero, PowerSGL Direct performs dithering to convert from the intermediate maps to the lower color resolution. This is ignored if a pre-processed intermediate map is supplied.
I `pixel_data`	This is a pointer to the texture map to use. If, in the case of user supplied intermediate maps, the resolution supplied is larger than the `map_size` requested, the extra pixels are ignored. If the resolution is smaller than the requested size, then the undefined pixels in the texture map are assumed to be black.

I `filtered_maps`	This parameter is used when MIP mapping and automatic generation of the lower resolution maps has not been requested. It is an array of pointers to intermediate texture maps. Element 0, points to the 1x1 map, element 1 to the 2x2 map etc., up to the resolution just smaller than that requested. For example, if the map size requested is `sgl_map_64x64`, then all the resolutions, from 1x1 to 32x32 inclusive, must be supplied. Again, extra pixels are ignored, and if there are too few, the texture is padded with black. Dithering is applied if requested. If a pointer in the array is `NULL`, then that map level is automatically generated from the next highest resolution map.
`returned value`	The value returned is either a name for the texture, or an error value.

Errors

`sgl_err_bad_parameter`	`pixel_data` or `filtered_maps` is a null pointer, or the fields of the intermediate map header are invalid.
`sgl_err_texture_memory_full`	There is not enough memory left for this new texture.
`sgl_err_no_mem`	There is not enough memory so the texture definition has been ignored.
`sgl_err_no_name`	The texture has not been created as there were no available names.

3.6.2 Pre-process Texture

Takes a user-defined intermediate texture map and generates a pre-processed version of the texture map. It takes similar parameters to the `sgl_create_texture` (see page 3-4), except that it fills in a supplied intermediate map with pre-processed data. It allocates the correct amount of space for the pixel data in that map.

The routine is intended for off-line processing of texture maps, so they can be stored and loaded quickly.

As with `sgl_create_texture` you must specify:

- Whether to automatically generate the MIP map levels, or use user-defined maps.
- Whether the system dithers the texture when converting between color resolutions.

All the fields of the returned map are filled in by the routine.

The function returns either the size, in bytes, of the pixels in the pre-processed map, or an error value.

Function Header

```
int sgl_preprocess_texture( sgl_map_types map_type,
                            sgl_map_sizes map_size,
                                 sgl_bool generate_mipmap,
                                 sgl_bool dither,
                       sgl_intermediate_map *pixel_data,
                       sgl_intermediate_map *filtered_maps[],
                       sgl_intermediate_map *processed_map);
```

Parameters

I `map_type`	The type of pre-processed texture map to create.
I `map_size`	The size of the texture map. If the texture is a MIP map, this specifies the dimensions of the highest resolution map.
I `generate_mipmap`	If the type of texture is a MIP map, then this specifies if PowerSGL Direct should automatically generate the lower resolution maps. If non-zero, the filtered textures are generated, otherwise the lower resolution maps are read from the `filtered_maps` parameter.
I `dither`	If non-zero, PowerSGL Direct dithers to convert from the intermediate maps to the lower color resolution.
I `pixel_data`	This is a pointer to the texture map to use. If, in the case of user supplied intermediate maps, the resolution supplied is larger than the `map_size` requested, the extra pixels are ignored. If the resolution is smaller than the requested size, then the undefined pixels in the texture map are assumed to be black.
I `filtered_maps`	This parameter is used when MIP mapping and automatic generation of the lower resolution maps has not been requested. It is an array of pointers to intermediate texture maps. Element 0, points to the 1x1 map, element 1 to the 2x2 map etc., up to the resolution just smaller than that requested. For example, if the map size requested is `sgl_map_64x64`, then all the resolutions, from 1x1 to 32x32 inclusive, must be supplied. Again, extra pixels are ignored, and if there are too few, the texture is padded with black. Dithering is applied if requested. If a pointer in the array is `NULL`, then that map level is automatically generated from the next highest resolution map.
`returned value`	The value returned is either the size, in bytes, of the pixel data in the intermediate texture map texture, or an error value.

Errors

`sgl_err_bad_parameter`	`pixel_data` or `filtered_maps` is a null pointer, or the fields of the supplied intermediate map header are invalid.
`sgl_err_no_mem`	There is not enough memory so the texture definition has been ignored.

3.6.3 Texture Size

Returns the size, in bytes, of the pixels in a given intermediate texture structure, either user-accessible or pre-processed.

Function Header

```
int sgl_texture_size( sgl_intermediate_map *texture_map);
```

Parameters

I	`texture_map`	This is a pointer to the texture map
	`returned value`	The value returned is either the size, in bytes, of the pixel data in the intermediate texture map texture, or an error value.

Errors

`sgl_err_bad_parameter`	`pixel_data` is a null pointer, or the fields of the supplied intermediate map header are invalid.
`sgl_err_no_mem`	There is not enough memory so the texture definition has been ignored.

3.6.4 Set Texture

Redefines the pixels of an existing texture. The dimensions and type of the texture are unchanged. PowerSGL Direct makes its own copies of pixel data during this call, and memory allocated for the `pixel_data` and `filtered_maps` parameters can be released using `free( )`.

