Glide 3.0 Programming Guide

Programming the 3Dfx Interactive Glide™ Rasterization Library 3.0

Document Release 021
June 1998

3Dfx Interactive, Inc.
4435 Fortran Avenue
San Jose, CA 95134

Table of Contents

List of Figures

List of Tables

1. An Introduction to Glide

The 3Dfx Interactive family of graphics accelerators enables personal computers and low cost video game platforms to host true 3D entertainment applications. Optimized for real-time texture-mapped 3D images, the graphics subsystem provides acceleration for advanced 3D features including true-perspective texture mapping with trilinear mipmapping and lighting, detail and projected texture mapping, texture anti-aliasing, and high precision subpixel correction. In addition, it supports general purpose 3D pixel processing functions, including triangle-based Gouraud shading, depth buffering, alpha blending, and dithering.

The Glide Rasterization Library is a set of low level rendering functions that serve as a software "micro-layer" to the graphics hardware, including the 3Dfx Interactive Texel*fx* ™ and the Pixel*fx*™ special purpose chips. Glide permits easy and efficient implementation of 3D rendering libraries, games, and drivers.

Why Glide?

Glide serves three primary purposes:

- It relieves programmers from hardware specific issues such as timing, maintaining register shadows, and working with hard-coded register constants and offsets.
- It defines an abstraction of the graphics hardware to facilitate ease of software porting.
- It acts as a delivery vehicle for sample source code providing in-depth hardware-specific optimizations for the graphics hardware.

By abstracting the low level details of interfacing with the graphics hardware into a set of C-callable functions, Glide allows developers to avoid working with hardware registers and memory directly, enabling faster development and lower probability of bugs. Glide also handles mundane and error prone chores such as initialization and shutdown.

Glide 2.x was designed for up to about 1 million triangles per second. Glide 3.0 is designed for the next order of magnitude: 1-10 million triangles per second. With the addition of vertex arrays, only one call need be made to draw a group of triangles. Tight inner loops, command packets, full triangle setup, and packed RGBA all contribute to being able to transfer and process millions of triangles per second.

Performance is one of Glide's top priorities. When decisions are made, performance is always one of the criteria taken into account, and is always an important criteria. The goal for Glide is to do as little as possible - it is a very thin API layer above the hardware. One rule of thumb is that Glide should impose no more than 5% to 10% overhead on an application when compared to what the application could do if it wrote hardware registers directly.

Glide is but one part of the 3Dfx Interactive Software Developer's Kit (SDK), which is designed to assist developers in creating tools and titles that are optimized for the graphics hardware. The SDK also includes the Texture Utility Software (TexUS™).

The Glide Utility Library contains utility routines that create fog tables, extensions that do significant pre-processing before calling Glide routines to access the graphics system, and obsolete routines that are provided for interim compatibility as Glide development continues.

Voodoo

The 3Dfx graphics accelerator subsystem, which may be called Voodoo Graphics, Voodoo Rush, or Voodoo^2, depending on it's age and functionality, sits on the PCI system bus of the host computer. The entry-level system configuration consists of two 3Dfx Interactive proprietary ASICs, Texel*fx* and Pixel*fx,* and memory. Figure 1.1 shows the entry level configuration as well as several ways to expand the system and enhance graphics performance. Increasing the number of Texel*fx* ASICs decreases the number of passes required to perform various texture mapping techniques. Systems with more than one 3Dfx Interactive graphics subsystem can utilize scanline interleaving to achieve the highest possible rendering performance.

Glide and the 3Dfx Interactive graphics hardware supports a rich set of rendering techniques, including:

- *Gouraud shading*. The programmer provides initial red, green, blue, and alpha values for each vertex. Glide calculates the associated gradients and the hardware automatically iterates the color across the defined triangle.
- *Texture mapping*. The programmer provides initial texture values *s/w*, *t/w*, and 1/*w* for each vertex and Glide computes the gradients. The hardware performs the proper iteration and perspective correction for true-perspective texture mapping. During each iteration of row/column walking, a division is performed by 1/*w* to correct for perspective distortion.
- *Texture mapping with lighting*. Texture-mapped rendering can be combined with Gouraud shading to introduce lighting effects during the texture mapping process. The programmer supplies initial color and texture values, Glide calculates the appropriate gradients, and the hardware performs the proper calculations to implement the lighting models and texture lookups. A texel is either modulated (multiplied by), added, or blended to the Gouraud shaded color. The selection of color modulation or addition is programmable.
- *Texture space decompression*. Texture map compression uses a patent-pending "narrow channel" **YAB** compression scheme that maps 24-bit RGB values to an 8-bit **YAB** format with little loss in precision.
- *Depth buffering*. 3Dfx Interactive graphics accelerators support hardware-accelerated, depth-buffered rendering with no performance penalty. The depth buffer is implemented in frame buffer memory: 2 Mbyte systems can utilize a 640×480 double buffered display buffer and a 16-bit *z* buffer. To eliminate many of the *z* aliasing problems typically encountered with 16-bit *z* buffer systems, the graphics subsystem allows a floating point representation of the 1/*w* parameter to be used as the depth component.

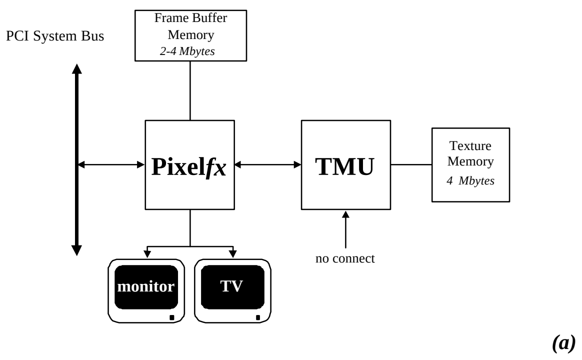
Figure 1.1 System configurations.
The Pixelfx chip interfaces with the host computer, the linear frame buffer, and the display monitor. It implements basic 3D primitives including Gouraud shading, alpha blending, depth buffering, dithering, and fog. The TMU (located on the Texelfx chip) implements true-perspective, detail, and projected texture mapping, bilinear and trilinear filtering, and level-of-detail mipmapping.

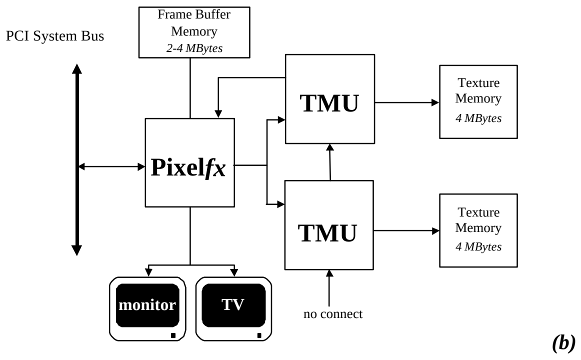
(a) The basic configuration has one Pixelfx chip and one TMU. The advanced texture mapping techniques of detail texture mapping, projected texture mapping, and trilinear texture filtering are two-pass operations, but there is no performance penalty for point-sampled or bilinear-filtered texture mapping with mipmapping.

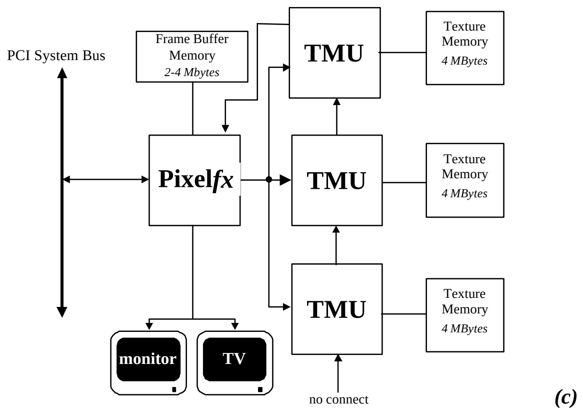
(b) A two TMU configuration allows single pass detail texture mapping, projected texture mapping, or trilinear filtering.

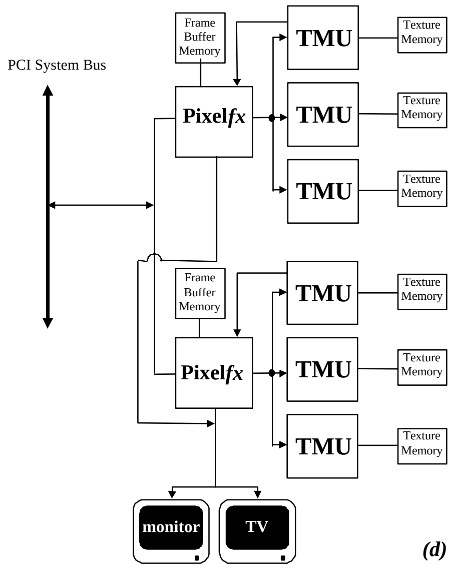
(c) Three TMUs can be chained together to provide single pass rendering of all supported advanced texture mapping features, including projected texture mapping.

(d) For the highest possible rendering performance, multiple 3Dfx Interactive graphics accelerator subsystems can be chained together utilizing scanline interleaving to effectively double the rendering rate of a single subsystem.







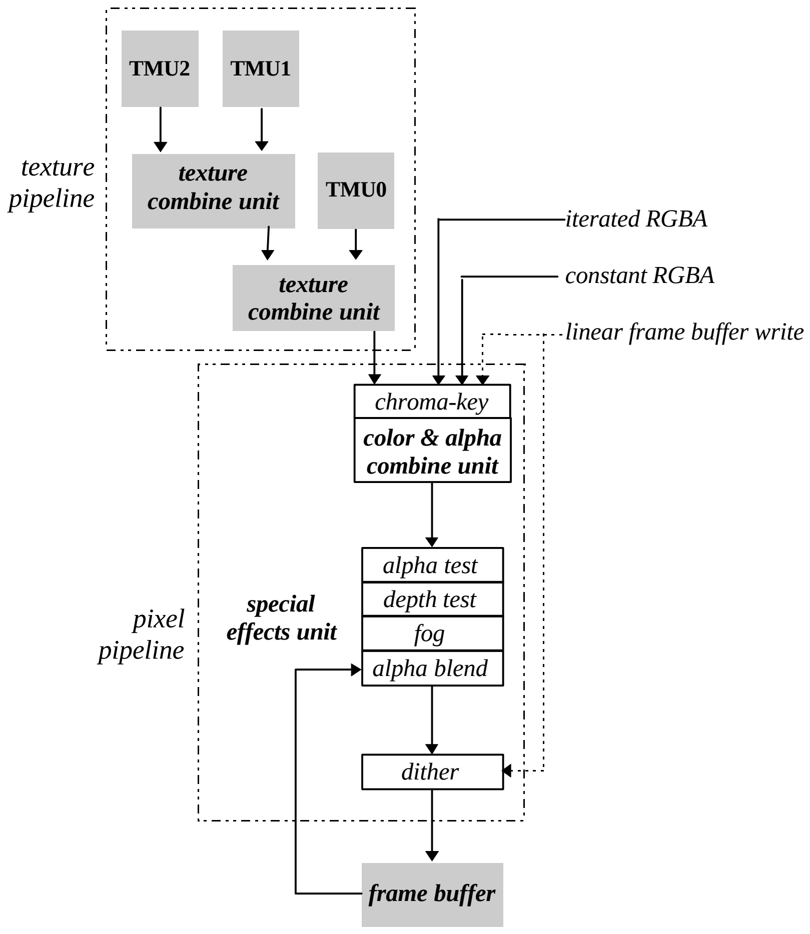


- *Pixel blending*. The hardware supports alpha blending functions that blend incoming source pixels with current destination pixels with no performance penalty. Alpha buffering is supported, but it is mutually exclusive with depth buffering and triple buffering. Note that alpha buffering is required only if destination alpha is used in alpha blending; alpha blending modes that do not use destination alpha can be used with depth buffering and triple buffering.
- *Fog*. The 3Dfx Interactive graphics accelerator subsystem supports a 64-entry lookup table to support atmospheric effects such as fog and haze. When enabled, a 14-bit floating point representation of 1/*w* is used to index into the 64-entry lookup table and interpolate between entries. The output of the lookup table is a value that represents the level of blending to be performed between a reference fog color and the incoming pixel.
- *Chroma-keying*. 3Dfx Interactive graphics accelerator supports a chroma-key operation used for transparent object effects. When enabled, an outgoing pixel is compared with the chroma-key register. If a match is detected, the outgoing pixel is invalidated in the pixel pipeline, and the frame buffer is not updated.
- *Color dithering*. Numeric operations are performed on 24-bit colors within the graphics subsystem. However, the final stage of the pixel pipeline dithers the color from 24 bits to 16 bits before storing it in the display buffer. The 16-bit color dithering allows for the generation of photo-realistic images without the additional cost of a true color frame buffer storage area.

The Rendering Engine

The graphics hardware has a very flexible lighting and texture mapping pipeline to support all of the features described above. Glide abstracts it into three distinct units: the texture combine unit, the color and alpha combine units, and the special effects unit. The basic architecture is illustrated in Figure 1.2.

Figure 1.2 The pixel pipeline.
The rendering engine is structured as a pipeline through which each pixel drawn to the screen must pass. The individual stages of the pixel pipeline modify or invalidate individual pixels based on mode settings. The input to the pixel pipeline can come from one of four sources: a texture value, an iterated RGBA value, a constant RGBA value, or data for a frame buffer write. Pixels that pass the chroma-key test go to the color combine unit where a user-specified lighting function is applied. The special effects unit further modifies the pixel with alpha and depth testing, fog, and alpha blending operations. The final 24-bit color value is then dithered to 16 bits and written to the frame buffer.



About This Manual

The *Glide Programming Guide* attempts to introduce a knowledgeable graphics programmer to the capabilities of the hardware through the Glide interface. The subroutines are introduced in a logical progression: initialization and termination requirements are first, then simple rendering capabilities, followed by more and more complex functions. The audience for this manual is the application programmer who just took delivery on 3Dfx Interactive graphics accelerator and wants to port existing applications or develop new applications in Glide. The experienced Glide programmer will use the *Glide Reference Manual* to research specific Glide functions, but will reach for this manual when trying out new features.

Chapter 2, *Glide in Style*, describes data types, data formats, and the programming model used in Glide and the graphics subsystem.

Chapter 3, *Getting Started*, describes the display buffers and the initialization and termination requirements for Glide and the graphics hardware. It also includes a very simple but complete program that clears the screen.

Chapter 4, *Rendering Primitives*, describes the functions that draw points, lines, triangles, and convex polygons in both aliased and anti-aliased forms. In addition, clipping and backface culling are discussed.

Chapter 5, *Color and Lighting*, describes the functions that control the color and alpha combine unit, which can produce effects that run the gamut from simple Gouraud shading to diffuse ambient lighting with specular highlights and other complex lighting models.

Chapter 6, *Using the Alpha Component*, describes the various ways to utilize the alpha channel: alpha blending, alpha buffering, and alpha testing.

Chapter 7, *Depth Buffering*, presents two techniques for depth buffering.

Chapter 8, *Special Effects*, describes other special rendering effects that can be produced in the pixel pipeline: atmospheric effects like fog, haze, and smoke; multi-pass alpha-blended fog; transparent objects implemented with chroma-keying; and alpha masking.

Chapter 9, *Texture Mapping*, describes the texture pipeline and texture mapping while Chapter 10, *Managing Texture Memory*, describes the process of downloading textures into texture memory.

Chapter 11, *Accessing the Linear Frame* Buffer, describes the Glide functions that provide a path for reading and writing the frame buffer directly.

Chapter 12, *Housekeeping Routines*, and Chapter 13, *Glide Extensions*, describes the routines in Glide and the Glide Utilities Library that haven't been discussed already.

Chapter 14, *Programming Tips and Techniques*, give some hints about how to head off trouble and get the best performance from your 3Dfx Interactive graphics accelerator.

The *Glide Programming Guide* concludes with two appendices, one containing a non-trivial example, and the other summarizing the Glide constants used to set state variables. There is also a *Glossary* of frequently used terms and a comprehensive *Index*.

Other Documentation

Available from 3Dfx Interactive, Inc.:

Glide 3.0 Reference Manual
SST1 Application Notes
TexUS Manual

Additional published references:

FOLE90	Foley, J., A. van Dam, S. Feiner, and J. Hughes, *Computer Graphics*, Addison-Wesley, Reading, MA, 1990
OPEN92	OpenGL Architecture Review Board, *OpenGL Reference Manual*, Addison-Wesley, Reading, MA, 1992
OPEN93	OpenGL Architecture Review Board with J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide*, Addison-Wesley, Reading, MA, 1992
PHIG88	PHIGS+ Committee, A. van Dam, Chair, "PHIGS+ Functional Description - Revision 3.0". *Computer Graphics*, 22(3), p. 125-218
SUTH74	Sutherland, I. E. and G. W. Hodgman, "Reentrant Polygon Clipping", *CACM* 17(1), p. 32-42
WATT92	Watt, A. and M. Watt, *Advanced Animation and Rendering Techniques: Theory and Practice*, Addison-Wesley, Reading, MA, 1992
WILL83	Williams, L., "Pyramidal Parametrics", *SIGGRAPH 83*, p. 1-11

Online references:

http://www.3dfx.com

http://www.sgi.com/grafica/texmap/index.html

http://reality.sgi.com/Fun/Free_graphics.html

2. Glide in Style

In this Chapter

You will learn about:

- the naming conventions for functions, types, and constants.
- the notational conventions that designate functions, types, variables, parameters, and constants in this manual.
- the state machine model that Glide uses to minimize bandwidth to the hardware and increase graphics performance.
- the functions that save and restore Glide state.
- the functions that establish a format for vertex information.
- the constraints and properties of numerical data representing geometric, color, and texture coordinates.

Naming and Notational Conventions

Functions are divided into families consisting of routines related in their duties. All Glide functions are prefixed with **gr**; all Glide Utility functions use **gu** as the prefix. The Glide prefix is immediately followed by the family name, for example **grDrawTriangle()** and **grDrawLine()** are both members of the **grDraw** family. Glide uses the mixed caps convention for function names. When function names appear in the text of this manual, they are shown in bold face type. Actual function names end with '()'; function family names do not.

The internal name for the graphics subsystem is "SST". Some function names, type definitions, and constants within Glide reflect this internal name, which is easier to type than Voodoo Graphics, Voodoo Rush, or Voodoo^2. For example, **grSstWinOpen()** initializes the hardware.

Constants are named values that are defined in `glide.h`. The names of constants use all uppercase letters, as in `MAX_NUM_SST` and `GR_TEXTUREFILTER_BILINEAR` and are shown in `Courier` font when they appear in the text of this manual.

C specifications for functions and data types are displayed in shaded rectangles throughout this manual. Glide *type definitions* are shown in *Helvetica* type to distinguish them from the C keywords and primitive types. Glide makes use of enumerated types for function arguments in order to restrict them to the defined set of values. Enumerated types end with *_t*, as in *GrColorFormat_t*.

Glide *variable names* and *function arguments* are italicized in both the C specifications and the text.

Code segments use `Courier` font.

The State Machine Model

Glide is state based: rendering "modes" can be set once and then remain in effect until reset. Parameter values like a reference value for depth comparisons and a specific depth test are set once and are used whenever depth testing is enabled (until they are given new values). The state machine model allows users to set modes and reference values only when they change, minimizing the host-to-hardware transfers.

For example, one of the state variables Glide maintains is the "current mipmap", used during texture mapping. A mipmap is a collection of hierarchically defined texture maps that are loaded into the texture memory that supports the TMUs. A stateless model would not retain information about the contents of the texture memory, so each rendering operation would have to include a texture memory address.

Sending redundant state information can lead to noticeable performance degradation. For example, if a system is attempting to render 200,000 triangles per second and the "current mipmap" is sent as a 4-byte address, bandwidth associated with updating this single state variable can amount to 800KB/sec. Compound this with all of the other state information necessary and the amount of unnecessary data sent across the system bus can become overwhelming.

Two library functions are used to save and restore state. Use **grGet(**`GR_GLIDE_STATE_SIZE,…`**)** to determine the size of the buffer in which the state will be saved (see Chapter 13).

void **grGlideGetState(**void **state* **)**

void **grGlideSetState(** const void **state* **)**

grGlideGetState() makes a copy of the current state of Glide in a buffer, *state*, provided by the user. The saved state can be restored at some later time with **grGlideSetState()**. These routines save and restore all Glide state, and therefore are expensive to use. If only a small subset of Glide state needs to be saved and restored, these routines should not be used.

Coordinate Spaces

Glide 3.0 supports two different coordinate spaces: native hardware device coordinates (the only option in previous versions of Glide), or clip coordinates. The choice is made with the **grCoordinateSpace()** command.

void **grCoordinateSpace(** *GrCoordinateSpaceMode_t mode***)**

The argument, *mode*, is either `GR_CLIP_COORDS` or `GR_WINDOW_COORDS`. Window coordinates are relative to the origin of the window. Clip coordinates are relative to a viewport defined with the new command **grViewport()**.

void **grViewport(** *FxI32 x*, *FxI32 y*, *FxI32 width*, *FxI32 height* **)**

grViewport() specifies the viewport transformation. The current **grSstOrigin()** setting determines whether *x* and *y* specify the upper left corner or the lower left corner. Negative *width* and *height* are allowed and mirror the image about the *x* or *y* axis. If (x_{clip}/w, y_{clip}/w) represent normalized device coordinates, then the window coordinates (x_{win}, y_{win}) are computed as:

$$x_{win} = (x_{clip}/w+1)(width/2) + x \qquad \text{and} \qquad y_{win} = (y_{clip}/w+1)(height/2) + y$$

When using clip coordinates, the **grDepthRange()** command specifies the viewport parameters for the depth component.

void **grDepthRange(** *FxFloat near, FxFloat far* **)**

If *z* buffering, clip-space *z* is in the range [-*w*..+*w*]. After division by *w*, *z* is in the range [-1..1] which is mapped to the depth buffer according to [*near*.. *far*], where [*near*=0.. *far*=1] represents the entire range of the depth buffer. **grDepthRange()** is ignored unless clip coordinates are being used and z buffering is enabled.

Choosing a Coordinate Space

When window coordinates are used, the application performs the coordinate divisions by *w*, providing *x*/*w*, *y*/*w*, *z*/*w*, 1/*w*, *s*/*w*, *t*/*w*, and *q*/*w* as necessary in the vertex structure (only *x*/*w* and *y*/*w* are mandatory). Window coordinates may be less than optimal on future hardware that can perform perspective division and viewport transformations.

When clip coordinates are used, the division by *w* is performed automatically. The minimal vertex specifies *x*, *y*, and *w*. If *z* buffering is enabled, *z* should be in the range [–*w*..+*w*]; otherwise, *z* data need not be given. Glide will automatically compute *x*/*w*, *y*/*w*, *z*/*w*, and 1/*w*, perform vertex snapping on the results, and then apply the viewport transformation to get window coordinates. Texture coordinates *s* and *t* are in the range [0..1] for all texture sizes and aspect ratios. Glide automatically computes *s*/*w*, *t*/*w*, and *q*/*w*.

Clip space coordinates are recommended for all new applications. It is likely that future hardware will perform the viewport transformation and depth range computations to further off-load the CPU.

PORTING NOTE

Window coordinate space was the only available option in previous releases of Glide. The *w* component should be ported to Glide 3.0 vertices as `GR_PARAM_Q` and stored as 1/*w*. All *x*, *y*, *s*, and *t* components should be multiplied by 1/*w*, as in Glide 2.x.

The `GR_PARAM_Q` value is used when using fog mode `GR_FOG_WITH_TABLE_ON_Q` (formerly `GR_FOG_WITH_TABLE` and `GR_FOG_WITH_TABLE_ON_W`) and when *w* buffering (which should properly be renamed to *q* buffering, but won't be).

Specifying Vertices

The 3Dfx Interactive graphics accelerator is a rendering engine. The user configures the texture and pixel pipelines (see Figure 1.2) and then sends streams of vertices representing points, lines, triangles, and convex polygons. (In fact, the hardware renders only triangles; Glide converts points and lines to triangles and triangulates polygons as needed.)

Vertices are specified as a collection of parametric values, chosen from the following:

- the geometric coordinates (*x*, *y*);
- the color components (*r*, *g*, *b*, *a*);
- the depth indicator *z* (for window coordinators), or *q* (for clip coordinates);
- the homogenous coordinates *w* (distance from the eye, required for clip coordinates) and *q* (distance from the projected source);

- the TMU-specific texture coordinates (s_i, t_i), where *i* is the TMU the texel resides in;
- the TMU-specific homogeneous coordinate q_i, where *i* is the TMU where the value will be used;
- if supported, a separate fog table index (*q* may also be used to index a fog table).

Every vertex must specify values for *x* and *y*, but the other parameters are optional and need only be set if the rendering configuration requires them.

Syntactically, a vertex is a structure containing all the parameter values that apply. The vertex structure may hold additional information of interest to the application as well. The vertex layout is communicated semantically to Glide by issuing a series of **grVertexLayout()** commands, one each of the parameters included in the vertex structure.

void **grVertexLayout(***FxU32 param*, *FxI32 offset*, *FxU32 mode* **)**

grVertexLayout() is called once for each value of *param*, chosen from the values in the first column of Table 2.1 or Table 2.2 (there is a table for each coordinate space option).

offset is either the offset in bytes of the parameter data from the vertex pointer. The offset can be either positive or negative. Align data on word boundaries for optimal performance.

mode is either `GR_PARAM_ENABLE` or `GR_PARAM_DISABLE`. Disabling a parameter will potentially cause it to inherit the last known value. When a parameter is disabled, the offset argument is ignored. Disabling a mandatory parameter like `GR_PARAM_XY` will cause a fatal Glide error.

