PowerSGL™

PowerVR Graphics Library/API

Programmer's Reference Manual

Edition 3

16th April 1997

COPYRIGHT

DISCLAIMER

TRADEMARKS

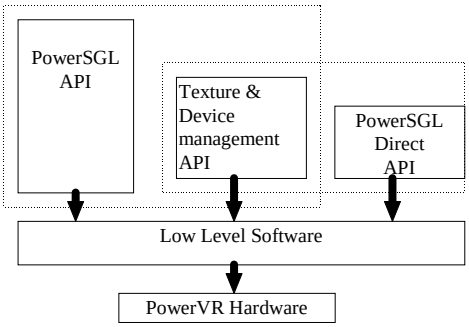
Table of Contents

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).

In contrast, 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.



Although the two APIs are supplied in the same library and both use common routines to manage the PowerVR texture memory and output devices, they are otherwise independent.

1.2 Overview of this Guide

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

1.2.1 Format and Conventions

The manual is divided into chapters covering functional areas of PowerSGL. PowerSGL Direct is covered in its own chapter.

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 code are shown in courier type, for example, `( int device)`.

All PowerSGL #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, destroying all viewports attached to it in the process. 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 Features

The PowerSGL graphics library is designed to fully exploit the rendering features of the PowerVR ASICs through a high-level, C-based interface, effectively hiding the complex interface to the hardware. A high-level interface also makes it possible to produce a software-only renderer for situations where there isn't any PowerVR hardware. A software-only version may not provide all the features and will only provide a fraction of the performance, but it would allow an upgrade path to real hardware.

PowerSGL features hidden surface removal rendering (to pixel or sub-pixel accuracy), with visual effects including perspective correct anti-aliased texturing, smooth shading with specular highlights, real-time shadows, and fogging (depending on the supporting hardware).

PowerSGL contains a number of concepts and facilities. These are briefly described below, but are covered in detail in later sections.

Devices and Viewports	are the destinations of rendering results. All rendering output goes to a viewport, which is a rectangular region (which can have rectangular holes) of the pixels of a device.
Display Lists	are used to define images. They contain virtually all the information, such as objects, lights etc., that are relevant to a scene. Multiple lists can be used for the creation of multiple scenes. The lists may be nested in order to build hierarchical structures, permitting complex animation and inheritance of material properties. They are also used as a powerful form of model instancing.

Primitives (Objects)	are the visible elements of any scene. Convex polyhedra and polygonal meshes are supported. Both may be smooth shaded to give the appearance of curved surfaces. Objects are built from arbitrary conglomerations of primitives.
Materials	define the surface properties of objects, including their diffuse and specular colors. These properties may also be controlled by textures. Objects may also be obscured by fog effects.
Transformations	are used to position objects, cameras, lights, materials, etc. in the scene, using combinations of rotation, scaling, and translation.
Lights and shadows	illuminate the objects and can interact with those objects to cast shadows. They can be placed arbitrarily in the scene.
Cameras	actually view the scene. A camera may be placed arbitrarily in the scene, giving a view of the scene from that position and orientation.
Quality	determines speed and quality trade-offs. The quality can be varied frequently throughout a scene. For example objects in the distance might not need to be smooth shaded if flat shading is sufficient.
Level of detail	can be used to automatically change between different object and material models depending on how significant the model's contribution is to the image.
Collision detection	provides some support for VR and games applications, for detecting collision of points with objects.

1.4 Names, Returned Values and Errors

All items put into PowerSGL's display lists are managed entirely by the library; you have no direct access to the stored data. To allow indirect manipulation of these items, the system maintains a set of names. A name is a positive integer assigned by PowerSGL, which acts as a handle to a particular item. You can then reference and modify items via their names.

If an item is not named when created, it is anonymous and cannot be directly modified. Anonymous items enable PowerSGL to optimize the display list.

PowerSGL 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 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 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 function that fails to initialize the software or hardware (usually the first PowerSGL 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 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 is used.

1.7 Common Types Used Throughout PowerSGL

The following is a list of types defined by PowerSGL. 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 function descriptions. FALSE is represented by 0, and TRUE by a non-zero value.

```
typedef int  sgl_bool;
```

`sgl_colour` A color in PowerSGL 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 AND VIEWPORTS

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.

A viewport is a rectangular subset of the pixels of a device, which is aligned with the *x* and *y* axes of that device. A viewport may also have rectangular holes. All images are rendered to a viewport. Viewports can be used to either produce completely independent views of potentially different scenes, or to separately render portions of a larger image, depending on the settings of the camera rectangle, which is described on page 2-2.

For convenience, each device has a default viewport which contains all the device pixels. Some routines that take a viewport as a parameter can also accept a device name instead, implying use of the default viewport for the device.

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 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:

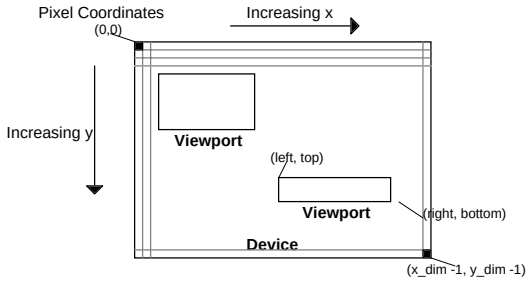
```
typedef enum
{
    sgl_device_16bit,
    sgl_device_24bit
}sgl_device_colour_types;
```

2.1.3 Double Buffering

If a device is double buffered, then it has two separate image areas. One is being displayed while PowerSGL renders into the other. You must explicitly instruct PowerSGL 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 2-7) can be used to find out what the hardware supports.

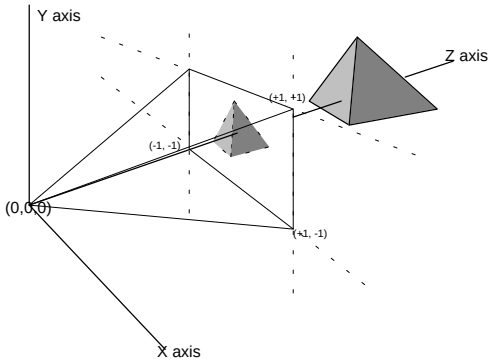
2.1.4 Pixel Coordinate System

The following diagram shows a device with its coordinate system, with two viewports. The coordinates specify the position of the center of each pixel.



2.1.5 Viewport Camera Rectangle

As stated on page 2-1, viewports can be used to display complete images, or to render portions of larger images. To understand how this is done, look at the diagram below:



The camera projects a 3D model onto an infinite projection plane, perpendicular to the *z* axis, and this 2D image is then mapped to pixels in the device or viewport. To specify this mapping in PowerSGL, you must define where the [-1, -1] to [1,1] square at the center of the projection plane maps to on the device—which is a rectangle aligned with the device axes. This rectangle is called the *camera rectangle*.

The boundaries are supplied in floating point device pixel coordinates, which allows fractional positioning, and can be larger than the device if desired. Two examples of using the camera rectangle are given in the following diagrams.

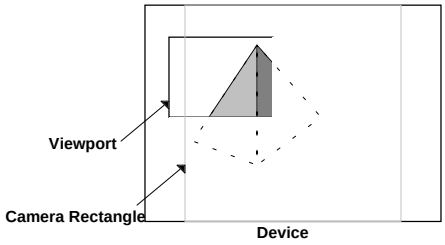


Figure A

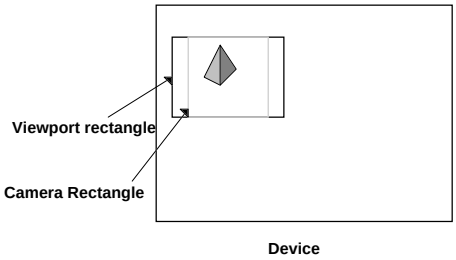


Figure B

In Figure A, only a portion of the complete image is rendered, as the camera rectangle is set up as if the image were scaled to fit the entire device.

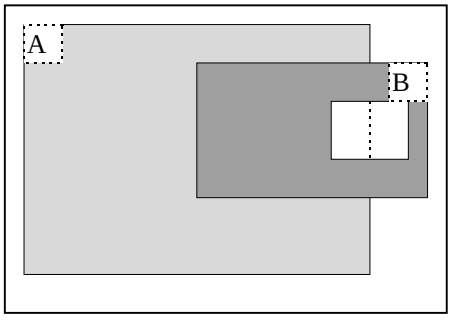
In Figure B, the entire image is to be scaled so that the *y* axis fits the viewport, but the aspect ratio is to be maintained. In this situation, the camera rectangle is set to have dimensions equal to the *y* dimension of viewport, and to be centered in that viewport.

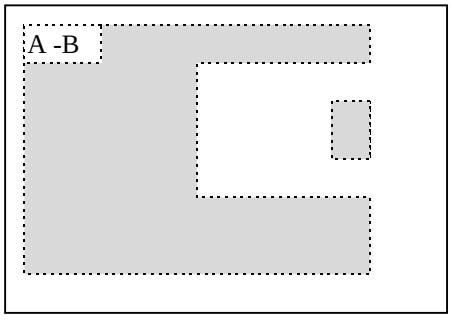
The camera rectangle can also be used to scale or stretch images to allow for non-square pixels (by using non-square camera rectangles), or for multipass anti-aliasing by offsetting the camera rectangle by sub-pixel amounts.

When a device's default viewport is used, the associated camera rectangle is set to be the largest centered square which fits in the device.

2.1.6 Defining Holes in Viewports

In some situations it is useful to be able to not render portions of a view—for example an application that has a viewport which only needs to updated every *n* frames. PowerSGL allows a viewport to be subtracted from another viewport provided both belong to the same device. This means that if we have two viewports, V_a and V_b, and we subtract V_b from V_a then any pixels which are in both viewports are removed from V_a. This is illustrated in the following diagrams—V_a has a hole cut out of it.





2.2 Device and Viewport 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, destroying all viewports attached to it in the process. 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.

2.2.4 Create Viewport

Creates a viewport on a rendering device. You specify the size and position of the viewport, and it returns a name for the viewport. The routine attempts to comply with the requested parameters, but because of possible hardware limitations, the actual values chosen may differ.

Function Header

```
int sgl_create_viewport( int parent_device,
                         int left,
                         int top,
                         int right,
                         int bottom,
                        float cam_rect_left,
                        float cam_rect_top,
                        float cam_rect_right,
                        float cam_rect_bottom);
```

Parameters

I	`parent_device`	The device on which to create the viewport.
I	`left, top`	The coordinate of the top left hand pixel. The values passed are clipped to the parent device.
I	`right, bottom`	The coordinates of the bottom right hand pixel. If (*left>right*) or (*top>bottom*) then the viewport contains no pixels.
I	`cam_rect_left,`	The coordinates of the top left hand corner of the camera rectangle.
	`cam_rect_top`	These values are not clipped to the device coordinates.
I	`cam_rect_right, cam_rect_bottom`	The coordinates of the bottom right hand corner of the camera rectangle. These values are not clipped to the device coordinates. However if the camera rectangle is specified to be unreasonably small or large then the size is clipped to a manageable dimension.
	`returned value`	This is either a name for the viewport, or an error value.

Errors

`sgl_err_bad_name`	There is no device of the given name.
`sgl_err_no_mem`	There was not enough memory to create the viewport.
`sgl_err_no_name`	There were no available names so the viewport information has been ignored.

2.2.5 Delete Viewport

Deletes a viewport. The name for the viewport becomes invalid.

Function Header

```
void sgl_delete_viewport( int viewport);
```

Parameters

I	`viewport`	The viewport to delete.

Errors

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

2.2.6 Get Viewport

Returns the dimensions for a viewport as defined by the `sgl_create_viewport` or `sgl_set_viewport` routines. It ignores any holes created by the `sgl_subtract_viewport` routine (see page 2-12).

Function Header

```
int sgl_get_viewport( int viewport,
                      int *left,
                      int *top,
                      int *right,
                      int *bottom
                    float *cam_rect_left,
                    float *cam_rect_top,
                    float *cam_rect_right,
                    float *cam_rect_bottom);
```

Parameters

I	`viewport`	The name of the viewport. If the viewport name is invalid, then the returned parameters are undefined.
O	`left, top`	As for Create Viewport.
O	`right, bottom`	As for Create Viewport.
O	`cam_rect_left, cam_rect_top`	As for Create Viewport.
O	`cam_rect_right, cam_rect_bottom`	As for Create Viewport.
	`returned value`	Error status.

Errors

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

2.2.7 Set Viewport

Modifies the dimensions of a viewport. The routine attempts to comply with the requested parameters, but due to possible hardware limitations, the actual values chosen may differ.

The routine also removes any holes in the viewport.

Function Header

```
void sgl_set_viewport( int viewport,
                       int left,
                       int top,
                       int right,
                       int bottom
                     float cam_rect_left,
                     float cam_rect_top,
                     float cam_rect_right,
                     float cam_rect_bottom);
```

Parameters

I	`viewport`	The name of the viewport. If the viewport name is invalid, then the routine is ignored.
I	`left, top`	As for Create Viewport.
I	`right, bottom`	As for Create Viewport.
I	`cam_rect_left, cam_rect_top`	As for Create Viewport.
I	`cam_rect_right, cam_rect_bottom`	As for Create Viewport.

Errors

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

2.2.8 Subtract Viewport

Creates holes in viewports so that certain pixels are not rendered. It takes two viewports, belonging to the same device as parameters, and removes all shared pixels from the first. This routine may be called many times to remove several regions from a viewport.

Function Header

```
void sgl_subtract_viewport( int viewport1,
                            int viewport2);
```

Parameters

I	`viewport1, viewport2`	The names of the viewports. If either viewport name is invalid, or they do not both belong to the same device, then the function sets an error, and exits.

Errors

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

3 LISTS

3.1 Introduction

PowerSGL is based on display lists, which are used for:

- storing the objects, materials, transformations etc. to be rendered
- creating hierarchies to allow the construction of complex animations
- instancing (i.e., efficient re-use or duplication) of objects
- grouping related elements together so that they can be replaced at the same time

During rendering, a specified list is traversed and the items in it are processed. The order in which items are encountered in the list is important, as is its hierarchical structure. A material definition, for example, can only affect the objects which come after it in the display list. Note that the properties of these objects, such as color, are determined dynamically at render time, not statically[1].

For the purposes of editing, PowerSGL has the concept of the current list. At all times, one list in the system is deemed to be the current list and the majority of the creation routines result in an additional item being appended to this list.

3.1.1 Default and Null List

At initialization, the library creates an empty list, which can be referred to by the constant name `SGL_DEFAULT_LIST`. You can add to and clear this list, but you can not delete it. This list is set to be the initial current list.

For the purposes of instance parameterization (described on page 3-3), an empty list is also defined, which can be referenced by the name `SGL_NULL_LIST`. This list is always empty, and cannot be edited.

3.2 Separate vs. Child Lists

When a new list is created, it can either be created as a child of the current list or as a separate entity. During rendering, a child list inherits all the currently active properties, such as position and material definitions, from its parent list.

A separate list, on the other hand, is one which has no parent. These can be used either as alternative display lists which can be rendered, or for convenient instancing from other lists. Instancing is explained on page 3-3.

Lists can be separated from parent lists, and reconnected to other lists, with the exception that the default list can never have a parent.

[1] Foley, van Dam, Feiner and Hughes, “Computer Graphics, Principles and Practice”, Second Edition, Addison and Wesley, 1990, pages 315 to 318

3.3 Preserving State Variables

Associated with each list is a `preserve_state` flag. When a child list (or an instance reference) is encountered during the rendering list traversal, this flag indicates whether the current state of certain system variables should be saved. When the end of the child list is reached, those variables which were saved are restored to their former values before the remainder of its parent list is processed. This state information consists of:

- the transformation (positioning and orientation)
- the material definition
- the light definitions
- the quality settings
- the instance parameterizations
- the point definitions

One potential use of the save feature is the creation of hierarchies. The fact that transformations can be saved allows simple definition of hierarchical motion. For example, an animation consisting of satellites orbiting planets, which in turn orbit a star, can be done by nesting lists which preserve the state variables. Each planet model is created along with orbiting information in a child list of the star, while each satellite is created as a child list of its planet. Because of the nested lists, the satellites inherit their planet's motion, while the saving of state information keeps the definition of the satellites' motion from interfering with that of the planet.

The effects of certain lights, materials etc., can similarly be localized by encasing them in state saving lists. The state save flag's value is set when the list is created.

Alternatively, there are times when the changes made in a list must carry on into the parent. A list might be used to conveniently bundle several material and quality definitions together for the subsequent objects in the parent list. Not saving the state allows the material and quality settings etc. to propagate back to the parent.

3.4 Ignoring Lists

To make repeated addition and deletion of complex models easier, each list has an ignore flag associated with it. Setting this flag causes the list, and all its contents, to be skipped over during display list traversal. By default, lists are not ignored.

3.5 Instancing

In complex scenes, you frequently use a material or an object or a transformation many times throughout a display list. Usually this means that the actual data is copied, consuming memory. For example, a car model might be generated, which is used multiple times in a road scene. Alternatively the same material might need to be applied to a number of objects throughout a display list. Instances address this re-use efficiently[1].

If a common object is created in its own list, that list can then be referenced, or instanced, many times from within other lists, as if it were duplicated in those lists. The data is not actually copied, so that changing the instance definition effectively changes all the display lists that reference it.

Instance definitions may also reference other instances. For example, the instanced car may have four references to a single wheel, which in turn may reference a wheel nut four times.

An instance inherits the state information that is defined prior to the reference. Again, if the car list definition contains no material definition, then different instance references could be used to build cars with different colors. The setting of the state save flag in the referenced list then determines if the changes to state values are passed back to the parent.

In general, instances should be created in their own separate lists. PowerSGL, however, doesn't prevent a definition being attached to another list. Note that deleting or clearing a list deletes all sub-lists contained within it, so it may be possible to accidentally delete the instance definition if it is not separate. Deleting an instance reference, however, does not delete the instance definition.

[1] Foley, van Dam, Feiner and Hughes, "Computer Graphics, Principles and Practice", Second Edition, Addison and Wesley, 1990, pages 291 to 325

3.5.1 Instance Substitutions and Parameterization

To extend the flexibility of instancing, PowerSGL provides a simplistic form of instance parameterization. If an instance references other instances (which can be used to define objects, materials, transformations etc.), then a set of substitutions can be given for those instances referenced. For example, assume there is an instance definition, R, which contains three references to other instances, A, B and X:

```
Define R
  reference A
  reference B
  reference X
```

If a display list, S, references R twice, and in the second reference specifies that A should be replaced by X, and B by Y:

```
List S
  reference R
  reference R (A=>X, B=>Y)
```

then S is effectively the same as:

```
List S...
  reference A
  reference B
  reference X
  ~~reference A~~ reference X
  ~~reference B~~ reference Y
  reference X
```

The substitutions also extend into the substitutions specified for other instance references. For example if S is now treated as an instance definition, which is referenced with the parameterization (replace X with Z) then that instance reference would effectively be:

```
Reference S (X=>Z) ...
  reference A
  reference B
  ~~reference X~~ reference Z
  ~~reference X~~ reference Z
  reference Y
  ~~reference X~~ reference Z
```

The order in which these substitutions occur is left to right. If the parameterization (A=>B, B=>C) is specified, then this is the same as (A=>C, B=>C). When substitutions are nested (as in the previous example) the lists of substitutions are effectively concatenated, with the ones occurring in the parent being appended to the list in the child. From the previous example, (A=>X, B=>Y) and (X=>Z) becomes (A=>Z, B=>Y, X=>Z).

3.6 List and Instance Routines

3.6.1 Create List

Creates a list. You can specify if the list is to be a child of the current list or separate, and whether it preserves the current state. You also specify if the list is to be named or anonymous.

Upon successful creation, the new list becomes the current list i.e., all further additions are made to it.

Function Header

```
int sgl_create_list( sgl_bool generate_name,
                     sgl_bool preserve_state,
                     sgl_bool separate_list);
```

Parameters

I `generate_name`	If non-zero, the routine returns a name for the list created (or an error if the routine fails). However, if the created list is separate, or if it is a child list which doesn't preserve the state information, then it also returns a name (or error). This is because, in these two cases, it is pointless to have an anonymous list. If none of the above applies, then the returned value is undefined.
I `preserve_state`	If non-zero, then the render list traversal saves (i.e., stacks) the current state as it enters the list, and restores it as it exits it. Note that this applies whether the list is a child, or if it is a separate list being accessed as an instance.
I `separate_list`	If non-zero, the list is created as a separate entity, i.e., not attached to a parent, otherwise it is appended to the current list.
`returned value`	If a name is requested (either directly or indirectly) the name, or an error value is returned. The value is otherwise undefined.