Function Header

```
void sgl_set_texture( int texture_name,
                      sgl_bool generate_mipmap,
                      sgl_bool dither,
     sgl_intermediate_map *pixel_data,
     sgl_intermediate_map *filtered_maps[]);
```

Parameters

I	`texture_name`	The name of the texture to be modified.
I	`generate_mipmap`	As for Create Texture.
I	`dither`	As for Create Texture.
I	`pixel_data`	As for Create Texture.
I	`filtered_maps`	As for Create Texture.

Errors

`sgl_err_bad_parameter`	`pixel_data` or `filtered_maps` is a null pointer.
`sgl_err_bad_name`	There is no texture of the given name.
`sgl_err_no_mem`	The display list has become too large for the system so the texture has been ignored.

3.6.5 Delete Texture Map

Deletes a texture map from PowerSGL Direct's texture memory, releasing the space for future use.

Function Header

```
void sgl_delete_texture( int texture_name);
```

Parameters

I	`texture_name`	The name of the texture to be deleted.

Errors

`sgl_err_bad_name`	There is no texture of the given name.

3.6.6 Free Texture Memory

Returns the total number of free bytes in the texture memory. Fragmentation may limit the amount that is useable.

Function Header

```
long sgl_get_free_texture_mem( );
```

Parameters

`returned value`	The number of free bytes in the texture memory.

Errors

None

4 POWERSGL DIRECT

4.1 Introduction

PowerSGL Direct is a low-level library, equivalent to Microsoft D3D and Apple RAVE. It operates at the triangle level, placing triangles into (*x,y*) coordinates, removing them if they are off the screen, calculating screen distances and clipping.

For each frame PowerSGL Direct needs to be given the list of triangles or quads for the frame, which must be added again for subsequent frames if they are still present in the scene. The triangles and quads are already transformed into screen coordinates and lit with their on-screen shade, as no camera or light information is required by the API. These processes may be carried out by the Direct3D (Microsoft) and RAVE (Apple) transformation and lighting stages, or directly by the calling application.

The polygons (faces) are added by specifying sets of triangles or quads that share a common material, although each vertex has its own color value. Each face indicates the vertices for that face, and each vertex contains a position, shade, and texture coordinate. A single instance of a context structure should be used by the application to indicate global constants and the material state for the current set of triangles or quads.

Multiple `SGLCONTEXT`s must not be used within the same frame because the structure is also used to store values required internally by the system on subsequent function calls. For textured sets of triangles the context contains the PowerSGL Direct name of the texture.

4.2 PowerSGL Functions used in PowerSGL Direct

On startup the application must call `sgl_create_screen_device` (see page 2-2) to set up the display parameters.

The PowerSGL Direct texture handling functions such as `ConvertBMPtoSGL`, `sgl_create_texture` or `LoadBMPTexture` must be used to handle the textures in PowerSGL Direct. The name values returned by these functions are used in the `SGLCONTEXT` structure, as demonstrated on page 4-2. Texture caching has not been implemented for PowerSGL Direct because the application has a greater knowledge of what textures are required in each frame.

The Windows 95 functions such as `sgl_use_address_mode`, `sgl_use_ddraw_mode` and `sgl_use_eor_callback` (see Appendix C) are also applicable to PowerSGL Direct. They are used in exactly the same way as in PowerSGL Direct.

4.3 Structures and the Material Flags Enumerated Type

4.3.1 Material Flags

Enumerated Type Header

```
typedef enum
{
    SGLTT_GOURAUD = 0x1,
    SGLTT_TEXTURE = 0x2,
    SGLTT_HIGHLIGHT = 0x4,
    SGLTT_DECAL = 0x8,
    SGLTT_MIPMAP = 0x10,
    SGLTT_GLOBALTRANS = 0x20,
    SGLTT_WRAPU = 0x40,
    SGLTT_WRAPV = 0x80,
    SGLTT_FORCEOPAQUE = 0x100,
    SGLTT_VERTEXTRANS = 0x200,
    SGLTT_MIPMAPOFFSET = 0x400,
    SGLTT_FACESIND3DFORMAT = 0x800,
    SGLTT_USED3DSTRIPFLAGS = 0x1000,
    SGLTT_DISABLEZBUFFER = 0x2000,
     SGLTT_AVERAGECOLOUR = 0x4000,
     SGLTT_VERTEXFOG = 0x8000,
     SGLTT_TRANSBACKGROUND = 0x10000


} SGLTRIANGLETYPE;
```