PORTING NOTE

The *GrVertex* structure is no longer necessary, since **grVertexLayout()** allows arbitrary layouts. Therefore *GrVertex* structure has been removed. To facilitate porting Glide 2.x applications, the old vertex structure needs to be defined in the application, and the vertex layout set accordingly. Example 2.2 shows you how.

Glide determines whether or not color and texture parameters are required based on other mode settings such as **grColorCombine()**. In addition, *s*, *t*, and *q* values can be inherited in order to reduce gradient calculations on older hardware. This situation is handled in Glide 3.0 by the addition of the *mode* argument to **grVertexLayout()**.   If an application wants a TMU-specific value for *s*, *t*, or *q*, the appropriate parameter will be enabled (`GR_PARAM_ENABLE`) in the vertex layout. Alternatively, if the application wants an *s*, *t*, or *q* value to be inherited, it will specify `GR_PARAM_DISABLE` instead.

PORTING NOTE

The `GR_HINT_STWHINT` hint is obsolete in Glide 3.0: it's functionality is implemented within **grVertexLayout()** as follows:

Glide 2.x: **grHints(**`GR_HINT_STWHINT`, `GR_STWHINT_W_DIFF_TMU0`**);**
Glide 3.0: **grVertexLayout(**`GR_PARAM_Q0`,…, `GR_PARAM_ENABLE`**);**

Glide 2.x: **grHints(**`GR_HINT_STWHINT`, `GR_STWHINT_ST_DIFF_TMU1`**);**
Glide 3.0: **grVertexLayout(**`GR_PARAM_ST1`,…, `GR_PARAM_ENABLE`**);**

Table 2.1 Specifying clip coordinate space vertices.
*The **grVertexLayout()** command is called once for each value of **param**, chosen from the table below.*

param	*type*	*size in bytes*	*description*	*values*	*usage*
`GR_PARAM_XY`	*FxFloat*	8	*x* and *y* coordinates. *Vertex snapping is no longer required*.	In the range [−*w*..*w*].	**Required**. Must be at *offset* 0.
`GR_PARAM_Z`	*FxFloat*	4	*z* coordinate.	In the range [−*w*..*w*].	When z buffering is enabled.
`GR_PARAM_W`	*FxFloat*	4	*w* coordinate.	In the range [1..64K].	Required.
`GR_PARAM_Q`	*FxFloat*	4	*Usage depends on choice of coordinate space*.	Depth/fog iterator.	When using fog mode `GR_FOG_WITH_TABLE_ON_Q` or *w* buffering is enabled. Defaults to 1 if not defined.
`GR_PARAM_ST`*n*	*FxFloat*	8	*s* and *t* coordinates for TMU *n*.	*s*, *t* in range [0,1] for one repeat of the texture. *Independent of aspect ratio*.	When texture mapping.
`GR_PARAM_Q`*n*	*FxFloat*	4	*q* coordinate for TMU *n*.		When texture mapping with projected textures. Defaults to GR_PARAM_Q if not defined.
`GR_PARAM_A`	*FxFloat*	4	*alpha* value.	In the range [0..1]	When using alpha blending, alpha testing, or anti-aliasing.
`GR_PARAM_RGB`	*FxFloat*	12	*RGB* triplet.	In the range [0..1]	Choose one of the two color formats.
`GR_PARAM_PARGB`	*FxU32*	4	Packed ARGB, one byte per component.	Each component is an integer in the range [0..255]	
`GR_PARAM_FOG_EXT` *(if FOGCOORD extension is supported)*	*FxFloat*	4	Fog table index.	*f/w* in the range [0..255]	When using fog mode `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT`

Table 2.2 Specifying window coordinate space vertices.
The ***grVertexLayout()*** *command is called once for each value of* ***param****, chosen from the table below. Note that `GR_PARAM_W` is not valid for window coordinate space.*

param	*type*	*size in bytes*	*description*	*values*	*usage*
`GR_PARAM_XY`	*FxFloat*	8	*x* and *y* coordinates. *Vertex snapping is no longer required*.	*x*/*w*, *y*/*w* in the range [–2048..2047]	**Required**. Must be at *offset* 0.
`GR_PARAM_Z`	*FxFloat*	4	*z* coordinate.	Stored as 1/z. In the range [0..64K]	When z buffering is enabled.
`GR_PARAM_Q`	*FxFloat*	4	*Usage depends on choice of coordinate space*.	1/*w*	Required.
`GR_PARAM_ST`*n*	*FxFloat*	8	*s* and *t* coordinates for TMU *n*.	Stored as *s*/*q*, *t*/*q* in the range [0..256] for one repeat of the texture. *The range of the smaller dimension is limited by the aspect ratio*. *See Chapter 9*.	When texture mapping.
`GR_PARAM_Q`*n*	*FxFloat*	4	*q* coordinate for TMU *n*.	In the range [0..255]	When texture mapping with projected textures. Defaults to GR_PARAM_Q if not defined or if disabled.
`GR_PARAM_A`	*FxFloat*	4	*alpha* value.	In the range [0..255]	When using alpha blending, alpha testing, or anti-aliasing.
`GR_PARAM_RGB`	*FxFloat*	12	*RGB* triplet.	In the range [0..255].	Choose one of the two color formats.
`GR_PARAM_PARGB`	*FxU32*	4	Packed ARGB, one byte per component.	Each component is an integer in the range [0..255].	
`GR_PARAM_FOG_EXT` *(if FOGCOORD extension is supported)*	*FxFloat*	4	Fog table index.	In the range [0..255].	When using fog mode `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT`

The application program has control over the order in which the selected parameters occur in the vertex array. For example, the code segment in Example 2.1 defines a vertex structure that has an (*x*,*y*,*z*) position and an RGB color. Other examples follow.

Example 2.1 Defining a vertex layout.
The code fragment below defines a vertex structure as an (x,y,z) position and an RGB color. It continues on to establish the layout semantically by calling ***grVertexLayout()****.*

```
Typedef struct {
   FxFloat x, y;
   FxFloat ooz;
   FxFloat r, g, b;
} myVertex;

grCoordinateSpace( GR_WINDOW_COORDS );
grVertexLayout( GR_PARAM_XY, 0, GR_PARAM_ENABLE );
grVertexLayout( GR_PARAM_Z, 8, GR_PARAM_ENABLE );
grVertexLayout( GR_PARAM_RGB, 12, GR_PARAM_ENABLE );
```

Example 2.2 Re-creating GrVertex in Glide 3.0.
The code segment below defines the vertex structure from previous versions of Glide and shows the grVertexLayout() that may be used to

```
typedef struct{
 float x, y, z;          /* X, Y, Z */
 float r, g, b;          /* R, G, B */
 float ooz;              /* 65535/Z (used for Z-buffering) */
 float a;                /* Alpha */
 float oow;              /* 1/W (used for W-buffering, texturing) */
 GrTmuVertex tmuvtx[GLIDE_NUM_TMU];
} myVertex;              /* old GrVertex */

grCoordinateSpace(GR_WINDOW_COORDS);
grVertexLayout(GR_PARAM_XY, 0, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_RGB, 12, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_Z, 24, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_A, 28, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_W, 32, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_ST0, 36, GR_PARAM_ENABLE);
```

Example 2.3 Creating a vertex definition using clip coordinates, a z buffer, and a fog table indexed by q.
The code fragment below creates a vertex layout that includes x, y, z, w, q, and a packed color.

```
typedef struct{
  FxFloat x, y, z;      /* X, Y, Z */
  FxFloat w, q;         /* W, Q */
  FxU32 pColor;         /* packed ARGB */
} myVertex;

grCoordinateSpace(GR_CLIP_COORDS);
grVertexLayout(GR_PARAM_XY, 0, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_Z, 8, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_W, 12, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_Q, 16, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_PARGB, 20, GR_PARAM_ENABLE);
```

Example 2.4 Creating a vertex definition using window coordinates and the FOGCOORD extension.
The code fragment below creates a vertex layout that includes x, y, q, f, and a packed color.

```
typedef struct{
  FxFloat x, y;         /* X, Y */
  FxFloat q;            /* Q */
  FxFloat f;            /* fog table index */
  FxU32 pColor;         /* packed ARGB */
} myVertex;

grCoordinateSpace(GR_CLIP_COORDS);
grVertexLayout(GR_PARAM_XY, 0, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_Q, 8, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_FOG_EXT, 12, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_PARGB, 20, GR_PARAM_ENABLE);
```

Example 2.5 Creating a vertex definition for projected texture mapping.
The code fragments below creates a vertex layout that includes x, y, w or q (depending on the coordinate space), a packed color, s and t values for two TMUs and a separate q for TMU1.

```
typedef struct{
  FxFloat x, y;         /* X, Y */
  FxFloat q;            /* Q */
  FxFloat f;            /* fog table index */
  FxU32 pColor;         /* packed ARGB */
} myVertex;

grCoordinateSpace(GR_CLIP_COORDS);
grVertexLayout(GR_PARAM_XY, 0, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_Q, 8, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_FOG_EXT, 12, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_PARGB, 20, GR_PARAM_ENABLE);
```

Using More than One Vertex Layout

Some applications may find it useful to use several vertex layouts during the course of the program. While only one layout is current at a time, you can save the current one, define and use a new one, then restore the saved one.

void **grGlideGetVertexLayout(** void **layout* **)**
void **grGlideSetVertexLayout(** void **layout* **)**

grGlideGetVertexLayout() makes a copy of the current vertex layout established by calls to **grVertexLayout()**. The application can restore the saved layout by calling **grGlideSetVertexLayout()**. Use **grGet(**`GR_GLIDE_VERTEXLAYOUT_SIZE,…`**)** to determine how much space is needed (and hence, how big the *layout* buffer should be).

PORTING NOTE

In Glide 3.0, vertices no longer need to be snapped to sub-pixel precision. The newer platforms perform snapping in hardware; Glide will do it for the older ones. There may be a slight performance degradation on platforms (e.g. SST-1 and SST-96) that don't have a triangle setup unit.

3. Getting Started

In This Chapter

You will learn how to:

- ▼ initialize Glide.
- ▼ configure and initialize the hardware.
- ▼ manage multiple 3Dfx Interactive graphics accelerators.
- ▼ terminate cleanly.
- ▼ manage the display buffers.
- ▼ detect and respond to errors.

Starting Up

Glide provides several functions to initialize Glide and to detect and configure a 3Dfx Interactive graphics subsystem. Three functions, **grGlideInit()**, **grSstSelect()**, and **grSstWinOpen()**, initialize Glide and the hardware and must be called, in the order listed, before calling any other Glide routines (except the **grGet()** and **grGetString()** calls that detect the presence of 3Dfx Interactive graphics subsystems). Failing to do this will cause the system to operate in an undefined (and, most likely, undesirable) state.

The first initialization function, **grGlideInit()**, sets up the Glide library and thus must be called before any other Glide functions are executed (with one exception, noted below). It allocates memory, sets up pointers, and initializes library variables and counters. There are no arguments, and no value is returned.

void **grGlideInit(** void **)**

Their is one exception to the rule stated above that **grGlideInit()**must be called before all other Glide routines. **grGet(**`GR_NUM_BOARDS,…`**)** may be called before **grGlideInit()** to determine the presence or absence of a graphics subsystem.

The next function called to initialize the system is **grSstSelect()**, which makes a specific graphics subsystem “current”. It must be called after **grGlideInit()** but before **grSstWinOpen()**.

void **grSstSelect(** int *whichSST* **)**

The argument is the ordinal number of the subsystem that will be made active and must be in the range [0..*numBoards*], where *numBoards* is the value returned when **grGet()** is called with argument `GR_NUM_BOARDS`. If *whichSST* is outside the proper range of values and the debugging version of Glide is used, a run-time error is generated. If the release version of Glide is loaded, use of an inappropriate value for *whichSST* will result in undefined behavior.

The final initialization function, **grSstWinOpen()**, initializes the currently active graphics subsystem, specified by the most recent call to **grSstSelect()**, to the default state. All hardware special effects (depth buffering, fog, chroma-key, alpha blending, alpha testing, etc.) are disabled. All global state constants (the chroma-key reference value, the alpha test reference, the constant depth value, the constant alpha value, the constant color value, etc.) and pixel rendering statistic counters are initialized to zero.

PORTING NOTE

Significant changes in Glide 3.0 pave the way for full support for windowed environments, including multiple windows. These changes are because resources are shared in a windowed environment. When programming a full screen Glide application, developers assume they have complete ownership of the graphics hardware, when in reality, it may be shared. Other processes (or the Window system) can appropriate resources owned by the Glide application at any time. Maintaining this illusion of complete ownership is impossible without severe performance penalties. So, instead of hiding the fact that 2D/3D resources are shared, Glide 3.0 ensures that applications can endure asynchronous reallocation of 2D/3D resources yet recover completely and gracefully.

grSstWinOpen() should be called once per installed graphics subsystem (note that scanline interleaved subsystems are treated as a single subsystem) and must be executed *after* **grGlideInit()** and **grSstSelect()**. It returns an opaque context handle if the initialization was successful and zero otherwise. Only one context at a time may be in use in Glide 3.0.

```
GrContext_t grSstWinOpen( FxU32                  hWin,
                          GrScreenResolution_t   res,
                          GrScreenRefresh_t      refresh,
                          GrColorFormat_t        cFormat,
                          GrOriginLocation_t     locateOrigin,
                          int                    numBuffers,
                          int                    numAuxBuffers
                        )
```

The arguments to **grSstWinOpen()** configure the frame buffer. The first argument, *hWin*, specifies a handle for the window in which the graphics will be displayed. The interpretation of *hWin* depends on the system environment. DOS applications must specify `NULL`. Applications run on SST-1 graphics hardware must specify `NULL` as well. Win32 full screen applications running on a SST-96 system must specify a window handle; a `NULL` value for *hWin* will cause the application's real window handle (i.e., what is returned by Microsoft's `GetActiveWindow` API) to be used. Since Win32 pure console applications do not have a window handle, they can be used only with SST-1 and a `NULL` window handle is required. Finally, Glide Win32 applications that run in a window may either specify `NULL` (if there is only one window), or the correct *hWin*, cast to *FxU32*.

Table 3.1 Specifying a window handle in grSstWinOpen().
*The interpretation of the **hWin** argument to **grSstWinOpen()** depends on the system environment, as shown below.*

system environment	***hWin** value*
DOS	`NULL`

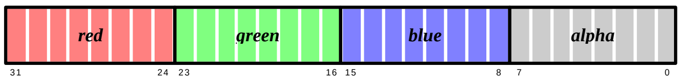
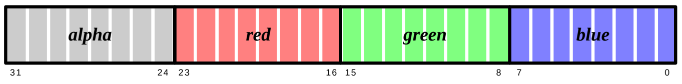
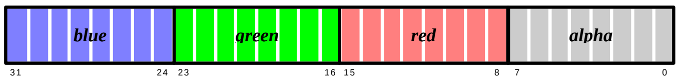
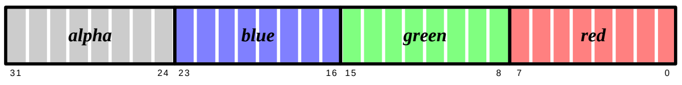
Win32, full screen	`NULL` or *hWin*
Win32, pure console	`NULL` (SST-1 only)
Win32 Glide application	`NULL` or *hWin* (SST-96 only)

The screen resolution and refresh rate are specified in the next two arguments, *res* and *refresh*. Both variables are given values chosen from enumerated types defined in the `sst1vid.h` header file. A typical application might set *res* to `GR_RESOLUTION_640x480` and *refresh* to `GR_REFRESH_60HZ`.

While not recommended, the screen resolution may be specified as `GR_RESOLUTION_NONE` on an SST-96 system. If so, Glide will use the user specified window (see the *hWin* parameter). The *ref* parameter is ignored when a Win32 application is running in a window. Specifying `GR_RESOLUTION_NONE` on an SST-1 system will cause the call to fail.

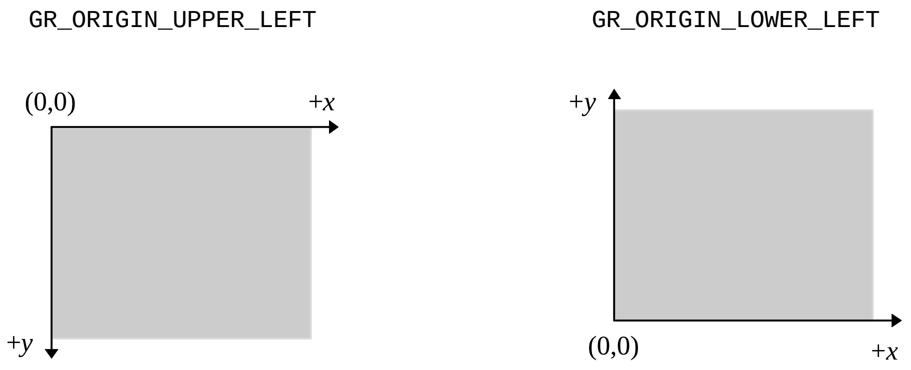
The fourth argument, *cFormat*, specifies the packed color RGBA ordering in the frame buffer. Different software systems assume different byte ordering formats for pixel color data. For the widest possible compatibility across a wide range of software, Glide provides “byte swizzling,” meaning that incoming pixels can have their color values interpreted in one of four different formats that are defined in the enumerated type *GrColorFormat_t* and are shown in Table 3.2. The color format affects data written to the linear frame buffer (the subject of Chapter 11) and parameters for the following Glide functions: **grBufferClear()** (described later in this chapter), **grChromakeyValue()** (described in Chapter 8), **grConstantColorValue()** (see Chapter 5), and **grFogColorValue()** (see Chapter 8).

Table 3.2 Frame buffer color formats.
Glide supports four different color byte orderings: RGBA, ARGB, BGRA, and ABGR. Color byte ordering determines how user-supplied color values are interpreted. The first column in the table shows the name of the format, as defined in the enumerated type GrColorFormat_t. The second column in the table shows the byte ordering of the color components within a 32-bit word.

color format	*byte ordering*
`GR_COLORFORMAT_RGBA`	
`GR_COLORFORMAT_ARGB`	
`GR_COLORFORMAT_BGRA`	
`GR_COLORFORMAT_ABGR`	

The fifth parameter to **grSstWinOpen()** specifies the location of the screen space origin. If *locateOrigin* is `GR_ORIGIN_UPPER_LEFT`, the screen space origin is in the upper left corner with positive *y* going down. `GR_ORIGIN_LOWER_LEFT` places the screen space origin at the lower left corner with positive *y* going up. Figure 3.1 shows the two possibilities for locating the origin.

Figure 3.1 Locating the origin.
The 3Dfx Interactive graphics accelerator allows the origin to be in the upper left or lower left corner of the screen. The choice of coordinate system is be made by passing the appropriate parameter to ***grSstWinOpen()****.*



The final two arguments to **grSstWinOpen()** select the buffering options. The first one, *numBuffers*, specifies double or triple buffering and is an integer value, either 2 or 3. The other argument, *numAuxBuffers*, specifies the number of auxiliary buffers required by an application. The auxiliary buffers are used for depth or alpha buffering. Permitted values are 0 or 1. For full screen applications, this parameter allows both SST-1 and SST-96 to validate whether the available video memory will support the application's requirements for color and auxiliary buffers at a specified screen resolution. For a windowed application running on SST-96, this parameter allows an application to run in a larger 3D window if a depth buffer is not necessary (depth and back buffers share the same off-screen video memory).

If there is not enough memory to support the desired resolution and buffering options, an error will occur.

Querying for Screen Parameters

Applications that are written to run on a variety of hardware configurations can query for available resolutions before calling **grSstWinOpen()**.

```
typedef struct  {
    GrScreenResolution_t    resolution;
    GrScreenRefresh_t       refresh;
    int                     numColorBuffers;
    int                     numAuxBuffers;
} GrResolution;

FxI32  grQueryResolutions(  const  GrResolution      *resTemplate,
                            GrResolution             *output
                          )
```

grQueryResolutions() returns all available frame buffer configurations that match the constraints specified in the template *resTemplate*. The constraints are specified as either `GR_QUERY_ANY` or a specific value in each of the four fields in the *GrResolution* structure. If *output* is NULL,

grQueryResolutions() returns the number of bytes required to contain the available resolution information. The application can then allocate space and call **grQueryResolutions()** again to return the information. This process is demonstrated in Example 3.1.

Example 3.1 Querying for possible frame buffer configurations.
*The code fragment below calls **grQueryResolutions()** twice, the first time to establish the amount of space required for all the possible configurations, and the second time to actually return the data.*

```
GrResolution   query;
GrResolution   *list;
int            listSize;

/* find all possible modes that include a z-buffer */
query.resolution      = GR_QUERY_ANY;
query.refresh         = GR_QUERY_ANY;
query.numColorBuffers = GR_QUERY_ANY;
query.numAuxBuffers   = 1;

listSize = grQueryResolutions( &query, NULL );
list = malloc( listSize );
grQueryResolutions( &query, list );
```

Example 3.2 The Glide initialization sequence.
*This code fragment calls the three Glide functions, in the required order, that initialize the software and the hardware subsystems. The parameters to **grSstWinOpen()** establish a double buffered full-screen frame buffer with 640´480 screen resolution and a 60Hz refresh rate. Colors are stored as RGBA, the origin is in the lower left corner, and there is no auxiliary buffer.*

```
GrContext_t gcon;

grGlideInit(void);
grSstSelect(0);
if ((gcon=(grSstWinOpen(NULL,GR_RESOLUTION_640x480,GR_REFRESH_60HZ,
              GR_COLORFORMAT_RGBA,GR_ORIGIN_LOWER_LEFT, 2, 0))==0)
   printf(“ERROR: failed to open graphics context!\n”);
```

When programming a full screen Glide application, the developer has complete ownership of the 3D hardware and texture ram. Many applications will be developed to run under Windows 95, however, and must be prepared to restore the graphics state after a period of inactivity.

To gracefully handle the loss of resources (e.g. to another 3D application being scheduled by the Windows 95 operating system), an application is required to periodically (typically once per frame) query with **grSelectContext()** to determine if Glide’s resources have be reallocated by the system. *context* is a context handle returned from a successful call to **grWinOpen()**.

FxBool **grSelectContext(** *GrContext_t context* **)**

If none of the rendering context’s state and resources have been disturbed since the last call, **grSelectContext()** will return `FXTRUE`. In this case no special actions by the application are required. If it returns `FXFALSE`, then the application must assume that the rendering state has changed and must be reestablished (by re-downloading textures, explicitly resetting the rendering state, etc.) before further rendering commands are issued.

Driving Multiple Systems

Glide supports two forms of multiple graphics subsystem support: multiple subsystems driving multiple displays and two subsystems driving a single display.

Selecting a Graphics Subsystem

At any given moment, only a single 3Dfx Interactive graphics accelerator is active. The **grSstSelect()**, presented above, activates a specific unit. All Glide functions, with the exception of the **grGlide** family and **grSstSelect()**, operate on only the currently active subsystem. Note that the global Glide state is bound to each graphics subsystem independently. So, to set the constant color in each unit to the same value, for example, you must write a loop that selects each one in turn and sets the color, as shown in Example 3.3.

Example 3.3 Setting a state variable in all graphics subsystems.
Each graphics subsystem has its own version of the Glide state variables, including a constant color value that is used to clear the screen. The constant color is zero by default. The code fragment below cycles through all the units found by a previous call to ***grGet()****, setting the constant color to black.*

```
int i, n;

n = grGet(GR_NUM_BOARDS, sizeof(n), &n) ;
for ( i = 0; i < n; i++ )
{
   grSstSelect( i );
   grConstantColorValue( ~0 ); /* only affects SST “i” */
}
```

Opening Multiple Graphics Subsystems

grSstWinOpen() must be called once for each graphics subsystem that will be used. In Glide 3.0, the current graphics context must be closed (by calling **grSstWinClose**(), described below) before **grSstWinOpen()** can be called to open a context for another subsystem. Note that two graphics subsystems linked together in a scanline interleaving configuration are treated in software as a single unit.

Scanline Interleaved Graphics Subsystems

Two 3Dfx Interactive graphics accelerators can be wired together in a configuration known as *scanline interleaving*, which effectively doubles rasterization performance. From an application’s perspective, two graphics subsystems in a scanline-interleaved configuration are treated as if a single subsystem is installed in the system, including during unit selection, initialization, state management, texture download, etc.

Shutting Down

Before a new graphics context can be created, the previous one must be closed by calling **grSstWinClose()**.

FxBool **grSstWinClose(** *GrContext_t context* **)**

grSstWinClose() will fail, returning `FXFALSE`, if *context* is not a valid handle to a graphics context. Otherwise, it returns the state of Glide to the one following **grGlideInit()**, so that **grSstWinOpen()** can be called to open a new context.

After an application has completed using Glide and the graphics subsystem, proper shutdown must be performed. This allows Glide to de-allocate system resources like memory, timers, address space, and file handles that were used during program execution.

The function **grGlideShutdown()** shuts down Glide and all graphics contexts previously opened with **grSstWinOpen()**. It should be called only when an application is finished using Glide, and should not be executed unless **grGlideInit()** and **grSstWinOpen()** have already been called.

void **grGlideShutdown(** void **)**

Example 3.4 shows a minimal Glide program: it executes the four function calls that initialize the graphics subsystem and then terminates.