Errors

`sgl_err_no_mem`	There was not enough memory to create the list.
`sgl_err_no_name`	There were no available names so the list information has been ignored.

3.6.2 Return To Parent List

Changes the list editing point to be the parent of the current list. If the list has no parent, i.e., it is a separate list, no action is taken and it remains the current list. If you wish to change to a different list when the current list is a separate list, use `sgl_modify_list`.

Function Header

```
void sgl_to_parent( );
```

Errors

None

3.6.3 Modify List

Sets a named list to be the current list. Subsequent creations of objects etc., are added to the end of this list.

A flag can be supplied which instructs PowerSGL to first clear the list. This removes its contents, recursively deleting any lists contained within it. Note that instances (i.e., references) are deleted but the actual definitions of those lists are not affected, provided they are not actually contained within the cleared list. For this reason, we recommend that instances be kept as separate lists. All names for items which were deleted when the list is cleared, e.g. named cameras and transforms, become invalid.

Function Header

```
void sgl_modify_list( int list_name,
                      sgl_bool clear_list);
```

Parameters

I	`list_name`	The name of the list to be modified (i.e., to become the current list).
I	`clear_list`	If non-zero, the contents of the list are removed.

Errors

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

3.6.4 Delete List

Given the name of a list, deletes that list, recursively deleting all its contents. If the current list was either the named list, or one of the deleted list's children, then:

- If the deleted list had a parent, that parent becomes the current list.
- If it had no parent, the current list is set to be the default list, i.e., `SGL_DEFAULT_LIST`.

All names for items within the deleted list, e.g. cameras and transforms, as well as the deleted list's name, become invalid.

Function Header

```
void sgl_delete_list( int list_name);
```

Parameters

I	`list_name`	The name of the list to be deleted.

Errors

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

3.6.5 Detach List

Given the name of a list, this removes it from its parent (if any) and leaves it as a separate list.

Function Header

```
void sgl_detach_list( int list_name);
```

Parameters

I	`list_name`	The name of the list to be detached from its parent.

Errors

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

3.6.6 Attach List

Given the name of a list, removes it from its current parent (if any) and adds it to the end of the current list. The routine prevents cycles occuring in the display list by checking that the new parent is not a descendent of the list to be added. If this is the case the routine flags an error, and returns without modifying either display list.

The routine merely re-arranges the tree structure of the display list. It makes no other modifications to the lists such as modifying transformation matrices.

The default list can not be attached to other lists. An error will be reported, but no other action is taken.

Function Header

```
void sgl_attach_list( int list_name);
```

Parameters

I	`list_name`	The name of the list to be detached from its parent and attached to the current list.

Errors

`sgl_err_bad_name`	There is no list of the given name.
`sgl_err_cyclic_reference`	The current list is a child of the list to be attached. The command is ignored.

3.6.7 Ignore List

Given the name of a list, sets the ignore flag of that list. Ignored lists are skipped during display list traversal.

Function Header

```
void sgl_set_ignore_list( int list_name,
                          sgl_bool ignore_list);
```

Parameters

I	`list_name`	The name of the list.
I	`ignore_list`	If non-zero, the list is ignored, otherwise it is processed.

Errors

`sgl_err_bad_name`	There is no list of the given name.
`sgl_err_cyclic_reference`	The current list is a child of the list to be attached. The command is ignored.

3.6.8 Use Instance

Adds an instance reference to the display list. If the referenced instance list is later deleted, then an error may be flagged during display list traversal. If any instance substitutions are in effect, then they may apply to this instance reference, and in turn, any instances referenced inside that list.

The list instanced must be a user-defined list, not one of the predefined lists such as `SGL_NULL_LIST`.

Function Header

```
void sgl_use_instance( int list_to_instance);
```

Parameters

I	`list_to_instance`	The name of the list to be referenced

Errors

`sgl_err_bad_name`	There is no list of the given name. The routine is ignored.
`sgl_err_no_mem`	The display list has become too large for the system so the instance information has been ignored.

3.6.9 Instance Substitutions

Defines instance substitutions that can apply to instances referenced within the scope of the instance substitutions.

Note that a referenced instance may be ignored by specifying that it be replaced with the null list, SGL_NULL_LIST. You may not replace the predefined lists.

Function Header

```
void sgl_instance_substitutions( int num_subs,
                                 int *instance_params);
```

Parameters

I num_subs	The number of instance substitution pairs given in the parameter instance_parameters. There is a limit to the number of parameter substitutions, SGL_MAX_INSTANCE_PARAMS, that can be passed. PowerSGL ignores any more than this number. There is also a limit to the maximum that can be active and inherited from parent instance references, SGL_MAX_INSTANCE_SUBS.
I instance_params	This is a pointer to an array of pairs of names of lists. The number of elements in this array should be (at least) 2 * num_subs. For example, it could either be a one dimensional array, or a 2 dimensional array (with the second dimension being 2). The first integer of each pair represents the original instance reference, and the second its replacement.

Errors

sgl_err_bad_name	At least one of the instance parameters contains an invalid list name. The bad parameters are ignored, but the remainder is added.
sgl_err_no_mem	The display list has become too large for the system so the instance information has been ignored.

4 TRANSFORMATIONS

4.1 Introduction

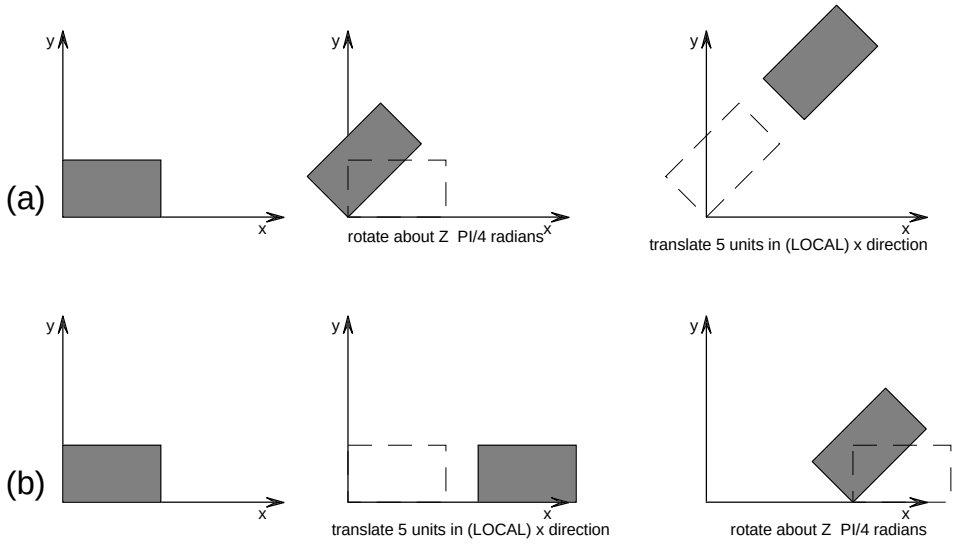
Transformations (which are effectively 4x4 homogenous matrices[1]) are used to position and orient objects, cameras, lights, and materials in scenes. They are constructed by concatenating a number of simple translation, rotation, and scaling operations. Transformations may be named and edited, allowing animation to be easily performed.

The coordinate system used in PowerSGL is a left-handed system. This may seem unusual, but it is more natural when dealing with computer coordinates to have increasing *z* going into the distance. If you need to render an object which has been specified in a right-handed system, negating the *z* coordinates (or using a transformation which scales the *z* by -1) converts it to a right-handed system.

Each transformation encountered in a display list redefines the local coordinate system. At the top of the display list, the local origin is the global origin, and the axes are aligned with the global axes. A translation, for example, would move the local coordinate systems origin, and a rotation would rotate the local coordinate axes. Further transformations are performed relative to the last local coordinate system.

Transformations are part of the state information which can be preserved when a child list is entered during display list traversal. This permits positioning operations to be localized, so that they do not affect the rest of the scene.

As matrix operations aren't commutative, the order in which transformation operations are performed is important. This is illustrated in the following diagrams. The axes show the global, or world coordinate system, the local ones are aligned with the block. The observer is looking down the *z* axis (left-handed coordinate system).



[1] Foley, van Dam, Feiner and Hughes, "Computer Graphics, Principles and Practice", Second Edition, Addison and Wesley, 1990, pages 213 to 217

In example (a), to get the third image, a rotation is added to the display list, followed by a translation, followed by the block definition. In (b), the translation is performed first, followed by the rotation followed by the object. In effect, if we represent A points by column vectors, then if transformation $\overline{A}$ occurs before transformation $\overline{B}$ in the display list, then point *p*, is transformed by $\overline{A}\overline{B}p = p'$.

4.2 Transformation Routines

4.2.1 Create Transformation

Creates an initial null (i.e., identity) transformation which may be modified by translations, rotations, and scales. The transformation may be named or anonymous. The transformation becomes the current transformation—it remains the current transformation until a non-transformation operation is performed, or the current transformation is explicitly changed by the modify routine.

Note: As a shortcut, whenever a transformation operation (i.e., scale, translation, or rotation) is performed, and there is no current transformation, then an anonymous transformation is automatically created in the current list.

Function Header

```
int sgl_create_transform( sgl_bool generate_name);
```

Parameters

I `generate_name`	If non-zero, the routine returns a name for the transform, otherwise an anonymous transform is created.
`returned value`	A meaningful value is returned only if a name is requested, and then is either the name of the created transform or an error value.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the transformation information has been ignored.
`sgl_err_no_name`	The transformation has not been created as there were no available names.

4.2.2 Modify Transformation

Sets the current transformation to an existing named transformation. Subsequent contiguous transformation operations (i.e., scaling, translation, and rotation) are then applied to the current transformation. As soon as a non transformation operation is performed, it is no longer the current transformation and ceases to be affected.

The caller can optionally reset the existing transformation back to an identity transformation, so that it can be rebuilt from scratch. Since transformation operations are not generally commutative, rebuilding is probably the easiest way to generate animation.

Function Header

```
void sgl_modify_transform( int name,
                      sgl_bool clear_transform);
```

Parameters

I `name`	The name of the transform to be modified.
I `clear_transform`	If non-zero, the transform is reset to the identity matrix.

Errors

`sgl_err_bad_name`	There is no transform of the given name in the display list. Applying further transform operations in the error case causes the transforms to be appended to the group where the last additions were made.

4.2.3 Translation

Adds a further translation to the current transformation. If there is no current transformation, then an anonymous one is created. This moves the local origin by the amount specified, relative to the local coordinate system.

Function Header

```
void sgl_translate( float x,
                    float y,
                    float z);
```

Parameters

I `x,y,z`	The amount to translate, in *x, y,* and *z*.

Errors

`sgl_err_no_mem`	There was not enough memory to create the translation.

4.2.4 Scale

Scales the current transformation independently in *x, y,* and *z*. If there is no current transformation, then an anonymous one is created. PowerSGL does not allow scaling by 0.0 (or very small values up to 10^{-6}). In such cases, the scaling is chosen to be the smallest legal scaling value.

Using a negative scaling value performs a mirror flip on the items affected by the transformation.

Function Header

```
void sgl_scale( float x,
                float y,
                float z);
```

Parameters

I `x,y,z`	The amount to scale, in *x, y,* and *z*.

Errors

`sgl_err_no_mem`	There was not enough memory to create the scale operation.

4.2.5 Rotate

This routine rotates the current transformation about an arbitrary axis passing through the local origin. If there is no current transformation, then an anonymous one is created. The axis is specified by a vector, and the angle is given in radians. The rotation is performed in a counterclockwise direction when looking in the direction of the vector. If the vector has zero length, no rotation is performed.

To perform a rotation about an arbitrary point, say (*x,y,z*), the local origin should first be moved to that point (by translating by (*x,y,z*)), the rotation should be performed, and then the origin moved back—a translation of (*-x,-y,-z*).

For convenience, the following axes are predefined in PowerSGL, which giving the usual Euler angle rotations:

```
const sgl_vector  sgl_x_axis = {1.0, 0.0, 0.0};
const sgl_vector  sgl_y_axis = {0.0, 1.0, 0.0};
const sgl_vector  sgl_z_axis = {0.0, 0.0, 1.0};
```

Function Header

```
void sgl_rotate( sgl_vector axis,
                      float angle);
```

Parameters

I	`axis`	The axis of rotation (passing through the origin).
I	`angle`	The angle of rotation expressed in radians.

Errors

`sgl_err_no_mem`	There was not enough memory to create the rotation.

5 OBJECTS AND PRIMITIVES

5.1 Introduction

The PowerSGL system supports two different types of object primitives: the polygonal mesh and the faster, but less general, convex polyhedron. Objects constructed with convex polyhedra also permit the library to automatically cast shadows from those objects, and also perform collision detection. PowerSGL also supplies a number of routines for the creation of common predefined primitives, such as boxes and prisms.

Primitives can be smooth shaded and/or textured. Smooth shading can make a faceted primitive appear to be curved, and texturing can increase the apparent detail on the surface.

5.1.1 Depth Priority of Faces

For both types of primitive, there is a deterministic approach to depth priority for faces which are deemed to be of equal depth at a pixel. This is important for two reasons. Firstly, to prevent a pixel from randomly flashing because two different objects intersect at that point and the system alternates between them Secondly, to allow you to exploit the order in scenes. Consider a painted line on a road. If both line and road are modeled as primitives, you want to be sure that the line is displayed in front of the road. You can set the positions of the line and road to be identical, knowing that the correct one will be displayed, by placing them in the correct order in the display list.

In PowerSGL, if two (or more) faces have the same *z* depth at a pixel, then the one which was encountered later in the display list, during rendering traversal, is displayed. Therefore, the line should be put in the list after the road.

The two types of primitive are described in the next two chapters of this manual.

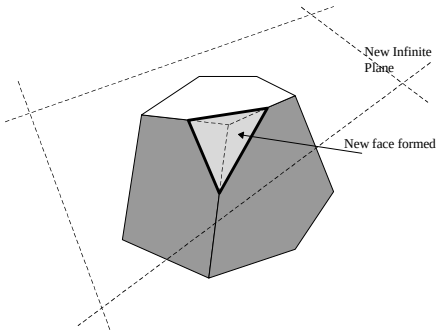
6 CONVEX POLYHEDRA

6.1 Introduction

PowerSGL's convex primitive is a set of faces which bound a convex[1] volume. Any convex polyhedron can be uniquely specified by describing just the plane equations of its faces—the individual polygons themselves do not have to be described. The edges of each face are defined by the intersection of the face's plane and its neighbors. PowerSGL uses this coherency in rendering.

Each convex primitive is specified by the intersection of a set of planar half-spaces. A planar half-space is an infinite plane with one side classed as 'inside' and the other 'outside'. (This is analogous to determining the back facing polygons in meshes.). A point is then inside the convex *if* it is inside all the planar half-spaces.

Another way of looking at this is to imagine a large block of stone which is progressively having pieces sliced off. This is illustrated in the following diagram:



Here, a new infinite plane is slicing off a corner from the convex object formed by earlier planes.

In fact a convex polyhedron does not have to be bounded, although such primitives will be be slighlty slower to render in comparison to non-infinite objects. A ground plane in a flight simulator-type application can therefore be just that—a single plane. The primitive it represents is infinitely large.

PowerSGL imposes a restriction on the number of surfaces that can be used to define a convex polyhedral object. This limitation is given by the constant `SGL_MAX_PLANES`.

6.2 Face Visibility

In most situations, it is expected that all faces of a convex polyhedron are visible. However there are occasions where it is useful to have some invisible faces. An example of this is an open cup, modeled as a cylinder with the top surface invisible, so that the interior can be seen. Such objects are slightly more expensive to render, as the faces which would normally be considered back-facing are now visible and

[1] The definition of convex, with regard to PowerSGL, is: *Given any two points inside a primitive, then all points on the line joining those must be inside the object.*

thus must be textured and shaded. However, PowerSGL renders these much more quickly than an object modeled from individual polygons.

6.3 Shading and Texturing Convex Polyhedra

Smooth shading is a means of simulating curved surfaces by varying the shading on faceted objects. To do this, you supply PowerSGL with data to allow it to linearly interpolate the surface normal across a face. This changing normal is then used in the shading equations.

The surface normal information is defined by providing 3 non-linear points on the surface, with three normals for the curved surface.

Smooth shading can be applied to both the diffuse color and optionally to specular highlights (depending on hardware support).

If the object is not to be smooth shaded, then it is better not to supply normal information (by passing `NULL` pointers) as this will save memory.

There are two methods in which textures can be applied to objects. You can either employ PowerSGL's automatic texture wrapping, or explicitly specify a texture mapping function for each face. In the latter method, you supply 3 sets of (*u*,*v*) texture coordinates with the points to which these are mapped. These points would be the same as for the smooth shading.

For the automatic texture wrapping, the *u* and *v* coordinates are computed automatically. Three points are chosen, to which the texture is pinned. You choose these points, which are the same as the smooth shading points, if smooth shading is enabled.

If the object is not to be textured then, as with smooth shading, it is better not to supply (*u*,*v*) coordinates.

6.4 Editing Convex Polyhedra

A named convex polyhedron may be edited, although the facilities supplied by PowerSGL are quite basic. The positional information, normal, and *uv* values of existing planes in an object may be modified, new planes may be added, and existing ones deleted. However, due to pre-processing requirements, it is not possible to extract the original parameters supplied when creating a plane.

Planes within an object are uniquely identified by the order in which they were put into the object. If there are *n* planes in an object, the first is identified by index *0*, and the last by *n-1*.

6.5 Definition of Local Materials within Convex Polyhedra

PowerSGL allows new material definitions to be made within a convex polyhedra, to the extent of a new material every plane. However, the more material changes that are made in an object, the more expensive the rendering becomes.

To group a material with the current convex polyhedron, a flag specifying that the new material is to be local is set when the material is created. This is specified in the section on materials.

If materials are grouped with a primitive, then when the primitive is cleared (using modify), they are deleted.

The material definitions made within a primitive only affect that primitive. After processing the primitive, during rendering, the material returns to the state prior to the primitive definition.

6.6 Convex Polyhedra Routines

6.6.1 Create Convex Polyhedron

Creates an empty convex polyhedron. The primitive can be specified to be either named or anonymous. All subsequent calls to `add_plane` routines result in planes being added to the primitive. This object definition phase is terminated whenever a non `add_plane` (or any plane modification) routine is called with the exception of a create material definition that specifies a local material. In effect, PowerSGL maintains a temporary current convex polyhedron which exists until the end of the primitive creation.

Function Header

```
int sgl_create_convex( sgl_bool generate_name);
```

Parameters

I `generate_name`	If non-zero, the routine returns a name for the object.
`returned value`	If a name has been requested then this either the name or an error value. The returned value is undefined if no name is requested.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the convex polyhedron has been ignored.
`sgl_err_no_name`	The convex polyhedron has not been created as there were no available names.

6.6.2 Modify Convex Polyhedron

Allows editing of a named convex polyhedron. In effect, the named primitive becomes the current convex polyhedron so that modifications, additions, and deletions of planes apply to this primitive until a non plane routine (or when creating a local material, a non material routine) is invoked.

If the `clear` flag is set, then all the contents of the polyhedron are removed (including any local material definitions which apply only to planes within the primitive).

Function Header

```
void sgl_modify_convex( int name,
                        sgl_bool clear);
```

Parameters

I `name`	The name of the convex primitive to be edited.
I `clear`	If non-zero, all the planes defining the convex polyhedron are removed.

Errors

`sgl_err_bad_name`	There is no convex polyhedron of the given name in the display list.

6.6.3 Add Plane

These two routines are used to define the planes that make up a convex polyhedron. The first is a simple version that is supplied with just a surface normal and a point on the surface, and does not allow specification of textures or smooth shading.

The second is defined by three points and allows the optional specification of texture coordinates and/or smooth shading surface normals.

These routines can be used both when creating a convex primitive, or when modifying an existing primitive during editing.

Note: As a shortcut, if a polyhedral primitive is not currently being defined or edited, (i.e., there is no current convex polyhedron) then on receipt of an `add_plane` command, a new anonymous primitive is created automatically.