Members

SGLTT_GOURAUD	Set to smooth shade the faces, or clear for flat shading.
SGLTT_TEXTURE	Texture the faces.
SGLTT_HIGHLIGHT	Specular highlights. The displayed color at a particular location on an object is (VERTEX COLOUR * TEXTURE COLOUR) + SPECULAR COLOUR. It is therefore possible to make textures go completely white when a specular color is applied to them. However, specular highlights are only available on flat shaded surfaces, not smooth shaded surfaces.
SGLTT_DECAL	This flag indicates if the colors supplied with the vertices are ignored when texturing. By default, these colors are used to modulate the texture pixels.
SGLTT_MIPMAP	This flag is currently ignored. It will be used to indicate whether the texture is to be MIP mapped. Currently mipmapping is always used for textures that are loaded with (or pre-processed to generate) MIP map data. Another way to turn MIP mapping off, or reduce its effect, is to use a MIP map offset (see SGLTT_MIPMAPOFFSET on page 4-3).
SGLTT_GLOBALTRANS	Set this flag to enable translucency for the entire set of polygons being added, according to the value of u32GlobalTrans. Smooth

	shading and translucency on the same polygon is not currently possible.
SGLTT_WRAPU/V	This is used to implement a feature of Direct3D. Adds 1 to *uv* values that are close to zero when they are used with values just less than 1.This is useful for objects such as cylinders where there is a line of vertices along the cylinder that should have zero texture coordinates for the faces on one side, and texture coordinates of 1 for the faces on the other side. Note: It is much more efficient to supply texture coordinates that already have the correct *uv* values.
SGLTT_FORCEOPAQUE	Not currently used.
SGLTT_VERTEXTRANS	Enables the use of the alpha (translucency) values at each vertex. If vertex alpha is enabled for a polygon then the translucency will be set according to the alpha value of the first vertex of that polygon.
SGLTT_MIPMAPOFFSET	Set this flag to shift the MIP map levels (see `n32MipmapOffset`).
SGLTT_FACESIND3DFORMAT	Used by the PowerVR Direct3D driver.
SGLTT_USED3DSTRIPFLAGS	Used by the PowerVR Direct3D driver. This flag is ignored if `SGLTT_FACESIND3DFORMAT` is not set.
SGLTT_DISABLEZBUFFER	Used by the PowerVR Direct3D driver. Note: Because this modifies depth values, the automatic per-pixel fogging does not function correctly on triangles and quads that have this flag set.
SGLTT_AVERAGECOLOUR	Averages the colors of three vertices when flat shading rather than just using the first vertex's color.
SGLTT_VERTEXFOG	Implements a feature of Direct3D, and its behaviour is subject to change without notice.
SGLTT_TRANSBACKGROUND	(Ignored by all but Midas 5 (PCX2) and later variants) - Considers the background to be transparent, only overwriting pixels in the 2D framestore when obscured by other objects.This is primarily for when 3D is rendered over a 2D backdrop.

4.3.2 Shadow/Light Volume Mode

Enumerated Type Header

```
typedef enum
{
    NO_SHADOWS_OR_LIGHTVOLS,
    ENABLE_SHADOWS,
    ENABLE_LIGHTVOLS

} SGLSHADOWTYPE;
```

4.3.3 Rendering Empty Regions

Enumerated Type Header

```
typedef enum
{
    ALWAYS_RENDER_ALL_REGIONS,
    DONT_RENDER_EMPTY_REGIONS

} SGLRENDERREGIONS;
```

4.3.4 Context

Structure Header

```
typedef struct tagSGLCONTEXT
{
        sgl_bool bFogOn;
           float fFogR,fFogG,fFogB;
      sgl_uint32 u32FogDensity;
        sgl_bool bCullBackfacing;
      sgl_uint32 u32Flags;
             int nTextureName;
 sgl_uint32 Reserved;
        sgl_bool bDoClipping;
      sgl_colour cBackgroundColour;
   SGLSHADOWTYPE eShadowLightVolMode;
           float u.fShadowBrightness;
      sgl_uint32 u.u32LightVolColour;
        sgl_bool bFlipU,bFlipV;
        sgl_bool bDoUVTimesInvW;
      sgl_uint32 u32GlobalTrans;
       sgl_int32 u32MipmapOffset;
SGLRENDERREGIONS RenderRegions;
      sgl_uint32 u32Reserve9[9];

} SGLCONTEXT,*PSGLCONTEXT;
```

Structure Parameters

I bFogOn	Set to non-zero to enable (or disable) fog on a set of triangles (or quads). Fog is calculated on a per-pixel basis by the hardware from the 1/w depth value specified by the vertex information. The fog model implemented by the hardware is an exponential function which simulates real-world uniform density fog.
I fFogR/G/B	The fog color: 0 ≤ component ≤ 1. This is a ‘per-render’ setting, and as such, only the value which is set at the time of the `sglti_render` call is used. The fog color affects all objects that have been supplied with bFogOn set.
I u32FogDensity	The fog density. The values range from 0 (hardly any fog) to 31 (very foggy). Each increment doubles the density of the fog. This only affects triangles that have been added with `bFogOn` set. As for the fog color, this is a ‘per-render’ setting, and as such, only the value which is set at the time of the `sglti_render` is used
I bCullBackfacing	Set to non-zero to remove backfacing triangles (those whose vertices appear in a counterclockwise order on the screen).
I u32Flags	Material type for the triangles, using ORed members of `SGLTRIANGLETYPE`. These are the flags that are specific to the set of triangles or quads currently being added to the frame for the scene.
I nTextureName	Texture name for the triangles.