Example 3.4 A minimal Glide program.
The complete program below includes the Glide initialization and termination procedure and nothing else.

```
#include <glide.h>

int n;

void main(void)
{  GrContext_t context;
   grGlideInit(void);
   if (! grGet(GR_NUM_BOARDS, sizeof(n), &n))
      printf(“ERROR: no 3Dfx Interactive Graphics Accelerator!\n”);
   grSstSelect(0) ;
   context = grSstWinOpen( NULL, GR_RESOLUTION_640x480, GR_REFRESH_60HZ,
                           GR_COLORFORMAT_RGBA, GR_ORIGIN_LOWER_LEFT, 2, 0);
   grSstWinClose(context);
   grGlideShutdown();
}
```

The Display Buffer

Glide manages several logical hardware graphics buffers, all of which are based out of the same area of memory known as the “frame buffer”. Depending on the amount of memory installed on the hardware, the frame buffer is typically arranged as three logical units: the front buffer, the back buffer, and, optionally, the auxiliary buffer.

void **grRenderBuffer(** *GrBuffer_t buffer* **)**

grRenderBuffer() selects the buffer for primitive drawing and buffer clears. Valid values are `GR_BUFFER_FRONTBUFFER` and `GR_BUFFER_BACKBUFFER`; the default is `GR_BUFFER_BACKBUFFER`.

The auxiliary buffer in a 3Dfx Interactive graphics accelerator subsystem can be used either as a depth buffer, an alpha buffer, or as a third rendering buffer for triple buffering. The auxiliary buffer is not available on systems with 2MB of frame buffer DRAM running at 800×600. However, it is always available on systems with 4MB of frame buffer DRAM installed or with the screen resolution set to 640×480.

Triple buffering allows an application to continue rendering even when a swap buffer command is pending. When triple buffering is enabled an application can act as if the hardware is operating in double buffer mode; intricacies of dealing with the third buffer are hidden from the application by the hardware. Since the auxiliary buffer can serve only a single use, depth buffering, alpha buffering, and triple buffering are mutually exclusive.

An application selects the purpose of the auxiliary buffer implicitly whenever depth buffering, alpha buffering, or triple buffering are enabled. For example, if **grDepthBufferMode()** is called with a parameter other than `GR_DEPTHBUFFER_DISABLE` (see Chapter 7), it is assumed that the auxiliary buffer will be used for depth buffering. Similarly, **grSstWinOpen()** enables triple buffering; alpha buffering is enabled if **grAlphaBlendFunction()** selects a destination alpha blending factor (see Chapter 6) or **grColorMask()** enables writes to the alpha buffer. The release build of Glide does not check for contention of the auxiliary buffer. Unexpected results may occur if the auxiliary buffer is used for more that one function (e.g., both depth buffering and triple buffering are enabled). The debugging version of the library will report the contention.

Note that source alpha blending can coexist with depth or triple buffering, but destination alpha blending cannot.

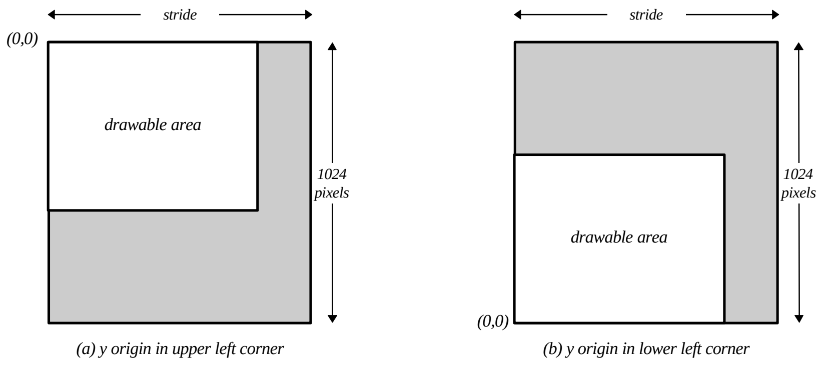
Table 3.3 Frame buffer resolution and configuration.
The frame buffer can be configured with two or three rendering buffers. In double buffer modes, an alpha or depth buffer can also be used. The available resolution depends on the amount of installed memory.

frame buffer memory	*double buffer mode*	*double buffer mode with 16-bit alpha/depth buffer*	*triple buffer mode*
2 Mbytes	800 by 600 by 16	640 by 480 by 16	640 by 480 by 16
4 Mbytes	800 by 600 by 16	800 by 600 by 16	800 by 600 by 16

Logical Layout of the Linear Frame Buffer

The frame buffer is logically organized as 1024 scanlines of 16 or 32-bit values, regardless of the amount of memory installed on the board, and is shown in Figure 3.2. Scanline length, or stride, is independent of screen resolution and dependent on the graphics hardware. The stride is returned in the *GrLfbInfo_t* structure, as described in Chapter 11. The data format within the frame buffer is programmable and is also described in detail in Chapter 11.

Figure 3.2 Logical layout of the linear frame buffer.
The frame buffer is logically organized as 1024 scanlines of 16 or 32-bit values, regardless of the amount of memory installed on the board and the screen resolution. The drawable area is a rectangular subset of the frame buffer; its location depends on the location of the y origin. The remainder of the board's memory (shaded area) is used as an auxiliary buffer that can be utilized as an alpha/depth buffer or as a third display buffer (triple buffering). This logical layout is independent of the user-specified origin location.



Masking Writes to the Frame Buffer

Writes to the frame buffer and depth buffer can be selectively disabled and enabled. The Glide functions **grColorMask()** and **grDepthMask()** control buffer masking: `FXTRUE` values allow writes to the associated buffer, and `FXFALSE` values disable writes to the associated buffer. Writes to the color and alpha buffers are controlled by **grColorMask()** whereas writes to the depth buffer are controlled by **grDepthMask()** (described in Chapter 7). Note that disabling writes to the alpha planes is the same as disabling writes to the depth planes, since they both share the same memory.

void **grColorMask**(*FxBool rgb*, *FxBool alpha*)

void **grDepthMask(** *FxBool enable* **)**

grColorMask() specifies whether the color and/or alpha buffers can or cannot be written to during rendering operations. If *rgb* is `FXFALSE`, for example, no change is made to the color buffer regardless of the drawing operation attempted. The *alpha* parameter is ignored if depth buffering is enabled since the alpha and depth buffers share memory.

grDepthMask() enables writes to the depth buffer.

The value of **grColorMask()** and **grDepthMask()** are ignored during linear frame buffer writes if the pixel pipeline is disabled (see Chapter 11). The default values are `FXTRUE`, indicating that the associated buffers are writable.

Swapping Buffers

In a double or triple buffered frame buffer, the next scene is rendered in a back buffer while the front buffer is being displayed. After an image has been rendered, it is displayed with a call to **grBufferSwap()**, which exchanges the front and back buffers every *swapInterval* vertical retraces. If the *swapInterval* is 0, then the buffer swap does not wait for vertical retrace. If the monitor frequency is 60 Hz, for example, a *swapInterval* of 3 results in a maximum frame rate of 20 Hz.

void **grBufferSwap**(int *swapInterval*)

A *swapInterval* of 0 may result in visual artifacts, such as 'tearing', since a buffer swap can occur during the middle of a screen refresh cycle. This setting is very useful in performance monitoring situations, as true rendering performance can be measured without including the time buffer swaps spend waiting for vertical retrace.

grBufferSwap() does not wait for the specified vertical blanking period; instead, it queues the buffer swap command and returns immediately. If the application is double buffering, the graphics subsystem will stop rendering and wait until the swap occurs before executing more commands. If the application is triple buffering and the third rendering buffer is available, then rendering commands will take place immediately in the third buffer.

A Glide application can poll the hardware using the **grGet()** function, described in Chapter 12, with argument `GR_PENDING_BUFFERSWAPS`, to determine the number of buffers waiting to be viewed, although this is generally not necessary.

The maximum value returned is 7, even though there may be more buffer swap requests in the queue. To minimize rendering latency in response to interactive input, **grGet(**`GR_PENDING_BUFFERSWAPS,`**…)** should be called in a loop once per frame until the returned value is less than some small number such as 1, 2, or 3.

Synchronizing with Vertical Retrace

Synchronization to vertical retrace is supported with the **grGet()** function with argument `GR_VIDEO_POSITION`, which returns the vertical and horizontal beam location. Vertical retrace is indicated by returning 0 for the vertical position.

Note that an application does not need to explicitly synchronize to vertical retrace if it only wishes to remove tearing artifacts. **grBufferSwap()** will automatically synchronize to vertical retrace if desired.

Monitoring Swapping Behavior

An application program can examine a history of swapping behavior: each entry shows the number of vertical retraces that occurred between the display of a frame and its predecessor. A call to **grGet(**`GR_NUM_SWAP_HISTORY_BUFFER,`**…)** returns the number of bytes of swapping history available. A call to **grGet(**`GR_SWAP_HISTORY,`**…)** returns the 4-byte entries and resets the recording buffer. Example 3.5shows and example.

Example 3.5 Retrieving the swapping history.
The code fragment below retrieves the swap history since the last time it was retrieved.

```
FxU32 sizeSwapHst, buffSwapHst[MAXBUFF];

grGet(GR_NUM_SWAP_HISTORY_BUFFER, 4, &sizeSwapHst);
grGet(GR_SWAP_HISTORY, sizeSwapHst << 2, buffSwapHst);
```

Clearing Buffers

The ability to clear a display buffer is fundamental to animation, since the remnants of a previously rendered scene must be reset before a new scene can be rendered. The hardware allows the back buffer and alpha or depth buffer to be cleared simultaneously.

A buffer clear fills pixels at twice the rate of triangle rendering or better. Therefore, the performance cost of clearing the buffer is, worse case, half the cost of rendering a rectangle. Clearing the buffer is not necessary when the scene paints a background that covers the entire area.

Buffers are cleared by calling **grBufferClear()**. The area within the buffer to be cleared is defined by **grClipWindow()**, described in the next chapter. The three parameters specify the values that are used to clear the display buffer (*color*), the alpha buffer (*alpha*), and the depth buffer (*depth*). Although the *color*, *alpha*, and *depth* parameters are always specified, the parameters actually used will depend on the current configuration of the hardware; the irrelevant parameters are ignored.

The *depth* parameter can be one of the depth constants found by calling **grGet()** with argument `GR_ZDEPTH_MIN_MAX` or `GR_WDEPTH_MIN_MAX`, or a direct representation of a value in the depth buffer. See Chapter 7 for more details.

void **grBufferClear(** *GrColor_t color*, *GrAlpha_t alpha*, *FxU32 depth* **)**

Any buffers that are enabled are automatically and simultaneously cleared by **grBufferClear()**. For example, if depth buffering is enabled (with **grDepthBufferMode()**, described in Chapter 7), the depth buffer is cleared to *depth*. If alpha buffering is enabled (with **grAlphaBlendFunction()**, described in Chapter 6), the alpha buffer is cleared to *alpha*. And if writes to the display buffer are enabled (with **grColorMask()**, described in Chapter 5), then it is cleared to *color*. If an application does not want a buffer to be cleared, it should mask off writes to the buffer using **grDepthMask()** and **grColorMask()** as appropriate.

Error Handling

Glide provides a family of error management functions to assist a developer with application debugging. This family of routines consists of Glide related error management (errors generated by Glide) and application level error management (errors generated by an application).

The debug build of Glide performs extensive parameter validation and resource checking. When an error condition is detected, a user-supplied callback function may be executed. This callback function is installed by calling **grErrorSetCallback()**. If no callback function is specified, a default error function that prints an error message to `stderr` is used.

typedef void (**GrErrorCallbackFnc_t*) (const char **string*, *FxBool fatal*)

void **grErrorSetCallback(***GrErrorCallbackFnc_t fnc*)**)**

The callback function accepts a string describing the error and a flag indicating if the error is fatal or recoverable. **grErrorSetCallback()** is relevant only when using the debugging version of Glide; the release build of Glide removes all internal parameter validation and error checking so the callback function will never be called.

4. Rendering Primitives

In This Chapter

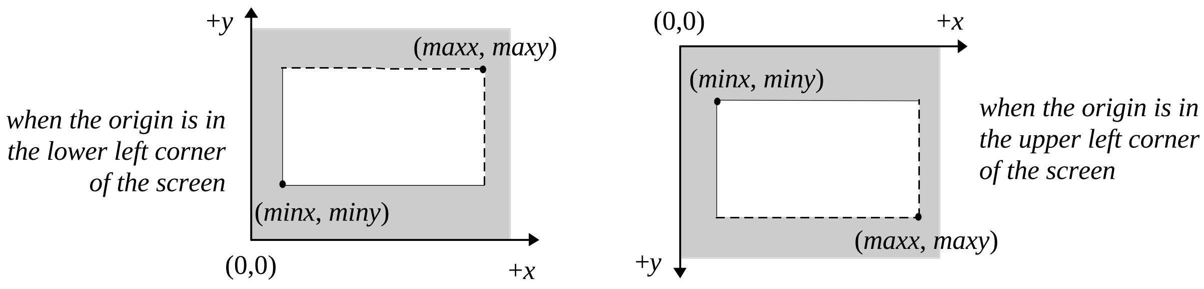
You will learn how to:

- ▼ establish a clipping window.
- ▼ draw a point, a line, a triangle, or a convex polygon on the screen.
- ▼ draw sets of points, lines, and triangles in a single operation.
- ▼ draw sets of connected lines and triangles in a single operation.
- ▼ cull back-facing polygons from the scene.
- ▼ draw anti-aliased points, lines, triangles, and convex polygons.

Clipping

The graphics hardware supports per-pixel clipping to an arbitrary rectangle that is defined with the Glide function **grClipWindow()**. Any pixels outside the clipping window are rejected. Values are inclusive for minimum *x* and *y* values and exclusive for maximum *x* and *y* values, as shown in Figure 4.1. The clipping window also specifies the area **grBufferClear()** will clear. (See Chapter 3.)

Figure 4.1 Specifying a clipping window.
The clipping window is defined by two pairs of integers in the range [0..1024) specifying the left and right edges and the top and bottom edges of the rectangle.


+y
(maxx, maxy)
when the origin is in the lower left corner of the screen
(minx, miny)
(0,0)
+x
(0,0)
+x
(minx, miny)
when the origin is in the upper left corner of the screen
(maxx, maxy)
+y

The **grClipWindow()** routine has four parameters that define the clipping rectangle. The values must be less than or equal to the current screen resolution and greater than or equal to 0; otherwise, they are ignored. Glide does not perform any geometric clipping outside of supporting a hardware clipping window. For optimal performance, an application should perform proper geometric clipping before passing any primitives to Glide. *The clipping window should not be used in place of true geometric clipping*.

void **grClipWindow**(*FxU32 minX*, *FxU32 minY*, *FxU32 maxX*, *FxU32 maxY*)

The default values for the clip window are the full size of the screen: (0,0,640,480) for 640×480 mode and (0,0,800,600) for 800×600 mode. To disable clipping, simply set the size of the clip window to the screen size. The clipping window should not be used for general purpose primitive clipping; since clipped pixels are processed but discarded, proper geometric clipping should be done by the application for best performance. The clip window should be used to prevent stray pixels that appear from imprecise geometric clipping. Note that if pixel pipeline is disabled, clipping is not performed on linear frame buffer writes (see Chapter 11 for more information).

Triangles

The triangle is the basic Glide rendering primitive. The Glide function **grDrawTriangle()** renders an arbitrarily oriented triangle. The arguments, *a*, *b*, and *c*, are pointers to vertices whose layout has been determined by the most recent call to **grVertexLayout()**, as described in Chapter 2.

void **grDrawTriangle(** const void **a*, const void **b*, const void **c* **)**

Triangles are rendered with the following filling rules:

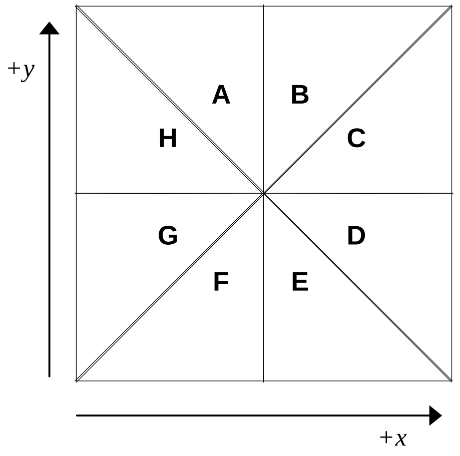
- Zero area triangles render zero pixels.
- Pixels are rendered if and only if their center lies within the triangle.

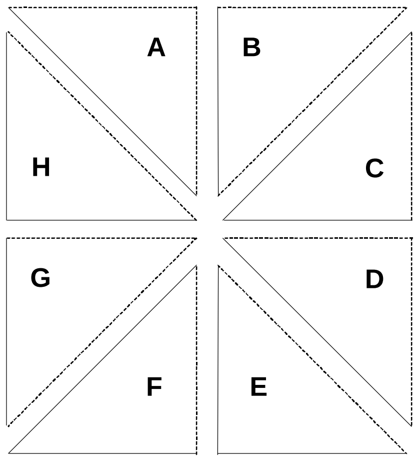
A pixel center is within a triangle if it is inside all three of the edges. When a pixel center lies exactly on an edge, it is inside the triangle if the edge is considered to be inside, and outside otherwise. Left edges are in; right edges are out. Horizontal edges with the smaller *y* value are in; those with a larger *y* value are out.

Figure 4.2 gives an example. Eight triangles are shown, all sharing a common vertex. Only one of the triangles renders the pixel whose center is the shared vertex. Can you guess which one?

The shared vertex is part of the “right edge” of triangles **A**, **H**, **G**, and **F**, and hence outside. It is part of the “top edge” (since the origin is in the lower left) of triangles **G**, **F**, **E** and **D**, and thus outside them as well. In triangle **B**, the vertex is on one inside edge and one outside edge and hence is considered to be outside the triangle. Only in triangle **C** does the vertex lie on two “inside” edges and thus lies inside the triangle.

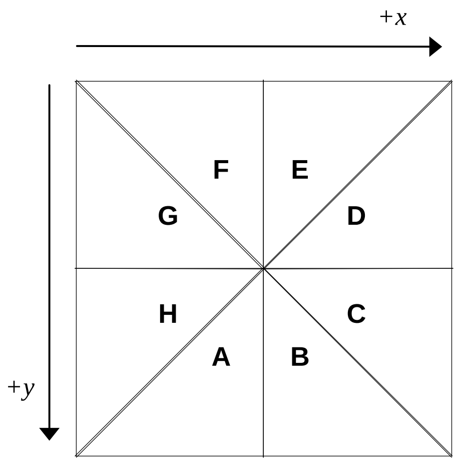
Figure 4.2 Pixel rendering.
Which of the eight triangles shown in diagram ***(a)*** *will render the pixel at the common vertex? In diagram* ***(b)****, solid edges are considered to be inside the triangle while dotted edges are outside. The top two diagrams are drawn with the origin in the lower left corner. The bottom row represents the other possibility: the origin is in the upper left corner. The two pairs of diagrams are mirror images of each other.*

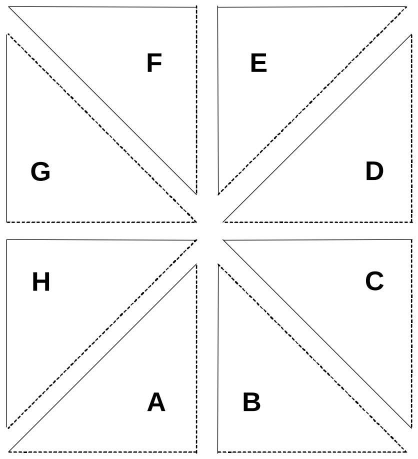




(a) *Which triangles will render the pixel in the center of the square? (If you like to think of the origin in the lower left corner, use the top row of diagrams; if you prefer an origin in the upper left corner, use the bottom row.)*

(b) *Pixels on solid edges lie inside the triangle; pixels on dotted lines do not. A vertex is inside the triangle (and hence, rendered) if both edges that radiate from it are inside the triangle. Thus, only triangle* ***C*** *will render the center point.*





Points

The Glide function **grDrawPoint()** renders a single point to the screen. The point is treated as a triangle with nearly coincident vertices (that is, a very small triangle) for rendering purposes. If many points will be rendered, noticeable performance improvement can be achieved by writing pixels directly to the frame buffer. (**grDrawPoint()** sends three vertices per point to the hardware and iterates along three edges; only one linear frame buffer write per point is required.)

void **grDrawPoint(** const void **a* **)**

The argument, *a*, is a pointer to a vertex whose layout has been determined by the most recent call to **grVertexLayout()**.

Example 4.1 A thousand points of light.
This code fragment clears the screen to black and then draws a thousand random points. By default, the rendering buffer is set to `GR_BUFFER_BACKBUFFER` and the color buffer is writable. The color white is made by specifying maximal values for red, green, and blue, and a quick way to do that is ~0. Some of the points will be clipped out: the random number generator selects points with coordinates in the range [0..1024]; the screen resolution is less than that. By default, the clipping window is set to the screen size.

```
typedef struct { float x, y; } myVertex;

int n;
myVertex p;

grVertexLayout(GR_PARAM_XY, 0, GR_PARAM_ENABLE);

/* clear the back buffer to black */
grBufferClear(0, 0, 0);

/* set color to white */
grColorCombine( GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
                GR_COMBINE_LOCAL_CONSTANT, GR_COMBINE_OTHER_NONE,FXFALSE );
grConstantColorValue(~0)

/* generate and draw 1000 random points */
for (n=0; n<1000; n++) {
   p.x = (float) (rand() % 1024);
   p.y = (float) (rand() % 1024);
   grDrawPoint(&p);
}
```

Lines

The Glide function **grDrawLine()** renders an arbitrarily oriented line segment. Enabled special effects (e.g., fog, blending, chroma-key, dithering, etc.) will affect a line’s appearance. The arguments, *a* and *b*, are pointers to vertices whose layout has been determined by the most recent call to **grVertexLayout()**.

void **grDrawLine(** const void **a*, const void **b* **)**

Drawing Sets of Disjoint Points, Lines, and Triangles

So far, we have talked about rendering commands that take one, two, or three vertex pointers as arguments and draw a single point, line, or triangle. Two more commands, **grDrawVertexArray()** and **grDrawVertexArrayContiguous()**, take an array of vertex pointers or of vertices and draw them according to a *mode* argument.

void **grDrawVertexArray(***FxU32 mode, FxU32 count,* void **pointers[]* **)**

void **grDrawVertexArrayContiguous(***FxU32 mode, FxU32 count,* void **vertex, FxU32 stride***)**

The first argument, *mode*, tells how to interpret the list of vertices. Valid values are `GR_POINTS`, `GR_LINES`, `GR_TRIANGLES`, `GR_LINE_STRIP`, `GR_TRIANGLE_STRIP`, `GR_TRIANGLE_FAN`, or `GR_POLYGON`, or two continuation modes, `GR_TRIANGLE_FAN_CONTINUE` and `GR_TRIANGLE_FAN_CONTINUE`. In this section, we will discuss the first three modes, which draw disjoint points, lines, and triangles. The other modes are discussed in later sections.

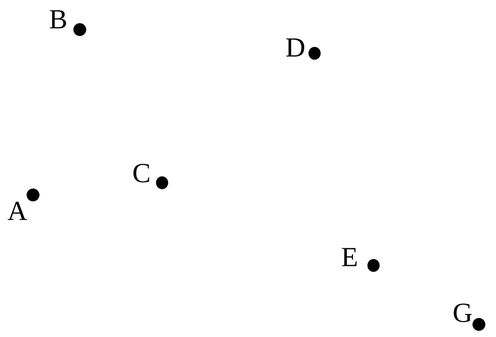
The second argument, *count*, gives the number of vertices to draw, and the final argument, *pointers*, is a pointer to an array of pointers to vertices. **grDrawVertexArrayContiguous()** assumes that all the vertices are stored in a linear array addressed by *vertex*, and that each vertex in the array is *stride* bytes long. In both cases, the *count* vertices are processed in the order given, according to *mode*.

Figure 4.3 gives a set of points and draws them with six of the modes.

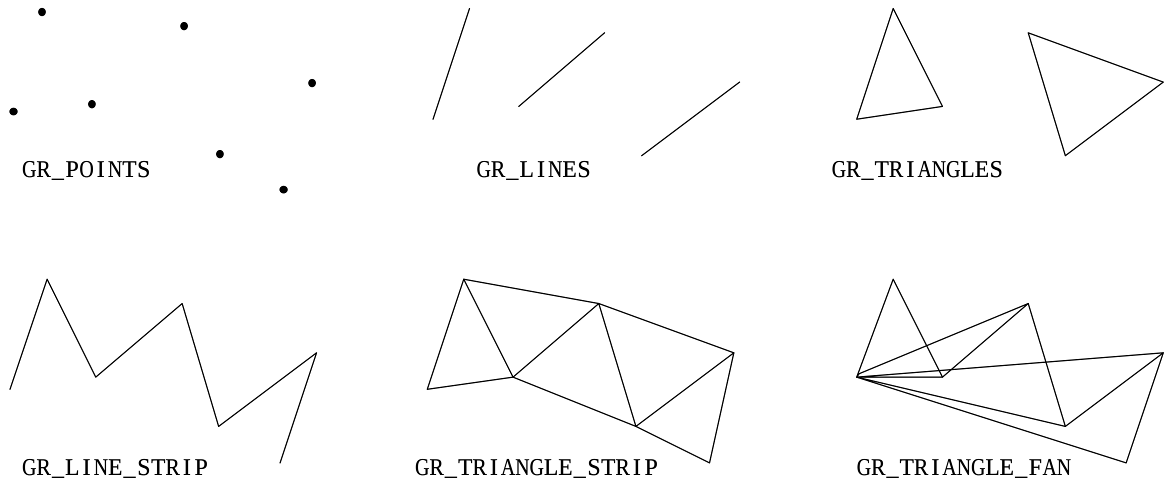
Drawing Sets of Connected Lines and Triangles

A *line strip* is a sequence of line segments in which each line segment shares an endpoint with the previous one. A *triangle strip* is a sequence of triangles in which each triangle (after the first one) shares two vertices with its predecessor. A *triangle fan* is a strip in which all triangles have one vertex in common. (See Figure 4.4.)