Function Header 1

```
void sgl_add_simple_plane( sgl_vector surface_point,
                           sgl_vector normal,
                             sgl_bool invisible);
```

Parameters

I `surface_point`	A point on the surface. Preferably this should be near the center of the face, however a corner point is sufficient. Apart from defining the surface, this point is also used to calculate directional lighting when flat shading.
I `normal`	The surface normal. It is assumed that the normal points outward from the primitive.
I `invisible`	If non-zero, the face acts as if it were completely transparent. In most cases this flag is set to 0.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the plane information has been ignored.
`sgl_err_too_many_planes`	The plane limit (`SGL_MAX_PLANES`) for the current convex polyhedron has been reached.

Function Header 2

```
void sgl_add_plane( sgl_vector surface_point,
                    sgl_vector point2,
                    sgl_vector point3,
                      sgl_bool invisible,
                    sgl_vector normal1,
                    sgl_vector normal2,
                    sgl_vector normal3
                    sgl_2d_vec uv1,
                    sgl_2d_vec uv2,
                    sgl_2d_vec uv3);
```

Parameters

I `surface_point`	A point on the surface. Preferably this should be near the center of the face, however a corner point is sufficient. This point is also used to calculate directional lighting when flat shading.
I `point2, point3`	The other two points on the face which are needed to define a plane. It is assumed that the 3 points are given in clockwise order when viewing the face from the outside of the object. These 3 points must be linearly independent.
I `invisible`	If non-zero, the face acts as if it were completely transparent. In most cases this flag is set to 0.
I `normal1,2,3`	The curved surface normals of the object at `surface_point`, `point2` and `point3` respectively. These are used for smooth shading. If no smooth shading is required then all three parameters should be `NULL`. Supplying `NULL` pointers saves memory.
I `uv1, uv2, uv3`	The texture *uv* coordinates for `surface_point`, `point2`, and `point3` respectively. The *uv* parameters are used to explicitly map a texture to a surface, which overrides any of the automatic wrapping methods. If no texturing is required then all three parameters should be `NULL`. Supplying `NULL` pointers saves memory.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so some or all of the plane information has been ignored.
`sgl_err_too_many_planes`	The plane limit (`SGL_MAX_PLANES`) for the current convex polyhedron has been reached.
`sgl_err_colinear_plane_points`	The surface points were not linearly independent.

6.6.4 Set Plane

Given an index for a plane in the current convex polyhedron, these two routines allow the planes to be redefined. Note that using the `set_simple_plane` routine destroys any exiting texturing or smooth shading data for that face.

Function Header 1

```
void sgl_set_simple_plane( int plane_index,
                   sgl_vector surface_point,
                   sgl_vector normal,
                     sgl_bool invisible);
```

Parameters

I	`plane_index`	The index for the plane to modify.
I	`surface_point`	As for Add Plane.
I	`normal`	As for Add Plane.
I	`invisible`	As for Add Plane.

Errors

`sgl_err_no_convex`	There is no current convex polyhedron.
`sgl_err_bad_index`	There is no plane of the given index.

Function Header 2

```
void sgl_set_plane( int plane_index,
          sgl_vector surface_point,
          sgl_vector point2,
          sgl_vector point3,
            sgl_bool invisible,
            sgl_bool supply_normals,
          sgl_vector normal1,
          sgl_vector normal2,
          sgl_vector normal3,
            sgl_bool supply_uv,
          sgl_2d_vec uv1,
          sgl_2d_vec uv2,
          sgl_2d_vec uv3);
```

Parameters

I	`plane_index`	The index for the plane to modify.
I	`surface_point`	As for Add Plane.
I	`point2, point3`	As for Add Plane.
I	`invisible`	As for Add Plane.
I	`supply_normals`	As for Add Plane.
I	`normal1,2,3`	As for Add Plane.
I	`supply_uv`	As for Add Plane.
I	`uv1, uv2, uv3`	As for Add Plane.

Errors

`sgl_err_no_convex`	There is no current convex polyhedron.
`sgl_err_bad_index`	There is no plane of the given index.
`sgl_err_no_mem`	The display list has become too large for the system, so some of the plane information has been ignored.
`sgl_err_colinear_plane_points`	The surface points were not linearly independent.

6.6.5 Delete Plane

Deletes a plane from the current convex polyhedron. If the plane being deleted is one of the middle planes in the list, then the remaining planes are renumbered to remove the gap.

Function Header

```
void sgl_delete_plane( int plane_index);
```

Parameters

I plane_index	The index for the plane to delete.

Errors

sgl_err_no_convex	There is no current convex polyhedron.
sgl_err_bad_index	There is no plane of the given index.
sgl_err_no_mem	There was not enough memory to generate a replacement convex polyhedron.

6.6.6 Plane Count

Returns the number of planes in the current convex polyhedron.

Function Header

```
int sgl_plane_count( );
```

Parameters

returned value	The number of planes in the current object, or an error (since error values are negative).

Errors

sgl_err_no_convex	There is no current convex polyhedron.

6.7 Light and Shadow Volumes

These are a class of convex objects, which, while defined as normal convex objects, do not appear directly in the rendered scene. Instead they produce areas of light of shadow by their intersection with other objects in the scene, i.e. part of an object inside a shadow volume will appear to be in shadow. Using these volumes gives performance improvements in scenes where shadows are required, but the shadow shape is constant with regard to the casting light and the shadowing object. This effectively precalculates the shadows for the duration of the scene, and avoids on-the-fly calculation of shadows at render time.

6.7.1 Create Light/Shadow Volume

Creates the definition of a light or shadow volume. The object can be named or anonymous.

Function Header

```
int sgl_create_light_volume( sgl_bool generate_name,
                             sgl_bool light_or_shadow,
                                  int light_name);
```

Parameters

I	`generate_name`	If TRUE, the routine returns a name for the object.
I	`light_or_shadow`	If TRUE, a light volume is created, otherwise a shadow volume is created.
I	`light_name`	The name of the light which acts as if casting the light or shadow.
	`return value`	If a name has been requested then this returns either the name or an error value. The returned value is undefined if no name is requested.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the convex polyhedron has been ignored.
`sgl_err_no_name`	The convex polyhedron has not been created as there were no available names.
`sgl_err_bad_name`	There is no light of the given name in the display list.

6.8 Hidden Convex Objects

All planes added to these objects are regarded as invisible. The purpose of defining such objects is so that, when casting shadows, a simple hidden convex can be co-located with a more complex convex object, and used to cast shadows in its place. The simpler shadows cast by the hidden object will be computationally less expensive to calculate, and faster to render.

6.8.1 Create Hidden Convex

This function creates a hidden convex object. The object can be named or anonymous.

Function Header

```
int sgl_create_hidden_convex( sgl_bool generate_name);
```

Parameters

I	`generate_name`	If TRUE, the routine returns a name for the object.
	`return value`	If a name has been requested then this either the name or an error value. The returned value is undefined if no name is requested.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the hidden convex and has been ignored.
`sgl_err_no_name`	The hidden convex has not been created as there were no available names.

6.9 Shadow Limit Planes

Shadow limit planes are new nodes which contain the definition of a single plane, thus limiting shadows cast by objects following this node in the display list. They allow you to place a plane just below a ground plane, thus turning automatically-generated shadows from semi-infinite volumes, to volumes which are bounded at ground level. Other uses could be to bound shadows within a room or other enclosed space.

Up to 16 shadow limiting planes can be used, any others are ignored. The 16 are determined by the first 16 encountered when traversing the display list. The limiting planes are treated like states, so any planes in a sub-list will not affect objects in other sub-lists.

6.9.1 Add Shadow Limit Plane

This function adds a new node to the current list.

Function Header

```
void sgl_add_shadow_limit_plane(sgl_vector surface_point,
                sgl_vector normal);
```

Parameters

I	`surface_point`	A point on the surface.
I	`normal`	A normal pointing outward on the plane. Shadows will be truncated on the outward side of the plane.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the shadow plane has been ignored.

7 POLYGON MESHES

7.1 Introduction

The polygon mesh, in various forms, is the most common method of modeling objects. A PowerSGL polygon mesh consists of a list of 3D vertices and a list of convex polygons. Each polygon, which may be either a triangle or quadrilateral, is built into the vertex table from a list of indices. Each vertex may optionally have normal and/or *uv* parameters associated with it to support smooth shading and/or arbitrary texturing.

PowerSGL maintains the concept of a current mesh. When a mesh is created, all mesh operations apply to that mesh until a non mesh operation is performed (with the exception of local material definitions) or the current mesh is explicitly changed by the `modify_mesh` function.

There are a set of limits on the data structure sizes for meshes. These are:

`SGL_MAX_VERTS` - the maximum number of vertices that can be defined for a mesh.

`SGL_MAX_FACES` - the maximum number of faces that can be in a mesh.

7.2 Back Face Removal

When rendering a solid object, the polygons on the back faces are invisible, and so should be removed from the rendering pipeline early on to reduce the rendering overhead. This is known as back face removal. To make this possible, polygons are generally defined in a standard order; for example, the vertices are listed in clockwise order when looking at the front of the polygon.

PowerSGL supplies three types of back face removal options. These are

- No back face removal. This is the default.
- Remove counterclockwise faces.
- Remove clockwise faces.

The first is supplied for the cases where an object is open, for example an open topped cylinder, where removing the back faces would be incorrect. The other two are supplied because there is no standard.

The culling mode is set on a mesh-by-mesh basis.

7.3 Mesh Routines

7.3.1 Create Mesh

Creates an empty named or anonymous polygon mesh in the current display list. It is the current mesh, and all subsequent add routines associated with meshes are added to this mesh.

Function Header

```
int sgl_create_mesh( sgl_bool generate_name);
```

Parameters

I `generate_name`	If non-zero, the routine returns a name for the mesh, otherwise it is anonymous.
`returned value`	If a name has been requested then this is either the name or an error value. The returned value is undefined if no name is requested.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the mesh was not created.
`sgl_err_no_name`	The mesh was not created as there were no available names.

7.3.2 Modify Mesh

Modifies a named or anonymous polygon mesh. The named mesh becomes the current mesh, so that the next set of contiguous mesh related operations apply to it. It remains the current mesh until a non mesh operation is performed (with the exception of material definitions).

There is also a `clear_mesh` parameter, which causes all faces and point definitions to be deleted from the mesh.

Function Header

```
void sgl_modify_mesh( int mesh_name,
                      sgl_bool clear_mesh);
```

Parameters

I `mesh_name`	The name of the mesh to modify.
I `clear_mesh`	If non-zero, the faces and points are removed from the mesh.

Errors

`sgl_err_bad_name`	There is no mesh of the given name in the display list.
`sgl_err_no_mem`	The display list has become too large for the system.

7.3.3 Set Cull Mode

This sets the face culling mode of the current mesh to one of three modes—no culling (the default), cull clockwise faces, or cull counterclockwise faces.

Function Header

```
typedef enum
{ sgl_no_cull,
  sgl_cull_anticlock,
  sgl_cull_clockwise
} sgl_cull_mode;
void sgl_set_cull_mode( sgl_cull_mode mode);
```

Parameters

I	`mode`	The required culling mode for the current mesh.

Errors

`sgl_err_no_mesh`	Not currently editing a mesh.
`sgl_err_bad_parameter`	An invalid value was passed.

7.3.4 Delete Mesh

Deletes the named mesh.

Function Header

```
void sgl_delete_mesh( int mesh_name);
```

Parameters

I	`mesh_name`	The name of the mesh to be deleted.

Errors

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

7.3.5 Add Vertices

Adds a set of vertices to the mesh. If *k* vertices are to be added to the mesh and there are already *n* in it, then the new points are indexed from *n* to *n+k-1* inclusive.

You can also supply normals for the vertices (for smooth shading) and explicit *u* and *v* coordinates for user-defined texture wrapping.

If there is no current mesh then a new anonymous mesh is created.

Function Header

```
void sgl_add_vertices( int num_to_add,
                sgl_vector *vertices,
                sgl_vector *vertex_normals,
                sgl_2d_vec *vertex_uvs);
```

Parameters

I `num_to_add`	The number of vertices to add to the mesh. If this exceeds the limit of points in a mesh, the extra points are ignored.
I `vertices`	A pointer to an array of the vertices to be added.
I `vertex_normals`	If this is not `NULL`, then this is a pointer to a set of surface normals, one for each vertex, which are used to calculate smooth shading. If `NULL`, then it is assumed that no normals are supplied.
I `vertex_uvs`	If this is not `NULL`, then this is a pointer to a set of *uv* coordinates for the texture definition. These are used if the texture wrapping mode is set to user-defined.

Errors

`sgl_err_bad_parameter`	`num_to_add` is either zero or negative, or `vertices` is a null pointer. No vertices were added.
`sgl_err_no_mem`	The display list has become too large for the system.

7.3.6 Add Face

Defines a new mesh face, given a list of indices into the vertex table of the current mesh. Faces are assumed to have a front and back, but only front facing faces are rendered. It is assumed that points are added in a clockwise order when viewed from the front face. There must be at least 3 points in the face.

The face is assumed to be convex and that the points are co-planar. The actual normal of the plane is calculated from the first 3 points in the face, which therefore must not be co-linear.

There is likely to be some speed advantage if adjacent faces are added in sequential order.

Function Header

```
void sgl_add_face( int num_face_points,
                   int *vertex_ids);
```

Parameters

I `num_face_points`	The number of vertices defining the face. If the number of vertices exceeds the maximum per face, the additional vertices are ignored.
I `vertex_ids`	A pointer to an array of the vertex IDs which define the face. These vertices must have already been added to the mesh.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.
`sgl_err_bad_parameter`	Either `vertex_ids` is a null pointer so the face was not added; or two or more of the `vertex_ids` are the same; or one or more `vertex_ids` refer to a vertex that does not exist.
`sgl_err_no_mem`	The display list has become too large for the system.

7.3.7 Num Vertices

Returns the number of vertices in the current mesh.

Function Header

```
int sgl_num_vertices( );
```

Parameters

`returned value`	The number of vertices or points defined in the mesh, or an error value if negative.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.

7.3.8 Num Faces

Returns the number of faces in the current mesh.

Function Header

```
int sgl_num_faces( );
```

Parameters

`returned value`	The number of faces defined in the mesh, or an error value if negative.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.

7.3.9 Num Face Vertices

Returns the number of vertices in the specified face of the current mesh. Faces are numbered from 0.

Function Header

```
int sgl_num_face_vertices( int face);
```

Parameters

`face`	The index of the face in the current mesh.
`returned value`	The number of vertices used in the face, or an error value if negative.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.
`sgl_err_bad_parameter`	`face` is not a valid face.

7.3.10 Get Vertex

Gets the values for the specified vertex of the current mesh object. Vertices are numbered from 0.

Function Header

```
int sgl_get_vertex( int vertex,
             sgl_vector position,
             sgl_vector normal,
             sgl_2d_vec uv);
```

Parameters

I `vertex`	The index of the vertex to retrieve.
O `position`	The position of the point in its local coordinate space.
O `normal`	The normal for the point (meaningful only for smooth shading).
O `uv`	The user-defined *u* and *v* texture coordinates (meaningful only for user-defined texture wrapping).
`returned value`	Error status.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.
`sgl_err_bad_parameter`	`vertex` is not a valid vertex or one or more of the other parameters is a null pointer.

7.3.11 Set Vertex

Sets the values for the specified vertex of the current mesh object. Vertices are numbered from 0.

Function Header

```
void sgl_set_vertex( int vertex,
              sgl_vector position,
              sgl_vector normal,
              sgl_2d_vec uv);
```

Parameters

I `vertex`	The index of the vertex to change.
I `position`	The position of the point in its local coordinate space.
I `normal`	The normal for the point. If you do not need to smooth shade, pass a dummy parameter.
I `uv`	Your *u* and *v* texture coordinates. If you do not need to texture map the object, pass `NULL`.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.
`sgl_err_bad_parameter`	`vertex` is not a valid vertex, or mesh optimization is set in the .INI file.

7.3.12 Get Face

Retrieves the points which make up the specified face of the current mesh object. Faces are numbered from 0.

Function Header

```
int sgl_get_face( int face,
                  int *vertex_indices);
```

Parameters

I `face`	The face to retrieve.
O `vertex_indices`	A pointer to an array of `int`s to hold the indices of the vertices that make up the face. It is assumed that this array is large enough to hold the list. Use `sgl_num_face_vertices` if necessary.
`returned value`	Error status.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.
`sgl_err_bad_parameter`	`face` is not a valid face or `vertex_indices` is a null pointer.

7.3.13 Set Face

Changes the definition of the specified face of the current mesh object. Faces are numbered from 0.

Function Header

```
void sgl_set_face( int face,
                   int num_vertices,
                   int *vertex_indices);
```

Parameters

I `face`	The face to redefine.
I `num_vertices`	The number of vertices in the following list.
I `vertex_indices`	A pointer to an array of `int`s that hold the indices of the vertices that redefine the face.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.
`sgl_err_bad_parameter`	`face` is not a valid face, `vertex_indices` is a null pointer, two or more of the `vertex_indices` are the same, or one or more refers to a vertex that does not exist.
`sgl_err_no_mem`	The display list has become too large for the system.

7.3.14 Delete Vertex

Deletes the specified vertex of the current mesh object. All faces that use that point are also deleted. Vertices are numbered from 0.

Function Header

```
void sgl_delete_vertex( int vertex);
```

Parameters

I `vertex`	The vertex to delete.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.
`sgl_err_bad_parameter`	`vertex` is not a valid vertex.

7.3.15 Delete Face

Deletes the specified face of the current mesh object. Faces are numbered from 0.

Function Header

```
void sgl_delete_face( int face);
```

Parameters

I `face`	The face to delete.

Errors

`sgl_err_no_mesh`	There is no current mesh selected.
`sgl_err_bad_parameter`	`face` is not a valid face.

8 MATERIALS

8.1 Introduction

Materials define the surface characteristics of an objects—how much light they reflect, whether they are matt or shiny, and whether they radiate their own light. Materials also govern translucency. These factors can either be constant across surfaces or varied by bitmap textures.

During display list traversal, PowerSGL maintains material properties as part of state information. Whenever it encounters a material definition in the display list, it updates just the properties that are affected by the new definition. For example, if the material definition covers just the specular properties (see below), then only the specular properties in the state information are altered—the diffuse values are not changed.

As with object primitives (i.e., convex polyhedra or meshes), a material definition can be named or anonymous. When a material is created, it becomes the current material, so that all contiguous material operations apply to it. Materials are the only entity that can be defined within an object primitive without terminating the object definition, provided they are specified to be local.

8.1.1 Ambient Reflectivity

This property determines the fraction of the ambient light that is reflected back by the objects. It gives values for each color component. This is generally a combination of the diffuse and specular properties, depending on the type of surface being modeled. For materials such as plastic, set the ambient reflectivity to be the same as the diffuse[1].

8.1.2 Diffuse Reflectivity

The diffuse color models the matt color component of an object. This assumes that when a directional light strikes the surface, it is bounced back equally in all directions. The amount of light that falls on the surface is proportional to the cosine of the angle between the surface normal and the direction of the incoming light.

The diffuse shading can either be flat or smooth shaded. Smooth shading makes polygonal objects look curved, but is computationally more expensive than flat shading[2].

8.1.3 Specular Reflectivity

The specular property is used to simulate the glossy highlights that lights can make on shiny surfaces. PowerSGL uses a phong-like model, which has a reflectance amount, and concentration and shininess value. The shininess simulates how smooth or polished the surface is. The larger the shininess value, the smaller the highlight. To get the best visual results, if the concentration is high, then the reflectance can be high, but if low, then the reflectance should also be low.

---

[1] Foley, van Dam, Feiner and Hughes, “Computer Graphics, Principles and Practice”, Second Edition, Addison and Wesley, 1990, page 723

[2] Foley, van Dam, Feiner and Hughes, “Computer Graphics, Principles and Practice”, Second Edition, Addison and Wesley, 1990, pages 723 to 727

The specular component can be either flat or smooth shaded, however smooth shaded specular highlights are more expensive to calculate. Smooth highlights also depend on the functionality of the hardware[1].

8.1.4 Glow Color

The glow color is the amount of light a surface radiates in all directions. It is not affected by other lights in the scene.

8.1.5 Effect of Textures on Material Properties

You can use textures to further modify the reflectivity and glow color. For example, you can specify that the actual diffuse color for an object is the product of the texture color at that pixel and the specified diffuse color. Note that this extra multiply depends on the hardware and software—it may not be possible in some implementations.