I	`Reserved`	Reserved for future use. This field will be ignored unless a specific flag (TBD) is set in `u32Flags`.
I	`bDoClipping`	Enable or disable clipping. If this is zero then all vertices must be on the screen.
I	`cBackgroundColour`	The scene background color.
I	`eShadowLightVolMode`	Enable or disable shadows and/or light volumes. This value must not be changed during a scene (except for switching between shadows and light volumes where both are required) as hardware lockups will occur. Refer to `sgltri_shadows` for more details.
I	`fShadowBrightness`	The shadow brightness. This takes values between 0 and 1. A value of 0 makes shadows completely dark, 0.5 makes them half the brightness of surrounding objects, and 1 gives no shadowing at all, but still sends the (ineffective) shadows to the hardware. The value is only used when `eShadowLightVolMode` is set to `ENABLE_SHADOWS`.
I	`u.u32LightVolColour`	The light volume color (D3DCOLOUR format). This is the color added to the environment whenever a light volume intersects it. The value is only used when `eShadowLightVolMode` is set to `ENABLE_LIGHTVOLS`. Different values may be used for different light volume creation calls. Refer to `sgltri_shadows` for more details.
I	`bFlipU/V`	Swap the directions of the horizontal and vertical texture axes respectively for subsequent texture repetitions.
I	`bDoUVTimesInvW`	If this is set, then it is assumed that all the *uv* coordinates in vertex structures have not already been multiplied by `fInvW`. This enables Direct3D `D3DTLVERTEX` structures to be cast to the `SGLVERTEX` type without changing the contents.
I	`n32MipmapOffset`	A signed mipmap offset value. Values for this will usually be small (for example, a value of 3 for a mipmapped ground plane to remove blurring too close to the camera).
I	`RenderRegions`	Whether or not to render empty regions. The current allowed values for this are enumerated in the `SGLRENDERREGIONS` type described above.
I	`u32Reserve9[9]`	Reserved words (currently nine of them). These should all be set to zero for compatibility with future versions of PowerSGL Direct.

4.3.5 Vertex

This structure is currently physically equivalent to Direct3D's D3DTLVERTEX (so one can be cast to the other), with the exception that Direct3D's structure specifies *u* and *v* values that have not been multiplied by the *1/w* value. When casting D3DTLVERTEX structures to the SGLVERTEX type set the bDoUVTimesInvW in the SGLCONTEXT structure. RAVE vertices already contain *u/w* and *v/w*.

The coordinate values are compatible with the Direct3D D3DTLVERTEX structure.

It is not necessary to sort the polygons in *z* order before sending them to PowerSGL Direct.

Structure Header

```
typedef struct tagSGLVERTEX
{
         float fX,fY;
         float fZ; /*"dummy" parameter */
        fInvW;
         float fFogR,fFogG,fFogB;
    sgl_uint32 u32Colour;
    sgl_uint32 u32Specular;
         float fUOverW,fVOverW;

} SGLVERTEX,*PSGLVERTEX;
```

Structure Parameters

I	fX fY	Projected screen coordinates. 0 ≤ fX, fY ≤ screen width/height (up to 1024).
I	fZ	This is a dummy value which is used only to keep the data structure compatible with D3D. All depth sorting information is obtained from fInvW.
I	fInvW	1/*w* for the perspective projection. This value should ideally lie in the range (0.0, 1.0), where 0.0 represents a distance of infinity, and 1.0 implies a 'distance' of 1.0f. Note that because precision is finite, positioning accuracy decreases as the distance approaches infinity. (In order to cope with some error in supplied values, PowerSGL Direct will, in fact, tolerate values up to 10.0f (i.e., a distance of 0.1f). As a last resort, this can be scaled, under Windows95 via the use of Application Hints 'Near Z Clip'. This reduces portability)
I	u32Colour	Material color (D3DCOLOUR format).
I	u32Specular	Specular color (D3DCOLOUR format).
I	fU/VOverW	Texture coordinates. Note that these are normally the *u* and *v* coordinates multiplied by fInvW, but if bDoUVTimesInvW is set in the context structure, PowerSGL Direct assumes that this multiplication has not been performed.

4.4 PowerSGL Direct Routines

4.4.1 Start a New Frame

Initializes the system for a new frame.

Function Header

```
void sgltri_startofframe( PSGLCONTEXT pContext);
```

Parameters

I	pContext	Pointer to the SGLCONTEXT structure that the application has created.

Errors

None

4.4.2 Add a Set of Triangles or Quads to the Frame

These routines send a set of triangles or quads to the hardware. `sgltri_startofframe` must be called (see page 4-8) before the first of these calls in a frame. `sgltri_triangles` and `sgltri_quads` can then be called many times to add subsequent sets of triangles or quads before the `sgltri_render` function (see page 4-11) is called to render the complete scene.

The errors set by these functions may be read by `sgl_get_errors` (see page 1-3). It is assumed that the face arrays refer to valid vertices.

The face arrays contain indices into the vertex arrays. The vertices for each face are listed in clockwise order when viewed from the front of the face. In flat shading the colors of the three or four vertices are averaged to determine the constant face color. All four vertices of a quad must lie on a flat plane.

Note: Using a single call to add a number of triangles/quadrilaterals is *much* more efficient than repeated calls to add a single face at a time.

Function Header

```
void sgltri_triangles( PSGLCONTEXT pContext,
                              int nNumFaces,
                              int pFaces[][3],
                       PSGLVERTEX pVertices);
```

Parameters

I	pContext	The pointer to the SGLCONTEXT structure that the application has created and used in the `sgltri_startofframe` call (see page 4-8).
I	nNumFaces	The number of faces.
I	pFaces	The list of three indices into the vertex array for each face.