Figure 4.3 Vertex arrays.
Suppose we have the following points:



They are stored in alphabetic order in a contiguous array of vertices and drawn with each of the possible modes, yielding the shapes below:

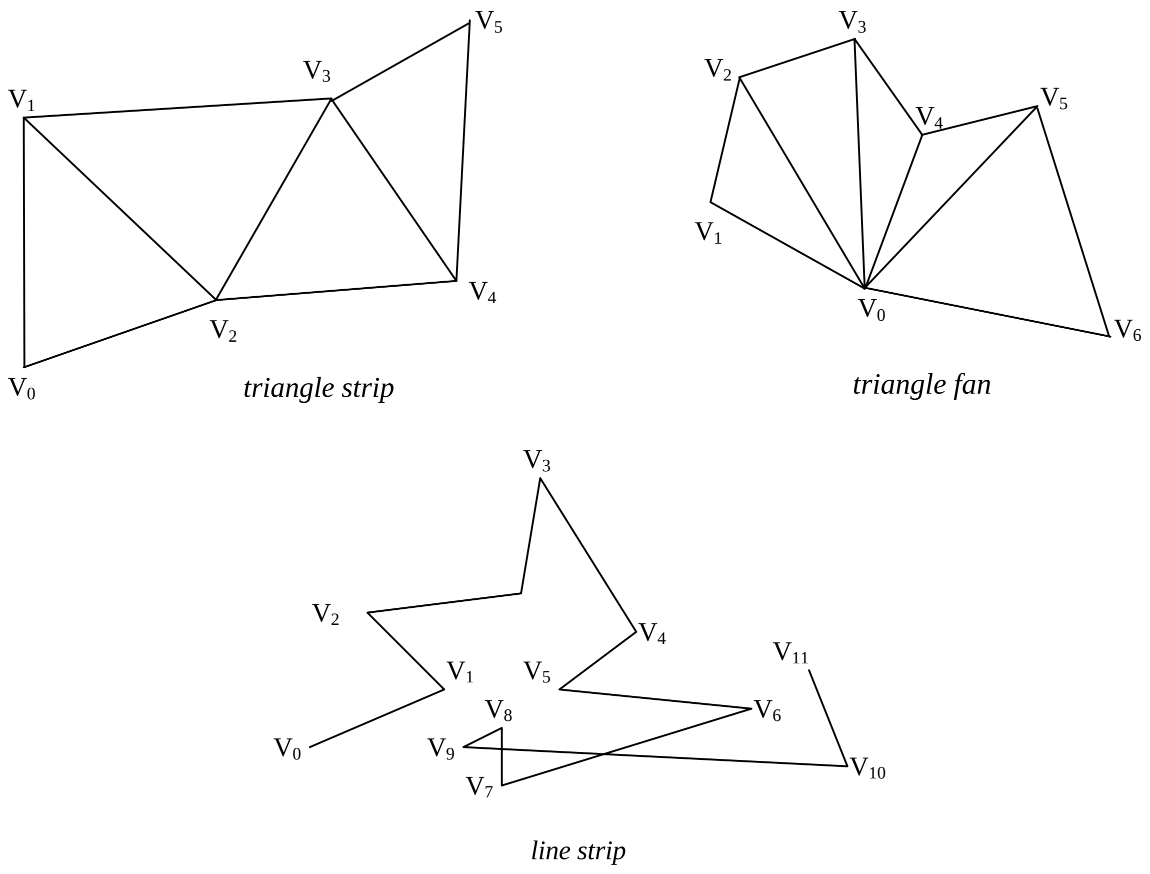


This set of points cannot be drawn in `GR_POLYGON` *mode as the resulting polygon would not be convex. An example later in the chapter uses the indirection of* ***grDrawVertexArray()*** *to draw a polygon by discarding points C and E.* `GR_TRIANGLE_FAN_CONTINUE` *and* `GR_TRIANGLE_STRIP_CONTINUE` *are continuation modes and are described later.*

Figure 4.4 Line strips, triangle strips, and triangle fans.
*The first line segment in a **line strip** provides two vertices. Subsequent line segments require only one new vertex, since their starting point is the endpoint of the previous one.*

*The first triangle in a **triangle strip** provides three vertices. Subsequent triangles in the strip share two vertices with their predecessor. All the triangles in a **fa**n share one vertex, the first one in the list. Furthermore, each triangle shares a second vertex with its predecessor.*

*When **grDrawVertexArray()** or **grDrawVertexArrayContiguous()** is used to draw the triangle aggregate, the mode argument identifies it as a strip or fan: the distinction is important because the shared vertices are handled differently. In a strip, each new vertex replaces the oldest of the previous three vertices. In a fan, the first vertex remains in use for the whole fan, and each new vertex replaces the oldest of the other two.*


V_0 V_1 V_2 V_3 V_4 V_5
triangle strip
V_0 V_1 V_2 V_3 V_4 V_5 V_6
triangle fan
V_0 V_1 V_2 V_3 V_4 V_5 V_6 V_7 V_8 V_9 V_{10} V_{11}
line strip

Two additional drawing modes, `GR_TRIANGLE_FAN_CONTINUE`, and `GR_TRIANGLE_STRIP_CONTINUE`, allow you to interrupt the rendering of a triangle strip or fan to do computations, then resume where you left off. The use of the continuation modes is subject to the following restrictions:

- **grDrawVertexArray(**`GR_TRIANGLE_STRIP_CONTINUE,`**…)** must follow either a **grDrawVertexArray(**`GR_TRIANGLE_STRIP,`**…)** or **grDrawVertexArray(**`GR_TRIANGLE_STRIP_CONTINUE,`**…)** command. Similarly, **grDrawVertexArray(**`GR_TRIANGLE_FAN_CONTINUE,`**…)** must be preceded by either **grDrawVertexArray(**`GR_TRIANGLE_FAN,`**…)** or **grDrawVertexArray(**`GR_TRIANGLE_FAN_CONTINUE,`**…)**.
- Intervening commands may not change Glide state. For example, the following sequence is not valid:

  grEnable(`GR_AA_ORDERED`**)**;
  grDrawVertexArray(`GR_TRIANGLE_FAN,`**…)**;
  grDisable(`GR_AA_ORDERED`**)**;   /* Wrong! No state changes between continuations */
  grDrawVertexArray(`GR_TRIANGLE_FAN_CONTINUE,`**…)**;

- No intervening rendering commands are allowed. For example, the following sequence is not valid:

  grDrawVertexArray(`GR_TRIANGLE_FAN,`**…)**;
  grDrawVertexArray(`GR_POINTS,`**…)**;   /* Wrong! No other rendering between continuations */
  grDrawVertexArray(`GR_TRIANGLE_FAN_CONTINUE,`**…)**;

Example 4.2 Using triangle continuation.
The code fragment below draws a triangle strip in three stages.

```
/* draw two triangles */
grDrawVertexArray(GR_TRIANGLE_STRIP, 4, pointers);
/* continue to draw a triangle strip, using the last two vertices in the
 * previous one
 */
grDrawVertexArray(GR_TRIANGLE_STRIP_CONTINUE, 1, pointers+4);
/* continue to draw one triangle using the last two vertices in the
previous triangle */
grDrawVertexArray(GR_TRIANGLE_STRIP_CONTINUE, 1, pointers+5);
```

This code fragment uses continuation to draw a triangle fan.

```
/* draw two triangles */
grDrawVertexArray(GR_TRIANGLE_FAN, 4, pointers);
/* continue to draw one triangle using the first vertex and last vertex
in the previous triangle */
grDrawVertexArray(GR_TRIANGLE_FAN_CONTINUE, 1, pointers+4);
/* continue to draw one triangle using the first vertex and last vertex
in the previous triangle */
grDrawVertexArray(GR_TRIANGLE_FAN_CONTINUE, 1, pointers+5);
```

Convex Polygons

A polygon is a planar area enclosed by a closed loop of line segments, specified by their endpoints. While the hardware does not render polygons directly, Glide provides a set of polygon rendering functions that are optimized for the hardware. The polygons rendered by the Glide functions are subject to some strong restrictions:

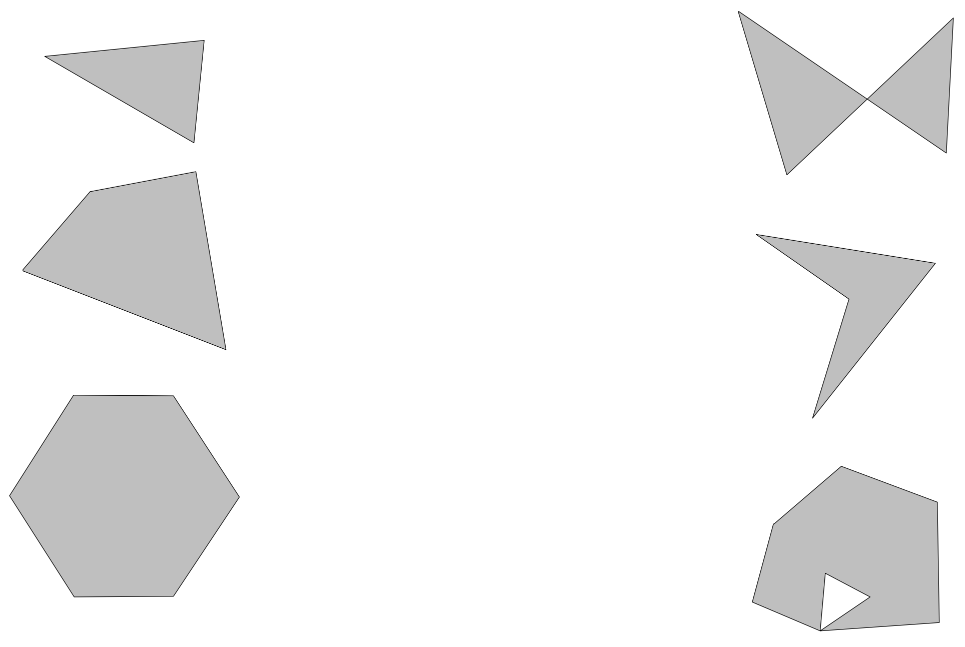
- The edges of the polygon cannot intersect.
- The polygon must be convex, that is, have no indentations. (*The glossary at the end of this manual gives a precise definition of convexity*.)

Figure 4.5 shows some examples of both valid and invalid polygons.

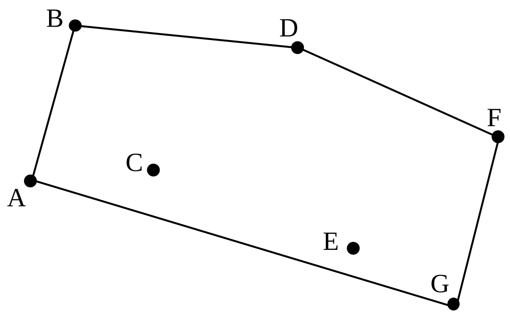
Figure 4.5 Polygons.

Valid polygons are convex and planar.

Invalid polygons have intersecting edges, indentations, or non-planar coordinates.

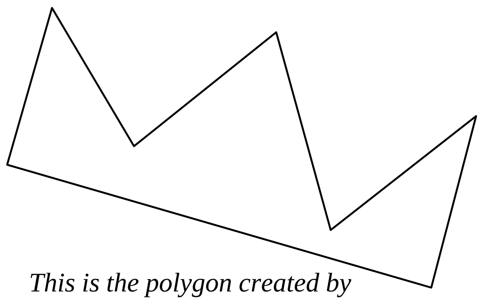


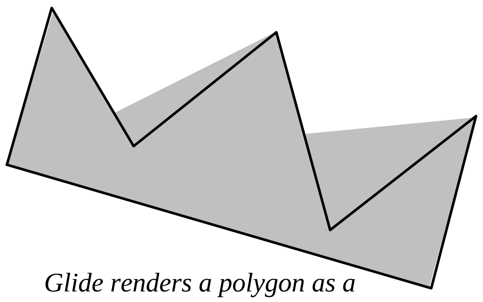
Example 4.3 Drawing a convex polygon in Glide 3.0.
The code fragment below assumes that the seven vertices shown below and in Figure 4.3 have been defined in an array of `myVertex` *structures called* `verts`*. By creating an array of* `myVertex` *pointers that drop out the C and E vertices, a convex polygon can be drawn.*



```
typedef struct { … } myVertex;
myVertex verts[7];

static struct myVertex *vlist[5] = {
     verts[0], verts[1],
     verts[3].
     Verts[5], verts[6] };

grDrawVertexArray(GR_POLYGON, 5, vlist);
```

So why not draw a polygon using all seven vertices? Because the resulting polygon is not convex. Polygons are rendered as a triangle fan. The illustration below demonstrates the fact that drawing a polygon that is not convex may yield unexpected results!


This is the polygon created by connecting all seven vertices in order. The deep indentations of the crown are not convex.


Glide renders a polygon as a triangular fan. The shaded area is what is drawn; the lines outline what was desired.

PORTING NOTE

Convex polygons are defined by an ordered set of vertices and drawn by calling **grDrawVertexArray(**`GR_POLYGON,`**…)** or **grDrawVertexArrayContiguous(**`GR_POLYGON,`**…)** in Glide 3.0. Table 4.1 provides guidance for porting the polygon rendering routines from Glide 2.x to the new regime.

Table 4.1 Porting obsolete grDrawPolygon() commands to Glide 3.0.
Glide 3.0 is the first release to support ***grDrawVertexArray()****. Six old routines for drawing polygons have been made obsolete by* ***grDrawVertexArray()****:* ***grDrawPolygon()****,* ***grDrawPlanarPolygon()****,* ***grDrawPolygonVertexList()****,* ***grDrawPlanarPolygonVertexList()****,* ***grAADrawPolygon()****, and* ***grAADrawPolygonVertexList()****. The table below shows how to convert calls to the obsolete routines with calls to* ***grDrawVertexArray()****. It assumes that the old GrVertex structure has been defined both syntactically and with calls to* ***grVertexLayout()****.*

old	*new*
`grDrawPlanarPolygon(`*nVerts*, *ilist*, *vlist*`)`	`grDrawVertexArray(GR_POYGON,` *nVerts*, *vlist* sorted according to *ilist*`)`
`grDrawPolygon(` *nVerts*, *ilist*, *vlist*`)`	`grDrawVertexArray(GR_POLYGON,` *nVerts*, *vlist* sorted according to *ilist*`)`
`grDrawPlanarPolygonVertexList(`*nVerts*, *vlist*`)`	`grDrawVertexArrayContiguous(GR_POLYGON,` *nVerts*, *vlist*, `sizeof(GrVertex))`
`grDrawPolygonVertexList(`*nVerts*, *vlist*`)`	`grDrawVertexArrayContiguous(GR_POLYGON,` *nVerts*, *vlist*, `sizeof(GrVertex))`
`grAADrawPolygon(`*nVerts*, *vlist*`)`	`grEnable(AA_ORDERED);` `grDrawVertexArray(GR_POLYGON,` *nVerts*, *vlist* sorted according to *ilist*`)`
`grAADrawPolygonVertexList(`*nVerts*, *vlist*`)`	`grEnable(AA_ORDERED);` `grDrawVertexArrayContiguous(GR_POLYGON,` *nVerts*, *vlist*, `sizeof(GrVertex))`

Example 4.4 L'embarras des richesses: The more alternatives, the more difficult the choice.
(Abbé D'Allainval, 1726). The code fragment below draw three triangles, It initializes an array of eight vertices, `vpool[8]`, and an array of pointers to them, verts. Vertex vpool[1] is shared by all three of the triangles; and two of them use vpool[2].

```
struct vert {
    FxFloat x,y,z,w;    // x,y,z(unused),1/w
    FxFloat s,t;        // s/w,t/w
} vpool[8];

static struct vert *verts[9] = {
              vpool+0, vpool+1, vpool+2,
              vpool+1, vpool+2, vpool+3,
              vpool+7, vpool+1, vpool+5};

//--------------------------------------------------------------
// set the scene
grCoordinateSpace(GR_WINDOW_COORDS);
grVertexLayout(GR_PARAM_XY, 0, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_Q0, 12, GR_PARAM_ENABLE);
grVertexLayout(GR_PARAM_ST0, 16, GR_PARAM_ENABLE);

// transform and deposit vertices into vpool
vpool[0].x = x;
vpool[0].y = y;
vpool[0].w = oow = 1.0F/w;
vpool[0].s = s*oow;
vpool[0].t = t*oow;
// similar for other vertices…
```

Here are three different ways to draw the same three triangles. Method 1: Draw them as three independent triangles.

```
static struct vert *verts[9] = {
              vpool+0, vpool+1, vpool+2,
              vpool+1, vpool+2, vpool+3,
              vpool+7, vpool+1, vpool+5};

grDrawVertexArray(GR_TRIANGLES, 9, verts);
```

Method 2: Draw them as a two triangle strip (remember the shared vertices?) and an independent triangle.

```
// 2 meshed triangles and 1 independent traingle
static struct vert *verts[7] = {
              vpool+0, vpool+1, vpool+2, vpool+3
              vpool+7, vpool+1, vpool+5};


grDrawVertexArray(GR_TRIANGLE_STRIP, 4, verts);
grDrawVertexArray(GR_TRIANGLES, 3, verts+4);
```

Method 3: Draw them using the **grDrawTriangle()** *command.*

```
grDrawTriangle(vpool+0, vpool+1, vpool+2);
grDrawTriangle(vpool+1, vpool+2, vpool+3);
grDrawTriangle(vpool+7, vpool+1, vpool+5);
```

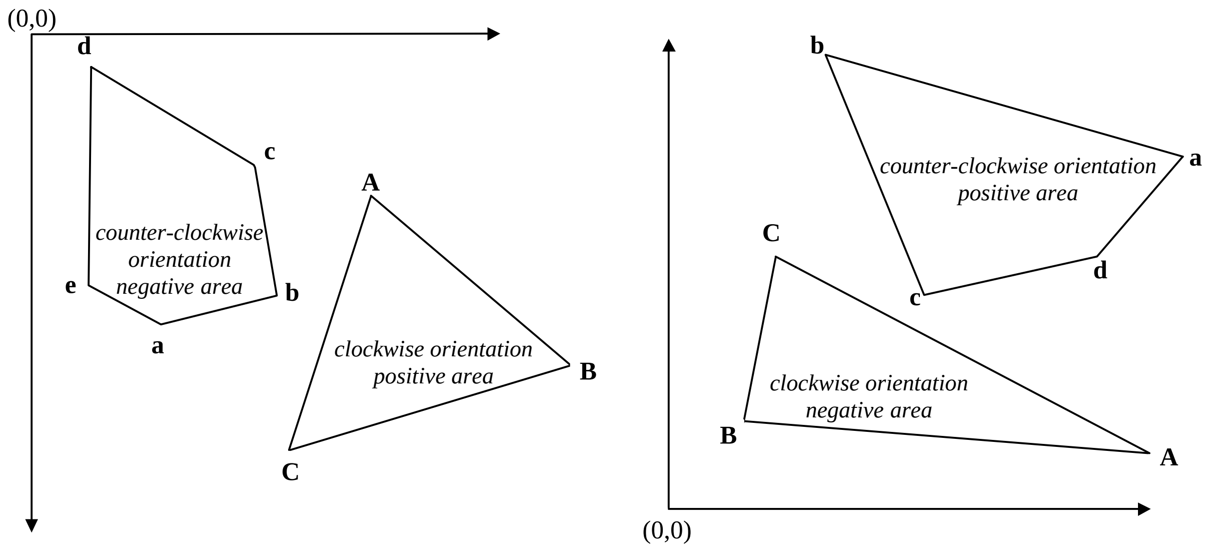
Method 4: Draw them as a contiguous triangle strip and an independent triangle.

```
grDrawVertexArrayContiguous(GR_TRIANGLE_STRIP, 4, vpool, sizeof(struct
vert));
grDrawTriangle(vpool+7, vpool+1, vpool+5);
```

Backface Culling

Glide supports backface culling based on the signed area of a polygon. When Glide renders a polygon, the first step is to divide the polygon into triangles, the hardware rendering primitive. Figure 4.6 shows a pair of triangles whose vertices have been labeled according to the rule given above.

Figure 4.6 Polygon orientation and the sign of the area.
The polygons on the left are defined relative to an origin in the upper left corner; the ones on the right have the origin in the lower left corner. Clockwise and counter-clockwise refer to the direction that the vertices are traversed in alphabetical order.


(0,0)
d
c
b
a
e
counter-clockwise orientation negative area
A
B
C
clockwise orientation positive area
b
a
c
d
counter-clockwise orientation positive area
C
B
A
clockwise orientation negative area
(0,0)

The sign of the area of the triangle can be used for backface culling (quickly discarding triangles that won't be visible on the screen before they are rendered). Because the area must be computed anyway, this is a cheap way to cull. However, removing back-facing triangles earlier may be advantageous. For example, if back face removal is performed before lighting, then the computationally expensive lighting calculations for invisible triangles can be skipped.

The Glide function **grCullMode()** has one parameter, a mode that can be set to `GR_CULL_NONE`, `GR_CULL_NEGATIVE`, or `GR_CULL_POSITIVE`. When the culling mode is `GR_CULL_NONE`, the default value, all polygons are rendered to the screen regardless of their signed area. Otherwise, if the sign of the area matches the *mode*, then the triangle is rejected. **grCullMode()** assumes that `GR_CULL_POSITIVE` corresponds to a counter-clockwise orientation when the origin is in the lower left corner of the screen, and a clockwise oriented triangle when the origin is in the upper left corner, as shown in Table 4.2.

void **grCullMode(** *GrCullMode_t mode* **)**

Note that **grCullMode()** has no effect on points and lines, but does effect the rendering of triangles and polygons.

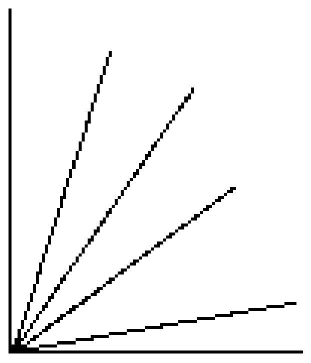
Table 4.2 The location of the origin affects triangle orientation and the sign of its area.

if the origin location is	*and the triangle orientation is*	*then the sign of the area is*
`GR_ORIGIN_LOWERLEFT`	clockwise	negative
`GR_ORIGIN_LOWERLEFT`	counter-clockwise	positive
`GR_ORIGIN_UPPERLEFT`	clockwise	positive
`GR_ORIGIN_UPPERLEFT`	counter-clockwise	negative

Anti-aliasing

If you look closely and critically at lines drawn on the screen, particularly lines that are nearly horizontal or nearly vertical, they may appear to be jagged. The screen is a grid of pixels and the line is approximated by lighting spans of pixels on that grid. The jaggedness is called aliasing; examples of aliased lines are shown in Figure 4.7(a). Anti-aliasing techniques reduce the jaggedness, as shown in Figure 4.7(b), by partially coloring neighboring pixels to simulate partial pixel coverage.

Figure 4.7 Aliased and anti-aliased lines.
These lines are drawn at a resolution of 50 pixels/inch in order to exaggerate the jagged edges of the aliased lines and highlight the widening and blending in the anti-aliased lines. These lines are examples of the general concepts; if you replicate this drawing on the screen, the results may be different in detail.


(a) aliased lines have jagged edges

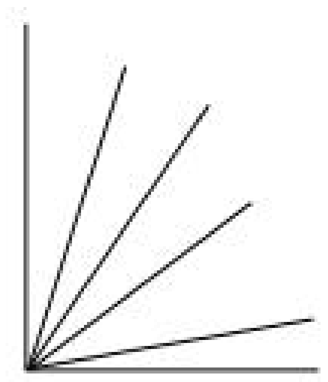
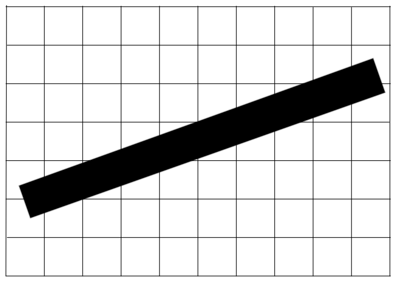
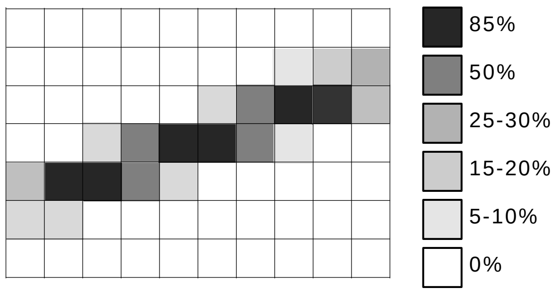

(b) anti-aliased lines soften the edges by shading surrounding pixels

Figure 4.8 shows an angled line segment one pixel wide, superimposed on a pixel grid. Some pixels are almost completely covered by the line, while others have only a small corner involved. Glide's anti-aliasing routines compute a coverage value for each pixel and uses that in combination with the source and destination alpha values to blend the pixel color.

Figure 4.8 Pixel coverage and lines.



(a) This angled one-pixel wide line segment doesn't cover any pixel completely.



(b) The shaded squares are touched by the line segment at the left; the shade of gray filling each square represents the area covered by the line.

Glide draws anti-aliased points, lines, triangles, and polygons by setting up the alpha iterator so that it represents pixel coverage. You must correctly configure the alpha combine unit (discussed in detail in Chapter 6) and enable alpha blending before using any of the anti-aliased drawing commands. The code segment in

PORTING NOTE

Previous versions of Glide contained a set of commands to draw anti-aliased primitives. Only one of these has been retained in Glide 3: grAADrawTriangle(). It operates independently of the GR_AA_ORDERED mechanism. A description follows.