This facility allows the same texture to be re-used with different variations. For example, the same leaf texture could be used on a number of trees, but be made to look different by varying the diffuse color.

8.1.6 Transparency

PowerSGL implements non-refractive transparency, which allows objects to appear partially see-through. To calculate the value of a pixel, PowerSGL takes a specified percentage, *k*, of the image behind the translucent object, and adds the color of the object to it.

Transparency can either be set as a constant per surface, or be under the control of textures. Note that texture maps fall into two categories—normal and translucent. Translucent maps have a translucency value per pixel. Using a translucent texture map automatically implies use of translucency. A normal texture can only have a constant tranlucency value, as set by `sgl_set_opacity`.

The early versions of PowerVR computed translucency by performing an explicit number of passes. The first pass computed the visible opaque surfaces. Subsequent passes handled the translucency layers. For each pixel in each translucency pass, the closest object, which was in front of the closest object from the previous passes, was kept and mixed with the results of the previous pass. PowerSGL must therefore sort[2] the passes into back to front order to get correct results. You must specify what objects are to be in separate translucency passes by using the `sgl_new_translucent` function. If you don't call this function, then on the current PowerVR hardware only one layer of translucency will be visible. Later versions of the PowerVR hardware will dynamically determine how many passes are needed, but for backward compatibility we recommend still using the `sgl_new_translucent` function.

Finally, note that every layer or pass of translucency increases the amount of texturing and shading performed per pixel, and so can represent a significant overhead on rendering.

---

[1] Foley, van Dam, Feiner and Hughes, "Computer Graphics, Principles and Practice", Second Edition, Addison and Wesley, 1990, pages 728 to 730

[2] This sorting is determined by approximate *z* depths across entire regions of the screen and not on a pixel-by-pixel basis. Translucent objects which are very close to each other in *x*,*y* and *z* may not be sorted correctly, resulting in one not being visible behind the other.

8.1.7 Default Material

When a display list is traversed during rendering, the initial material values are set to a flat gray color. Ambient reflectivity starts off as [0.5, 0.5, 0.5] (R,G,B), the default diffuse color is [0.5, 0.5, 0.5], and there are no specular, glow, texture, or translucent components.

8.2 Material Routines

8.2.1 Create Material

Creates a new material definition, which becomes the current material.

Note: As a shortcut, if there is no current material, then any material routine (other than create, delete or modify) creates a new anonymous material.

Function Header

```
int sgl_create_material( sgl_bool generate_name,
                         sgl_bool make_local);
```

Parameters

I `generate_name`	If non-zero, the routine returns a name for the new material, or an error value in the event of failure. If zero, no valid value is returned.
I `make_local`	If non-zero and you are creating either a convex polyhedron or mesh primitive, then the material is created local to that primitive. If zero, or there is no current primitive, then the material is appended to the display list as a separate entity.
`returned value`	Either the name of the new material, or an error, provided a name was requested.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the material has been ignored.
`sgl_err_no_name`	The material has not been created as there were no available names.

8.2.2 Modify Material

Sets the named material to be the current material. You can optionally clear out any definitions within the existing material.

Function Header

```
void sgl_modify_material( int name,
                          sgl_bool clear_material);
```

Parameters

I	`name`	The name of the material to be modified.
I	`clear_material`	If non-zero, this resets the material definition to be empty (i.e., the way it was when first created).

Errors

`sgl_err_bad_name`	There is no material of the given name in the display list.

8.2.3 Set Ambient Color

Defines a new ambient color in the current material, replacing any existing ambient definition.

Function Header

```
void sgl_set_ambient( sgl_colour colour);
```

Parameters

I	`colour`	The ambient color.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the color has been ignored.

8.2.4 Set Diffuse Color

Defines a new diffuse color in the current material, replacing any existing diffuse definition.

Function Header

```
void sgl_set_diffuse( sgl_colour colour);
```

Parameters

I	`colour`	The diffuse color.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the color has been ignored.

8.2.5 Set Specular Color

Defines a new specular color and shininess and concentration value in the current material. The shininess value ranges from 1 to about 1000, corresponding to the power function defined by Phong[1].

If the shininess is set ≤ 0 then specular reflectivity is turned off.

Function Header

```
void sgl_set_specular( sgl_colour colour,
                               int shininess);
```

Parameters

I `colour`	The specular color.
I `shininess`	The shininess of the surface.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the color has been ignored.

8.2.6 Set Glow Color

Sets the glow color for subsequent surfaces.

Function Header

```
void sgl_set_glow( sgl_colour colour);
```

Parameters

I `colour`	The glow color.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the color has been ignored.

[1] Foley, van Dam, Feiner and Hughes, “Computer Graphics, Principles and Practice”, Second Edition, Addison and Wesley, 1990, pages 728 to 730

8.2.7 Set Opacity

Sets the opacity (or translucency) of the object. The values are in the range [0.0,1.0] with 1.0 being totally opaque and 0.0 being totally transparent. The default value is 1.0. Note that a totally transparent surface is not the same as an invisible plane in a convex polyhedron.

Function Header

```
void sgl_set_opacity( float opacity);
```

Parameters

I opacity	The opacity of the material.

Errors

sgl_err_no_mem	The display list has become too large for the system so the opacity change has been ignored.

8.2.8 New Translucent Layer

Forces the next group of translucent objects to be processed as a separate pass. When the PowerVR technology achieves full, per-pixel sorting of translucent objects, this function will be ignored, but should still be used for backward compatibility.

Function Header

```
void sgl_new_translucent( );
```

Errors

sgl_err_no_mem	The display list has become too large for the system so the new translucent command has been ignored.

8.2.9 Set Texture Map

Defines which texture map to use for subsequent objects, planes and polygons. The current texture wrapping function is used to apply the texture to the surface. The shading parameters affected by the texture are determined by the `sgl_set_textures_effect` routine (see page 8-8).

The routine also specifies how the texture map tiles when textures repeat. You may optionally request that the texture flip along the *u* and/or *v* boundaries.

If the special texture name `SGL_NULL_TEXTURE` is used then texture mapping is turned off.

Function Header

```
void sgl_set_texture_map( int texture_name,
                          sgl_bool flip_u,
                          sgl_bool flip_v);
```

Parameters

I `texture_name`	The name of the texture to use.
I `flip_u`	The texture reverses direction each time it repeats in the *u* direction.
I `flip_v`	The texture reverses direction each time it repeats in the *v* direction.

Errors

`sgl_err_bad_name`	There is no texture of the given name in the display list.
`sgl_err_no_mem`	The display list has become too large for the system so the texture has been ignored.

8.2.10 Set Textures Effect

Determines which properties are affected by the current texture, allowing the ambient, diffuse, specular and glow colors to be determined by the current texture and texture wrap.

If no properties are set to be modified by the texture, then the texture mapping is turned off. The other texture settings (i.e., texture name, flip settings, and texture wrapping) remain set.

Function Header

```
void sgl_set_textures_effect( sgl_bool affect_ambient,
                              sgl_bool affect_diffuse,
                              sgl_bool affect_specular,
                              sgl_bool affect_glow);
```

Parameters

I `affect_ambient`	If non-zero, the texture affects the ambient reflectivity color.
I `affect_diffuse`	If non-zero, the texture affects the diffuse reflectivity color.
I `affect_specular`	If non-zero, the texture affects the specular reflectivity color.
I `affect_glow`	If non-zero, the texture affects the glow color.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the texture effect has been ignored.

8.2.11 Use Material Instance

This function allows instantation of a previously defined material.

Function Header

```
void sgl_use_material_instance(int material_name, sgl_bool make_local);
```

Parameters

I	`material_name`	Name of the previously defined material.
I	`make_local`	TRUE if the instance is to be local to a mesh or convex currently being defined. (See sgl_create_material for more detailed defintion)

Errors

sgl_err_bad_name	There is no material of the given name in the display list.
sgl_err_no_mem	The display list has become too large for the system so the material instance has been ignored.

9 TEXTURE DEFINITION AND TEXTURE WRAPPING

9.1 Introduction

This chapter describes how texture maps are defined, and how they can be applied to the surfaces of an object. It also describes MIP mapping—the technique used for anti-aliasing of textures.

9.2 Texture Definitions and MIP Maps

PowerSGL 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 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.

9.2.1 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 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.

9.3 Texture Memory and Texture Caching

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 9-6 and 9-11), and then reference its name in the display list.

In some applications, however, the potential texture requirements for a scene may exceed that of the texture memory. In this situation, it is necessary to load only those textures that are needed for the current image. In some situations, the application knows what textures are likely to be in the current view, but in others it may not. The solution is for PowerSGL, during the display list traversal, to determine what textures are required by the current view, and report these via a call back function. You can then add and delete textures as required before allowing the rendering to continue.

---

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

There are limitations on what can be performed during the call back function in order for it to work correctly:

- Cached textures may be loaded and unloaded freely—this is what the system is intended to do.
- Materials may be modified, and lists may be marked as ignored. Further modification of the display list structure is inadvisable.

The operations that must not be performed during the call back are:

- The camera position must not be moved, because this makes the list of required textures incorrect, and the display list traversal still assumes the current camera position.
- Lists must not be deleted, as the traversal could crash.

To enable the call back system, you must specify a call back routine with a data structure for storing the cached texture details. PowerSGL then automatically calls this routine when the function and a cacheable texture have been defined. The routine is always called, even if all required textures are loaded, as this gives the opportunity to change to textures of higher resolutions.

9.4 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 9-9).It can then be loaded, say, directly from disk.

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

9.4.1 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 “preprocessed” version sets this to “PT”xy  where xy depend
  // on the type of the texture map. (x is set to a sgl_map_types value,
  // and y to a 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;
```

9.4.2 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 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.

9.4.3 Texture Types

PowerSGL 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;
```

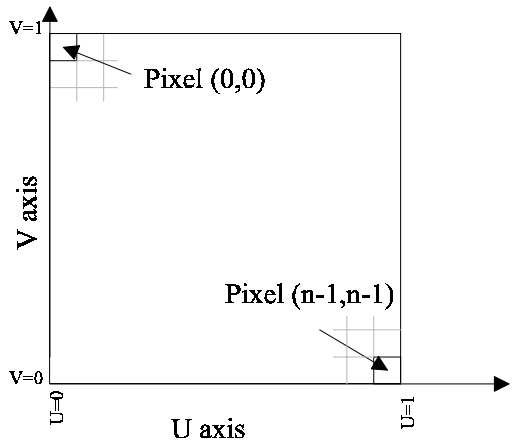
9.4.4 Conversion Between Color Formats

Because the intermediate format is 24-bit, whereas the internal maps have fewer bits, the library converts between the formats. PowerSGL 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 to dither when going from the intermediate map to the internal format.

9.5 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.

9.6 Texture Caching Call Back Definitions

The cached texture call back facility requires you to create a data structure to hold the report of the texture caching and the call back function. The data structure is an array of `structs` with the type `sgl_tex_callback_struct`. This `struct` has the following fields:

```
typedef struct
{
       int texture_name;
      long user_id;
  sgl_bool loaded;
  sgl_bool needed;
  sgl_bool used_previous_render;
       int approx_object_size;
}sgl_tex_callback_struct;
```

`texture_name`	This is the name assigned by PowerSGL when the cached texture is created (using `sgl_create_cached_texture`).
`user_id`	This is a value that you supply. It is always associated with the texture, but is otherwise ignored by PowerSGL. It can be used, for example, as an index into an array, or a pointer where pointers occupy the same number of bits as longs.
`loaded`	Indicates if the texture is currently loaded in memory.

`needed`	Indicates that the texture is required in the current render.
`used_previous_render`	Indicates that the texture is still being used by the hardware.
`approx_object_size`	If the texture is needed this gives the approximate size of the object using it in screen pixels.

You must create an array of these structures, which are then passed to the call back registering function.

The call back function takes the following form:

```
void MyTextureCallBack( sgl_tex_callback_struct TexDataArray[],
                                                 int num_used,
                                               long free_texture_memory);
```

When called, the function is given the array in which the cached texture data is stored, the number of entries that are used, and the amount of free texture memory available. Note that the free texture memory is unlikely to be contiguous. You can destroy the `TexDataArray`, as PowerSGL uses its own internal structure.

9.7 Texture Definition Routines

9.7.1 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'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 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 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 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.

9.7.2 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 9-6), 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 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 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.

9.7.3 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.

9.7.4 Set Texture

Redefines the pixels of an existing texture. The dimensions and type of the texture are unchanged. PowerSGL 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.

9.7.5 Delete Texture Map

Deletes a texture map from PowerSGL'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.

9.7.6 Create Cached Texture

Creates a cached texture. It returns a name for the texture, but does not load any actual texture data. It also takes a user value which is stored with the texture. The texture can be deleted using `sgl_delete_texture` (see page 9-12).

Function Header

```
int sgl_create_cached_texture( long user_id);
```

Parameters

I `user_id`	A value which you supply.
`returned_value`	The name of the create texture, or an error value if negative.

Errors

`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.

9.7.7 Load Cached Texture

Sets the pixel data for a cached texture. If data was previously defined, it is replaced. The function takes parameters similar to `sgl_create_texture` (i.e., takes intermediate maps of either user-accessible or predefined varieties, see page 9-6) except that, in certain circumstances of pre-processed type (see below), it can override the predefined type.

If a predefined, MIP mapped texture type is supplied, you can choose not to load the higher resolution MIP maps if texture memory is scarce.

Function Header

```
void sgl_load_cached_texture( int name,
                    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_bool override_preprocessed_type);
```

Parameters

I `name` The name of the cached texture.

I `map_type` As for Create Cached Texture.

I `map_size` As for Create Cached Texture.

I `generate_mipmap` As for Create Cached Texture.

I `dither` As for Create Cached Texture.

I `pixel_data` As for Create Cached Texture.

I `filtered_maps` As for Create Cached Texture.

I `override_preprocessed_type` For certain cases of pre-processed type, this allows you to load a smaller resolution map. If set to non-zero, the `map_type` and `map_size` variables are examined and the portion of the pre-processed map is loaded. The current limitations are:

- The pre-processed map must be a MIP map.
- The pre-processed map and the `map_type` must both be the same type.
- The size of the supplied map must be larger than that requested.

If these conditions are not satisfied, the override parameter is effectively ignored, and the pre-processed map is loaded without modification.

`returned_value` The name of the create texture, or an error value, if negative.

Errors

`sgl_err_bad_name`	An invalid cached texture name has been supplied.
`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.

9.7.8 Unload Cached Texture

Frees the space occupied by the pixels of a cached texture, if any, but does not delete the texture.

Function Header

```
void sgl_unload_cached_texture( int name);
```

Parameters

I `name`	The name of the cached texture.

Errors

`sgl_err_bad_name`	An invalid cached texture name has been supplied.

9.7.9 Register Texture Call Back Function

Specifies the texture call back function and associated data structure.

Definitions

```
typedef struct {
  /*
  // SGL and User name for texture
  */
   int texture_name;
  long user_id;
  /*
  // Whether it is currently loaded,
  // needed for this render,
  // still in use by a previous render,
  // (and, if needed) the approximate size in pixels, of the largest
  //  object that does use it. (Note this is the whole object, even if
  // only a small surface of that object uses the texture).
  */
  sgl_bool loaded;
  sgl_bool needed;
  sgl_bool used_previous_render;
       int approx_object_size;
}sgl_tex_callback_struct;
typedef void (CALL_CONV *sgl_tex_callback_func)
sgl_tex_callback_struct tex_data_array[],
                    int num_used,
                   long free_space);
```

Function Header

```
void sgl_register_texture_callback( sgl_tex_callback_func func,
                                    sgl_tex_callback_struct tex_data_array[],
                                                            int array_size);
```

Parameters

I	`func`	Your texture call back function. Passing `NULL` disables texture callbacks.
I	`tex_data_array`	An array of texture result structures that you allocate and define.
I	`array_size`	The size of the array.

9.7.10 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

9.8 Two-Step Texture Wrapping Functions

Texture wrapping is a technique of automatically placing textures on objects. PowerSGL implements a two-step wrapping process, based on the work by Bier and Sloan[1] .

A 2D texture is first mapped onto a simple 3D shape, which is either a plane, a cylinder, or a sphere. This mapping is called the S mapping.

$$S(u,v) \rightarrow (x_i, y_i, z_i)$$

The second mapping takes the now 3D texture and maps it onto the object's surface. This mapping is referred to as the O mapping. Again, there is a small set of supplied mappings.

$$O(x_i, y_i, z_i) \rightarrow (x_o, y_o, z_o)$$

The combination of these two mappings can produce a number of useful texture wraps.

Note: If you have explicitly defined *u* and *v* mappings for a surface, then the texture wrapping information is ignored in preference to this explicit mapping.

---

[1] Watt and Watt, "Advanced Animation and Rendering Techniques. Theory and Practice.", First Edition, Addison and Wesley, 1992, pages 181 to 182

9.8.1 S Maps

The three S maps supplied by PowerSGL—planar, cylindrical, and spherical—define a plane, cylinder, or sphere, in 3D space.

The orientation and position of these simple surfaces depends on where they are declared in the display list, that is, they are affected by the transformation matrices. The textures can therefore be animated relative to the objects to which they are applied.

9.8.1.1 Planar Map

The planar map is a plane perpendicular to the *z* axis, with the *u* and *v* coordinates aligned with the *x* and *y* axes. This mapping, and its equation, are shown in Figure 1 below. Note that the s_u, s_v, o_u and o_v are scale and offset values used to adjust the position of the texture. The scaling values would normally be set to 1.0, and the offsets to 0.0. The scaling values s_u and s_v must be non-zero.

$$S_p(u,v) = \begin{pmatrix} s_u u + o_u \\ s_v v + o_v \\ 0 \end{pmatrix}$$

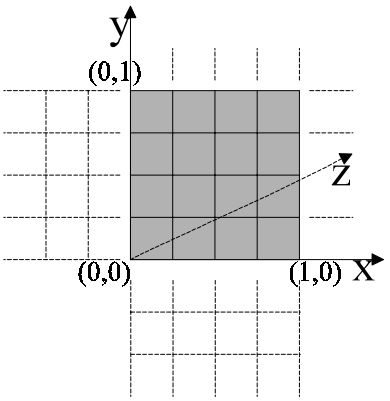


Figure 1

9.8.1.2 Cylindrical Map

The cylindrical map is shown in Figure 2 below. It is a right cylinder of radius *r*, and height, s_v, with its axis aligned with the *y* axis. The *v* values run vertically, and the *u* goes around the cylinder.

$$S_c(u,v) = \begin{pmatrix} r\cos\left(2\pi\left(s_u u + o_u\right)\right) \\ s_v v + o_v \\ r\sin\left(2\pi\left(s_u u + o_u\right)\right) \end{pmatrix}$$

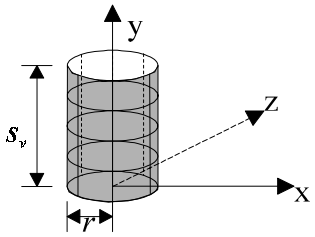


Figure 2

9.8.1.3 Spherical Map

One common way of spherical mapping is to map *u* to the longitude and *v* to the latitude, as is sometimes used in cartography. However, mapping *v* to the sine of the latitude[1] means that equal areas of the texture are mapped to equal areas of the sphere. This utilizes the texture map more effectively, and so is employed in PowerSGL (as maps can be pre-distorted, a texture which is correct for a latitude-style mapping, can be easily converted to the sine-latitude variety). This mapping is then:

$$S_s(u,v) = \begin{pmatrix} r\cos\left(2\pi\left(s_u u + o_u\right)\right)\sqrt{1-\left(s_v(2v-1)+o_v\right)^2} \\ r s_v(2v-1)+o_v \\ r\sin\left(2\pi\left(s_u u + o_u\right)\right)\sqrt{1-\left(s_v(2v-1)+o_v\right)^2} \end{pmatrix}$$

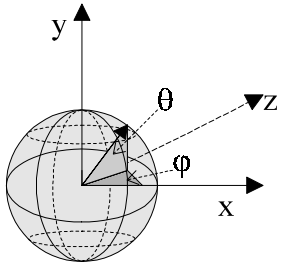


Figure 3

Note that not all values of *v* are valid, and that its domain depends on the values of s_v and o_v. We recommend starting values of 1.0 for the scaling and 0.0 for the offsets.