I	`pVertices`	The pointer to the first vertex in the array of vertices, some or all of which are referred to by `pFaces`.

Errors

sgl_no_err	Success.
sgl_err_bad_parameter	One or more of the pointer parameters was null, or nFaces was negative.
sgl_err_failed_init	Initialization failed (probably out of memory).

Function Header

```
void sgltri_quads( PSGLCONTEXT pContext,
                          int nNumFaces,
                          int pFaces[][4],
                   PSGLVERTEX pVertices);
```

Parameters

I pContext	Pointer to the SGLCONTEXT structure that the application has created and used in the sgltri_startofframe call (see page 4-8).
I nNumFaces	Number of faces.
I pFaces	List of four indices into the vertex array for each face.
I pVertices	Pointer to the first vertex in the array of vertices, some or all of which are referred to by pFaces.

Errors

sgl_no_err	Success.
sgl_err_bad_parameter	One or more of the pointer parameters was null, or nFaces was negative.
sgl_err_failed_init	Initialization failed (probably out of memory).

4.4.3 Add a Convex Shadow or Light Volume

This function is similar to sgltri_triangles (see page 4-8) except that it defines an infinite convex shadow or light volume with the triangle faces defining the finite or infinite surfaces of the volume. The vertices in each face must be in clockwise order when viewed from outside the volume.

To put shadows in a scene, set eShadowLightVolMode in the SGLCONTEXT structure to ENABLE_SHADOWS before the call to sgltri_startofframe, and keep that state for the entire frame. For light volumes, set the value to ENABLE_LIGHTVOLS in the same way for the entire frame. To have both shadows and light volumes, set the state as for light volumes only, but change the value to ENABLE_SHADOWS before adding each shadow, and back to ENABLE_LIGHTVOLS for the remainder of the scene.

The entire shadow volume must be defined by a single call to sgltri_shadow. Separate sgltri_shadow calls create separate shadow volumes. It is essential that the entire shadow volume is convex. If not, it will be a completely incorrect shape on the screen.

Shadows make the scene darker than it otherwise would be, and light volumes make the scene lighter. This means that you will need to reduce the colors of the faces to values less than full saturation (e.g. white) in order to see the light volumes.

Where shadow planes are identical to object planes (i.e., those used to define part of the shadow volume) those object planes are not put in shadow.

If the bounding box is `NULL`, PowerSGL Direct takes the smallest and largest values of the vertex coordinates of the shadow volume to determine the rectangle of the screen that the shadow affects. It is usually necessary to specify the rectangle explicitly by having a bounding box, such as {0,0,640,480} for the whole screen. If there are many shadow planes in the scene PowerVR is faster if the bounding boxes are made as small as possible.

Adding shadows has about the same impact on performance as adding triangles the same size, so adding some simple shadow volumes will increase the quality of the image without slowing the game down significantly.

Function Header

```
void sgltri_shadow( PSGLCONTEXT pContext,
                           int nNumFaces,
                           int pFaces[][3],
                    PSGLVERTEX pVertices,
                         float fBoundingBox[2][2]);
```

Parameters

I	`pContext`	The pointer to the `SGLCONTEXT` structure that the application has created and used in the `sgltri_startofframe` call (see page 4-8).
I	`nNumFaces`	The number of faces.
I	`pFaces`	The list of three indices into the vertex array for each face.
I	`pVertices`	The list of vertices referenced by the faces
I	`fBoundingBox`	Bounding box (see description above).

Errors

`sgl_no_err`	Success.
`sgl_err_bad_parameter`	One or more of the pointer parameters was null, `nFaces` was negative, or `nFaces` was greater than `SGL_MAX_PLANES`.
`sgl_err_failed_init`	Initialization failed (probably out of memory).

4.4.4 Render a Frame

This routine renders the entire scene of triangles.

Function Header

```
void sgltri_render( PSGLCONTEXT pContext);
```

Parameters

I	pContext	Pointer to the SGLCONTEXT structure that the application has created and used in previous triangle API function calls for the frame.

Errors

None

4.4.5 Determine Whether the Render is Complete

Loops until the hardware either finishes or times out, otherwise the state is sampled and returned.

Enumerated Type Header

```
typedef enum
{
    IRC_RENDER_COMPLETE,
    IRC_RENDER_NOT_COMPLETE,
    IRC_RENDER_TIMEOUT,
} IRC_RESULT;
```

Function Header

```
IRC_RESULT  sgltri_isrendercomplete( PSGLCONTEXT pContext,
                                     sgl_uint32 u32Timeout);
```

Parameters

I	pContext	Pointer to the SGLCONTEXT structure that the application has created and used in previous triangle API function calls for the frame.
I	u32Timeout	Maximum length of time in milliseconds to wait for the hardware to finish rendering.

Errors

None

5 RANDOM NUMBER GENERATION

5.1 Introduction

The routines in this section are designed to offer pseudo-random number generation for applications independent of the host environment. Setting the seed of the generator to a certain value produces the same sequence of pseudo-random numbers on all platforms running PowerSGL Direct.