Example 4.5 details the proper sequence of Glide commands that must precede the actual anti-aliased drawing commands. Briefly, you must:

- Set the alpha combine unit to produce *iterated alpha*.
- Set the alpha blending function. Blending functions are specified for source and destination color components and for source and destination alpha values, and the choice of function depends on whether the scene is rendered front to back or back to front.
- *Set the alpha value for each vertex*. The chosen alpha value should represent the transparency of the object being rendered, with opaque objects setting alpha to 255. This alpha value is multiplied by the pixel coverage to obtain the final alpha value used for alpha blending.
- Call **grEnable(**`GR_AA_ORDERED`**)** to enable anti-aliasing.
- Sort the vertices by depth and draw with a **grDraw** routine. *You cannot draw anti-aliased strips and fans*.

PORTING NOTE

Previous versions of Glide contained a set of commands to draw anti-aliased primitives. Only one of these has been retained in Glide 3: **grAADrawTriangle()**. It operates independently of the `GR_AA_ORDERED` mechanism. A description follows.

Example 4.5 Drawing an anti-aliased triangle.
The alpha combine unit must be configured to produce an iterated alpha value in order to use the Glide anti-aliasing drawing functions. Consider the following code segment a recipe for success in this chapter; the alpha combine unit, alpha buffering, and alpha blending are the subject of Chapter 6.

The objects in the picture must be pre-sorted on depth. The alpha blending factors depend on whether the scene is drawn from front to back or back to front. The first code shows the alpha blending factors if the scene is drawn from front to back.

```
/* set alpha combine unit to produce an iterated alpha */
grAlphaCombine(GR_COMBINE_SCALE_OTHER, GR_COMBINE_FACTOR_ONE, GR_LOCAL_NONE,
   GR_LOCAL_INTERATED, FXFALSE);

/* blend colors based on alpha */
grAlphaBlendFunction(GR_BLEND_ ALPHA_SATURATE, GR_BLEND_ONE, GR_BLEND_
SATURATE,               GR_BLEND_ONE);

grEnable(GR_AA_ORDERED);
/* draw the scene using the grDraw routines */
```

Substitute the alpha blending factors shown below if the scene is drawn from back to front.

```
grAlphaBlendFunction(GR_BLEND_SRC_ALPHA, GR_BLEND_ONE_MINUS_SRC_ALPHA,
   GR_BLEND_ZERO, GR_BLEND_ZERO);
```

void **grAADrawTriangle (** *GrVertex *va*, *GrVertex *vb*, *GrVertex *vc*,
FxBool aaAB, *FxBool aaBC*, *FxBool aaCA*
)

grAADrawTriangle() has three more arguments than its aliased counterpart **grDrawTriangle()**. The arguments, *aaAB*, *aaBC*, and *aaBC* are Boolean values that allow the edges of the triangle to be selectively anti-aliased.

Glide draws a triangle with the specified edges anti-aliased by setting up the alpha iterator so that it represents pixel coverage. **grAlphaCombine()** must select iterated alpha and **grAlphaBlendFunction()** should select `GR_BLEND_SRC_ALPHA`, `GR_BLEND_ONE_MINUS_SCR_ALPHA` as the RGB blend functions and `GR_BLEND_ZERO`, `GR_BLEND_ZERO` as the alpha blend functions if sorting from back to front and `GR_BLEND_ALPHA_SATURATE`, `GR_BLEND_ONE` as the RGB blend functions and `GR_BLEND_SATURATE`, `GR_BLEND_ONE` as the alpha blend functions if sorting from front to back. Opaque anti-aliased primitives *must* set alpha=255 in the vertex data. Transparent anti-aliased primitives are drawn by setting alpha to values less than 255; this alpha value is multiplied by the pixel coverage to obtain the final alpha value for alpha blending.

If there is a steep gradient in a particular color space (i.e., green goes from 255.0 to 0.0 in a small number of pixels), then there will be visual anomalies at the edges of the resultant anti-aliased triangle. The workaround for this ‘feature’ is to reduce the gradient by increasing small color components and decreasing large ones. This can be demonstrated by changing the values of *maxColor* and *minColor* in `test25` of the Glide distribution. Note that this ‘feature’ is only present when the color combine mode includes iterated RGB or alpha as one of the parameters in the final color.

5. *Color and Lighting*

In This Chapter

You will learn about:

- specifying colors.
- configuring the color combine unit that produces shading and lighting effects.
- drawing a flat-shaded object.
- drawing a smooth-shaded object.
- simulating various lighting effects.

Specifying Colors

A color consists of three or four *color components*: *red*, *green*, *blue*, and optionally, *alpha*. The color component values should be clamped to the range [0..255] where 0 is black and 255 is maximum intensity.

The color components are packed together into a word to form a color. Glide supports four different color byte orderings, defined in the enumerated type *GrColorFormat_t* (see Figure 3.1 for a pictorial representation). Color byte ordering determines how linear frame buffer writes and color arguments are interpreted and is established in the call to **grSstWinOpen()** when Glide and the graphics hardware are initialized (see Chapter 3).

The *GrColor_t* type definition represents a packed color value and is used in routines that set a constant color: **grBufferClear()** (see Chapter 3), **grConstantColorValue()** (described below), **grFogColorValue()** and **grChromakeyValue()** (both described in Chapter 8).

void **grConstantColorValue**(*GrColor_t color*)

Glide refers to a global constant color when performing flat-shaded primitive rendering, set with **grConstantColorValue()**. The default value is `0xFFFFFFFF`.

Vertex colors are specified as individual color components, each stored as an *FxFloat* value, or as four bytes packed into a word.

Dithering

The graphics hardware represents color internally as 32-bit quadruplets in a format specified by the color format argument passed to **grSstWinOpen()** (see Chapter 3). This color is eventually dithered to 16-bit RGB for storage in the frame buffer, then expanded and (optionally) filtered up to 24-bits for final display. From an application's perspective, the 32-to-16-bit RGB dithering operation is transparent.

Dithering is a technique for increasing the perceived range of colors in an image by applying a pattern to surrounding pixels to modify their color values. When viewed from a distance, these colors appear to blend into an intermediate color that can't be represented directly. Dithering is similar to the half-toning used in black and white publications to produce shades of gray.

void **grDitherMode**(*GrDitherMode_t mode*)

grDitherMode() selects the form of dithering the hardware uses when converting 24-bit RGB values to the 16-bit RGB color buffer format. Valid values are `GR_DITHER_DISABLE`, `GR_DITHER_2x2`, and `GR_DITHER_4x4`. `GR_DITHER_DISABLE` forces a simple truncation that may result in noticeable banding. `GR_DITHER_2x2` uses a 2x2 ordered dither matrix, and `GR_DITHER_4x4` uses a 4x4 ordered dither matrix.

The default dithering mode is `GR_DITHER_4x4`.

The Color Combine Unit

TAKE NOTE

*Control of high level rendering functions is managed by three functions, **grColorCombine()**, **grAlphaCombine()** (see Chapter 6), and **grTexCombine()** (described in Chapter 9). While the three routines will be presented individually, settings for one function can potentially affect the inputs to the other routines.*

The color combine unit computes an RGB color for each pixel as it is rendered. User-selected inputs are added, blended, and/or scaled to produce flat or smooth (Gouraud) shading with optional lighting effects. The color combine unit computes each RGB color component separately, but all three are computed using the same formula. The alpha combine unit computes the alpha component and is discussed in the next chapter.

The color combine unit computes a color component as

$$c = f * a + b$$

where *c* is the red, green, or blue color component, *f* is a scale factor, and *a* and *b* are sums and differences of the various input choices.

The Glide routine that configures the color combine unit is **grColorCombine()**. It specifies the function that computes the color and selects the inputs.

```
void grColorCombine( GrCombineFunction_t   func,
                     GrCombineFactor_t     factor,
                     GrCombineLocal_t      local,
                     GrCombineOther_t      other,
                     FxBool                invert
                   )
```

Fourteen combining functions are defined in the *GrCombineFunction_t* enumerated type; one is selected with *func*, the first argument to **grColorCombine()**. Table 5.1 gives the symbolic names and formulas for each color combine function.

The *f* variable in the combining formulas is defined by *factor*, the second argument to **grColorCombine()**. The choices for this scale factor are given in Table 5.2. Note that alpha values from the texture combine unit ($a_{texture}$) or specified by **grAlphaCombine()** arguments (a_{local} and a_{other}) appear in some of the scale factors.

Table 5.1 Configuring the color combine unit.
The first argument to ***grColorCombine()****, func, specifies the color combine function; its value is chosen from among the symbols list in the left hand column of the table below. The right hand column gives the combining function that corresponds to each symbolic name.* ***F*** *is a scale factor and is defined by the factor argument to* ***grColorCombine()****.* c_{local} *and* c_{other} *are specified by the third and fourth arguments. Some of the formulas specify an alpha value,* a_{local}*, that is defined in the* ***grAlphaCombine()*** *function described in the next chapter.*

color combine function	*computed color*
GR_COMBINE_FUNCTION_ZERO	0
GR_COMBINE_FUNCTION_LOCAL	c_{local}
GR_COMBINE_FUNCTION_LOCAL_ALPHA	a_{local}
GR_COMBINE_FUNCTION_SCALE_OTHER GR_COMBINE_FUNCTION_BLEND_OTHER	$f * c_{other}$
GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL	$f * c_{other} + c_{local}$
GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL_ALPHA	$f * c_{other} + a_{local}$
GR_COMBINE_FUNCTION_SCALE_OTHER_MINUS_LOCAL	$f * (c_{other} - c_{local})$
GR_COMBINE_FUNCTION_SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL GR_COMBINE_FUNCTION_BLEND	$f * (c_{other} - c_{local}) + c_{local}$ $\equiv f * c_{other} + (1 - f) * c_{local}$
GR_COMBINE_FUNCTION_SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL_ALPH A	$f * (c_{other} - c_{local}) + a_{local}$
GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL GR_COMBINE_FUNCTION_BLEND_LOCAL	$f * (- c_{local}) + c_{local}$ $\equiv (1 - f) * c_{local}$
GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL_ALPHA	$f * (- c_{local}) + a_{local}$

Table 5.2 The color combine function scale factor.
The second argument to ***grColorCombine()****, factor, specifies a scale factor, called f in the formulas delineated in Table 5.1; its value is chosen from among the symbols listed in the left hand column of the table below. The right hand column gives the scale factor that corresponds to each symbolic name.* C_{local} *is specified by the third argument to* ***grColorCombine()****,* a_{local} *and* a_{other} *are defined in the* ***grAlphaCombine()*** *function described in the next chapter, and* $a_{texture}$ *comes from the texture combine unit, described in Chapter 9.*

combine factor	*scale factor (f)*
GR_COMBINE_FACTOR_NONE	*unspecified*
GR_COMBINE_FACTOR_ZERO	0
GR_COMBINE_FACTOR_LOCAL	$c_{local} / 255$
GR_COMBINE_FACTOR_OTHER_ALPHA	$a_{other} / 255$
GR_COMBINE_FACTOR_LOCAL_ALPHA	$a_{local} / 255$
GR_COMBINE_FACTOR_TEXTURE_ALPHA	$a_{texture} / 255$
GR_COMBINE_FACTOR_ONE	1
GR_COMBINE_FACTOR_ONE_MINUS_LOCAL	$1 - c_{local} / 255$
GR_COMBINE_FACTOR_ONE_MINUS_OTHER_ALPHA	$1 - a_{other} / 255$
GR_COMBINE_FACTOR_ONE_MINUS_LOCAL_ALPHA	$1 - a_{local} / 255$
GR_COMBINE_FACTOR_ONE_MINUS_TEXTURE_ALPHA	$1 - a_{texture} / 255$

The third and fourth arguments to **grColorCombine()** set values for the c_{local} and c_{other} variables that appear in the combining functions; the choices are shown in Table 5.3. Iterated colors are computed by iterating the colors specified in the vertices passed to drawing functions. The texture color comes from the texture combine unit (see Chapter 9), and the constant color is set by **grConstantColorValue()** (described earlier in this chapter).

The *func* formula computes the red, green, and blue color components. The result of the computation is clamped to [0..255] and may be bit-wise inverted, based on the final argument to **grColorCombine()**, *invert*. Inverting the bits in a color component *c* is the same as computing (1.0 – *c*) for floating point values in the range [0..1] or (255 – *c*) for 8-bit values in the range [0..255].

Table 5.3 Choosing local and other colors for the color combine unit.
*The third and fourth arguments to **grColorCombine()**, local and other, specify the sources for the c_{local} and c_{other} values that appear in the color combine formulas delineated in Table 5.1; their values are chosen from among the symbols in the tables below. Iterated colors are computed by iterating the colors specified in the vertices passed to drawing functions. The texture color comes from the texture combine unit, and the constant color is set by **grConstantColorValue()**.*

local combine source	*local color (c_{local})*
`GR_COMBINE_LOCAL_NONE`	unspecified color
`GR_COMBINE_LOCAL_ITERATED`	iterated vertex color (Gouraud shading)
`GR_COMBINE_LOCAL_CONSTANT`	constant color

other combine source	*other color (c_{other})*
`GR_COMBINE_OTHER_NONE`	unspecified color
`GR_COMBINE_OTHER_ITERATED`	iterated vertex color (Gouraud shading)
`GR_COMBINE_OTHER_TEXTURE`	color from texture map
`GR_COMBINE_OTHER_CONSTANT`	constant color

The color combine unit computes the source color for the remainder of the rendering pipeline. The default color combine mode is

```
grColorCombine( GR_COMBINE_FUNCTION_SCALE_OTHER,
                GR_COMBINE_FACTOR_ONE,
                GR_COMBINE_LOCAL_ITERATED,
                GR_COMBINE_OTHER_ITERATED
                FXFALSE );
```

A series of examples follows.

Example 5.1 Drawing a constant color triangle.
The code segment below draws a teal colored triangle by setting the constant color and directing the color combine unit to use it as c_{other}. The code assumes that the vertex layout has already been established.

```
myVertex a, b, c;

/* set color to teal (assumes ARGB format) */
grConstantColorValue( (100<<8) + 150 );

/* configure color combine unit for constant color */
grColorCombine(GR_COMBINE_FUNCTION_SCALE_OTHER, GR_COMBINE_FACTOR_ONE,
    GR_COMBINE_LOCAL_NONE, GR_COMBINE_OTHER_CONSTANT, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

The code segment below will produce the same result as the one above, but it points c_{local} to the constant color.

```
myVertex a, b, c;

/* set color to teal (assumes ARGB format) */
grConstantColorValue( (100<<8) + 150);

/* configure color combine unit for constant color */
grColorCombine(GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
    GR_COMBINE_LOCAL_CONSTANT, GR_COMBINE_OTHER_NONE, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

Example 5.2 Drawing a flat-shaded triangle.
The code segment below draws a flat-shaded triangle using the color for vertex A. It sets the constant color to the vertex color and proceeds as in the previous example. The code assumes that the vertex layout has already been established.

```
myVertex A, B, C;

/* set constant color to color of vertex A (assumes ARGB format) */
grConstantColorValue((((int)A.a)<<24)||(((int)A.r)<<16)||(((int)A.g)<<8)||(int)
A.b);

/* configure color combine unit for constant color */
grColorCombine(GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
    GR_COMBINE_LOCAL_CONSTANT, GR_COMBINE_OTHER_NONE, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&A, &B, &C);
```

Alternatively, you could set the colors of all three vertices to the colors in Vertex A and proceed as in the next example.

```
myVertex A, B, C;

/* set all vertices to same color */
B.a = C.a = A.a;
B.r = C.r = A.r;
B.g = C.g = A.g;
B.b = C.b = A.b;

/* configure color combine unit for iterated colors */
grColorCombine(GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
    GR_COMBINE_LOCAL_ITERATED, GR_COMBINE_OTHER_NONE, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&A, &B, &C);
```

Example 5.3 Drawing a smooth-shaded triangle.
In this example, a Gouraud-shaded triangle is drawn, with the color blending smoothly from vertex to vertex. The hardware automatically iterates the colors to achieve the smooth shading. The color combine unit is configured with c_{local} set to the iterated color components. The code assumes that the vertex layout has already been established.

```
myVertex a, b, c;

/* configure color combine unit for iterated color */
grColorCombine(GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
     GR_COMBINE_LOCAL_ITERATED, GR_COMBINE_OTHER_NONE, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

Alternatively, c_{other} can be directed at the iterated color components.

```
myVertex a, b, c;

/* configure color combine unit for iterated color */
grColorCombine(GR_COMBINE_FUNCTION_SCALE_OTHER, GR_COMBINE_FACTOR_ONE,
     GR_COMBINE_LOCAL_NONE, GR_COMBINE_OTHER_ITERATED, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

Example 5.4 Drawing a flat-shaded textured triangle.
The following code produces a textured flat-shaded triangle using the constant color. The code assumes that the vertex layout has already been established.

```
myVertex a, b, c;

/* set color to teal (assumes ARGB format) */
grConstantColorValue( (100<<8) + 150);

/* configure color combine unit for iterated color */
grColorCombine(GR_COMBINE_FUNCTION_SCALE_OTHER, GR_COMBINE_FACTOR_LOCAL,
   GR_COMBINE_LOCAL_CONSTANT, GR_COMBINE_OTHER_TEXTURE, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

Example 5.5 Drawing a smooth-shaded textured triangle.
This example configures the color combine unit for a smoothly shaded textured triangle by directing c_{local} to the iterated color and c_{other} to the output from the texture combine unit. The code assumes that the vertex layout has already been established.

```
myVertex a, b, c;

/* configure color combine unit for iterated color */
grColorCombine(GR_COMBINE_FUNCTION_SCALE_OTHER, GR_COMBINE_FACTOR_LOCAL,
   GR_COMBINE_LOCAL_ITERATED, GR_COMBINE_OTHER_TEXTURE, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

Example 5.6 Drawing a smooth-shaded triangle with specular lighting.
This example produces a textured triangle with specular lighting provided by iterating the RGB color. The code assumes that the vertex layout has already been established.

```
myVertex a, b, c;

/* configure color combine unit for iterated color */
grColorCombine(GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL,
GR_COMBINE_FACTOR_ONE,  GR_COMBINE_LOCAL_ITERATED, GR_COMBINE_OTHER_TEXTURE,
FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

Example 5.7 Drawing a smooth-shaded textured triangle with specular highlights.
*By using the alpha component to model monochrome specular highlights, you can produce shiny, textured, smooth-shaded triangles ((texture RGB * iterated RGB) + iterated a). The code assumes that the vertex layout has already been established.*

```
myVertex a, b, c;

/* configure color combine unit for iterated color */
grColorCombine(GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL_ALPHA,
   GR_COMBINE_FACTOR_LOCAL, GR_COMBINE_LOCAL_ITERATED,
GR_COMBINE_OTHER_TEXTURE,                     FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

Example 5.8 Drawing a smooth-shaded triangle with monochrome diffuse and colored specular lighting.
*Alternatively, monochrome diffuse lighting and colored specular lighting can be produced by using the alpha component to model monochrome diffuse lighting and iterated RGB to model colored specular lighting ((texture RGB * iterated a) + iterated RGB). Iterated alpha is chosen to be either a_{local} or a_{other} with a call to* ***grAlphaCombine()*** *that is not shown here. In the first code segment, iterated alpha is assumed to be available as a_{local}. The code assumes that the vertex layout has already been established.*

```
myVertex a, b, c;

/* configure color combine unit for iterated color */
grColorCombine(GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL,
   GR_COMBINE_FACTOR_LOCAL_ALPHA, GR_COMBINE_LOCAL_ITERATED,
   GR_COMBINE_OTHER_TEXTURE, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

Alternatively, iterated alpha can be specified for a_{other} in ***grAlphaCombine()****. In that case the following* ***grColorCombine()*** *configuration is needed.*

```
myVertex a, b, c;

/* configure color combine unit for iterated color */
grColorCombine(GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL,
   GR_COMBINE_FACTOR_OTHER_ALPHA, GR_COMBINE_LOCAL_ITERATED,
   GR_COMBINE_OTHER_TEXTURE, FXFALSE);

/* assumes that some coordinates have been assigned to a, b, and c */
grDrawTriangle(&a, &b, &c);
```

Other Color Combine Options

The routine **grAlphaControlsITRGBLighting()** can be used to specify that if the high order bit of $a_{texture}$ is 1, then the constant color set by **grConstantColorValue()** is used instead of the iterated RGB values. This is useful if a portion of a texture is to appear to be illuminated from behind the surface, instead of by an external light source.

void **grAlphaControlsITRGBLighting(** *FxBool enable* **)**

When enabled, the normal color combine controls for local color (c_{local}) are overridden, and the most significant bit of texture alpha ($a_{texture}$) selects between iterated vertex RGB and the constant color set by **grConstantColorValue()**. By default, this alpha controlled lighting mode is disabled. Table 5.4 shows how c_{local} is determined.

Table 5.4 Overriding the local color when the high order bit of $a_{texture}$ is set.
*You can get hybrid effect between smooth and flat shading by using **grAlphaControlsITRGBLighting()** to enable a technique whereby the high order bit of $a_{texture}$ is used to switch c_{local} between iterated RGB and the constant color. The state table below shows how the c_{local} value is determined.*

*when **enable** is*	*and the high order bit of $a_{texture}$ is*	*the local color c_{local} will be*
`FXTRUE`	`0`	iterated RGB
`FXTRUE`	`1`	**grConstantColorValue()**
`FXFALSE`	`0`	set by **grColorCombine()**
`FXFALSE`	`1`	set by **grColorCombine()**

Some possible uses for this mode are self-lit texels and specular paint. If a texture contains texels that represent self-luminous areas, such as windows, then multiplicative lighting can be disabled for these texels as follows. Choose a texture format that contains one bit of alpha and set the alpha for each texel to 1 if the texel is self-lit. Set the Glide constant color to white and enable alpha-controlled lighting mode. Finally, set up texture lighting by multiplying the texture color by iterated RGB, where iterated RGB is the *local* color in the color combine unit. When a texel's alpha is 0, the texture color will be multiplied by the local color, which is iterated RGB. This applies lighting to the texture. When a texel's alpha is 1, the texture color will be multiplied by the Glide constant color that was previously set to white, so no lighting is applied.

If the color combine unit is configured to add iterated RGB to a texture for the purpose of a specular highlight, then texture alpha can be used as specular paint. In this example, the Glide constant color is set to black and iterated RGB iterates the specular lighting. Where a texel's alpha is 0, the texture color will be added to iterated RGB and specular lighting is applied to the texture. Where a texel's alpha is 1, the texture color will be added to the Glide constant color that was previously set to black, so no lighting is applied. The result is that the alpha channel in the texture controls where specular lighting is applied to the texture and specularity can be *painted* onto the texture in the alpha channel.

Gamma Correction

By default, Glide does not perform gamma correction (i.e., a linear ramp is used). However, gamma correction is available. The **guGammaCorrectionRGB()** function computes a hardware-dependent gamma correction table.

void **guGammaCorrectionRGB**(*FxFloat red, FxFloat green, FxFloat blue*)

guGammaCorrectionRGB() computes a gamma correction curve for each color component using the following formula:

$$C_{gamma} = [(C_{fb}/255)^{1/gamma}]*255$$

The red, green, and blue gamma values are positive floating point numbers in the range [0.0..20.0]. Typical values are 1.3 to 2.2. The default value is 1.0 (i.e. a linear ramp is used).

While it is not recommended, an application can cook up its own gamma correction table and download it to the hardware using **grLoadGammaTable()**.

void **grLoadGammaTable**(*FxU32 nEntries,* const *FxU32 *red,* const *FxU32 *green,* const *FxU32 *blue*)

The first argument, nEntries, is the number of elements in each of the three arrays of color values. The other three arguments are pointers to arrays of red, green, and blue values, respectively, that will be interpolated to generate an output gamma value.

If *nEntries* is less than the size of the hardware-dependent gamma table, the first part of the table is overwritten by the new values; if *nEntries* is greater than the gamma table size, the excess elements are discarded. The size of the gamma table may be obtained by calling **grGet(**`GR_GAMMA_TABLE_ENTRIES`**)**. The entries in the gamma table must be monotonically increasing in each color component or the results are undefined. It is strongly recommended that **guGammaCorrectionRGB()** be used instead of **grLoadGammaTable()**.

PORTING NOTE

guGammaCorrectionRGB() is new to Glide 3.0, replacing **grGammaCorrectionValue()**.**grLoadGammaTable()** is also new, and allows an application to use a customized gamma correction table. However, it is strongly recommended that **guGammaCorrectionRGB()** be used instead.

---

6. *Using the Alpha Component*

In This Chapter

Several different rendering techniques using the alpha component of the color are discussed. You will learn about:

- specifying alpha values.
- configuring the alpha combine unit that produces alpha values for pixels being rendered.
- using the auxiliary buffer to store alpha values.
- alpha blending, a technique for creating translucent objects in a scene.
- alpha testing, a technique for accepting or rejecting pixels based on their alpha value.

Specifying Alpha

Alpha values, like the red, green, and blue components of a color, are 8-bit values in the range [0..255]. Glide maintains a constant alpha value as part of the constant color described in the previous chapter that is set with **grConstantColorValue()**. Alpha values, if used, are part of the user-defined vertex layout defined with calls to **grVertexLayout()**, as described in Chapters 2 and 4.

The Alpha Combine Unit

TAKE NOTE

*Control of high level rendering functions is managed by three functions, **grColorCombine()**, **grAlphaCombine()** (see Chapter 6), and **grTexCombine()** (described in Chapter 9). While the three routines are presented individually, settings for one function can potentially affect the inputs to the other routines.*

The alpha combine unit is similar to the color combine unit that produces RGB values for the pixel being rendered. A user-selectable combining function specifies a scale factor, and *local* and *other* alpha values, and a formula for combining them to produce a new alpha value. The α_{local} and α_{other} inputs selected by the arguments to **grAlphaCombine()** can also be used in the scale factor chosen by **grColorCombine()**, described in the previous chapter.