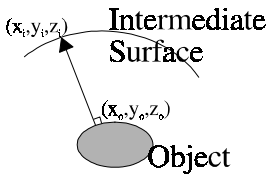
9.8.2 O Maps

The second stage of the texture wrapping uses the O map, i.e., $O(x_i,y_i,z_i) \rightarrow (x_o,y_o,z_o)$. PowerSGL provides four different mappings. Two are view-independent, while two can be used for environment mapping. The O maps provided are:

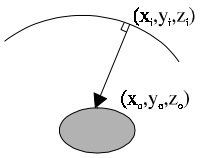
Object Normal	The object point, (x_o,y_o,z_o), gets mapped from the intersection of the objects normal at (x_o,y_o,z_o) and the intermediate surface. Note that if there is more than one possible point, then the closest is chosen.
Intermediate Surface Normal	The intermediate surface point, (x_i,y_i,z_i), is the point whose normal intersects the object point (x_o,y_o,z_o). Again, the closest solution is chosen.
Reflected Ray	The intermediate point is the intersection of the reflected viewing ray and the intermediate surface.
Transmitted Ray	The intermediate point is the intersection of a transmitted ray and the intermediate surface. A refractive index is specified. Note that the transmitted ray assumes that it only hits one surface of the object. It is refracted only once.

[1] Foley, van Dam, Feiner and Hughes, "Computer Graphics, Principles and Practice", Second Edition, Addison and Wesley, 1990, pages 758 to 759

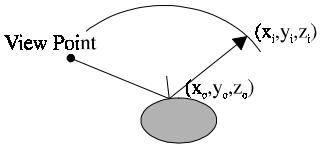
These are illustrated in the following diagrams:



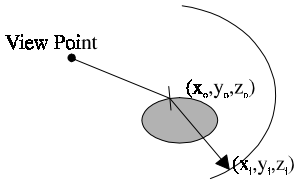
Object Normal



Intermediate Surface Normal



Reflected Ray



Transmitted Ray

9.8.3 Combined Mappings

There are a number of ways to combine these mappings. A few examples are:

Shrink wrap	by using a cylindrical S map and the Intermediate Surface Normal O-Map[1].
Reflection map	by using a spherical S map, and the reflected ray O map.
Magnifying glass effect	by using a planar S map, and the transmitted ray O map. Some adjusting would have to be done to align the map correctly.

[1] Watt and Watt, “Advanced Animation and Rendering Techniques. Theory and Practice.”, First Edition, Addison and Wesley, 1992, page 182

9.9 Texture Wrap Routines and Type Definitions

9.9.1 Wrap Types

The following two types specify the S and O maps available in PowerSGL, as described previously.

```
typedef enum
{
  sgl_smap_plane,
  sgl_smap_cylinder,
  sgl_smap_sphere
} sgl_smap_types;
typedef enum
{
  sgl_omap_obj_normal,
  sgl_smap_inter_normal,
  sgl_smap_reflection,
  sgl_smap_transmission
} sgl_omap_types;
```

9.9.2 Set S Map

Defines the type of intermediate surface used. The S Map is placed in the current coordinate system. If transformations are performed between the S Map definition and the object definition, these transformations orient the object in relation to the S Map.

Function Header

```
void sgl_set_smap( sgl_smap_types smap_type,
                               float su,
                               float sv,
                               float ou,
                               float uv,
                               float r);
```

Parameters

I `smap_type`	The type of S map to use. If this is a user-defined map, then the other parameters are ignored, and no texture mapping is performed on object surfaces that do not have *u* and *v* parameters explicitly defined.
I `su, sv`	These are the texture scaling values given in the equations detailed on pages 9-17 and 9-17. They specify how rapidly the texture is repeated. For the cylindrical and spherical maps, it is advisable to make these integer values. We recommend that you set these to starting values of 1.0f. Values of zero or very small values up to 10^{-6} are considered illegal, and are set to an internally-defined safe value.
I `ou, ov`	These are the texture offset values in the above equations. We recommend that you set these to starting values of 0.0f.
I `r`	This is the radius value for the cylindrical and spherical maps. This is ignored for planar maps.

Errors

`sgl_err_bad_parameter`	`smap_type` is not a valid type.
`sgl_err_no_mem`	The display list has become too large for the system so the S Map has been ignored.

9.9.3 Set O Map

Defines the type of mapping from the intermediate surface to the object.

Function Header

```
void sgl_set_omap( sgl_omap_types omap_type,
                   float refractive_index);
```

Parameters

I `omap_type`	The type of O map to use.
I `refractive_index`	If a transmission O map is required, this specifies the refractive index to use when calculating the transmitted ray direction.

Errors

`sgl_err_bad_parameter`	`omap_type` is not a valid type.
`sgl_err_no_mem`	The display list has become too large for the system so the O Map has been ignored.

10 LIGHTS AND SHADOWS

10.1 Introduction

Lights in PowerSGL are used to illuminate objects, creating specular highlights if desired, and to generate shadows. There are several light models, and you can place several lights in the scene. The scope of the lights, i.e., which objects they illuminate, is governed by their placement within the display list, as is their position in 3D space.

10.1.1 Scope and Position of Lights

Lights form part of the PowerSGL state information which can be preserved when child lists are entered during display list traversal. When a light is created, it can only illuminate the objects which follow after it in the display list. As soon as the display list traversal leaves a display list which restores the system state to the one before the light was created, the light ceases to exist and does not illuminate any further objects. Therefore, if a light must illuminate the entire scene, it must be created near the start of the scene display list. In effect, lights behave like the identifier scoping rules in a high-level computer language.

The position of a light can be determined in two ways. After a light has been created in the list, its position can be set up in another part of the list, provided the light is still in scope. As a simple example, consider the landing lights on an aircraft. We would like these to illuminate the whole scene (or at least the runway), but the positions of the lights need to be determined by the nested lists that define the moving aircraft. As long as the lights are declared in (probably) the top level list, their positions can be set in the lists for the aircraft. To determine the positions of the lights, the tree is traversed from the position marker backwards to the top of the display list. If, for some reason, the light position markers are not in the scope of the light, then the positioning may be unusual.

If a light's position is not explicitly set, its location is determined by the transformations in effect at the point of creation.

10.1.2 Light Types

The following are the light types supported by PowerSGL.

Ambient Light — simulates light which is coming equally from every direction as a result of diffuse inter-reflections. You can set the ambient light level many times in the scene. For example, one end of a large room may be darker than the other—the ambient level could be changed to approximate this. The ambient light level cannot generate shadows.

Parallel Lights — simulate lights which are effectively an infinite distance away. For example, the sun could be considered a parallel light. The shading for parallel lights is relatively inexpensive to calculate, as the lighting direction stays constant across the object surfaces. Shadows can be cast from parallel lights.

You define the light by specifying the direction in which the light rays travel.

Point Lights/Spotlights

a point source light simulates a point in space which is radiating light in all directions. A point source light can also behave like a spotlight[1] by using the principal lighting direction and concentration value. The higher the concentration value, the smaller the spot. If the concentration value is 0, then the light radiates equally in all directions.

To get a gradually changing pool of light from a point to a spotlight, enable smooth shading.

10.1.3 Light Colors

Light colors are defined by red, green and blue components. If all three values are identical (resulting in a shade of gray) then internal optimizations are performed which may significantly increase the rendering speed where shading, especially smooth shading, is used. In most situations it is adequate to assume that lights are white or gray, as sunlight, tungsten and florescent lights are effectively white. Colored lights are useful for special effects, such as simulating red light from a fireplace, or yellow sodium vapor lamps along a road.

10.1.4 Special Light Attributes and Limits

PowerSGL imposes a limit to the number of lights that can be in action at a time—further light definitions are ignored. There is also a maximum number of lights that can cast shadows, and a maximum number that can generate smooth specular highlights. Any request to use such facilities, when the maximum number has been allocated, are ignored.

These limits are specified in the library as:

- `SGL_MAX_ACTIVE_LIGHTS`
- `SGL_MAX_SHADOW_LIGHTS`
- `SGL_MAX_SPECULAR_LIGHTS`

A light's capability to cast shadows or generate specular highlights is determined when the light is created. The capabilities can be turned on or off many times within the scope of the light. This can be used by the application to speed up rendering when a particular feature is not required.

When the scope of a light ends the number of active lights is decremented, as if it were set to generate specular highlights, and this facility becomes available for new lights. Due to the global nature of shadows, the shadow casting facility is not released for other lights.

10.1.5 Shadow Algorithm Limitations

To enable the real-time calculation of shadows, the algorithm in PowerSGL is constrained to work only on convex polyhedral primitives. Furthermore, it assumes that an object casting a shadow is fully opaque and ignores the invisible status of the faces. In general this is not a significant restriction, as the shadow generally does not need to be as detailed as the casting object to look effective.

---

[1] Foley, van Dam, Feiner and Hughes, “Computer Graphics, Principles and Practice”, Second Edition, Addison and Wesley, 1990, pages 731 to 733

10.2 Light Routines

10.2.1 Create Ambient Light

Creates an ambient light level that can either be set absolutely or relative to the current ambient level. Using relative mode could be useful in a model of a room, where the ambient light beneath a table is, say, 90% of the ambient light in the rest of the room. Increasing the brightness for the room automatically changes the light level beneath the table.

The initial ambient light level is gray with an intensity of 0.1.

Function Header

```
int sgl_create_ambient_light( sgl_bool generate_name,
                              sgl_colour colour,
                                sgl_bool relative);
```

Parameters

I generate_name	If non-zero, the routine returns a name for the ambient light. This can be used to change the light level.
I colour	The color/intensity of the light. In general, values are in the range [0.0,1.0].
I relative	If non-zero, the color and brightness level is determined to be relative,i.e., is obtained by multiplying the current value by the supplied parameters. If zero, then the color or grayscale is set absolutely. If in relative mode, when either the previous ambient definition or the new definition is colored, then the ambient becomes colored. It remains gray in all other cases.
returned value	If a name is to be generated, then the returned value is either the name or an error value. It is otherwise undefined.

Errors

sgl_err_no_mem	The display list has become too large for the system so the light has been ignored.
sgl_err_no_name	The light has not been created as there were no available names.

10.2.2 Create Parallel Light

Creates a new parallel light source. If a name is returned, this can be used to edit and position the light.

Function Header

```
int sgl_create_parallel_light( sgl_bool generate_name,
                               sgl_colour colour,
                               sgl_vector direction,
                                 sgl_bool casts_shadows,
                                 sgl_bool smooth_highlights);
```

Parameters

I `generate_name`	If non-zero, the routine returns a name for the light.
I `colour`	The color/intensity of the light. In general, values are in the range [0.0,1.0].
I `direction`	Specifies the direction of the light rays. This is given in the local coordinate system, and so can be animated by manipulating transformations.
I `casts_shadows`	If non-zero, then this light generates shadows. If the maximum number of lights are already casting shadows, the flag is ignored.
I `smooth_highlights`	If non-zero, when smooth shading is enabled, this light can produce smooth shaded highlights on a surface. If the maximum number of lights are already allowed to do so, the flag is ignored. This maximum number depends on hardware support.
`returned value`	If a name is to be generated, then the returned value is either the name or an error value. It is otherwise undefined.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the light has been ignored.
`sgl_err_no_name`	The light has not been created as there were no available names.

10.2.3 Create Point Light

Creates a new point light source. If a name is returned, this can be used to both edit and position the light.

Function Header

```
int sgl_create_point_light( sgl_bool generate_name,
                            sgl_colour colour,
                            sgl_vector direction,
                            sgl_vector position,
                                   int concentration,
                              sgl_bool casts_shadows,
                              sgl_bool smooth_highlights);
```

Parameters

I `generate_name`	If non-zero, the routine returns a name for the light.
I `colour`	The color and intensity of the light. In general, values are in the range [0.0,1.0].
I `direction`	If creating a spotlight then this is the principle light direction. This is given in the local coordinate system, and so can be animated by manipulating transformations.
I `position`	If creating a point source light, this is the location of that light. The parameter is ignored for parallel lights.

I concentration	This value determines if a point source light behaves as a point source light (radiating equally in all directions) or as a spotlight. A value of 0 implies a point light source, with increasing values producing more spot-like effects. A sensible maximum is around 32, as values that are too high start to cause aliasing effects.
I casts_shadows	If non-zero, then this light generates shadows. If the maximum number of lights are already casting shadows, the flag is ignored.
I smooth_highlights	If non-zero, when smooth shading is enabled, this light can produce smooth shaded highlights on a surface. If the maximum number of lights are already allowed to do so, the flag is ignored. This maximum number depends on hardware support.
returned value	If a name is to be generated, then the returned value is either the name or an error value. It is otherwise undefined.

Errors

sgl_err_no_mem	The display list has become too large for the system so the light has been ignored.
sgl_err_no_name	The light has not been created as there were no available names.

10.2.4 Position Light

Positions a given light in the display list. Note that positioning an ambient light has no effect. If this routine is called several times, only the last call has any effect.

Function Header

```
void sgl_position_light( int name);
```

Parameters

I name	The name of the light.

Errors

sgl_err_bad_name	There is no light of the given name.
sgl_err_no_mem	The display list has become too large for the system so the new position has been ignored.

10.2.5 Set Ambient Light

Changes the values for the named ambient light.

Function Header

```
void sgl_set_ambient_light( int name,
                    sgl_colour colour,
                    sgl_bool relative);
```

Parameters

I	`name`	The name of the ambient light to change.
I	`colour`	As for Create Ambient Light.
I	`relative`	As for Create Ambient Light.

Errors

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

10.2.6 Set Parallel Light

Changes the values for the named parallel light.

Function Header

```
void sgl_set_parallel_light( int name,
                      sgl_colour colour,
                      sgl_vector direction,
                        sgl_bool casts_shadows,
                        sgl_bool smooth_highlights);
```

Parameters

I	`name`	The name of the light to modify.
I	`colour`	As for Create Parallel Light.
I	`direction`	As for Create Parallel Light.
I	`casts_shadows`	As for Create Parallel Light.
I	`smooth_highlights`	As for Create Parallel Light.

Errors

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

10.2.7 Set Point Light

Changes the values for the named parallel light.

Function Header

```
void sgl_set_point_light( int name,
                  sgl_colour colour,
                  sgl_vector direction,
                  sgl_vector position,
                         int concentration,
                    sgl_bool casts_shadows,
                    sgl_bool smooth_highlights);
```

Parameters

I	`name`	The name of the light to modify.
I	`colour`	As for Create Point Light.
I	`direction`	As for Create Point Light.
I	`position`	As for Create Point Light.
I	`concentration`	As for Create Point Light.
I	`casts_shadows`	As for Create Point Light.
I	`smooth_highlights`	As for Create Point Light.

Errors

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

10.2.8 Switch Light

Allows you to switch a light on or off and define whether or not it casts shadows and generates specular highlights. If a light did not have this facility enabled at the time it is encountered in the display list traversal, then this function is ignored.

Function Header

```
int sgl_switch_light( int name,
                      sgl_bool on,
                    sgl_colour casts_shadows,
                      sgl_bool smooth_highlights);
```

Parameters

I	`name`	The name of the light to be modified.
I	`on`	If non-zero, the light is switched on, otherwise the light is switched off.
I	`cast_shadows`	If non-zero, shadow casting is enabled, otherwise it is disabled. The facility to cast shadows must have been set with the relevant Create Light or Set Light routine.
I	`smooth_highlights`	If non-zero, smooth highlights enabled, otherwise they are disabled. The facility to smooth shade must have been set with the `sgl_create_light` or `sgl_set_light` routine.

Errors

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

11 CAMERAS

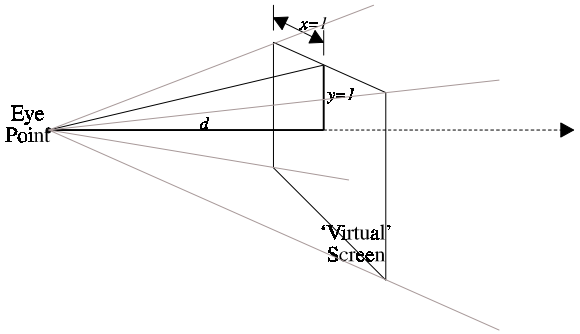
11.1 Introduction

Cameras are used to view a display list. You can place several cameras in a scene, and then choose one to perform a render. The image is produced from the camera's viewpoint and orientation. If a camera is placed in the same part of a display list as a moving object, it can automatically allow a view from that object.

PowerSGL currently only supports cameras that produce perspective projection.

11.1.1 Simple Perspective Camera Model

For the simple perspective camera model, the basic image control is the zoom factor. For a pinhole camera this is illustrated in the following diagram. (*Note:* the virtual screen corresponds to that described in the viewport section.)



$$zoom\_factor = \frac{d}{y}$$

The zoom factor gives the camera zoom. A large value gives a telescopic effect, while a small value gives a wide angle view. As a guide, the following conversions give approximate zoom factors for typical 35 mm camera lenses.

Focal length	15	24	35	50	75	100	200
Zoom factor	1.2	2	3	4	6	8	16

11.1.2 Foreground and Background Distances

Because it is impossible to render all objects simultaneously and perfectly, from microscopic objects near the camera to mountains in the distance, you must specify foreground and background distances. The foreground distance represents the closest distance at which objects are represented accurately, and the background distance is the furthest. Effectively both distances are planes, perpendicular to the camera viewing direction. PowerSGL performs clipping against these two planes.

The background distance defines the limit of the view. Anything beyond this distance is rejected from the render processing.

Both distances are specified in world or global coordinates, the foreground as a real number value, the background as an inverse so that it can be placed at infinity.

11.1.3 Camera Positioning

When a camera is added to a scene, the center of the virtual screen is positioned at the local coordinates origin, looking down the *z* axis. The virtual screen *x* and *y* axes are aligned with the local coordinates axes. Note that viewport and device *y* axes go down the screen, but the camera axes goes up.

The camera position and orientation are therefore determined like other items in the scene with the exception of scaling. Cameras are treated as points—the only effect scaling has on a camera is for translation. Stretching in *x,* for example, might move the camera, but does not cause a change in the size of the image. The exception to this rule is if a negative scale is used. In this situation the image appears flipped.

11.1.4 Default Cameras

For convenience in simple images, each display list has an implied default camera. Normally the render command takes a camera name as a parameter. If the name of a list is given instead, the default camera is used.

This default camera is located at the origin (in global coordinates), viewing down the *z* axis. The zoom factor is 4.0, which corresponds to the standard focal length lens on cameras.

11.2 Camera Routines

11.2.1 Create Camera

Creates a simple perspective camera at the end of the current list.

Function Header

```
int sgl_create_camera( float zoom_factor,
                       float foreground,
                       float inv_background);
```

Parameters

I `zoom_factor`	Specifies the camera zoom. This value should be positive, but PowerSGL clips values to lie in a reasonable range. Some typical values are given on page 11-1.
I `foreground`	The closest accuracy or clipping distance. If this value is less than a system-dependent minimum value, it is set to the legal minimum.
I `inv_background`	The inverse of the distance to the background. A value of 0.0 implies that the background is an infinite distance away.
`returned value`	This returns either a name for the camera, or an error value.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the camera has been ignored.
`sgl_err_no_name`	There were no available names so the camera information has been ignored.

11.2.2 Get Camera

Retrieves the parameters of a camera.

Function Header

```
int sgl_get_camera( int name,
                    float *zoom_factor,
                    float *foreground,
                    float *inv_background);
```

Parameters

I	`name`	The name of the camera to retrieve.
O	`zoom_factor`	The camera zoom.
O	`foreground`	The foreground distance.
O	`inv_background`	The inverse background distance.
	`returned value`	Error status.

Errors

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

11.2.3 Set Camera

Changes the parameters of a camera.

Function Header

```
void sgl_set_camera( int name,
                     float zoom_factor,
                     float foreground,
                     float inv_background);
```

Parameters

I	`name`	The name of the camera to modify.
I	`zoom_factor`	As for Create Camera.
I	`foreground`	As for Create Camera.
I	`inv_background`	As for Create Camera.

Errors

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

12 RENDERING AND SELECTION

12.1 Introduction

Rendering is the process which takes a display list and produces an image. Selection is a related operation which, when given a point on the image, returns the name of the object that is visible beneath that point.

12.2 Render Routines

12.2.1 Render Command

Produces an image in a named viewport (or device) from a named display list (or camera). If a camera is supplied as a parameter, then to get the list which is actually rendered, PowerSGL traverses up through the list's parents until the root list is reached. If a list is supplied, then that list is rendered using the default camera.