5.2 Random Number Routines

5.2.1 Set Seed of Random Number Generator

This routine defines the seed of the pseudo-random number generator.

Function Header

```
void sgl_srand( unsigned long seed);
```

Parameters

I	seed	The value of the new seed.

Errors

None

5.2.2 Generate Random Number

This routine generates the next 31-bit positive integer value in the pseudo-random sequence.

Function Header

```
long sgl_rand( );
```

Parameters

returned value	The pseudo-random number.

Errors

None

6 VERSION INFORMATION

6.1 Introduction

6.2 Retrieving Current Versions

The routine below returns platform-independent version information. Platform-dependent information is covered in Appendix C.

6.2.1 Return Library Version and Required sgl.h Version

Returns a pointer to the internally-held instance of the structure, which contains valid pointers to the appropriate zero-terminated information.

Structure Header

```
typedef struct
{
    char *library;
    char *required_header;
} sgl_versions;
```

Structure Parameters

0 `library`	String representation of the PowerSGL Direct library version.
0 `required_header`	The version number of the `sgl.h` file that corresponds to the library.

Function Header

```
sgl_versions *sgl_get_versions ( );
```

Parameters

`returned value`	Pointer to the internal versions structure.

Errors

None

Appendix A - DOCUMENT HISTORY

Date	Changes
2 May 1995 Draft A	Initial version.
23 Jun 1995 Draft C	Removed Delete routines, simplified Texture definition.
18 Dec 1995 Draft F	Added error code descriptions, third normal to smooth shading parameters for convex objects and descriptions for `sgl_set_ambient`,`sgl_delete_mesh`,`sgl_srand` and `sgl_rand`.
23 May 1996	New functions added, and some corrections to the mesh definitions.
24 Jun 1996	Triangle API documentation added.
14 Oct 1996	Updated and reformatted for publication.
4 Feb 1997	Added more information on Context fields, and corrected SGLTT values.

Appendix B - BIBLIOGRAPHY

“Computer Graphics. Principles and Practice”

Foley, van Dam, Feiner, and Hughes

Second Edition

Addison and Wesley, 1990

“Advanced Animation and Rendering Techniques. Theory and Practice”

Watt and Watt

First Edition

Addison and Wesley, 1992

Appendix C - PowerSGL Direct Windows 95 Extensions

Introduction

PowerSGL Direct has been extended for use with DirectDraw graphics cards in Windows 95. There are two basic modes of operation for Windows 95: PowerSGL Direct DirectDraw mode, and PowerSGL Direct address mode.

PowerSGL Direct's DirectDraw mode allows the application to leave all of the buffer management to PowerSGL Direct. In this mode the application runs full-screen 640x480 16-bit RGB; you do not have to manage page flipping or display memory, as it all happens automatically. PowerSGL Direct creates and controls the DirectDraw objects. Other display modes are supported depending on the amount of graphics memory.

PowerSGL Direct's low-level address mode allows the application to create and control the display buffers. PowerSGL Direct renders the 3D image at the start address defined by the application. This allows the application to implement various modes, including 3D-in-a-window. 3D-in-a-window requires the use of techniques such as overlay and blting to avoid overwriting menus and dialog boxes.

Use Address Mode

Allows the application to specify exactly where PowerVR renders the 3D image. This address must point to a buffer of contiguous, PCI-addressable page locked memory. PowerSGL Direct converts the address from logical (CPU) address to physical (PCI) address.

Function Header

```
int sgl_use_address_mode ( PROC_ADDRESS_CALLBACK ProcNextAddress,LPDWORD
                *pStatus);
```

Parameters

I	`ProcNextAddress`	The address of a function which is the application's Address Mode callback. This callback function is called by PowerSGL Direct during an `sgl_render` command to get the address of the buffer for rendering.
O	`pStatus`	This is the address of a pointer which PowerSGL Direct sets to point to the status DWORD.

Errors

None

Notes

This must be called before `sgl_create_screen_device` (see page 2-2).

The application must provide the following information to PowerSGL Direct on return of the callback:

```
typedef struct
{
  LPVOID    pMem;
  WORD     wStride;
  BYTE     bBitsPerPixel;
} CALLBACK_ADDRESS_PARAMS,*P_CALLBACK_ADDRESS_PARAMS;
```

In address mode the application must provide a callback similar to the following example:

```
/*This function is called by SGL during sgl_render */
int ProcNextAddress( P_CALLBACK_ADDRESS_PARAMS pParamBlk)

{
  /*Lock next render buffer to get address and stride */

  /*Complete the CALLBACK_ADDRESS_PARAMS structure */
  pParamBlk ->pMem = pNextBuffer;
  pParamBlk ->wStride = wStride;
  pParamBlk ->bBitsPerPixel = 16;
  return sgl_no_err;
}
```

The status DWORD `pStatus` is modified at interrupt time and so must be declared by the application as volatile to avoid optimizations. This flag currently has bit 0 set if rendering has ended, bit 0 cleared if rendering is active.