```
void grAlphaCombine( GrCombineFunction_t func,
                     GrCombineFactor_t factor,
                     GrCombineLocal_t local,
                     GrCombineOther_t other,
                     FxBool invert
```

)

Table 6.1 lists the possible values for *func*, the first argument to **grAlphaCombine()**. The *f* that appears in the formulas in Table 6.1 is a scale factor that is chosen by the second argument, *factor*. Table 6.2 lists the possible scale factors. a_{local} and a_{other} are chosen by the third and fourth arguments, *local* and *other*; the candidates are listed in Table 6.3. As with **grColorCombine()**, the final argument, *invert*, is a Boolean that is set if a bit-wise inversion of the computed alpha value is desired. Inverting the bits in a color component *c* is the same as computing (1.0 – *c*) for floating point color values in the range [0..1] or (255 – *c*) for 8-bit color values in the range [0..255].

The default alpha combine unit configuration is

```
grAlphaCombine( GR_COMBINE_FUNCTION_SCALE_OTHER,
                GR_COMBINE_FACTOR_ONE,
                GR_COMBINE_LOCAL_NONE,
                GR_COMBINE_OTHER_CONSTANT,
                FXFALSE
               );
```

Two examples in the previous chapter, Example 5.7 and Example 5.8, use the a_{local} or a_{other} value.

Table 6.1 Combining functions for alpha.
*The first argument to **grAlphaCombine()**, func, specifies the alpha combine function; its value is chosen from among the symbols list in the left hand column of the table below. The right hand column gives the combining function that corresponds to each symbolic name. **f** is a scale factor and is defined by the factor argument to **grAlphaCombine()**. a_{local} and a_{other} are specified by the third and fourth arguments.*

combine function	*computed alpha*
`GR_COMBINE_FUNCTION_ZERO`	0
`GR_COMBINE_FUNCTION_LOCAL`	a_{local}
`GR_COMBINE_FUNCTION_LOCAL_ALPHA`	a_{local}
`GR_COMBINE_FUNCTION_SCALE_OTHER` `GR_COMBINE_FUNCTION_BLEND_OTHER`	$f * a_{other}$
`GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL`	$f * a_{other} + a_{local}$
`GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL_ALPHA`	$f * a_{other} + a_{local}$
`GR_COMBINE_FUNCTION_SCALE_OTHER_MINUS_LOCAL`	$f * (a_{other} - a_{local})$
`GR_COMBINE_FUNCTION_SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL` `GR_COMBINE_FUNCTION_BLEND`	$f * (a_{other} - a_{local}) + a_{local}$ $\equiv f * a_{other} + (1 - f) * a_{local}$
`GR_COMBINE_FUNCTION_SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL_ALPHA`	$f * (a_{other} - a_{local}) + a_{local}$
`GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL` `GR_COMBINE_FUNCTION_BLEND_LOCAL`	$f * (- a_{local}) + a_{local}$ $\equiv (1 - f) * a_{local}$
`GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL_ALPHA`	$f * (- a_{local}) + a_{local}$

Table 6.2 Scale factors for the alpha combine function.
*The second argument to **grAlphaCombine()**, factor, specifies a scale factor, called **f** in the formulas delineated in Table 6.1; its value is chosen from among the symbols listed in the left hand column of the table below. The right hand column gives the scale factor that corresponds to each symbolic name.* $\boldsymbol{a}_{\textbf{local}}$ *and* $\boldsymbol{a}_{\textbf{other}}$ *are defined by the third and fourth arguments to **grAlphaCombine()** and* $\boldsymbol{a}_{\textbf{texture}}$ *comes from the texture combine unit, described in Chapter 9.*

combine factor	*scale factor (f)*
`GR_COMBINE_FACTOR_NONE`	*unspecified*
`GR_COMBINE_FACTOR_ZERO`	0
`GR_COMBINE_FACTOR_LOCAL`	a_{local} / 255
`GR_COMBINE_FACTOR_OTHER_ALPHA`	a_{other} / 255
`GR_COMBINE_FACTOR_LOCAL_ALPHA`	a_{local} / 255
`GR_COMBINE_FACTOR_TEXTURE_ALPHA`	$a_{texture}$ / 255
`GR_COMBINE_FACTOR_ONE`	1
`GR_COMBINE_FACTOR_ONE_MINUS_LOCAL`	1 – a_{local} / 255
`GR_COMBINE_FACTOR_ONE_MINUS_OTHER_ALPHA`	1 – a_{other} / 255
`GR_COMBINE_FACTOR_ONE_MINUS_LOCAL_ALPHA`	1 – a_{local} / 255
`GR_COMBINE_FACTOR_ONE_MINUS_TEXTURE_ALPHA`	1 – $a_{texture}$ / 255

Table 6.3 Specifying local and other alpha values.
*The third and fourth arguments to **grAlphaCombine()**, local and other, specify the sources for the* a_{local} *and* a_{other} *values that appear in the alpha combine formulas delineated in Table 6.1 and in the color combine formulas shown in Table 5.1 and Table 5.2; their values are chosen from among the symbols in the tables below. Iterated alpha values are computed by iterating the alpha specified in the vertex structures passed to drawing functions. The texture alpha comes from the texture combine unit, and the constant alpha is set by **grConstantColorValue()**.*

local combine source	*local alpha (a_{local})*
`GR_COMBINE_LOCAL_NONE`	unspecified α
`GR_COMBINE_LOCAL_ITERATED`	iterated vertex α
`GR_COMBINE_LOCAL_CONSTANT`	constant α
`GR_COMBINE_LOCAL_DEPTH`	high 8 bits from iterated vertex *z*

other combine source	*other alpha (a_{other})*
`GR_COMBINE_OTHER_NONE`	unspecified α
`GR_COMBINE_OTHER_ITERATED`	iterated vertex α
`GR_COMBINE_OTHER_TEXTURE`	α from texture map
`GR_COMBINE_OTHER_CONSTANT`	constant α

Alpha Buffering

As pixels are rendered, a full 32-bit RGBA color is maintained internally. At the end of the rendering pipeline, the 24-bit RGB portion is dithered to 16 bits and stored in the display buffer. The alpha value component is discarded, unless the auxiliary buffer is being used as an alpha buffer.

With alpha buffering enabled, the graphics hardware stores an 8-bit alpha value for each pixel in the auxiliary buffer. To enable alpha buffering, set the *alpha* parameter of **grColorMask()** or blend using a function that calls for a destination alpha (see the following section for a discussion of alpha blending). Since the auxiliary buffer can only serve a single use at a time, depth buffering, alpha buffering, and triple buffering are mutually exclusive. If depth buffering is currently enabled (by calling **grDepthMask()** with argument `FXTRUE`), the *alpha* parameter specified in a **grColorMask()** call is ignored.

void **grColorMask**(*FxBool rgb*, *FxBool alpha*)

The alpha buffer is cleared by calling **grBufferClear()**. If alpha buffering is enabled, then the alpha buffer is cleared using the *alpha* parameter. The graphics display buffer and alpha buffer can be cleared simultaneously.

void **grBufferClear(** *GrColor_t color*, *GrAlpha_t alpha*, *FxU32 depth* **)**

In the anti-aliasing discussion in Chapter 4, alpha was used as a pixel coverage value for objects being rendered. Alpha blending is then used to blur the edge color with the background color and reduce unsightly “jaggies”.

The final example in this chapter, Example 6.3, shows another way to use the alpha buffer. In this case, a background scene is drawn with one alpha value, a polygonal cropping window is drawn with a second alpha value, and a foreground is mapped into the cropping window by discarding parts of the new scene that fall outside the cropping window. The example uses the alpha combine unit, alpha buffering, and alpha blending.

Alpha Blending

In Chapter 4, routines to draw anti-aliased points, lines, triangles and polygons were presented. They use *alpha blending* to smooth the jagged edges.

PORTING NOTE

Previous versions of Glide contained a set of commands to draw anti-aliased primitives. Only one of these has been retained in Glide 3: **grAADrawTriangle().** It operates independently of the `GR_AA_ORDERED` mechanism. A description follows.

Example 4.5 calls **grAlphaBlendFunction()** to configure alpha blending to accomplish anti-aliasing.

Another use for alpha blending is to create translucent objects in a scene. Without blending, a newly calculated color value will overwrite any color value already computed for that pixel and stored in the frame buffer. With blending, the alpha value is used to combine the new color value with the previous one so that the previous color “shows through”.

Think of the RGB values of a pixel as its color, and the A, or alpha, value as its opacity. Transparent or translucent objects have lower opacity values than opaque objects. For example, objects seen through a window are less defined than those viewed directly, but are still visible (unlike objects behind a solid wall). The window glass has a color and a small alpha value that is used to scale the window color before adding it to the existing color.

The graphics hardware supports alpha blending of pixels. When alpha blending is enabled, the alpha value of a pixel is used to combine the color value of the pixel being processed with that of the pixel already stored in the frame buffer.

Alpha blending allows an application to control the degree to which the two pixels have their colors blended, i.e., alpha blending allows translucent surfaces. The alpha component of a pixel represents its opacity; transparent or translucent surfaces have lower opacity than opaque ones. An alpha value of `0x00` corresponds to absolute transparency and an alpha value of `0xFF` corresponds to absolute opacity.

When using alpha blending for translucency/transparency, a scene must be sorted so that translucent/transparent surfaces are rendered correctly.

Just as with the color combine and alpha combine functions, the color components can be blended differently than the alpha component. The blending functions are defined as follows:

$$c_{dst} \leftarrow (c_{src} \cdot \mathbf{f}_{src}) + (c_{dst} \cdot \mathbf{f}_{dst})$$

$$a_{dst} \leftarrow (a_{src} \cdot \mathbf{g}_{src}) + (a_{dst} \cdot \mathbf{g}_{dst})$$

where c_{dst} is the RGB color of the destination pixel, c_{src} is the incoming source pixel RGB, and $\mathbf{f}_{src}$ and $\mathbf{f}_{dst}$ are the source and destination blending factors for the RGB components. Similarly, α_{dst} is the alpha value of the destination pixel, α_{src} is the incoming alpha value, and $\mathbf{g}_{src}$ and $\mathbf{g}_{dst}$ are the source and destination blending factors for the alpha component. Note that the current value of the destination pixel is used to compute the blended value that will overwrite it. The source of incoming alpha and color are determined by **grAlphaCombine()** and **grColorCombine()** respectively. C_{dst} and a_{dst} are clamped to the range [0..255].

The manner in which incoming pixels (source) are combined with the existing pixel (destination) is defined by two blending factors. These factors are controlled by the Glide function **grAlphaBlendFunction()**.

```
void grAlphaBlendFunction( GrAlphaBlendFnc_t rgbSrcFactor,
                           GrAlphaBlendFnc_t rgbDestFactor,
                           GrAlphaBlendFnc_t alphaSrcFactor,
                           GrAlphaBlendFnc_t alphaDestFactor
                         )
```

The first two arguments specify blending factors for the RGB components while the third and fourth arguments give the blending factors for the alpha component. The choices for all source and destination blending factors are shown in Table 6.4.

Alpha blending that requires a destination alpha is mutually exclusive of either depth buffering or triple buffering. Attempting to use `GR_BLEND_DST_ALPHA`, `GR_BLEND_ONE_MINUS_DST_ALPHA`, or `GR_BLEND_ALPHA_SATURATE` when depth buffering or triple buffering are enabled will have undefined results.

Example 6.1 Blending two images, part I.
In this example, two images are blended so that the final color of each pixel is the sum of colors from the two images.

```
grAlphaBlendFunction(GR_BLEND_ONE, GR_BLEND_ZERO, GR_BLEND_ONE,
GR_BLEND_ZERO);

/* draw the first image */

grAlphaBlendFunction(GR_BLEND_ONE, GR_BLEND_ONE, GR_BLEND_ONE,
GR_BLEND_ZERO);

/* draw the second image */
```

Example 6.2 Blending two images, part II.
In this example, two images are blending so that the final color of each pixel is 75% of the first image and 25% of the second. When the second image is drawn, alpha is given a constant value of ¼ by setting the constant color and pointing the a_{other} *in the alpha combine unit to it.*

```
grAlphaBlendFunction(GR_BLEND_ONE, GR_BLEND_ZERO, GR_BLEND_ONE,
GR_BLEND_ZERO);


/* draw the first image */

/* assumes RGBA format for colors */
grConstantColorValue(64);


grAlphaCombine(GR_COMBINE_FUNCTION_BLEND_OTHER, GR_COMBINE_FACTOR_ONE,
              GR_COMBINE_LOCAL_NONE, GR_COMBINE_OTHER_CONSTANT, FXFALSE);


grAlphaBlendFunction(GR_BLEND_SRC_ALPHA, GR_BLEND_ONE_MINUS_SRC_ALPHA,
              GR_BLEND_ONE, GR_BLEND_ZERO);


/* draw the second image */
```

Table 6.4 Alpha blending factors.
Four blending factors are specified in the ***grAlphaBlendFunction()****. The* ***rgbSrcFactor*** *and* ***alphaSrcFactor*** *choices are given in the first table. The specified factors are multiplied by the incoming RGBA values from the color and alpha combine units and added to the product of the destination factors and the alpha values stored in the alpha buffer. The possible destination factors are shown in the second table.*

For alpha source and destination blend function factor parameters, only `GR_BLEND_ZERO` *and* `GR_BLEND_ONE` *are supported.*

*if **rgbSrcFactor** or **alphaSrcFactor** is*	*the source blending factor $\mathbf{f}_{src}$ or $\mathbf{g}_{src}$ is*
`GR_BLEND_ZERO`	0
`GR_BLEND_ONE`	1
`GR_BLEND_DST_COLOR`	$c_{dst}/255$
`GR_BLEND_ONE_MINUS_DST_COLOR`	$1 - c_{dst}/255$
`GR_BLEND_SRC_ALPHA`	$a_{src}/255$
`GR_BLEND_ONE_MINUS_SRC_ALPHA`	$1 - a_{src}/255$
`GR_BLEND_DST_ALPHA`	$a_{dst}/255$
`GR_BLEND_ONE_MINUS_DST_ALPHA`	$1 - a_{dst}/255$
`GR_BLEND_ALPHA_SATURATE`	$\min(a_{src}/255, 1 - a_{dst}/255)$

*if **rgbDestFactor** or **alphaDestFactor** is*	*the destination blending factor $\mathbf{f}_{dst}$ or $\mathbf{g}_{dst}$ is*
`GR_BLEND_ZERO`	0
`GR_BLEND_ONE`	1
`GR_BLEND_SRC_COLOR`	$c_{src}/255$
`GR_BLEND_ONE_MINUS_SRC_COLOR`	$1 - c_{src}/255$
`GR_BLEND_SRC_ALPHA`	$a_{src}/255$
`GR_BLEND_ONE_MINUS_SRC_ALPHA`	$1 - a_{src}/255$
`GR_BLEND_DST_ALPHA`	$a_{dst}/255$
`GR_BLEND_ONE_MINUS_DST_ALPHA`	$1 - a_{dst}/255$
`GR_BLEND_PREFOG_COLOR`	c_{src} before fog is applied. See the *Multi-Pass Fog* section in Chapter 8.

Example 6.3 A compositing example.
A background scene is drawn with one alpha value, a polygonal cropping window is drawn with a second alpha value, and a foreground is mapped into the cropping window by discarding parts of the new scene that fall outside the cropping window. This example uses the alpha combine unit, alpha buffering, and alpha blending.

```
/* enable the alpha buffer */
grColorMask(FXTRUE, FXTRUE);

/* set alpha combine to generate zero alpha */
grAlphaCombine(GR_COMBINE_FUNCTION_ZERO, GR_COMBINE_FACTOR_NONE,
   GR_COMBINE_LOCAL_NONE, GR_COMBINE_OTHER_NONE, FXFALSE);

/* draw background scene */

/* clear out the cropping polygon */
grColorCombine(GR_COMBINE_FUNCTION_ZERO, GR_COMBINE_FACTOR_NONE,
   GR_COMBINE_LOCAL_NONE, GR_COMBINE_OTHER_NONE, FXFALSE);
grAlphaCombine(GR_COMBINE_FUNCTION_ZERO, GR_COMBINE_FACTOR_NONE,
   GR_COMBINE_LOCAL_NONE, GR_COMBINE_OTHER_NONE, FXFALSE);

/* draw cropping window */

/* set alpha blend unit to use destination alpha to select */
/* new pixel or old one */
grAlphaBlendFunction(GR_BLEND_DST_ALPHA, GR_BLEND_ONE_MINUS_DST_ALPHA,
   GR_BLEND_ZERO, GR_BLEND_ONE);

/* set color combine and alpha combine back to defaults */
grColorCombine(GR_COMBINE_FUNCTION_SCALE_OTHER, GR_COMBINE_FACTOR_ONE,
   GR_COMBINE_LOCAL_ITERATED, GR_COMBINE_OTHER_ITERATED, FXFALSE);
grAlphaCombine(GR_COMBINE_FUNCTION_ SCALE_OTHER, GR_COMBINE_FACTOR_ONE,
   GR_COMBINE_LOCAL_NONE, GR_COMBINE_OTHER_CONSTANT, FXFALSE);

/*draw the foreground scene */
```

7. Depth Buffering

In This Chapter

One potential use of the auxiliary buffer is as a 16-bit depth buffer. Each pixel may have an associated *z* or *q* value and either one may be used to represent the distance between the pixel and the viewer. A user-selectable depth test determines when an incoming pixel replaces one previously stored in the frame buffer. One common use for a depth buffer is pixel-accurate hidden surface removal, allowing nearer surfaces to obscure surfaces further away regardless of the order they are drawn in.

You will learn how to:

▼ enable depth buffering.

▼ specify a depth test.

▼ implement a fixed point *z* buffer.

▼ implement a floating point *w* buffer.. (It's really a "*q* buffer" in Glide 3.0 but history demands that we stick with the old name.)

▼ use a depth bias to reduce poke-through artifacts introduced by coplanar polygons.

The type of depth buffering in use is controlled using **grDepthBufferMode()**. The comparison function is selected with the function **grDepthBufferFunction()**. Writes to the depth buffer are controlled by **grDepthMask()**. Since the auxiliary buffer can serve only a single use, depth buffering, alpha buffering, and triple buffering are mutually exclusive.

Enabling Depth Buffering

The Glide function **grDepthBufferMode()** enables and disables depth buffering.

void **grDepthBufferMode(** *GrDepthBufferMode_t mode* **)**

The *mode* argument specifies the type of depth buffering to be performed. Valid modes are `GR_DEPTHBUFFER_DISABLE`, `GR_DEPTHBUFFER_ZBUFFER`, `GR_DEPTHBUFFER_WBUFFER`, `GR_DEPTHBUFFER_ZBUFFER_COMPARE_TO_BIAS`, or `GR_DEPTHBUFFER_WBUFFER_COMPARE_TO_BIAS`. If `GR_DEPTHBUFFER_ZBUFFER` or `GR_DEPTHBUFFER_ZBUFFER_COMPARE_TO_BIAS` is selected, the depth buffer is a 16-bit fixed point *z* buffer. A 16-bit floating point *w* buffer is used if *mode* is `GR_DEPTHBUFFER_WBUFFER` or `GR_DEPTHBUFFER_WBUFFER_COMPARE_TO_BIAS`. By default, the depth buffer mode is `GR_DEPTHBUFFER_DISABLE`.

Since alpha, depth, and triple buffering are mutually exclusive of each other, enabling depth buffering when using either the alpha or triple buffer will have undefined results.

If `GR_DEPTHBUFFER_ZBUFFER_COMPARE_TO_BIAS` or `GR_DEPTHBUFFER_WBUFFER_COMPARE_TO_BIAS` is selected, then the bias specified with **grDepthBiasLevel()** is used as a pixel's depth value for comparison purposes only. Depth buffer values are compared against the depth bias level, and if the compare passes and the depth buffer mask is enabled, the pixel's unbiased depth value is written to the depth buffer. This mode is useful for clearing beneath cockpits and other types of overlays without affecting either the color or depth values for the cockpit or overlay.

Consider the following example: the depth buffer is cleared to `0xFFFF` and a cockpit is drawn with a depth value of zero. Next, the scene beneath the cockpit is drawn with depth buffer compare function of `GR_CMP_LESS`, rendering pixels only where the cockpit is not drawn. To render the next frame, you need to clear the last scene. If you use **grBufferClear()**, you will remove everything, including the cockpit. To clear the color and depth buffers underneath the cockpit without disturbing the cockpit, the area to be cleared is rendered using triangles with the depth bias level set to zero, a depth buffer compare function of `GR_CMP_NOTEQUAL`, and a depth buffer mode of `GR_DEPTHBUFFER_ZBUFFER_COMPARE_TO_BIAS` or `GR_DEPTHBUFFER_WBUFFER_COMPARE_TO_BIAS`. All pixels with non-zero depth buffer values will be rendered and the depth buffer will be set to either unbiased *z* or *q*, depending on the mode. Using this method, the color and depth buffers can be cleared to any desired value beneath a cockpit or overlay without affecting the cockpit or overlay. Sorted background polygons that cover the visible area can be rendered in this manner, eliminating the need to clear the whole buffer and then redraw the overlay for each frame. Once the depth buffer is cleared beneath the cockpit, the depth buffer mode is returned to either `GR_DEPTHBUFFER_ZBUFFER` or `GR_DEPTHBUFFER_WBUFFER` by calling **grDepthBufferMode()** and the depth comparison function is returned to its normal setting (`GR_CMP_LESS` in this example) by calling **grDepthBufferFunction()**.

Note that since this mode of clearing is performed using triangle rendering, the fill rate is about one half that of a rectangular clear using **grBufferClear()**. In the case where sorted background polygons are used to clear beneath the cockpit, this method should always be faster than the alternative of calling **grBufferClear()** and then drawing the background polygons. In the case where background polygons are not used, the two methods:

- clearing the buffers with **grBufferClear()** and then repainting the cockpit, or
- clearing beneath the cockpit with triangles and not repainting the cockpit

should be compared and the faster method chosen. Avoiding a cockpit repaint is important: cockpits are typically rendered with linear frame buffer writes and while the writes are individually fast, the process can be lengthy if the cockpit covers many pixels.

`GR_DEPTHBUFFER_ZBUFFER_COMPARE_TO_BIAS` and `GR_DEPTHBUFFER_WBUFFER_COMPARE_TO_BIAS` modes are not available in revision 1 of the Pixel*fx* chip (use **grGet()** to obtain the revision number).

When depth buffering is enabled, the **grDepthMask()** routine enables writes to the depth buffer.

void **grDepthMask(** *FxBool enable* **)**

If *enable* is `FXFALSE`, depth buffer writing is disabled. Otherwise, it is enabled. Initially, writing to the depth buffer is disabled. Since the alpha, depth, and triple buffers share the same memory, **grDepthMask()** should be called only if depth buffering is being used.

The depth buffer can be cleared to a specific value with **grBufferClear()**, as described in Chapter 3. The depth buffer is typically cleared to a value that is further away from the viewpoint than any object in the scene.

The Depth Test

grDepthBufferFunction() specifies the function used to compare each rendered pixel's depth value with the depth value present in the depth buffer. The comparison is performed only if depth testing is enabled with **grDepthBufferMode()**. The choice of depth buffer function is typically dependent upon the depth buffer mode currently active. The default comparison function is `GR_CMP_LESS`.

The single argument, *func*, specifies the depth comparison function. Table 7.1 lists the valid comparison functions and the conditions under which a pixel will "pass" the test and overwrite the pixel in the frame buffer and depth buffer.

Table 7.1 The depth test.
*The **func** argument to **grDepthBufferFunction()** can take on any of the values listed in the first column of the table below. The second column specifies the depth test, and the third column describes the conditions under which an incoming pixel will "pass" the test and overwrite the appropriate location in the frame buffer and depth buffer.*

*if **func** is*	*the depth comparison is*	*and the pixel*
`GR_CMP_NEVER`	FALSE	*never passes*
`GR_CMP_LESS`	$\text{depth}_{new} < \text{depth}_{old}$	*passes if the pixel's depth value is less than the stored depth value*
`GR_CMP_EQUAL`	$\text{depth}_{new} = \text{depth}_{old}$	*passes if the pixel's depth value is equal to the stored depth value*
`GR_CMP_LEQUAL`	$\text{depth}_{new} \leq \text{depth}_{old}$	*passes if the pixel's depth value is less than or equal to the stored depth value*
`GR_CMP_GREATER`	$\text{depth}_{new} > \text{depth}_{old}$	*passes if the pixel's depth value is greater than the stored depth value*
`GR_CMP_NOTEQUAL`	$\text{depth}_{new} \neq \text{depth}_{old}$	*passes if the pixel's depth value is not equal to the stored depth value*
`GR_CMP_GEQUAL`	$\text{depth}_{new} \geq \text{depth}_{old}$	*passes if the pixel's depth value is greater than or equal to the stored depth value*
`GR_CMP_ALWAYS`	TRUE	*always passes*

Fixed Point *z* Buffering

When 16-bit linear *z* buffering is enabled, *z* values for each pixel are linearly interpolated across a polygon's face. Since observer space *z* values are not linear in screen space, the graphics hardware must instead interpolate 1/*z* values, which *are* linear in screen space. When linear *z* buffering is enabled, the graphics hardware interpolates a high precision fixed point 1/*z* value (provided by the application), but it stores only the 16-bit integer portion of the 1/*z* value. This can lead to some precision problems, and thus an application's objects and database must be constructed and scaled carefully to minimize *z* aliasing. Linear *z* buffering is enabled by calling **grDepthBufferMode()** with the constant `GR_DEPTHBUFFER_ZBUFFER`.

Example 7.1 Configuring a z buffer.
The following code sequence configures Glide for z buffering:

```
grDepthBufferMode( GR_DEPTHBUFFER_ZBUFFER );
grDepthBufferFunction( GR_CMPFNC_GREATER );  // 1/Z decreases as Z
increases!
grDepthMask( FXTRUE );
grBufferClear(0, 0, 0);
```

Floating Point *w* Buffering

The graphics hardware can also derive a depth value from the *q*/*w* factor computed for texture mapping and fog. Such an approach has many advantages over linear *z* buffering, including much greater dynamic range and less aliasing and accuracy artifacts. The graphics hardware uses a proprietary 16-bit floating point format for *w* buffering, however, an application typically does not need to manipulate this data directly, except when an application must read data directly from the depth buffer and then convert this depth value to an application dependent format. Floating point *w* buffering is enabled by calling **grDepthBufferMode()** with the constant `GR_DEPTHBUFFER_WBUFFER`.

Example 7.2 Configuring a w buffer.
The following code sequence configures Glide for w buffering. The depth buffer is initially cleared to a value representing the farthest point, so that all objects in the scene are closer to the viewer than empty space is.