With a double buffered device, the rendering is done to the hidden display buffer—the `swap_buffer` parameter indicates whether to swap the hidden and displayed buffers when rendering is complete. The parameter is ignored for single buffered displays.

You can also supply a device name instead of the viewport. The rendering then uses the default viewport for the device.

Note that this routine may take some time to complete, but if there is hardware assistance, then the rendering may overlap with other operations.

Function Header

```
void sgl_render( int viewport_or_device,
                 int camera_or_list,
            sgl_bool swap_buffers);
```

Parameters

I	`viewport_or_device`	The name of a viewport or a device.
I	`camera_or_list`	The name of a camera or a display list.
I	`swap_buffers`	If non-zero, and the device is double buffered, then the displays are swapped at the end of rendering. The parameter is ignored for single buffered displays.

Errors

`sgl_err_bad_name`	`viewport_or_device` is not a valid viewport or device, or `camera_or_list` is not a valid camera or list.
`sgl_err_list_too_deep`	The maximum depth of lists was exceeded.

13 LEVEL OF DETAIL

13.1 Introduction

When an object appears small because it is a long way from the viewer, there is little point in trying to render it with vast numbers of polygons or complex shading, because the detail is lost, and processing time is wasted. Alternatively, when the object is close to the camera, a detailed model is probably required.

To make management of these models easier, PowerSGL provides a system to automatically choose between different model descriptions in scenes. You supply two diagonally opposite corners of a bounding box which contains the model, and up to three different model descriptions. PowerSGL projects the supplied bounding volume onto the screen and examines its width and height in pixels. It then compares the larger of the two with user-supplied pixel sizes, to determine which model to use.

The choice of model is based on the size and distance of the object, not its shadow: a distant object at a low level of detail with a light behind it could cast a large local shadow which would be constructed from the low level of detail.

The different models are treated as instances, i.e., built-in separate lists. This allows the more complex models to reference the simpler model if necessary. If more than three levels of detail are needed, then levels of detail can be nested. For example, the most complex model might in turn contain a level of detail structure for its fine details.

Instance substitutions (see page 3-4) also apply to the named lists supplied in the level of detail constructs.

13.2 Level of Detail Routines

13.2.1 Create Level of Detail Definition

Creates a level of detail definition at the end of the current list.

Function Header

```
int sgl_create_detail_levels( sgl_bool generate_name,
                              sgl_vector box_corner1,
                              sgl_vector box_corner2,
                                     int models[3],
                                     int change_sizes[2]);
```

Parameters

I `generate_name`	Specifies whether a name is required for this level of detail definition.
I `box_corner1,2`	Two points in local coordinates that define the bounding box. They should therefore be diagonally opposite corners, such as the minimum and maximum values for the three dimensional space of the box.
I `models`	Names of the three alternative models, with the most detailed model in index 0. A model list can be repeated, and the null list, `SGL_NULL_LIST`, can be used if a model must be emptied. `SGL_DEFAULT_LIST` must not be used.

I `change_sizes`	The two values are the threshold pixel lengths to compare with the size of the projected bounding sphere. If the projected width is smaller than the smallest of the two given values then the least detailed model (index 2) is used, while if it is larger than the largest of the two values, the most detailed (index 0) is used. The middle model is used for intermediate sizes.
`returned value`	If a name was requested this returns either a name for the level of detail definition, or an error value, else the value is undefined.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the level of detail information has been ignored.
`sgl_err_no_name`	The level of detail definition has not been created as there were no available names.
`sgl_err_bad_parameter`	Either there is no model list of the given name (`SGL_NULL_LIST` has been used for this model), or at least one of the `change_sizes` was either zero or negative and has been replaced by a value of 1.

13.2.2 Get Level of Detail Definition

Retrieves a copy of a level of detail definition.

Function Header

```
int sgl_get_detail_levels( int name,
                    sgl_vector box_corner1,
                    sgl_vector box_corner2,
                           int models[3],
                           int change_sizes[2]);
```

Parameters

I `name`	The name of the level of detail definition to retrieve.
O `box_corner1,2`	Two points defining the bounding box in local coordinates.
O `models`	Names of the three models.
O `change_sizes`	Threshold sizes, sorted with the smallest first.
`returned value`	Error status.

Errors

`sgl_err_bad_name`	There is no level of detail definition of the given name. The contents of the output parameters have not been changed.

13.2.3 Set Level of Detail Definition

This routine changes a level of detail definition.

Function Header

```
void sgl_set_detail_levels( int name,
                      sgl_vector box_corner1,
                      sgl_vector box_corner2,
                             int models[3],
                             int change_sizes[2]);
```

Parameters

I	`name`	The name of the level of detail definition to modify.
I	`box_corner1,2`	As for Get Level of Detail Definition.
I	`models`	As for Get Level of Detail Definition.
I	`change_sizes`	As for Get Level of Detail Definition (not necessarily sorted).

Errors

`sgl_err_bad_name`	There is no level of detail definition of the given name.
`sgl_err_bad_parameter`	Either there is no model list of the given name (`SGL_NULL_LIST` has been used for this model), or at least one of the `change_sizes` was either zero or negative and has been replaced by a value of 1.

14 COLLISION DETECTION AND POSITION FEEDBACK

14.1 Introduction

In an interactive 3D application, such as a flight simulator, it is often necessary to know if objects have collided or are about to collide and to take the appropriate action. To make this testing easier, PowerSGL enables you to define a number of points which, during the rendering traversal, can be tested against the convex polyhedral objects in the scene. Each point is then flagged with the first object that it is inside.

In the case of a flight simulator, a point might be placed just in front of the aircraft—if it goes inside a building, part of the terrain,or another aircraft, then it is about to crash. You can specify if certain objects are to be ignored for the purposes of collision detection—both as a requirement of the application (say to ignore clouds) and for efficiency, since performing this testing imposes an overhead on the system.

During display list traversal, the detection of collisions can be disabled and re-enabled when testing against particular objects is unnecessary.

The points can also be used for positional feedback. After a render command has been executed you can read a point’s position in world coordinates, regardless of its collision detection status.

Points are not rendered.

14.1.1 Definition and Positioning of Points

The creation, positioning and scope of collision detection points is similar to that of lights. Within a list, a point can be declared and then, optionally, positioned. It is then tested against the subsequent objects in the display list until a collision is detected or the scope of the list is ended. If a point is not placed, its position is set in the local coordinate system when the point is declared.

If a point is declared within an instance, and that instance is referenced many times within a display list, then the result (i.e., both position and collision detection) only applies to the last instance in the display list.

14.1.2 Maximum Number of Active Points

PowerSGL imposes a limit on the number of collision points that can be in action at a time—any other definitions are ignored. This limit is specified in the library as `SGL_MAX_ACTIVE_POINTS`. Points used only for position information are not included in this number, so there is no limit on the number of position points.

When the scope of a point ends, the number of active points is decremented and this facility becomes available for new points.

14.2 Collision Detection Routines

14.2.1 Create Point

Creates a point that you can use for position feedback, and optionally for collision detection, at the end of the current list.

If you are using the point for collision detection, you must set the appropriate flag in the create call (although this can be modified using the `sgl_set_point` routine, see page 14-2).

This routine declares the point. The actual position of the point is determined by the `sgl_set_point` routine.

Collision detection is only performed on objects that come after the points entry in the display list and only if those objects are within the scope of the point.

The return value is always needed. This is the name for the check point which is required to turn the point on (and off) and to get any information about the point.

Function Header

```
int sgl_create_point( sgl_vector offset,
                      sgl_bool collision_check);
```

Parameters

I `offset`	The offset from the local coordinate system origin to the point. If the point is explicitly positioned, then this offset is relative to the local coordinate system of the new position in the display list. Otherwise it uses the coordinate system at the point of creation.
I `collision_check`	Specifies whether collision detection is globally enabled for the point. This must be set to non-zero if any collision detection is to be performed.
`returned value`	The name of the created point, or an error.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the point information has been ignored.
`sgl_err_no_name`	There were no available names so the point information has been ignored.

14.2.2 Set Point

This routine allows the offset vector and `collision_check` flag for a point to be changed. The routine must be supplied with the name of a point, and the new values for `offset` and `collision_check`.

Function Header

```
void sgl_set_point( int point_name,
             sgl_vector offset,
               sgl_bool collision_check);
```

Parameters

I	`point_name`	The name of the point that is to be modified.
I	`offset`	As for Create Point.
I	`collision_check`	As for Create Point.

Errors

`sgl_err_bad_name`	An invalid point name was supplied.

14.2.3 Switching a Check Point On or Off

Allows you to temporarily disable or re-enable collision point checking. Only points which have the collision check flag set can be involved in collision checking (see `sgl_create_point` and `sgl_set_point` on pages 14-2 and 14-2).

You must supply the name of the point and the flag to start or stop checking. The setting for this point applies until it either explicitly changed, or the display list traversal exits the current scope.

The best usage is to determine the objects that you want to check the point against and surround only those with an on and off pair.

Function Header

```
void sgl_switch_point( int name,
                       sgl_bool enable_check);
```

Parameters

I	`name`	The name of the point.
I	`enable_check`	Sets collision checking until it is either re-set or the current display list scope is exited, and the checking returns to its previous setting.

Errors

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

14.2.4 Position Point

Fixes a given point relative to the current local coordinate system. Note that, as with the position light routine, only one position point can apply—if this routine is called many times, only the last one has any effect.

Function Header

```
void sgl_position_point( int name);
```

Parameters

I	`name`	The name of the point to position.

Errors

`sgl_err_no_mem`	The display list has become too large for the system so the position information has been ignored.
`sgl_err_bad_name`	There is no point of the given name.

14.2.5 Query Point

Returns the position of a given named point in world coordinates and, if required, information on its collision status for the most recently rendered frame.

The collision test determines whether the point was inside any objects that were in its scope in the display list. If so, it returns the details for the first object encountered that the point is inside. The details consist of the name of the object (if named), the plane equation of the closest surface to the point in world coordinates, and the names of the named lists all the way up to the root of the display list. This path information allows unique identification of the collisions with instanced objects.

The collision data is returned in a client-created `sgl_collision_data` structure. If collision information is required then a pointer to the structure is passed to the function, otherwise `NULL` is supplied.

Structure Header

```
typedef struct tag_collision_data
{
      sgl_bool collision;
           int object_name;
           int object_plane;
    sgl_vector normal;
         float d;
           int path_length;
           int path[SGL_MAX_PATH];
} sgl_collision_data;
```

Structure Parameters

0 `collision`	A non-zero value if the point is inside an object, otherwise zero.
0 `object_name`	If a collision occurred, and the object is named, this is the name of the object. If the object has no name, then the value returned is `SGL_ANON_OBJECT`.
0 `object_plane`	If a collision occurred this is the number of the object's plane that is closest to the point.
0 `normal, d`	These parameters specify the plane equation, in global coordinates, of the object's surface which is closest to the point. This is given by $$normal \cdot X = d$$ where X is any point on the plane.

O `path_length`	This gives the number of valid elements given in `path`.
O `path`	This is an array of the names of the named lists the renderer has traversed in reaching the collided object. The array is given in the order from most deeply-nested up to the top of the tree. If there are more than `SGL_MAX_PATH` named lists in the path, then only the most deeply nested lists are given.

Function Header

```
int sgl_query_point( int name,
                     sgl_vector point_pos,
           sgl_collision_data *collision_data);
```

Function Parameters

I `name`	The name of the point to check.
O `point_pos`	Returns the location of the point in world coordinates.
O `collision_data`	Returns collision information for the point. This is optional, depending on whether the point is being used for collision detection - use `NULL` for no collision data.
`returned value`	Error status.

Errors

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

15 LEVELS OF QUALITY AND SYSTEM DEFAULTS

15.1 Introduction

In any type of system, there is almost always a tradeoff between quality and speed, and rendering systems are no different. The more highly detailed an image, the longer it takes to render. To allow you to control the balance between image quality and rendering speed, PowerSGL provides switch routines that can enable or disable rendering features. These features include smooth shading and shadow generation.

The quality switch settings form part of the system state information, which may be saved when a child list is traversed during rendering. For example, if a child list contains some fine detail of a model, and it is not necessary for that detail to cast shadows, then shadow generation can be switched off for that section of the display list, and can automatically reset itself back to the original value.

At the beginning of a render, the system initializes each of the switches to a default, the values of which are covered in the sections below.

15.2 Shading Control

PowerSGL supports two modes of shading. These are:

- flat shading
- smooth diffuse shading

This can be changed throughout the display list. The initial state setting enables smooth shading on objects with normal information.

15.3 Shadow Controls

The generation of shadows can be enabled and disabled throughout the display list. The initial state setting enables the casting of shadows from convex objects and the lights which cast shadows.

15.4 Texturing Controls

Texturing can be enabled and disabled throughout the display list, with the initial state setting enabling texturing.

15.5 Fog Controls

Fog can be switched on or off particular objects throughout the display list, with the initial setting enabling fog.

15.6 Quality Setting Routines

15.6.1 Smooth Shading

Enables or disables smooth shading.

Function Header

```
void sgl_qual_smooth_shading( const sgl_bool enable );
```

Parameters

I `enable`	Enables smooth shading if non-zero.

Errors

`sgl_err_no_mem`	There is insufficient system memory.

15.6.2 Shadows

Enables or disables the generation of shadows from objects.

Function Header

```
void sgl_qual_generate_shadows( const sgl_bool enable );
```

Parameters

I `enable`	Enables generation of shadows if non-zero.

Errors

`sgl_err_no_mem`	There is insufficient system memory.

15.6.3 Texturing

Enables or disables texturing of objects.

Function Header

```
void sgl_qual_texturing( const sgl_bool enable );
```

Parameters

I `enable`	Enables texturing if non-zero.

Errors

`sgl_err_no_mem`	There is insufficient system memory.

15.6.4 Fog

Enables or disables fog on objects. Note that the overall density and colour cannot be altered unless the fog affects a group of objects.

Function Header

```
void sgl_qual_fog( const sgl_bool enable );
```

Parameters

I	`enable`	Enables fogging if non-zero.

Errors

16 GLOBAL EFFECTS

16.1 Introduction

Fogging is a function which mixes a fog color with the pixels of an object based on their distance from the camera. The background is what is seen when no user-defined objects cover a pixel.

16.2 Fogging

Fogging is a function which mixes a fog color with the pixels of an object based on their distance from the camera. It can be used to simulate not only fog, but distance haze, and dim lighting effects as objects gradually darken to black.

An approximation for the amount of fog at a particular distance is defined by the following function:

$$pixel\_colour = (1-k).\ fog\_colour + k.object\_colour$$

$$\text{where:} \quad k = (1-\ fog\_density)^{\text{distance}}$$

Distance is the distance from the camera in world coordinates.

The parameters for fogging are associated with cameras. This can be used for view-dependent fog values. For example, a camera which is inside a cloud could have higher fogging values.

The default fog is no fog at all, i.e., `fog_density = 0`.

Fog is a global function, and although you can disable the fogging for particular sets of objects using the quality settings, you cannot vary the fog density or color from object to object.

16.2.1 Set Fog Level

Defines the color of the fog and the rate at which it fades in.

Function Header

```
void sgl_set_fog( int camera_name,
           sgl_colour colour,
                float fog_density);
```

Parameters

I	`camera_name`	The camera to which the fogging function applies.
I	`colour`	The color of the fog.
I	`fog_density`	The rate at which fog becomes more dense, with values in the range [0.0,1.0]. A fog of 0.0 means no fog at all. A `fog_density` of 0.5 means that an object becomes 50% fogged in one world unit.

Errors

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

16.3 Background

The background is what is seen when no user-defined objects cover a pixel, and is placed at an infinite distance from the camera. The background is not affected by fog, but in many applications the two colors would be set the same.

As with the fog, the background color is associated with a camera.

The default background color is black.

16.3.1 Set Background Color

Defines the background color for a camera.

Function Header

```
void sgl_set_background_colour( int camera_name,
                                sgl_colour colour);
```

Parameters

I	`camera_name`	The camera to which the background color applies.
I	`colour`	The color of the background

Errors

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

17 POWERSGL DIRECT

17.1 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.

17.2 PowerSGL Functions used in PowerSGL Direct

On startup the application must call `sgl_create_screen_device` (see page 2-5) 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 17-2. 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.

17.3 Structures and the Material Flags Enumerated Type

17.3.1 Material Flags

Enumerated Type Header

```
typedef enum
{
    SGLTT_GOURAUD = 0x1,
    SGLTT_TEXTURE = 0x2,
    SGLTT_HIGHLIGHT = 0x4,
    SGLTT_MODULATE = 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,

} 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_MODULATE	This flag is currently ignored. It will be used to indicate whether or not you want to modulate (multiply) the texture pixel color with the vertex colors.
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 17-3).
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	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.
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.

17.3.2 Shadow/Light Volume Mode

Enumerated Type Header

```
typedef enum
{
    NO_SHADOWS_OR_LIGHTVOLS,
    ENABLE_SHADOWS,
    ENABLE_LIGHTVOLS
} SGLSHADOWTYPE;
```

17.3.3 Rendering Empty Regions

Enumerated Type Header

```
typedef enum
{
    ALWAYS_RENDER_ALL_REGIONS,
    DON'T_RENDER_EMPTY_REGIONS
} SGLRENDERREGIONS;
```

17.3.4 Context

Structure Header

```
typedef struct tagSGLCONTEXT
{
        sgl_bool bFogOn;
           float fFogR, fFogG, fFogB;
      sgl_uint32 u32FogDensity;
        sgl_bool bCullBackfacing;
      sgl_uint32 u32Flags;
             int nTextureName;
           float fTranslucentPassDepth;
        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  n32MipmapOffset;
SGLRENDERREGIONS RenderRegions;
      sgl_uint32 u32Reserve9[9];
} SGLCONTEXT, *PSGLCONTEXT;
```

Structure Parameters

I `bFogOn`	Set to non-zero to enable fog.
I `fFogR/G/B`	The fog color: 0 ≤ component ≤ 1.
I `u32FogDensity`	The fog density. The values range from 0 (hardly any fog) to 31 (very foggy). This only affects triangles that have been added with `bFogOn` set.
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 `fTranslucentPassDepth`	Describes the ordering of triangles within passes. With the current hardware, at each pixel PowerVR only displays the closest objects in each pass. Each call to `sgltri_triangles` puts those triangles into a different pass. `fTranslucencyPassDepth` orders these passes to get the correct result. The value is positive, and the larger the value, the closer the object.
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.

17.3.5 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.
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.

17.4 PowerSGL Direct Routines

17.4.1 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

17.4.2 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 17-8) 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 17-12) is called to render the complete scene.

The errors set by these functions may be read by `sgl_get_errors` (see page 1-4). 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.

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 17-8).
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 17-8).
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).

17.4.3 Add a Convex Shadow or Light Volume

This function is similar to `sgltri_triangles` (see page 17-8) 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 e`ShadowLightVolMode` 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 17-8).
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).

17.4.4 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

17.4.5 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

18 RANDOM NUMBER GENERATION

18.1 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.

18.2 Random Number Routines

18.2.1 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

18.2.2 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

19 VERSION INFORMATION

19.1 Introduction

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

19.1.1 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 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

20 EXAMPLES

20.1 Example 1 - Rotating Tube

This complete example shows how to set up a simple scene with a couple of light sources; a camera, and an object, which in this case is a hollow tube. First the device and viewport are defined, followed by a single camera which is put in as a separate list so that its local transform, which displaces it away from the origin does not affect other items on the scene. Lights are then added, followed by a named transform which is then used to rotate the tube. The tube is built up using simple non-textured planes with invisible planes to bound the ends of the tube.