Use DirectDraw Mode

Switches PowerSGL Direct into DirectDraw mode,i.e., PowerSGL Direct takes responsibility for creating and maintaining the DirectDraw surfaces. The default mode is 640x480 16-bit full-screen double buffered. The display mode is defined by the dimensions passed into the `sgl_create_screen_device` call (see page 2-2).

Function Header

```
int sgl_use_ddraw_mode ( HWND hWnd,PROC_2D_CALLBACK Proc2d);
```

Parameters

I	`hWnd`	The application's window handle.
I	`Proc2d`	The application’s 2D callback routine. `NULL` if none.

Errors

None

Notes

The application can pass in the address of its 2D callback function which PowerSGL Direct then calls just before the DirectDraw flip of a rendered surface. This allows the application to modify the surface before it becomes visible. Enough data is passed back to the application to allow it to:

- use GDI to write text or graphics onto the image
- use DirectDraw to do fast Blting from off-screen surfaces
- write to the frame buffer memory directly

In DirectDraw mode the application can provide a 2D callback similar to the following example:

```
/*This function is called by SGL during sgl_render */
int Proc2d( P_CALLBACK_ SURFACE_PARAMS lpSurfaceInfo)

{
    HDC                 hdc;
    /*Get device context for surface */
    if ( IDirectDrawSurface_GetDC ( lpSurfaceInfo->p3DSurfaceObject,and hdc) ==
DD_OK)
    {
     /*Output some text on top of the 3d image */
        SetBkMode ( hdc,TRANSPARENT);
        SetBkColour ( hdc,RGB( 0,0,255));
        SetTextColour ( hdc,RGB( 255,255,0));
        strcpy ( szTmp,"2d text goes here");
        TextOut ( hdc,50,10,szTmp,lstrlen( szTmp));
        IDirectDrawSurface_ReleaseDC ( lpSurfaceInfo->p3DSurfaceObject,hdc);
    }
    return sgl_no_err;
}
```

Use End of Render Callback

Allows an application to give PowerSGL Direct the address of a routine that is called as soon as the hardware has finished rendering the previous frame. The callback is in fact called during an `sgl_render` command (see page 4-11) when PowerSGL Direct first detects that the previous frame has been rendered.

Function Header

```
int sgl_use_eor_callback ( PROC_END_OF_RENDER_CALLBACK ProcEOR);
```

Parameters

I	`ProcEOR`	End of render callback.

Errors

None

Notes

The End Of Render callback allows the application to unlock the render buffer (address mode) and process any pending windows messages as soon as possible to prevent windows from locking up.

A side-effect of locking a DirectDraw surface is that the Windows display freezes until the surface is unlocked. The DirectDraw documentation says that to stop VRAM from being lost when accessing a surface, DirectDraw holds the Win16 lock between **Lock** and **Unlock** operations. The Win16 lock is the critical section when serializing access to GDI and USER, but while preventing the mode from being changed by other applications, it stops Windows from running. Therefore, you should keep **Lock**/**Unlock** and **GetDC**/**ReleaseDC** periods short. However, stopping Windows prevents debuggers from being used in between **Lock**/**Unlock** and **GetDC**/**ReleaseDC** operations.

`sgl_use_eor` is used in sample code to overcome this limitation.

In practice, most graphics cards do not need the buffer to be locked during render, and so this facility may not be needed for an OEM solution. We recommend the use of the sample code in the PowerVR SDK as a basis for understanding these issues.

Get Windows Versions

Returns Windows-specific PowerSGL Direct version information, i.e., a pointer to the internally-held instance of the structure shown below, which contains valid pointers to the appropriate zero-terminated information.

Function Header

```
sgl_win_versions *sgl_get_win_versions ( );
```

Structure Header

```
typedef struct
{
  char *required_sglwin32_header;
  char *sgl_vxd_rev;
  char *pci_bridge_vendor_id;
  char *pci_bridge_device_id;
  char *pci_bridge_rev;
  char *pci_bridge_irq;
  char *pci_bridge_io_base;
  char *tsp_rev;
  char *tsp_mem_size;
  char *isp_rev;
  char *mode;
  char *status;
  char *build_info;
} sgl_win_versions;
```

Structure Parameters

0	`required_sglwin32_header`	Required sglwin32.h version
0	`sgl_vxd_rev`	Required VxD version
0	`pci_bridge_vendor_id`	PCI bridge vendor ID
0	`pci_bridge_rev`	PCI bridge revision

0	`pci_bridge_irq`	PCI Interrupt
0	`pci_bridge_io_base`	PCI Base address of bridge
0	`tsp_rev`	TSP chip revision
0	`tsp_mem_size`	TSP on-board memory (MB)
0	`isp_rev`	ISP chip revision
0	`mode`	Reserved (for mode)
0	`status`	Reserved (for status)
0	`build_info`	Reserved (for build info)

Parameters

returned `value`	Pointer to the internal windows-specific versions structure.

Errors

None

Appendix D - FUTURE CHANGES

Below are changes to PowerSGL Direct planned for the next release.

General

`sgl_err_bad_parameter` will be more widely used to indicate invalid parameter cases, such as `NULL` pointers.

Bitmap To Internal Texture Conversion

The functions described below are currently available (but subject to change) for converting high color Windows bitmap files to the PowerSGL Direct intermediate map format (see section 3.3).