```
FxU8 wLimits[2];

grGet(GR_WDEPTH_MIN_MAX,2,*wLimits);
grDepthBufferMode( GR_DEPTHBUFFER_WBUFFER );
grDepthBufferFunction( GR_CMP_LESS );  // larger W values are farther
away
grDepthMask( FXTRUE );
grBufferClear(0, 0, wLimits[1]);
```

Establishing a Depth Bias

When depth buffering coplanar polygons (e.g. when one polygon is used as a "detail" polygon on another), precision problems with coplanar polygons may result in "poke through" artifacts if the vertices of the two polygons are not the same. To eliminate the artifacts, an application should apply a "depth bias" when it renders two coplanar polygons, so that Glide understands which polygon is on top of the other. **grDepthBiasLevel()** allows an application to specify a depth bias.

void **grDepthBiasLevel(** *FxU32 level* **)**

Specifically, if two polygons are coplanar but do not share vertices (e.g., a surface detail polygon sits on top of a larger polygon), the depth bias *level* should be incremented or decremented as appropriate for the depth buffer mode and function, per coplanar polygon. For left-handed coordinate systems, where `0x0000` corresponds to "nearest to viewer" and `0xFFFF` corresponds "farthest from viewer", depth bias levels should be decremented on successive renderings of coplanar polygons. When the coplanar polygons have been rendered, the depth bias mode should be reset to 0.

Example 7.3 Using a depth bias.
In this code segment, an underlying triangle is rendered, a depth bias is established, and then another triangle is rendered on top of the first one.

```
/* Render the underlying polygon */
grDrawTriangle( /* base polygon’s parameters */ );

/* Render the composite polygon by first enabling depth bias */
grDepthBiasLevel( -1 );
grDrawTriangle( /* composite polygon’s parameters */ );

/* Disable depth bias */
grDepthBiasLevel( 0 );
```

An Example: Hidden Surface Removal

When a scene is rendered, some of the objects will undoubtedly obscure other objects. If the viewpoint never changes, you can sort the polygons on *z*, and draw the scene from back to front.

But what if the viewpoint can change from one frame to the next? Say it's tracking a cursor controlled by a mouse. The computation cost of re-sorting the scene for each frame can be prohibitive, depending on the complexity of the scene. But a *z* buffer will solve the problem.

You will still need to transform world coordinates to screen coordinates for each object in the scene, but the transformed vertices can be drawn in any order, without regard to their distance from the viewpoint.

The code segment in Example 7.4 shows the depth buffer in action.

Example 7.4 Hidden surface removal using a z buffer.
The code segment below leaves out the details of converting a mouse position or movement into a viewpoint and transforming the world coordinates to new screen coordinates.

```
/* set up a z buffer and depth test */
grDepthBufferMode( GR_DEPTHBUFFER_ZBUFFER );
grDepthBufferFunction( GR_CMPFNC_GREATER ); // 1/Z decreases as Z increases!
grDepthMask( FXTRUE ) ;

while (1) {
   /* clear the buffers for each frame */
   grBufferClear(0, 0, 0);

   /* get the new viewpoint and transform the coordinates */
   set_viewpoint_from_mouse();
   transform_coordinates();

   /*draw the objects in the scene */
   draw_objects();

   /* display the frame */
   grBufferSwap(1);
}
```

8. Special Effects

In This Chapter

Glide supports several different types of special effects, including fog, chroma-keying, and alpha testing. *Fog* simulates atmospheric conditions like fog, mist, smog, or smoke that partially obscure distant objects. *Chroma-keying* can be used to create a blue screen effect, removing all pixels that are a specific color. *Alpha masking* uses the low order bit of the incoming alpha value to invalidate pixels.

You will learn how to:

- produce fog using the alpha iterator.
- create a fog table and use it to create atmospheric effects.
- configure the fog and alpha blending units for multi-pass fogging.
- use chroma-keying to simulate a blue screen.
- use alpha testing to simulate a blue screen.

Fog

Fog is a rendering technique that adds realism to computer-generated scenes by making distant objects appear to fade away. Fog is a general term representing all atmospheric effects: haze, mist, smoke, smog. It is essential in visual simulations like flight simulators to produce the effect of limited visibility. When fogging is enabled, distant objects fade into the fog color. Both the fog color and the fog density (the rate at which objects fade as a function of their distance from the viewer) are programmable.

Glide and the graphics hardware support per-pixel fog blending operations. The fog unit is separate from the alpha blending unit, so both fog and transparency may be applied simultaneously. Fog is applied after texturing and lighting, and it may improve performance in large simulations: some objects may be lost in the fog and can be culled before rendering.

Fog is applied after color combining and before alpha blending, as shown in the pixel pipeline flow diagram in Figure 1.2.

The fog operation blends the fog color (c_{fog}) with each rasterized pixel's post-texturing color (c_{in}) using a blending factor ***f***. Factor ***f*** is retrieved from a user downloaded fog table indexed with the pixel's *q* for *fog* component, depending on **grFogMode()**. The fog operation blends a global (c_{fog}) with each rasterized pixel's color (c_{in}) using a blending factor ***f***. A value of ***f****=0* indicates minimum fog density and a value of ***f***=255 indicates maximum fog density.

The general fog equation is shown below.

$$c_{out} = \boldsymbol{f}\, c_{fog} + (1-\boldsymbol{f}\,)c_{in}$$

The fog mode, set with **grFogMode()**, shapes the fog equation to the situation, as shown in Table 8.1.

```
void grFogMode( GrFogMode_t mode )
```

The *mode* argument can be one of five values: `GR_FOG_DISABLE`, `GR_FOG_WITH_TABLE_ON_Q`, `GR_FOG_ADD2`, `GR_FOG_MULT2`, or, if supported, `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT`. The `GR_FOG_ADD2` and `GR_FOG_MULT2` modes facilitate multi-pass fogging applications and are used in conjunction with `GR_FOG_WITH_TABLE_ON_Q` or `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT`.

Table 8.1 The fog mode shapes the fog equation.
The general form of the fog equation is $c_{out} = f\, c_{fog} + (1 - f\,)c_{in}$. The mode argument to **grFogMode()** *tailors the general equation for a specific situation, as shown below. The first three modes are mutually exclusive: choose one. Modes `GR_FOG_ADD2` and `GR_FOG_MULT2` are used in tandem with either `GR_FOG_WITH_TABLE_ON_Q` or `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT`.*

*if **mode** sets*	*the fog equation is*	*where c_{in} is the color entering the fog unit, c_{out} is the result of fogging, c_{fog} is the fog color and*
`GR_FOG_DISABLE`	$c_{out} = c_{in}$	
`GR_FOG_WITH_TABLE_ON_Q`	$c_{out} = f_{\text{fog}[q]} \bullet c_{fog} + (1 - f_{\text{fog}[q]}) \bullet c_{in}$	$f_{\text{fog[w]}}$ *is computed by interpolating between entries in a fog table indexed with **q**.*
`GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT`	$c_{out} = f_{\text{fog}[v\cdot fog]} \bullet c_{fog} + (1 - f_{\text{fog}[v\cdot fog]}) \bullet c_{in}$	$f_{\text{fog}[v\cdot\text{fog}]}$ *is computed by interpolating between entries in a fog table indexed with v.fog, the `GR_PARAM_FOG_EXT` parameter to **grVertexLayout()**. This mode is valid only when the FOGCOORD extension is supported. See **grGetString()** in Chapter 13.*
`GR_FOG_MULT2`	$c_{out} = f\, c_{fog}$	***f*** *is computed from a fog table.*
`GR_FOG_ADD2`	$c_{out} = (1 - f\,)c_{in}$	***f*** *is computed from a fog table.*

The fogging factor ***f*** is determined by *mode*. If *mode* is `GR_FOG_WITH_TABLE_ON_Q`, then ***f*** is computed by interpolating between fog table entries, where the fog table is indexed with a floating point representation of the pixel's *q* component. If *mode* is `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT`, then the fog table is indexed with a special vertex parameter, `GR_PARAM_FOG_EXT`. Fog is applied after color combining and before alpha blending.

The global fog color (c_{fog}) is set by calling **grFogColorValue()**. The argument, *value*, is an RGBA color and is specified in the format defined in the *cFormat* parameter to **grSstWinOpen()** (see Chapter 3).

void **grFogColorValue(** *GrColor_t value* **)**

Fogging With A Fog Table

The application may supply a fog table to the hardware via the function **grFogTable()**. To enable table-based fogging, the fog mode must be set to `GR_FOG_WITH_TABLE_ON_Q`. The number of entries in the fog table depends on the hardware and can be retrieved with **grGet(**`GR_FOG_TABLE_ENTRIES`,…**)**. The entries are density values of type *GrFog_t*, an unsigned 8-bit quantity. A value of 0 indicates minimum density, and 255 indicates maximum density. This density determines the amount of blending that occurs between the incoming pixel and the global fog color, set by **grFogColorValue()**. The order of the entries within the table corresponds roughly to their distance from the viewer. Entries within the table are calculated as a function of world q where world $q \cong 2^{i/4}$, where i is the index into the fog table. To minimize "fog banding", the graphics hardware linearly blends between adjacent fog levels within the fog table. *The difference between consecutive fog values must be less than 64.*

void **grFogTable(** const *GrFog_t table*[] **)**

grFogTable() downloads a new table of 8-bit values that are logically viewed as fog opacity values corresponding to various depths. The table entries control the amount of blending between the fog color and the pixel's color. A value of `0x00` indicates no fog blending and a value of `0xFF` indicates complete fog.

The fog operation blends the fog color (c_{fog}) with each rasterized pixel's color (c_{in}) using a blending factor f. When **grFogMode()** is set to `GR_FOG_WITH_TABLE_ON_Q`, then the factor f is computed by interpolating between fog table entries, where the fog table is indexed with a floating point representation of the pixel's q component.

$$c_{out} = \boldsymbol{f}_{\mathrm{fog}[q]} \bullet c_{fog} + (1 - \boldsymbol{f}_{\mathrm{fog}[q]}) \bullet c_{in}$$

The order of the entries within the fog table corresponds roughly to their distance from the viewer. The exact fog coordinate or q value corresponding to fog table entry i can be found by calling **guFogTableIndexToW()** with argument i.

guFogTableIndexToW(int *i***)**

guFogTableIndexToW() returns the floating point fog coordinate value associated with entry i in a fog table. Because fog table entries are non-linear, it is not straight forward to initialize a fog table. **guFogTableIndexToW()** assists by converting fog table indices to eye-space w, and returns the following:

```
pow(2.0, 3.0+(double)(i>>2)) / (8-(i&3))
```

An exponential fog table can be generated by computing $(1–e^{–kw})\bullet 255$ where k is the fog density and w is world distance. It is usually best to normalize the fog table so that the last entry is 255.

Example 8.1 Creating a fog table.
The code fragment below creates storage for a fog table. Then, two different techniques for filling in the entries are explored.

```
int nFog;
GrFog_t *fog;

grGet(GR_FOG_TABLE_ENTRIES, 4, &nFog);
fog = (GrFog_t) malloc(nFog * sizeof(GrFog_t));
```

The first code segment shows a linear fog table that has a steep ramp at the beginning and end, with slow growing values in the middle.

```
int i;

fog [0] = 0;
for (i=1; i<12; i++) fog[i]= fog[i-1]+ 12;
for (i=12; i<56; i++) fog[i]= fog[i-1] + 1;
for (i=56; i< nFog-1; i++) fog[i]= fog[i-1] + 7;
fog[nFog-1] = 255;
```

The second table is an exponential fog table. It computes q from i using ***guFogTableIndexToW()*** *and then computes the fog table entries as* $fog[i]=(1-e^{-kw})\cdot 255$ *where k is a user-defined constant,* `FOG_DENSITY`*.*

```
#define FOG_DENSITY .5
int i;

for (i=0; i<nFog; i++) {
    fog[i] = (1 - exp((- FOG_DENSITY) * guFogTableIndexToW(i))) * 255;
}
fog[nFog-1] = 255;
```

Example 8.2 Fogging with q and a fog table.
The code segment below assumes that a fog table has been defined. It is loaded using ***grFogTable()****, a fog color is defined, and the appropriate fog mode set. All that remains is to draw the scene.*

```
GrFog_t fog[];
int i;

/* load the fog table */
grFogTable(fog);

/* set a fog color - how about smoke? */
grFogColorValue(0);

/* set mode to fog table */
grFogMode(GR_FOG_WITH_TABLE_ON_Q);

/* draw the scene */
```

PORTING NOTE

In previous versions of Glide, a fog table has a constant number of entries, namely GR_FOG_TABLE_SIZE. The number of entries has become a run-time constant in Glide 3.0 and is retrieved by calling **grGet(**`GR_FOG_TABLE_ENTRIES`**,...)**. Check your code for hardcoded numbers like “64” in loops and so forth.

Generating a Fog Table Automatically

The Glide Utilities Library includes three routines that generate fog tables with different characteristics.

void **guFogGenerateExp(** *GrFog_t fogTable[]*, float *density* **)**

guFogGenerateExp() generates an exponential fog table according to the equation:

$$e^{-density \bullet w}$$

where *w* is the eye-space *q* coordinate associated with the fog table entry. The resulting fog table is copied into *fogTable*. The fog table is normalized (scaled) such that the last entry is maximum fog (255).

void **guFogGenerateExp2(** *GrFog_t fogTable[]*, float *density* **)**

guFogGenerateExp2() generates an exponentially squared fog table according to the equation:

$$e^{-(density \bullet w)\,(density \bullet w)}$$

where *w* is the eye-space *q* coordinate associated with the fog table entry. The resulting fog table is copied into *fogTable*. The fog table is normalized (scaled) such that the last entry is maximum fog (255).

void **guFogGenerateLinear(** *GrFog_t fogTable[]*, float *near*, float *far* **)**

guFogGenerateLinear() generates a linear (in eye-space) fog table according to the equation

$$(w - near)/(far - near)$$

where *w* is the eye-space *w* coordinate associated with the fog table entry. The resulting fog table is copied into *fogTable*. The fog table is clamped so that all values are between minimum fog (0) and maximum fog (255). Note that **guFogGenerateLinear()** fog is linear in eye-space w*q*, *not* in screen-space.

Multi-Pass Fog

Special actions must be taken when applying fog to pixels generated with multi-pass techniques. Recall from Figure 1.2 that the fog unit is sandwiched between the color combine unit and the alpha blending unit in the pixel pipeline. This ordering facilitates anti-aliasing but may result in repeated fogging of intermediate values in multi-pass alpha blending applications. Special modes for the fog unit and a special alpha blending function have been provided to identify and handle this situation.

The `GR_FOG_ADD2` and `GR_FOG_MULT2` modes, passed as arguments to **grFogMode()**, suppress the first and second terms, respectively, of the fog equation. In `GR_FOG_ADD2` mode, the first term of the fog equation is suppressed, resulting in a fog equation shown below:

$$c_{out} = (1-\boldsymbol{f})c_{in}$$

and no fog is applied. In `GR_FOG_MULT2` mode, the second term is suppressed, making the fog equation effectively:

$$c_{out} = \boldsymbol{f}\, c_{fog}$$

leaving only the scaled fog color.

In the **grAlphaBlendFunction()** routine, presented in Chapter 6, the `GR_BLEND_PREFOG_COLOR` factor selects the pre-fogged value of the pixel as the destination RGBA blending factor.

The following sections present recipes for correctly applying fog to common multi-pass rendering applications. The generalized fog and blending equations are tailored to the specific situations and are the starting point for the derivations presented in the text. In case you've forgotten, the general fog equation is

$$Fog(\mathrm{c}_{in}) = \boldsymbol{f}\mathrm{c}_{fog} + (1-\boldsymbol{f})\mathrm{c}_{in}$$

where c_{in} is the pre-fogged color, and the blending equation is

$$\mathrm{c}_{dst} = \boldsymbol{f}_{src} \bullet Fog(\mathrm{c}_{in}) + \boldsymbol{f}_{dst} \bullet \mathrm{c}_{dst}$$

where c_{dst} is the value stored in the frame buffer and $\boldsymbol{f}_{src}$ and $\boldsymbol{f}_{dst}$ are the source and destination blending factors.

Table 8.2 summarizes the required fog mode and blending factor settings for the multi-pass fog scenarios presented here. Detailed discussion follows.

Table 8.2 Configuring the fog and alpha blending units for multi-pass fog generation.
The table below describes the proper settings for the fog mode and source and destination alpha blending factors for three different multi-pass fogging applications. If the fog mode is specified as ***mode****, either* `GR_FOG_WITH_TABLE_ON_Q` *or* `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT`*, if supported, may be used.*

pass	**grFogMode()** *and* **grAlphaBlendFunction()** *parameters*	*simple two pass blending* $\alpha \bullet Fog(\mathrm{c}_1) + (1-\alpha) \bullet Fog(\mathrm{c}_2)$	*additive blending* $Fog(\Sigma \mathrm{c}_i)$	*modulated blending* $Fog(\Pi \mathrm{c}_i)$
1	*mode*	*mode*	*mode*	(*mode* | `GR_FOG_ADD2`)
	rgbSrcFactor	`GR_BLEND_ONE`	`GR_BLEND_ONE`	`GR_BLEND_ONE`
	rgbDstFactor	`GR_BLEND_ZERO`	`GR_BLEND_ZERO`	`GR_BLEND_ZERO`
2 thru *n*–1	*mode*		(*mode* | `GR_FOG_ADD2`)	`GR_FOG_DISABLE`
	rgbSrcFactor	*n/a*	`GR_BLEND_ONE`	`GR_BLEND_DST_COLOR`
	rgbDstFactor		`GR_BLEND_ONE`	`GR_BLEND_ZERO`
n	*mode*	*mode*	(*mode* | `GR_FOG_ADD2`)	(*mode* | `GR_FOG_MULT2`)
	rgbSrcFactor	`GR_BLEND_SRC_ALPHA`	`GR_BLEND_ONE`	`GR_BLEND_ONE`
	rgbDstFactor	`GR_BLEND_ONE_MINUS_SRC_ALPHA`	`GR_BLEND_ONE`	`GR_BLEND_PREFOG_COLOR`

Simple Blends

Simple two-pass blending using α and 1–α can be used to produce translucent fog and requires no special actions. The goal here is to produce

$$\mathrm{c}_{dst} = \alpha \bullet Fog(\mathrm{c}_2) + (1-\alpha) \bullet Fog(\mathrm{c}_1)$$

where c_i is the color entering the fog unit from the color combine unit on pass *i*, $Fog(\mathrm{c}_i)$ is the color output by the fog unit on pass *i*, and c_{dst} is the color that is stored in the frame buffer. The first pass will generate and store $Fog(\mathrm{c}_1)$. The second pass will generate $Fog(\mathrm{c}_2)$ and blend it with the result of the first pass.

For the first pass, set the fog mode to `GR_FOG_WITH_TABLE_ON_Q` and the source and destination factors for alpha blending to `GR_BLEND_ONE` and `GR_BLEND_ZERO`, respectively, as shown in Table 8.2 and demonstrated in Example 8.3. After pass one is complete,

$$\begin{aligned} c_{dst} &= 1\bullet Fog(c_1) + 0\bullet c_{dst} \\ &= Fog(c_1) \end{aligned}$$

For the second pass, specify the source and destination factors for alpha blending as `GR_BLEND_SRC_ALPHA` and `GR_BLEND_ONE_MINUS_SRC_ALPHA`, respectively. Thus,

$$\begin{aligned} c_{dst} &= \alpha\bullet c_{in} + (1-\alpha)\bullet c_{dst} \\ &= \alpha\bullet Fog(c_2) + (1-\alpha)\bullet Fog(c_1) \end{aligned}$$

Note that there is nothing special about using `GR_BLEND_SRC_ALPHA` and `GR_BLEND_ONE_MINUS_SRC_ALPHA` as the blending factors. Any of the blending factors listed in Table 6.4 can be used.

Example 8.3 Simple two-pass blending.
The code segment below assumes that a fog table has been defined. It loads the table, then sets a fog color. For the first pass, the fog mode is set to use the fog table and the alpha blending function to write fogged colors into the frame buffer. For the second pass, the fog mode and color remain the same, but the blending factors change blending the newly-generated fogged colors with the previous ones.

```
const GrFog_t fog[];
int i;

/* load the fog table */
grFogTable(fog);

/* set a fog color - how about smoke? */
grFogColorValue(0);

/* set mode to fog table */
grFogMode(GR_FOG_WITH_TABLE_ON_Q);
grAlphaBlendFunction(GR_BLEND_ONE, GR_BLEND_ZERO, GR_BLEND_ONE,
GR_BLEND_ZERO);

/* draw the first pass */
…

/* reconfigure alpha blending for the second pass */
grAlphaBlendFunction(GR_BLEND_SRC_ALPHA, GR_BLEND_ONE_MINUS_SRC_ALPHA,
              GR_BLEND_ONE, GR_BLEND_ZERO);

/* draw the second pass */
…
```

Additive Multi-Pass Fog

The additive case assumes that the results of each pass are being added together, and we wish to fog the final result:

$$c_{dst} = Fog(\Sigma c_i) \text{ where } c_i \text{ is the color entering the fog unit in pass } i$$

Here is the procedure for the two-pass case. This can be generalized to multiple passes by induction. We wish to obtain:

$$c_{dst} = Fog(c_1 + c_2) = \boldsymbol{f}c_{fog} + (1-\boldsymbol{f})(c_1 + c_2)$$

For the first pass, choose either `GR_FOG_WITH_TABLE_ON_Q` or `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT` (if supported) as the fog mode and set the source and destination alpha blending factors to `GR_BLEND_ONE` and `GR_BLEND_ZERO`, respectively. After the first pass,

$$\begin{aligned} c_{dst} &= 1\bullet Fog(c_1) + 0\bullet c_{dst} \\ &= Fog(c_1) \\ &= \boldsymbol{f}c_{fog} + (1-\boldsymbol{f})c_1 \end{aligned}$$

For the second pass, add `GR_FOG_ADD2` to the fog mode, causing the blended fog term to be suppressed (if you forget to do this, the c_{fog} term will occur twice). Set the source and destination alpha blending factors to `GR_BLEND_ONE` and `GR_BLEND_ONE`, respectively. Thus,

$$\begin{aligned} Fog(c_2) &= (1-\boldsymbol{f})c_2 \\ c_{dst} &= 1\bullet c_{in} + 1\bullet c_{dst} \\ &= (1-\boldsymbol{f})c_2 + (\boldsymbol{f}c_{fog} + (1-\boldsymbol{f})c_1) \\ &= \boldsymbol{f}c_{fog} + (1-\boldsymbol{f})(c_1 + c_2) \end{aligned}$$

Example 8.4 Two-pass additive fogging.
The code segment below assumes that a fog table has been defined.

```
const GrFog_t fog[];
int i;

/* load the fog table */
grFogTable(fog);

/* set a fog color - how about smoke? */
grFogColorValue(0);

/* set mode to fog table */
grFogMode(GR_FOG_WITH_TABLE_ON_Q);
grAlphaBlendFunction(GR_BLEND_ONE, GR_BLEND_ZERO, GR_BLEND_ONE,
GR_BLEND_ZERO);

/* draw the first pass */
…

/* set mode to fog table */
grFogMode(GR_FOG_WITH_TABLE_ON_Q | GR_FOG_ADD2);
grAlphaBlendFunction(GR_BLEND_ONE, GR_BLEND_ONE, GR_BLEND_ONE,
GR_BLEND_ZERO);

/* draw the second pass */
…
```

Modulation Multi-Pass Fog

The modulation case assumes that the results of each pass are being multiplied together, and we wish to fog the final result:

$$c_{dst} = \mathit{Fog}(\prod c_i) \text{ where } c_i \text{ is the color entering the fog unit in pass } i$$

This case occurs most commonly when applying light maps to a scene, and it is more complex to implement than the additive case. Here is the procedure for the three-pass case; it can be generalized by induction. We wish to obtain:

$$c_{dst} = \mathit{Fog}(c_1c_2c_3) = \boldsymbol{f}c_{fog} + (1-\boldsymbol{f})(c_1c_2c_3)$$

For the first pass, choose either `GR_FOG_WITH_TABLE_ON_Q` or `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT` (if supported) as the fog mode and OR in `GR_FOG_ADD2`, as shown in Table 8.2 and demonstrated in Example 8.5. Set the source and destination alpha blending factors to `GR_BLEND_ONE` and `GR_BLEND_ZERO`, respectively. After the first pass,

$$\begin{aligned} c_{dst} &= 1\bullet \mathit{Fog}(c_1) + 0\bullet c_{dst} \\ &= \mathit{Fog}(c_1) \\ &= (1-\boldsymbol{f})c_1 \end{aligned}$$

For the second pass (and all intermediate passes in the general case), disable fogging (**grFogMode(**`GR_FOG_DISABLE`**)**) and set the source and destination alpha blending factors to `GR_BLEND_DST_COLOR` and `GR_BLEND_ZERO`, respectively. (Using source and destination factors of `GR_BLEND_ZERO` and `GR_BLEND_SRC_COLOR`, respectively, will work as well.) After the second pass we have:

$$\begin{aligned} c_{dst} &= c_{dst}\bullet c_{in} + 0\bullet c_{dst} \\ &= c_{dst}\bullet c_2 \\ &= (1-\boldsymbol{f})c_1c_2 \end{aligned}$$

For the final pass, enable fogging again, choosing either `GR_FOG_WITH_TABLE_ON_Q` or `GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT` (if supported), and OR in `GR_FOG_MULT2`, which causes the blended pixel term to be suppressed. Set the source and destination alpha blending factors to `GR_BLEND_ONE` and `GR_BLEND_PREFOG_COLOR`, respectively. The result is:

$$\begin{aligned} \mathit{Fog}(c_3) &= \boldsymbol{f}c_{fog} \\ c_{dst} &= 1\bullet \mathit{Fog}(c_3) + c_3\bullet c_{dst} \\ &= \boldsymbol{f}c_{fog} + c_3\bullet(1-\boldsymbol{f})c_1c_2 \\ &= \boldsymbol{f}c_{fog} + (1-\boldsymbol{f})c_1c_2c_3 \end{aligned}$$

Example 8.5 Three-pass modulation fogging.
The code segment below assumes that a fog table has been defined.