```
/*****************************************************************************
 * Name    : tube.c
 * Title   : Simple Tube Demo using SGL
 *
 * Copyright : 1996 by VideoLogic Limited. All rights reserved.
 * : No part of this software, either material or conceptual
 * : may be copied or distributed, transmitted, transcribed,
 * : stored in a retrieval system or translated into any
 * : human or computer language in any form by any means,
 * : electronic, mechanical, manual or other-wise, or
 * : disclosed to third parties without the express written
 * : permission of VideoLogic Limited, Unit 8, HomePark
 * : Industrial Estate, King's Langley, Hertfordshire,
 * : WD4 8LZ, U.K.
 *
 * Description : Rotates a simple open ended tube using PowerSGL
 *
 * Platform : ANSI compatible
 *
*****************************************************************************/

#include "math.h"
#include "sgl.h"

#define NAMED_ITEM        TRUE
#define UNAMED_ITEM       FALSE

#define COLOUR            TRUE
#define GREY              FALSE

#define SHADOW_LIGHT      TRUE
#define NO_SHADOWS        FALSE

#define SMOOTH_SHADING    TRUE
#define NO_SMOOTH_SHADING FALSE

#define INVISIBLE         TRUE
#define VISIBLE           FALSE

#define LENS_50MM         4.0f
```

```
int  main()
{
  int frame;
  int tubeName;
  int logicalDevice;
  int viewport1;
  int transformName;
  int camera1;

  sgl_colour lightGreen = {0.6f,1.0f,0.6f};
  sgl_colour lightBlue  = {0.8f,0.8f,1.0f};
  sgl_colour darkGrey   = {0.25f,0.25f,0.25f};
  sgl_vector lightDir   = {1.0f,-1.0f,1.0f};
  sgl_vector lightPos   = {-100.0f,100.0f,-100.0f};
         int lightCon   = 32;

  sgl_vector topPnt     = {0.0f,25.0f,0.0f};
  sgl_vector topNorm    = {0.0f,1.0f,0.0f};
  sgl_vector botPnt     = {0.0f,-25.0f,0.0f};
  sgl_vector botNorm    = {0.0f,-1.0f,0.0f};

  sgl_vector sidePnt;
  sgl_vector sideNorm;

  float tubeRadius,tAngle,angleStep;
  float yAngle,yInc;

  /*  Create screen device 0 with 640x480 screen resolution and 24-bit
colour and no double buffering  */

  logicalDevice = sgl_create_screen_device ( 0,640,480,
sgl_device_24bit,FALSE)

  if ( logicalDevice<0)
  {
    printf( stderr,"ERROR %01d: Failed to Open Screen
Device\n",logicalDevice);
    exit(1);
  }

  /*  Set the viewport for the opened device so that there is a 20 pixel
border around the rendered image, and the image is fitted to this rectangle
*/

  /*  The following is the largest square centered in the viewport  */

  viewport1 = sgl_create_viewport( logicalDevice,20,20,620,460,
        100.0f,20.0f,540.0f,460.0f);

  /*  Put camera in a list so that its positioning transformation does not
affect the tubes position  */

  sgl_create_list( UNAMED_ITEM,TRUE,FALSE);

  sgl_create_transform( UNAMED_ITEM); /*  this could be omitted  */
  sgl_translate( 0.0f,0.0f,-200.0f);

  /*  Add Camera with the zoom factor of a 50mm lens, an aspect ratio of 1,
a front clipping plane at 10.0, and no background clipping plane  */

  camera1 = sgl_create_camera( LENS_50MM,1.0f,10.0f,0.0f);
```

```
  /*  Go back up the list  */

  sgl_to_parent();

  /*  Define background colour as light blue.  */

  sgl_set_background_colour( camera1,lightBlue);

  /*  Ambient colour is set to a grey slightly lighter than default. Note
that only the red component is relevant for grey lights. This is also set as
an absolute level  */

  sgl_create_ambient_light( UNAMED_ITEM,darkGrey,GREY,FALSE);

  /*  A single spotlight is added with a concentration of 32 and is
positioned to shine down from the top left onto the tube  */

  sgl_create_point_light( UNAMED_ITEM,GREY,lightDir,lightPos,lightCon,
SHADOW_LIGHT,NO_SMOOTH_SHADING);

  /*  Create a named transform which will be used later to rotate the tube
*/

  transformName = sgl_create_transform( NAMED_ITEM);

  /*  Create an unnamed transform then set it to translate and rotate about
the x-axis  */

  sgl_create_transform( UNAMED_ITEM); /*  this could be omitted  */
  sgl_translate( 0.0f,25.0f,0.0f);
  sgl_rotate( sgl_x_axis,( PI/2.0f));

  /*  Set tube material properties  */

  materialName = sgl_create_material( NAMED_ITEM);
  sgl_set_diffuse( lightGreen);
  sgl_set_specular( lightGreen,0) /*  turn specular reflectivity off  */

  /*  Build up tube by adding two invisible planes for the top and bottom
and 32 visible planes for the actual tube  */

  tubeName = sgl_create_convex( NAMED_ITEM);

  sgl_add_simple_plane( topPnt,topNorm,INVISIBLE);
  sgl_add_simple_plane( botPnt,botNorm,INVISIBLE);

  /*  Add the sides of the tube using 32 calls to sgl_add_plane  */

  tubeRadius = 10.0f;
  tAngle     = 0.0f;
  angleStep  = (2.0f*PI)/32.0f;

  for ( tubeSides=0;tubeSides<32;tubeSides++)
  {
    tAngle+=angleStep;

    sideNorm[0] = cos(tAngle);
    sideNorm[1] = 0.0f;
    sideNorm[2] = sin(tAngle);
    sidePnt[0]  = sideNorm[0]*tubeRadius;
    sidePnt[1]  = sideNorm[1]*tubeRadius;
    sidePnt[2]  = sideNorm[2]*tubeRadius;
```

```
    sgl_add_simple_plane( sidePnt,sideNorm,VISIBLE);
  }

  /*  Animate tube by rotating it about the global y-axis  */

  yAngle = 0.0f;
  yInc   = PI/75.0f;

  for (frame=0;frame<300;frame++)
  {

    sgl_modify_transform( transformName,TRUE);
    sgl_rotate( sgl_y_axis,yAngle);

    yAngle+ = yInc;

    /*  Render a single image using the defined camera1 into viewport1  */

    sgl_render( viewport1,camera1);
  }

  /*  Close down the display device in an orderly fashion  */

  sgl_delete_device(logicalDevice);

  exit(0);

}  /*  end of example 1  */
```

20.2 Example 2 - Tower

This example demonstrates shadows, more complex convex models, texturing, infinte planes to model the sky and ground, and use of translucency. It assumes only a single level of transparency is sufficient, and so does not use the `sgl_new_translucent`.

```
/*****************************************************************************
 * Name : textower.c
 * Title : tower example
 *
 * Copyright : 1996 by VideoLogic Limited. All rights reserved.
 * : No part of this software, either material or conceptual
 * : may be copied or distributed, transmitted, transcribed,
 * : stored in a retrieval system or translated into any
 * : human or computer language in any form by any means,
 * : electronic, mechanical, manual or other-wise, or
 * : disclosed to third parties without the express written
 * : permission of VideoLogic Limited, Unit 8, HomePark
 * : Industrial Estate, King's Langley, Hertfordshire,
 * : WD4 8LZ, U.K.
 *
 * Description : tower example using PowerSGL
 *
 * Platform : ANSI compatible
 *
*****************************************************************************/

#include <stdio.h>
#include <stdlib.h>
#include "math.h"

#include <windows.h>
#include <windowsx.h>
#include "ddraw.h"

#include "sgl.h"
#include "sglwin32.h"
#include "textower.h"

#pragma warning ( disable : 4244)
#pragma warning ( disable : 4056)

#define NAMED_ITEM        TRUE
#define UNAMED_ITEM       FALSE

#define COLOUR            TRUE
#define GREY              FALSE

#define SHADOW_LIGHT      TRUE
#define NO_SHADOWS        FALSE

#define SMOOTH_SHADING    TRUE
#define NO_SMOOTH_SHADING FALSE

#define INVISIBLE         TRUE
#define VISIBLE           FALSE

#define ON                TRUE
#define OFF               FALSE
```

```
#define LENS_200MM  16.0f
#define LENS_50MM    4.0f
#define LENS_35MM    3.0f
#define LENS_24MM    2.0f

#define PI     3.142f
#define TWOPI  ( 2.0f * PI)

void AddTower( );
void AddGenerator( );
void AddHead( );

int finished = 0;
int viewport1;
int logicalDevice;

int groundTex,pan1Tex,pan2Tex,pan3Tex,barbTex,glowyTex,badTex;
sgl_intermediate_map First,Second,Third,Fourth,Fifth,Sixth,Seventh;

int camTran,towTran,lightTran,lightMat;

static sgl_vector Light3Pos = { -40.0f,200.0f,-40.0f};
static sgl_vector Light3Dir = { -1.0f,-1.0f,-1.0f};


static sgl_colour White     = {0.8f,0.8f,0.8f};
static sgl_colour black     = {0.0f,0.0f,0.0f};
static sgl_colour Yellow    = {0.7f,0.7f,0.3f};
static sgl_colour SkyBlue   = {0.5f, 0.6f, 0.9f};
static sgl_colour fogWhite  = {1.0f,0.8f,0.8f};
static sgl_colour blue      = {0.2f,0.2f,1.0f};
static sgl_colour Red       = {0.8f,0.1f,0.1f};
static sgl_colour darkgrey  = {0.25f,0.25f,0.25f};

/*  Variables for named items  */

static int camera1,light1,pointLight;

static int frame;

sgl_2d_vec uv1,uv2,uv3;

static float camAngle,camOff;

char pszTmp [120];

int SetupScene ( HWND hWnd, int xsize, int ysize, int pixdepth)
{
  sgl_vector ParLightDir = {-1.0f,-1.0f,1.0f};

  sgl_vector groundPnt  = {0.0f,0.0f, 2000.0f };
  sgl_vector point2     = {-20.0f,0.0f, 2000.0f };
  sgl_vector point3     = {0.0f,0.0f, 2020.0f };

  float zoomFac         = 15.0f;
  float cam_rect_left   = 80.0f;
  float cam_rect_top    = 0.0f;
  float cam_rect_right  = 560.0f;
  float cam_rect_bottom = 480.0f;

  int nSGLPixSize;
```

```
  /*  Create screen device 0 with 640x480 screen resolution and 24 bit
colour and no double buffering  */

  if ( pixdepth > 16)
  {
    nSGLPixSize = sgl_device_24bit;
  }
  else
  {
    nSGLPixSize = sgl_device_16bit;
  }

  logicalDevice = sgl_create_screen_device ( 0,xsize,( ysize-1),
nSGLPixSize,FALSE);

  if ( logicalDevice<0)
  {
    sprintf( pszTmp,"ERROR %01d: Failed to Open Screen
Device\n",logicalDevice);
    OutputDebugString( pszTmp);
    return ERR_CREATE_SCREEN_DEVICE;
  }

  /*  Set the viewport for the opened device  */
   switch ( ysize)
  {
    /*  800x600  */
    case 600:
    {
      cam_rect_left   = 80.0f;
      cam_rect_top    = 0.0f;
      cam_rect_right  = 560.0f;
      cam_rect_bottom = 480.0f;
      break;
    }

    /*  640x480  */
    case 480:
    {
      cam_rect_left   = 80.0f;
      cam_rect_top    = 0.0f;
      cam_rect_right  = 560.0f;
      cam_rect_bottom = 480.0f;
      break;
    }

    /*  600x400  */
    case 400:
    {
      cam_rect_left   = 80.0f;
      cam_rect_top    = 0.0f;
      cam_rect_right  = 560.0f;
      cam_rect_bottom = 480.0f;
      break;
    }
```

```
  /*  320x240  */
    case 240:
    {
      cam_rect_left   = 100.0f;
      cam_rect_top    = 100.0f;
      cam_rect_right  = 300.0f;
      cam_rect_bottom = 300.0f;
      break;
    }
  }

  viewport1 = sgl_create_viewport ( logicalDevice,0,0,xsize,(
ysize-1),cam_rect_left, cam_rect_top, cam_rect_right, cam_rect_bottom);

     if ( viewport1<0)
  {
    sprintf( pszTmp,"ERROR %01d: Failed to Create Viewport\n",viewport1);
    OutputDebugString( pszTmp);
    return ERR_CREATE_VIEWPORT;
  }

  /*  Load up and assign names to textures  */

  First   = ConvertBMPtoSGL( "grass2.bmp", FALSE);
  Second  = ConvertBMPtoSGL( "badblk2.bmp",TRUE);
  Third   = ConvertBMPtoSGL( "pan1.bmp",   FALSE);
  Fourth  = ConvertBMPtoSGL( "pan2.bmp",   FALSE);
  Fifth   = ConvertBMPtoSGL( "pan3.bmp",   FALSE);
  Sixth   = ConvertBMPtoSGL( "barbie.bmp", TRUE);
  Seventh = ConvertBMPtoSGL( "glowy.bmp",  FALSE);

  groundTex = sgl_create_texture( sgl_map_16bit_mm, sgl_map_256x256,
              TRUE, TRUE, &First,  NULL);
  badTex    = sgl_create_texture( sgl_map_trans16, sgl_map_128x128,
              FALSE, FALSE, &Second, NULL);
  pan1Tex   = sgl_create_texture( sgl_map_16bit_mm, sgl_map_128x128,
              TRUE, TRUE, &Third, NULL);
  pan2Tex   = sgl_create_texture( sgl_map_16bit_mm, sgl_map_128x128,
              TRUE, TRUE, &Fifth, NULL);
  pan3Tex   = sgl_create_texture( sgl_map_16bit_mm, sgl_map_128x128,
              TRUE, TRUE, &Fourth, NULL);
  barbTex   = sgl_create_texture( sgl_map_trans16_mm, sgl_map_128x128,
              TRUE, TRUE, &Sixth, NULL);
  glowyTex  = sgl_create_texture( sgl_map_16bit_mm, sgl_map_128x128,
              TRUE, TRUE, &Seventh, NULL);

  /*  Create lights  */

  sgl_create_ambient_light( UNAMED_ITEM,darkgrey,FALSE);


  light1 = sgl_create_parallel_light( NAMED_ITEM,darkgrey,
            ParLightDir,NO_SHADOWS,FALSE);


  pointLight = sgl_create_point_light( NAMED_ITEM, White,
          Light3Dir,Light3Pos,0,SHADOW_LIGHT,FALSE);
```

```
    /*  Set up camera  */


    sgl_create_list( UNAMED_ITEM,TRUE,FALSE);

      /*  The following transform is used to animate to camera  */

      camTran = sgl_create_transform( TRUE);

      sgl_create_transform( FALSE);   /*  We need to create a new transform to
store the following transformation  */
      /*  Orient camera  */

      sgl_rotate( sgl_y_axis,-0.05f);
      sgl_rotate( sgl_x_axis,-0.24f);

      #if 0

        camera1 = sgl_create_camera( LENS_24MM,10.0f,0.0f);

      #else

        camera1 = sgl_create_camera( LENS_35MM,10.0f,0.0f);

      #endif

    sgl_to_parent( );

    /*  set the fog and background  */

    sgl_set_fog( camera1,fogWhite,0.0005f);

    sgl_set_background_colour( camera1,fogWhite);


    /*  Add the ground plane  */

    sgl_create_list( UNAMED_ITEM,TRUE,FALSE);

      uv1[0] = 0.0f;
      uv1[1] = 0.0f;
      uv2[0] = 0.04f;
      uv2[1] = 0.0f;
      uv3[0] = 0.0f;
      uv3[1] = 0.04f;

      sgl_set_diffuse( White);

      sgl_set_texture_effect( TRUE,TRUE,TRUE,FALSE);
      sgl_set_texture_map( groundTex,FALSE,FALSE);
      sgl_add_plane( groundPnt,point2,point3, VISIBLE,
             NULL,NULL,NULL,uv1,uv2,uv3);
    sgl_to_parent( );
```

```
    /*  Add the hologram generator model  */

    /*  Switch off shadow generation for the hologram generator object  */

    sgl_switch_light( pointLight,TRUE,NO_SHADOWS,FALSE);

    AddGenerator( );

    sgl_switch_light( pointLight,TRUE,SHADOW_LIGHT,FALSE);


    /*  Add the tower model  */

       sgl_create_list( UNAMED_ITEM,TRUE,FALSE);

      sgl_translate( -60.0f,0.0f,60.0f);
         AddTower( );

    sgl_to_parent( );


    /*  Add light ball model  */

      sgl_create_list( UNAMED_ITEM,TRUE,FALSE);

      lightMat  = sgl_create_material( NAMED_ITEM,TRUE);
      lightTran = sgl_create_transform( TRUE);

      sgl_create_sphere( 0,5.0f,2,FALSE,FALSE,FALSE);

    sgl_to_parent( );


    #if 1

    camAngle = 60.0f;
    camOff   = -40.0f;

    #else

    camAngle = 0.0f;
    camOff   = -700.0f;

    #endif

      frame = 0;

    return 0;
}



void NextFrame( )
{
    /*  Animate Scene  */

    if ( frame>1500)
    {
      frame    = 0;
      camAngle = 60.0f;
      camOff   = -40.0f;
    }
```

```
  Light3Pos[1] = 600.0f*fabs( sin( camAngle*8.0f))+5.0f;

  /*  Make the red point light throb  */

  sgl_modify_material( lightMat,TRUE);
  sgl_set_glow( Red);

  Red[0] = ( 0.9f*fabs( sin( camAngle*20.0f)))+0.1f;
  Red[1] = ( 0.9f*fabs( sin( camAngle*6.0f)))+0.1f;
  Red[2] = ( 0.9f*fabs( sin( camAngle*12.0f)))+0.1f;

  sgl_set_point_light( pointLight,Red,Light3Dir,Light3Pos,0,
            SHADOW_LIGHT,FALSE);

  sgl_modify_transform( lightTran,TRUE);
  sgl_translate( Light3Pos[0],Light3Pos[1],Light3Pos[2]);


  /*  Animate the camera  */

  camAngle+ = 0.008f;
  camOff-   = 0.50f;

  sgl_modify_transform( camTran,TRUE);
  sgl_rotate( sgl_y_axis,camAngle);
  sgl_rotate( sgl_x_axis,0.75f);
  sgl_translate( 0.0f,0.0f,camOff*( ( float)frame/500.0f+1.0f));


  /*  Animate the revolving tower top  */

  sgl_modify_transform( towTran,TRUE);
  sgl_rotate( sgl_y_axis,-camAngle*4.0f);

  /*  Render image seen from camera2 to viewport 1  */

     sgl_render( viewport1,camera1,TRUE);

  frame++;

}


void Finish( )
{
  if( !finished)
  {
    /*  Close down the display device in an orderly fashion  */

    FreeAllBMPTextures( );

    sgl_delete_device( logicalDevice);
    finished=1;
  }

}
```

```
/*************************************************************************
 * Function Name  : sgl_cross_prod
 * Inputs         : Vec1 ,Vec2
 * Outputs        :
 * Input/Output   : Vec3
 * Returns        :
 * Global Used    : None
 * Description    : returns cross product of two given vectors in Vec3
 *
 *          NOTE: The result MUST NOT be the same as either of
 *          the input vectors
 *
*************************************************************************/


void sgl_cross_prod( const sgl_vector Vec1,
                     const sgl_vector Vec2,
                           sgl_vector Vec3)
{
  Vec3[0] = ( Vec1[1] * Vec2[2]) - ( Vec1[2] * Vec2[1]);
  Vec3[1] = ( Vec1[2] * Vec2[0]) - ( Vec1[0] * Vec2[2]);
  Vec3[2] = ( Vec1[0] * Vec2[1]) - ( Vec1[1] * Vec2[0]);
}



/*************************************************************************
 * Function Name  : sgl_vec_sub
 * Inputs         : Vec1 ,Vec2
 * Outputs        :
 * Input/Output    : Vec3
 * Returns        :
 * Global Used    : None
 * Description    : Does Vec1 - Vec2 putting result in Vec3
 *
*************************************************************************/

void  sgl_vec_sub( const sgl_vector Vec1,
                   const sgl_vector Vec2,
                         sgl_vector Result)
{
  Result[0] = Vec1[0] - Vec2[0];
  Result[1] = Vec1[1] - Vec2[1];
  Result[2] = Vec1[2] - Vec2[2];
}
```