ConvertBMPtoSGL

```
sgl_intermediate_map ConvertBMPtoSGL( char *filename,sgl_bool translucent)
```

Loads a texture from a 24-bit .BMP file. If `translucent` is set a second BMP is opened with a *t* prefixed to the filename, and the red channel of that bitmap is used for opacity.

LoadBMPTexture

```
int LoadBMPTexture( char *filename,sgl_bool translucent,sgl_bool
generate_mipmap,sgl_bool dither)
```

Loads a texture from a 24-bit .BMP file. If `translucent` is set a second BMP is opened with a t prepended to the filename, and the red channel of that bitmap is used for opacity. If an identical texture has already been created the handle from the original is returned and no new texture is created. `generate_mipmap` determines whether the texture is to be mipmapped. If `dither` is set then the texture is dithered into mipmaps. The return value is the texture name or a negative error value.

FreeBMPTexture

```
void FreeBMPTexture( int texture_name)
```

Looks for the given texture in the cache. If the texture is found then its usage count is decremented. If this count reaches zero then the texture is deleted.

FreeAllBMPTextures

```
void FreeAllBMPTextures( )
```

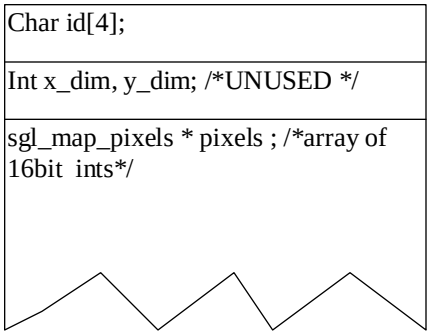
Removes all textures that have been loaded with `LoadBMPTexture`.

Appendix E - PRE-PROCESSED TEXTURE FORMAT

This appendix describes the pre-processed texture format which is produced by the function `sgl_preprocess_texture`. This is not intended to be the only format, but will be supported in all future versions of PowerSGL Direct.

Pre-processed Header Values

All versions of the pre-processed textures will use the same header as described earlier. Diagrammatically this looks like this:



In the only format of pre-processed texture implemented at present, the `id[]` values are:

`id[0] == 'P'`

`id[1] == 'T'`

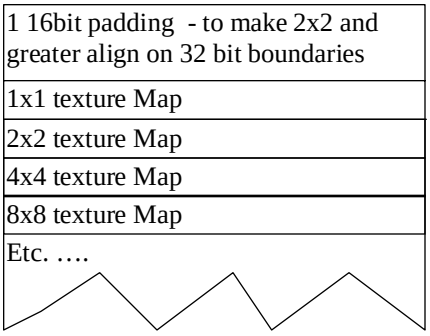
`id[2] == map_type` (as specified when preprocessed texture was created)

`id[3] == map_size` (as specified when preprocessed texture was created)

All other values are reserved.

Arrangement of Maps in Pixel Data

Although the `pixels` field in the `sgl_intermediate_map` type is defined to point to `sgl_map_pixels`, in the case of the pre-processed texture, it will actually point to an array of 16 bit integers (i.e. `shorts`). In the case of a MIP map, this array is formatted as:



Non-MIP map data has no padding and only stores a single resolution map.

Format of Individual Pixels

Each pixel occupies 16 bits and is in one of two formats. Non-translucent (i.e. no alpha channel) pixels are in the format:

1 bit (reserved)	Red (5 bits)	Green (5 bits)	Blue (5 bits)

while the translucent textures are:

Alpha (4 bits)	Red (4 bits)	Green (4 bits)	Blue (4 bits)

Pixel Order within Texture Maps

Pixels are not stored in row order—storing in row order can result in huge numbers of expensive page breaks during the texture mapping process. Instead, they are arranged in a pattern which increases the likelihood that adjacent pixels in any direction are stored in the same page of texture RAM.

If the pixels were stored in row order, then the offset to pixel (*x*,*y*) in, say, a 64x64 texture map, where the bit patterns for *x* and *y* are $x_5x_4x_3x_2x_1x_0$ and $y_5y_4y_3y_2y_1y_0$ respectively, would be $y_5y_4y_3y_2y_1y_0x_5x_4x_3x_2x_1x_0$. That is, the *x* and *y* offsets are effectively concatenated. (Or alternatively, *y**64+*x*). Note that an increment in *y* results in a large address change, which is very likely to result in a page break.

With the arrangement used in the pre-processed, the offset for pixel (*x*,*y*) is given by:

$$x_5y_5x_4y_4x_3y_3x_2y_2x_1y_1x_0y_0$$

The bits of the *x* and *y* offsets are alternated, so that small increments in *x* and *y* result in small changes in the actual texel address.

Stepping Through the Pixels Incrementing *x*

A quick way to step to the next pixel in the *x* direction (in software), is to keep the *x* and *y* parts of the index separate. To increment *x*, simply ADD 0x5556 and then AND with 0xAAAA, which removes the *y* position bits.

To get the actual offset of pixel (*x*,*y*), just OR in the *y* value. This can either be generated by a similar technique, or by a 256 element lookup table.

Appendix F - LIST OF FUNCTIONS

Index

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W

X

Y