```
/*****************************************************************************
 * Function Name  : sgl_simple_plane2
 * Inputs         : a, b, c - plane vertices
 *        visible - plane visibility
 * Outputs        : None
 * Input/Output   : None
 * Returns        : None
 * Global Used    :
 * Description    : Adds a simple plane, specified as three vertices
 *
*****************************************************************************/

void sgl_simple_plane2( sgl_vector a,
                        sgl_vector b,
                        sgl_vector c,
                           sgl_bool visible)
{
  sgl_vector ab,ac;

  float norm[3];

  sgl_vec_sub( a,b,ab);
  sgl_vec_sub( a,c,ac);

  sgl_cross_prod( ab,ac,norm);

  sgl_add_simple_plane( a,norm,visible);
}




/*****************************************************************************
 * Function Name  : sgl_texture_plane
 * Inputs         : texName
 *        a, b, c - plane vertices
 *        visible - plane visibility
 *
 *      xScale, yScale - texture scaling
 * Outputs        : None
 * Input/Output   : None
 * Returns        : None
 * Global Used    :
 * Description    : Adds a simple plane, specified as three vertices
 *
*****************************************************************************/

void  sgl_texture_plane( int texName,
                         float a[3],
                         float b[3],
                         float c[3],
                           int visible,
                         float xScale,
                         float yScale,
                      sgl_bool FirstPlane)
{
  static int LastTextureName = -1;
  sgl_2d_vec uv1,uv2,uv3;

  uv1[0] = 0.0f;
  uv1[1] = 0.0f;
```

```
  uv2[0] = xScale;
  uv2[1] = 0.0f;
  uv3[0] = 0.0f;
  uv3[1] = yScale;

  /*  create a new local material if a new texture is specified  */

  if ( ( ( LastTextureName==-1) || ( texName!=LastTextureName)) ||
FirstPlane)
  {
    sgl_create_material( UNAMED_ITEM,TRUE);
    sgl_set_texture_map( texName,FALSE,FALSE);
  }

     sgl_add_plane( a,b,c,visible,NULL,NULL,NULL,uv1,uv2,uv3);

  LastTextureName = texName;
}




/*************************************************************************
 * Function Name  : RevolveSection
 * Inputs         : Vertices - array of 2d vertices, these must form a
 *                  closed convex polygon
 *                  NumVertices - number of vertices
 *                  NumSections - number of elements in full object
 *                  SectionStep - Number of section steps per iteration.
 *                  If this is one, all section will have a convex
 *                  primitive, otherwise every n sections will have one.
 * Outputs        : None
 * Input/Output   : None
 * Returns        : None
 * Global Used    :
 * Description    : Rotates a closed convex polygon about the y axis
 *                  forming a 3d object.
 *
*************************************************************************/

void RevolveSection( sgl_2d_vec Vertices[],
                            int NumVertices,
                            int NumSections,
                            int SectionStep,
                            int textureName)
{
       float AngleStep,
             CurAngle,
             SecAngle;
         int CurVertice;
         int CurSection;
  sgl_vector p1,p2,p3,p4,p5,p6;

  AngleStep = TWOPI/( ( float) NumSections);
  CurAngle  = 0.0f;
```

```
  for( CurSection = 0; CurSection < NumSections; CurSection++)
  {
    /*  CurAngle and SecAngle define the two boundaries of the section  */

       SecAngle = CurAngle+AngleStep;

    /*  Step through the vertices, adding bounding planes as we go  */

    for ( CurVertice=0; CurVertice < ( NumVertices - 1); CurVertice++)
    {
      p1[0] = Vertices[CurVertice][0]*cos( CurAngle);
      p1[1] = Vertices[CurVertice][1];
      p1[2] = Vertices[CurVertice][0]*sin( CurAngle);

      p2[0] = Vertices[CurVertice][0]*cos( SecAngle);
      p2[1] = Vertices[CurVertice][1];
      p2[2] = Vertices[CurVertice][0]*sin( SecAngle);

      p3[0] = Vertices[CurVertice+1][0]*cos( CurAngle);
      p3[1] = Vertices[CurVertice+1][1];
      p3[2] = Vertices[CurVertice+1][0]*sin( CurAngle);

      p4[0] = Vertices[CurVertice+1][0]*cos( SecAngle);
      p4[1] = Vertices[CurVertice+1][1];
      p4[2] = Vertices[CurVertice+1][0]*sin( SecAngle);

      /*  Add the two bounding planes at first iteration  */

            if ( CurVertice==0)
           {
        p5[0] = Vertices[NumVertices-2][0]*cos( CurAngle);
        p5[1] = Vertices[NumVertices-2][1];
        p5[2] = Vertices[NumVertices-2][0]*sin( CurAngle);

        p6[0] = Vertices[NumVertices-2][0]*cos( SecAngle);
        p6[1] = Vertices[NumVertices-2][1];
        p6[2] = Vertices[NumVertices-2][0]*sin( SecAngle);

        /*  Must create a new polyhedron  */

        sgl_create_convex( UNAMED_ITEM);

        /*  sgl_simple_plane2 defines a plane using three point as opposed
to a point and a vector as in sgl_simple_plane  */

        sgl_texture_plane( textureName,p1,p5,p3,VISIBLE, 1.0f,
                    1.0f,TRUE);
        sgl_texture_plane( textureName,p2,p4,p6,VISIBLE, 1.0f,
                    1.0f,FALSE);

            }

      /*  add outline plane  */

      sgl_texture_plane( textureName,p1,p3,p2,VISIBLE,1.0f,1.0f,FALSE);

    }

    CurAngle+=( AngleStep*( ( float)SectionStep));
      }

}
```

```
/*************************************************************************
 * Function Name  : Tube
 * Inputs         : Vertices -  two 2d vertices
 *                  NumVertices - number of vertices
 *                  NumSides - tubes number of sides
 *                  Hollow - if hollow is true invisible planes are used for
 *                  top and bottom
 * Outputs        : None
 * Input/Output   : None
 * Returns        : None
 * Global Used    :
 * Description    : Forms a very simple surface of revolution using two
 *                  given 2d vertices
 *
*************************************************************************/

void  Tube( sgl_2d_vec Vertices[],
                   int NumVertices,
                   int NumSides,
                   int textureName,
              sgl_bool Hollow)
{

       float AngleStep, SecAngle;
       float CurAngle = 0.0f;
         int CurSide  = 0;
  sgl_vector Base;
  sgl_vector Top;
  sgl_vector p1, p2, p3, p4;
    sgl_bool First;


  /*  AngleStep is the angle of rotation per iteration  */

  AngleStep = TWOPI/NumSides;
   SecAngle = CurAngle+AngleStep;
      First = TRUE;

  /*  Loop through the sides of the object  */

  for ( CurSide=0; CurSide<NumSides; CurSide++)
  {
    p1[0] = Vertices[0][0]*cos( CurAngle);
    p1[1] = Vertices[0][1];
    p1[2] = Vertices[0][0]*sin( CurAngle);

    p2[0] = Vertices[0][0]*cos( SecAngle);
    p2[1] = Vertices[0][1];
    p2[2] = Vertices[0][0]*sin( SecAngle);

    p3[0] = Vertices[1][0]*cos( CurAngle);
    p3[1] = Vertices[1][1];
    p3[2] = Vertices[1][0]*sin( CurAngle);
```

```
    /*  For first iteration check to see if we need to add bounding plane at
top and bottom  */

    if ( CurSide == 0)
    {
      p4[0] = Vertices[1][0]*cos( SecAngle);
      p4[1] = Vertices[1][1];
      p4[2] = Vertices[1][0]*sin( SecAngle);

      Base[0] = 0.0f;
      Base[1] = Vertices[0][1];
      Base[2] = 0.0f;

      Top[0] = 0.0f;
      Top[1] = Vertices[1][1];
      Top[2] = 0.0f;

      /*  If the x component for either vertex isnt zero, then add a
bounding plane  */

      if ( Vertices[1][0] != 0.0f)
      {
        sgl_create_convex( UNAMED_ITEM);
        First = FALSE;
        sgl_texture_plane( textureName,p3,Top,p4,Hollow, 1.0f, 1.0f,
                    TRUE);
      }

      if ( Vertices[0][0] != 0.0f)
      {
        if ( First)
        {
          sgl_create_convex( UNAMED_ITEM);
          sgl_texture_plane( textureName,Base,p1,p2,Hollow,1.0f,
                    1.0f, TRUE);
          First = FALSE;
        }
        else
          sgl_texture_plane( textureName,Base,p1,p2,Hollow,1.0f,
                    1.0f,FALSE);
      }
    }

       if ( First)
       {
         sgl_create_convex( UNAMED_ITEM);
         sgl_texture_plane( textureName,p1,p3,p2,VISIBLE,1.0f,1.0f,TRUE);
         First = FALSE;
       }
    else
         sgl_texture_plane( textureName,p1,p3,p2,VISIBLE,1.0f,1.0f,FALSE);

    CurAngle = SecAngle;
    SecAngle += AngleStep;

  }
}
```

```
/*************************************************************************
 * Function Name    : AddTower
 * Inputs           : None
 * Outputs          : None
 * Input/Output     : None
 * Returns          : None
 * Global Used      :
 * Description      : Constructs Model of Tower
 *
*************************************************************************/

void AddTower( )
{


  /*  Data for the tower  */

  sgl_2d_vec VoutlineTop[6] = { {75.0f,0.0f},
                                {72.0f,2.0f},
                                {60.0f,5.0f},
                                {55.0f,5.0f},
                                {50.0f,0.0f},
                                {75.0f,0.0f}};

  sgl_2d_vec VoutlineCen[5] = { {73.0f,0.0f},
                                {73.0f,5.0f},
                                {50.0f,5.0f},
                                {50.0f,0.0f},
                                {73.0f,0.0f}};

  sgl_2d_vec Frame1[5]      = { {65.0f,0.0f},
                                {50.0f,10.0f},
                                {45.0f,5.0f},
                                {50.0f,0.0f},
                                {65.0f,0.0f}};

  sgl_2d_vec Frame2[5]      = { {5.0f,25.0f},
                                {5.0f,20.0f},
                                {45.0f,5.0f},
                                {50.0f,10.0f},
                                {5.0f,25.0f}};

  sgl_2d_vec Tow1[2]        = { {30.0f,0.0f},
                                {25.0f,50.0f}};

  sgl_2d_vec Tow2[2]        = { {25.0f,50.0f},
                                {15.0f,250.0f}};

  sgl_2d_vec Tow3[2]        = { {15.0f,250.0f},
                                {12.5f,300.0f}};

  sgl_2d_vec Tow4[2]        = { {12.5f,300.0f},
                                {5.0f,360.0f}};
```

```
  /*  Add the tower  */
  sgl_create_list( UNAMED_ITEM,TRUE,FALSE);
  sgl_set_diffuse( White);
  sgl_set_specular( SkyBlue,1);
  sgl_scale( 1.4f,1.45f,1.4f);
    /*  Add main central part of the tower  */
    /*  NOTE: it would be more plane efficient with a routine that took a
convex arc of points in this particular case  */
      Tube( Tow1,2,10,pan2Tex,FALSE);
      Tube( Tow2,2,10,pan2Tex,FALSE);
      Tube( Tow3,2,10,pan2Tex,FALSE);
      Tube( Tow4,2,10,pan2Tex,FALSE);

    /*  Translate to rotating part of tower  */
    sgl_translate( 0.0f,280.0f,0.0f);
    /*  A named transform is used to rotate the tower top  */
    towTran = sgl_create_transform( TRUE);
     /*  NOTE: TRUE for preserve state stores a copy of the current state
which is what becomes the state after sgl_to_parent  */
    sgl_create_list( UNAMED_ITEM,TRUE,FALSE);
      /*  sgl_y_axis is a predefined axis type - 0.0f,1.0f,0.0f  */
      /*  Add central dougnut shaped observation area  */
      RevolveSection( VoutlineTop,6,12,1,pan1Tex);
       /*  Add the top struts  */
             RevolveSection( Frame2,5,24,3,pan1Tex);

      /*  Disable shadowing for some parts of the tower to speed up
rendering  */
      sgl_switch_light( pointLight,TRUE,NO_SHADOWS,FALSE);
           RevolveSection( Frame1,5,24,3,pan1Tex);
           sgl_translate( 0.0f,-5.0f,0.0f);
           sgl_set_glow( Yellow);
           sgl_set_diffuse( black);
      RevolveSection( VoutlineCen,5,12,1,pan1Tex);
```

```
       /*  A scale with a -1 causes a relection about the axis  */

          sgl_scale( 1.0f,-1.0f,1.0f);
          sgl_set_glow( black);
          sgl_set_diffuse( White);

       RevolveSection( VoutlineTop,6,12,1,pan1Tex);

        /*  Add the bottom strutts  */

          RevolveSection( Frame1,5,24,3,pan1Tex);
          RevolveSection( Frame2,5,24,3,pan1Tex);

       sgl_switch_light( pointLight,TRUE,NO_SHADOWS,FALSE);

    sgl_to_parent( );

    sgl_to_parent( );

}



/*************************************************************************
 * Function Name    : AddGenerator
 * Inputs           : None
 * Outputs          : None
 * Input/Output     : None
 * Returns          : None
 * Global Used      :
 * Description      : Constructs Model of Generator
 *
*************************************************************************/


void AddGenerator( )
{

  sgl_2d_vec Gen1[2]  = {  {35.0f,0.0f},
                           {35.0f,35.0f}};

  sgl_2d_vec Gen2[2]  = {  {30.0f,35.0f},
                           {30.0f,135.0f}};

  sgl_2d_vec Gen3[2]  = {  {35.0f,135.0f},
                           {35.0f,170.0f}};

  sgl_2d_vec Fence[2] = {  {150.0f,5.0f},
                           {150.0f,30.0f}};

  sgl_2d_vec Topoutline[6] = { {152.5f,0.0f},
                               {152.5f,5.0f},
                               {147.5f,5.0f},
                               {147.5f,0.0f},
                               {152.5f,0.0f}};


    sgl_create_list( UNAMED_ITEM,TRUE,FALSE);

    sgl_scale( 1.8f,0.9f,0.9f);
    sgl_translate( 50.0f,0.0f,0.0f);
```

```
    sgl_set_diffuse( White);
    sgl_set_specular( SkyBlue,1);
       Tube( Gen1,2,6,pan1Tex,FALSE);

    /*  Set global translucency value  */
    sgl_set_opacity( 0.45f);
       Tube( Gen2,2,12,glowyTex,FALSE);
    /*  Set global translucency back to fully opaque  */

    sgl_set_opacity( 1.0f);
       Tube( Gen3,2,6,pan1Tex,FALSE);

    /*  Add translucent textured fence  */
       Tube( Fence,2,18,barbTex,TRUE);
       RevolveSection( Topoutline,5,18,1,pan1Tex);
    sgl_translate( 0.0f,30.0f,0.0f);
    sgl_switch_light( pointLight,TRUE,SHADOW_LIGHT,FALSE);
       RevolveSection( Topoutline,5,18,1,pan1Tex);
    sgl_switch_light( pointLight,TRUE,NO_SHADOWS,FALSE);

       sgl_create_list( UNAMED_ITEM,TRUE,FALSE);
       sgl_translate( 0.0f,190.0f,0.0f);
       sgl_rotate( sgl_x_axis,PI/2.0f);
       AddHead( );
       sgl_to_parent( );
    sgl_to_parent( );
}
```

```
/*************************************************************************
 * Function Name    : AddHead
 * Inputs           : None
 * Outputs          : None
 * Input/Output     : None
 * Returns          : None
 * Global Used      :
 * Description      : Adds a translucent textured map of a head, bounded
 *                    by invisible planes.
 *
*************************************************************************/


#define BOXSIZE  ( 48.0f)

void AddHead( )
{

  sgl_vector sidePnt;
  sgl_vector sideNorm;

  sgl_vector pnt2;
  sgl_vector pnt3;

  /*  Build up a cube with 5 simple planes, and one textured plane  */

     sgl_create_convex( UNAMED_ITEM);

  /*  Add plane with head texture  */

  sgl_create_material( FALSE,TRUE);   /*  TRUE makes this a local material
*/

  sgl_set_texture_map( badTex,FALSE,FALSE);

  sideNorm[0] = 0.0f;
  sideNorm[1] = -1.0f;
  sideNorm[2] = 0.0f;

  sidePnt[0]  = -BOXSIZE;
  sidePnt[1]  = -1.0f;
  sidePnt[2]  = -BOXSIZE;

  uv1[0]  = 0.0f;
  uv1[1]  = 0.0f;

  pnt2[0] = BOXSIZE;
  pnt2[1] = -1.0f;
  pnt2[2] = -BOXSIZE;

  uv2[0]  = 0.0f;
  uv2[1]  = 1.0f;

  pnt3[0] = -BOXSIZE;
  pnt3[1] = -1.0f;
  pnt3[2] = BOXSIZE;

  uv3[0]  = 1.0f;
  uv3[1]  = 0.0f;

  sgl_add_plane( sidePnt,pnt2,pnt3,VISIBLE,NULL,NULL,NULL,uv1,uv2,uv3);
```

```
    /*  Add bounding invisible planes  */

    sideNorm[0] = 1.0f;
    sideNorm[1] = 0.0f;
    sideNorm[2] = 0.0f;
    sidePnt[0]  = sideNorm[0]*BOXSIZE;
    sidePnt[1]  = sideNorm[1]*1.0f;
    sidePnt[2]  = sideNorm[2]*BOXSIZE;

    sgl_add_simple_plane( sidePnt,sideNorm,INVISIBLE);

    sideNorm[0] = 0.0f;
    sideNorm[1] = 0.0f;
    sideNorm[2] = 1.0f;
    sidePnt[0]  = sideNorm[0]*BOXSIZE;
    sidePnt[1]  = sideNorm[1]*1.0f;
    sidePnt[2]  = sideNorm[2]*BOXSIZE;

    sgl_add_simple_plane( sidePnt,sideNorm,INVISIBLE);

    sideNorm[0] = -1.0f;
    sideNorm[1] = 0.0f;
    sideNorm[2] = 0.0f;
    sidePnt[0]  = sideNorm[0]*BOXSIZE;
    sidePnt[1]  = sideNorm[1]*1.0f;
    sidePnt[2]  = sideNorm[2]*BOXSIZE;

    sgl_add_simple_plane( sidePnt,sideNorm,INVISIBLE);

    sideNorm[0] = 0.0f;
    sideNorm[1] = 0.0f;
    sideNorm[2] = -1.0f;
    sidePnt[0]  = -BOXSIZE;
    sidePnt[1]  = -1.0f;
    sidePnt[2]  = -BOXSIZE;

    sgl_add_simple_plane( sidePnt,sideNorm,INVISIBLE);

    sideNorm[0] = 0.0f;
    sideNorm[1] = 1.0f;
    sideNorm[2] = 0.0f;
    sidePnt[0]  = sideNorm[0]*BOXSIZE;
    sidePnt[1]  = sideNorm[1]*1.0f;
    sidePnt[2]  = sideNorm[2]*BOXSIZE;

    sgl_add_simple_plane( sidePnt,sideNorm,INVISIBLE);

}


/*  -------------------------- End of File ------------------------------
*/
```

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.

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 Windows 95 Extensions

Introduction

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

PowerSGL's DirectDraw mode allows the application to leave all of the buffer management to PowerSGL. 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 creates and controls the DirectDraw objects. Other display modes are supported depending on the amount of graphics memory.

PowerSGL's low-level address mode allows the application to create and control the display buffers. PowerSGL 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 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 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 sets to point to the status DWORD.

Errors

None

Notes

This must be called before `sgl_create_screen_device` (see page 2-5).

The application must provide the following information to PowerSGL 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 into DirectDraw mode,i.e., PowerSGL 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 2-5).

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 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 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 12-1) when PowerSGL 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 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 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 intermediate map format (see section 9-3).

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( );
```

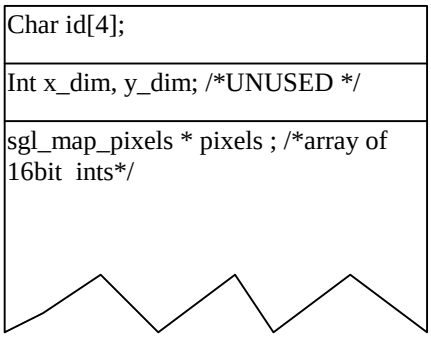
Removes all textures that have been loaded with `LoadBMPTexture`.

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 early. 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'`

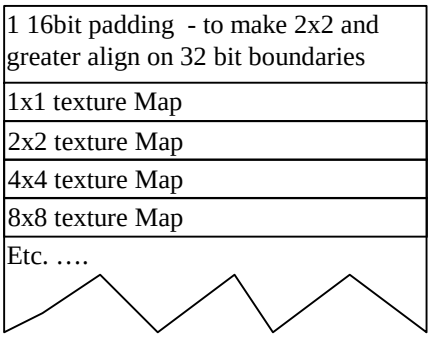
`id[2] == map_type` (as specified when preprocessed texture was created)

`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

Q

R

S

T

U

V

W

X

Y

Z