```
const GrFog_t fog[];
int i;

/* load the fog table */
grFogTable(fog);

/* set a fog color - how about smoke? */
grFogColorValue(0);

/* set fog mode and alpha blending function for pass 1*/
grFogMode(GR_FOG_WITH_TABLE_ON_Q | GR_FOG_ADD2);
grAlphaBlendFunction(GR_BLEND_ONE, GR_BLEND_ZERO, GR_BLEND_ONE,
GR_BLEND_ZERO);

/* draw pass 1 */
…

/* set fog mode and alpha blending function for pass 2*/
grFogMode(GR_FOG_DISABLE);
grAlphaBlendFunction(GR_BLEND_DST_COLOR, GR_BLEND_ZERO, GR_BLEND_ONE,
GR_BLEND_ZERO);

/* draw pass 2 */
…

/* set fog mode and alpha blending function for final pass */
grFogMode(GR_FOG_WITH_TABLE_ON_Q | GR_FOG_MULT2);
grAlphaBlendFunction(GR_BLEND_ONE, GR_BLEND_PREFOG_COLOR, GR_BLEND_ONE,
GR_BLEND_ZERO);

/* draw pass 3 */
…
```

Chroma-keying

When chroma-keying is enabled, color values are compared to a global chroma-key reference value set by **grChromakeyValue()**. If the pixel's color is the same as the chroma-key reference value, the pixel is discarded. The chroma-key comparison takes place before the color combine function; the *other* color selected by color combine function is the one compared (see **grColorCombine()** in Chapter 5). By default, chroma-keying is disabled.

Chroma-keying is useful for certain types of sprite animation or blue-screening of textures. Only one color value is reserved for chroma-keyed transparency, while alpha blending reserves a variable number of color bits for transparency.

void **grChromakeyMode(** *GrChromakeyMode_t mode* **)**

Use **grChromakeyMode()** to enable or disable chroma-keying. The argument, *mode*, specifies whether chroma-keying should be enabled or disabled. Valid values are `GR_CHROMAKEY_ENABLE` and `GR_CHROMAKEY_DISABLE`.

void **grChromakeyValue(** *GrColor_t value* **)**

The function **grChromakeyValue()** sets the global chroma-key reference value as a packed RGBA value in the format specified in the *cFormat* parameter to **grSstWinOpen()** (see Chapter 3).

Glide 3.0 introduces the concept of a *chroma-range* as an extension. The extension capability and the two chroma-range extensions, one for pixels and one for texels, are described in Chapter 13.

Example 8.6 Simulating a blue-screen with chroma-keying.
A blue screen is a compositing mechanism used in live video where a second scene overlays all the "blue" pixels in the first scene. This technique is used to stand a weathercaster in front of a weather map, for example, and explains why they don't wear blue suits or ties! With chroma-keying, pixels of any one specific color can be discarded, not just blue.

```
/* draw the background */
draw_weather_map();

/* enable chroma-keying */
grChromakeyMode(GR_CHROMAKEY_ENABLE);

/*set the reference color - assumes ARGB format */
grChromakeyValue(0xFF);

/* draw the inserted scene - most of it is blue */
draw_weatherman();
```

Alpha Testing

The alpha test function is a technique for accepting or rejecting a pixel based on its alpha value. The incoming alpha value (the output from the alpha combine unit) is compared with a reference value and accepted or rejected based on a user-defined comparison function.

One application of the alpha compare function is billboarding: if you create a texture with some transparent and some opaque areas, you can indicate the degree of opacity with the alpha value. Set alpha to zero if the texel is transparent, and to one if it's opaque. With a reference alpha value of .5 (or any number greater than 0) and a "greater than" comparison function, transparent texels are rejected and the destination pixel is displayed.

Incoming pixels can be rejected based on a comparison between their alpha values and a global alpha test reference value. The nature of the comparison is user definable through the function **grAlphaTestFunction()**. This is useful for some effects such as partially transparent texture maps. Also, alpha testing can prevent the depth buffer from being updated for nearly transparent pixels. To disable alpha testing, set the alpha test function to `GR_CMP_ALWAYS`. The global alpha test reference is set via a call to **grAlphaTestReferenceValue()**. Because alpha testing does not require alpha storage (i.e. an alpha buffer), it is always available regardless of the use of depth or triple buffering.

void **grAlphaTestFunction(** *GrCmpFnc_t func* **)**

The incoming alpha value is compared to the constant alpha test reference value using the function specified by *func*. The possible values for *func* are shown in Table 8.3. The incoming alpha is the output of the alpha combine unit (see **grAlphaCombine()**, described earlier in this chapter). The reference value is set with **grAlphaTestReferenceValue()**.

void **grAlphaTestReferenceValue(** *GrAlpha_t value* **)**

The incoming alpha value is compared to the *value* using the function specified by **grAlphaTestFunction()**. If the comparison fails, the pixel is not drawn.

Table 8.3 Alpha test functions.
Alpha testing is a technique whereby the incoming alpha value is compared to a reference value and the pixel is discarded if the test fails. The test is user-selectable; the choices are shown below.

*If **func** is*	*the comparison function*
`GR_CMP_NEVER`	never passes.
`GR_CMP_LESS`	passes if the α value produced by the alpha combine unit is less than the constant α reference value.
`GR_CMP_EQUAL`	passes if the α value produced by the alpha combine unit is equal to the constant α reference value.
`GR_CMP_LEQUAL`	passes if the α value is less than or equal to the constant α reference value.
`GR_CMP_GREATER`	passes if the α value is greater than the constant α reference value.
`GR_CMP_NOTEQUAL`	passes if the α value is not equal to the constant α reference value.
`GR_CMP_GEQUAL`	passes if the α value is greater than or equal to the constant α reference value.
`GR_CMP_ALWAYS`	always passes.

Alpha testing is performed on all pixel writes, including those resulting from scan conversion of points, lines, and triangles, and from direct linear frame buffer writes. Alpha testing is implicitly disabled during linear frame buffer writes if the pixel pipeline is bypassed (see Chapter 11).

Stenciling

Stenciling is not directly supported by the graphics family graphics hardware. However, a stencil effect is possible with depth buffering by setting the depth buffer (using linear frame buffer writes) to its minimum value in the areas to be stenciled out.

9. Texture Mapping

In This Chapter

The discussion thus far has described how to produce a polygon that is filled with a solid color or smoothly shaded from one color to another. This chapter describes the process of filling a polygon with a pattern: a brick wall pattern, for example, or a veined marble texture.

Texture mapping is a technique in which a two-dimensional image, a texture map, is pasted like wall-paper onto a three-dimensional surface. This allows for very realistic images without requiring the use of many small detail polygons. The graphics hardware provides accelerated perspective-correct texture mapping.

You will learn about:

▼ textures and texels and how they relate to pixels.

▼ magnification and minification.

▼ point sampling and bilinear filters.

▼ texture clamping.

▼ specifying magnification and minification filters and texture clamping options.

▼ adding, modulating, and blending textures in the texture combine unit.

A Look at Texture Mapping and Glide

A texture map is a square or rectangular array of texture elements, or texels, that are addressed by (*s*, *t*) coordinates. The TMU, or texture mapping unit, contains memory for storing textures, circuitry to map texels to pixels, and more circuitry to add, scale, and blend texels.

A 3Dfx Interactive graphics subsystem includes at least one TMU and may have as many as three; Figure 9.1 shows the connectivity. Each TMU will produce an RGBA color from its own texture memory that will be pairwise combined to produce a texture RGBA color that can be selected as an input to the color combine and alpha combine units described in Chapters 5 and 6.

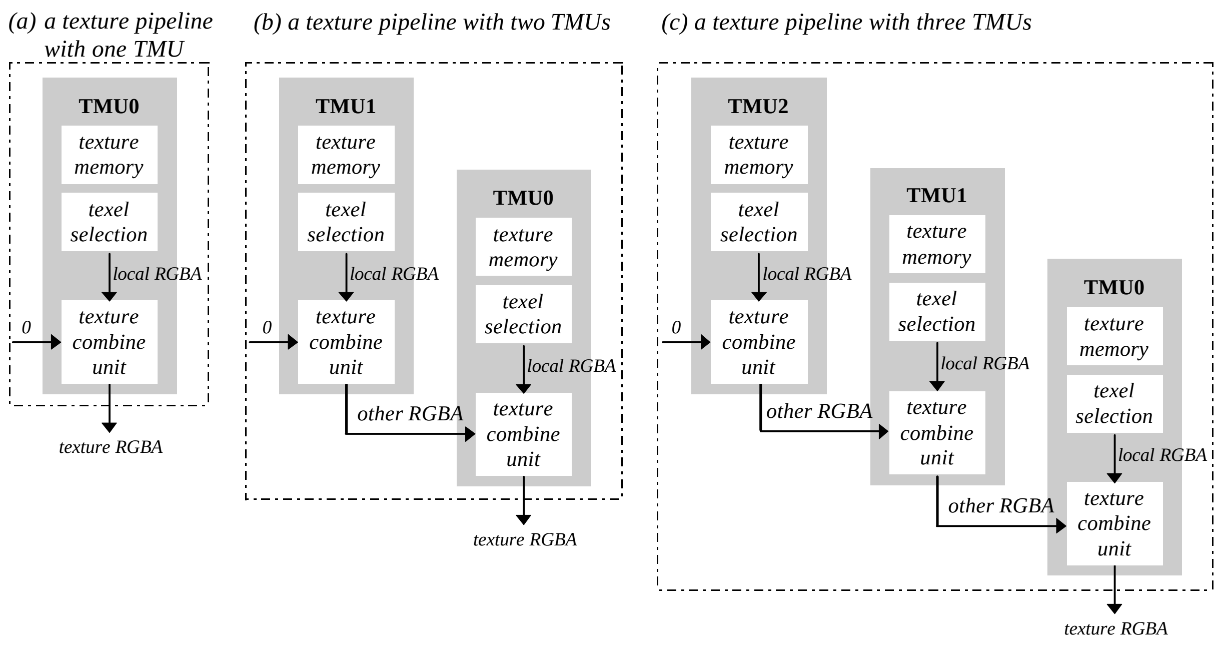
Texture memory is described in the next chapter. In this chapter, we assume that textures are already loaded into texture memory and concern ourselves with configuring the texel selection function and using the texture combine unit.

Figure 9.1 TMU connectivity.
A TMU contains texture memory, texture selection circuitry, and a texture combine unit. The texture combine units have ***other*** *and* ***local*** *datapaths just like the color and alpha combine units.*

(a) A system with one TMU extracts the appropriate texel or texels from texture memory, minifies or magnifies it, filters it, and clamps or wraps it according to texture map parameters or local overrides. The texture combine unit can scale the result.

(b) When the system has two TMUs they are chained together. The result from one TMU becomes an input to the texture combine unit of the next one and the texture RGBA that results is a user-selectable combination of the two textures.

(c) A three TMU system continues the cascading of texels.



Glide Textures and Texels

Textures are square or rectangular arrays of data; an individual value within a texture is called a *texel* and has an (*s*, *t*) address. The *s* and *t* texel coordinates are in the range [–32768..32767]. The large range for *s* and *t* allows a texture to be repeated many times across a polygon. A large number of fraction bits allows for precise *s* and *t* representation and iteration even when divided by a large *q* value.

For *one repeat* of the texture, the choice of coordinate systems determines the properties of *s* and *t*.

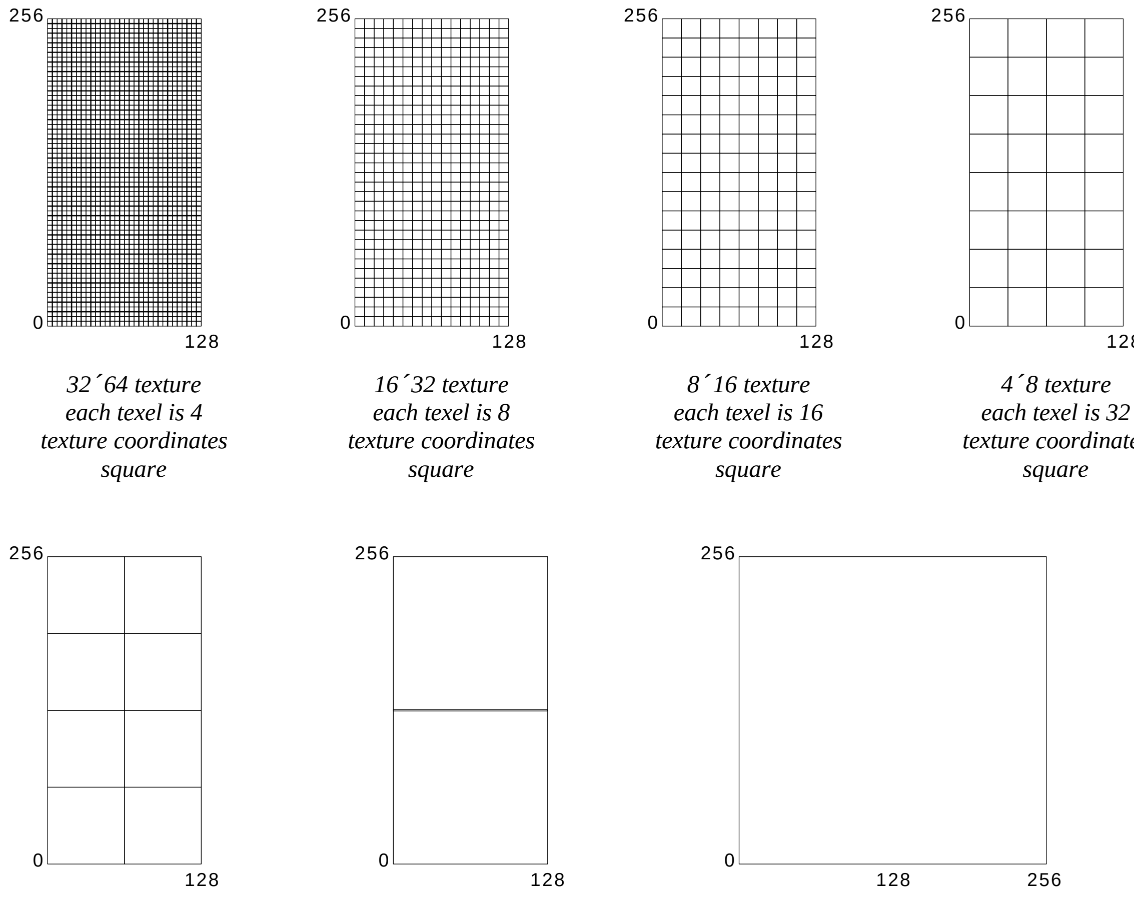
When Using Window Coordinates

All square texture maps have their origin at (*s*,*t*) = (0,0) and their opposite corner at (256,256). This is true even for a 1×1 texture map. Note that these texture coordinates are *before* division by *q*. Texture coordinate (0.5, 0.5) represents the exact center of the first texel in a 256×256 texture map, and

(255.5, 255.5) represents the exact center of the texel in the opposite corner; (256.5, 256.5) wraps to the center of the first texel. In general, the center of the first texel in a $2^n \times 2^n$ texture map (where $0 \le n \le 8$) is at ($128/2^n$, $128/2^n$).

Rectangular textures also have their origin at (0, 0). If the rectangular texture is wider than tall (*s* is larger than *t*) then the opposite corner is at (256, *n*) where *n*/256=*t/s*. For example, if the texture is four times as wide as high, then n=64. Likewise, if the rectangular texture is taller than it is wide, the opposite corner is at (*n*, 256) and *n*/256=*s/t*. Therefore, the longer texture axis always has texture coordinates running from 0 to 256, while the shorter texture axis is proportionally smaller. Table 9.1 shows the texel coordinates of the first and last pixel for all supported aspect ratios and texture map dimensions.

Figure 9.2 Mapping texels onto texture maps in window coordinate systems.
The textures shown below all have a 1:2 aspect ratio, and range in size from 32´64 to 1´2. In each one, the texture coordinates (s,t) range from (0,0) to (128,256). Thus, the texels get bigger (in terms of coverage of coordinate space) as the texture map size decreases. The degenerate case of 1´1 is shown for completeness.



32´64 texture each texel is 4 texture coordinates square

16´32 texture each texel is 8 texture coordinates square

8´16 texture each texel is 16 texture coordinates square

4´8 texture each texel is 32 texture coordinates square

2´4 texture each texel is 64 texture coordinates square

1´2 texture each texel is 128 texture coordinates square

1´1 texture single texel degenerate case

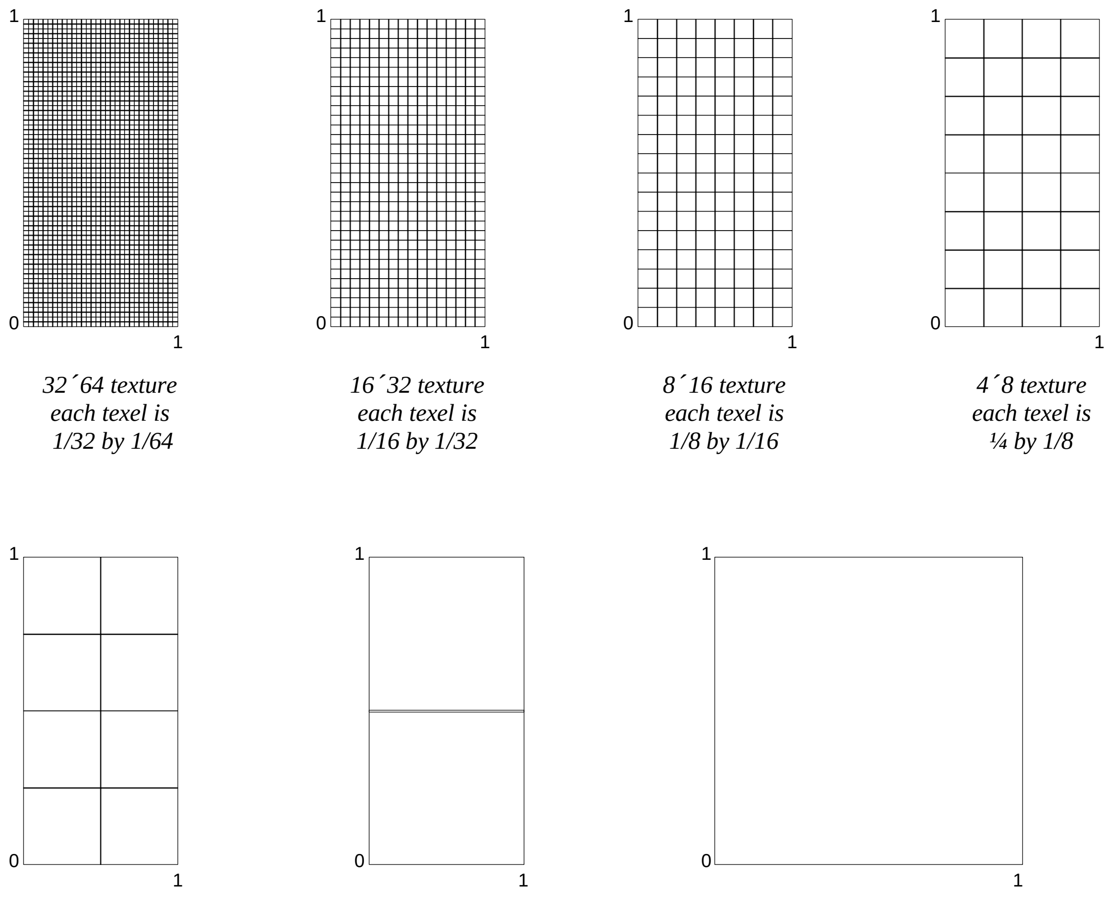
Table 9.1 Mapping pixels to texture coordinates in texture maps in window coordinate systems.
The texel coordinate on the long side of a texture map always goes from 0 to 256, regardless of the size of the texture map. Since texels are square, the texture coordinate on the short side of the texture map is scaled accordingly: it ranges from 0 to 256·(the ratio of the short to the long side). The degenerate cases are shaded.

if the aspect ratio is	*and the texture map size is*		*a texel is*	*the center of the first texel is at*	*the center of the last texel is at*	
1:1	256×256		1×1	(.5, .5)	(255.5, 255.5)	
(*a square texture*)	128×128		2×2	(1, 1)	(255, 255)	
	64×64		4×4	(2, 2)	(254, 254)	
	32×32		8×8	(4, 4)	(252, 252)	
	16×16		16×16	(8, 8)	(248, 248)	
	8×8		32×32	(16, 16)	(240, 240)	
	4×4		64×64	(32, 32)	(224, 224)	
	2×2		128×128	(64, 64)	(192, 192)	
	1×1		256×256	(128, 128)	(128, 128)	
2:1 or 1:2	*2:1*	*1:2*			*2:1*	*1:2*
(*the long side is twice the length of the short side*)	256×128	128×256	1×1	(.5, .5)	(255.5, 127.5)	(127.5, 255.5)
	128×64	64×128	2×2	(1, 1)	(255, 127)	(127, 255)
	64×32	32×64	4×4	(2, 2)	(254, 126)	(126, 254)
	32×16	16×32	8×8	(4, 4)	(252, 124)	(124, 252)
	16×8	8×16	16×16	(8, 8)	(248, 120)	(120, 248)
	8×4	4×8	32×32	(16, 16)	(240, 112)	(112, 240)
	4×2	2×4	64×64	(32, 32)	(224, 96)	(96, 224)
	2×1	1×2	128×128	(64, 64)	(192, 64)	(64, 192)
	1×1	1×1		(128, 128)	(128, 128)	(128, 128)
4:1 or 1:4	*4:1*	*1:4*			*4:1*	*1:4*
(*the long side is four times the length of the short side*)	256×64	64×256	1×1	(.5, .5)	(255.5, 63.5)	(63.5, 255.5)
	128×32	32×128	2×2	(1, 1)	(255, 63)	(63, 255)
	64×16	16×64	4×4	(2, 2)	(254, 62)	(62, 254)
	32×8	8×32	8×8	(4, 4)	(252, 60)	(60, 252)
	16×4	4×16	16×16	(8, 8)	(248, 56)	(56, 248)
	8×2	2×8	32×32	(16, 16)	(240, 48)	(48, 240)
	4×1	1×4	64×64	(32, 32)	(224, 32)	(32, 224)
	2×1	1×2		(64, 64)	(192, 64)	(64, 192)
	1×1	1×1		(128, 128)	(128, 128)	(128, 128)
8:1 or 1:8	*8:1*	*1:8*			*8:1*	*1:8*
(*the long side is eight times the length of the short side*)	256×32	32×256	1×1	(.5, .5)	(255.5, 31.5)	(31.5, 255.5)
	128×16	16×128	2×2	(1, 1)	(255, 31)	(31, 255)
	64×8	8×64	4×4	(2, 2)	(254, 30)	(30, 254)
	32×4	4×32	8×8	(4, 4)	(252, 28)	(28, 252)
	16×2	2×16	16×16	(8, 8)	(248, 24)	(24, 248)
	8×1	1×8	32×32	(16, 16)	(240, 16)	(16, 240)
	4×1	1×4		(32, 32)	(224, 32)	(32, 224)
	2×1	1×2		(64, 64)	(192, 64)	(64, 192)
	1×1	1×1		(128, 128)	(128, 128)	(128, 128)

When Using Clip Coordinates

All square texture maps have their origin at $(s,t) = (0,0)$ and their opposite corner at (1,1). This is true even for a 1×1 texture map. Note that these texture coordinates are *before* division by q, which is performed automatically. Rectangular textures also have their origin at (0, 0) and their opposite corner at (1,1). The center of the first texel in an $n \times m$ texture map is at $(1/2^{n+1}, 1/2^{m+1})$, and the center of the texel in the opposite corner is at $(1-(1/2^{n+1}), 1-(1/2^{m+1}))$.

Figure 9.3 Mapping texels onto texture maps in clip coordinate systems.
The textures shown below all have a 1:2 aspect ratio, and range in size from 32´64 to 1´2. In each one, the texture coordinates (s,t) range from (0,0) to (1,1). Thus, the texels get bigger (in terms of coverage of coordinate space) as the texture map size decreases. We have shown square texels and different scales on the s and t axis to parallel Table 9.1; however, this introduces distortion. The degenerate case of 1´1 is shown for completeness.



32´64 texture each texel is 1/32 by 1/64

16´32 texture each texel is 1/16 by 1/32

8´16 texture each texel is 1/8 by 1/16

4´8 texture each texel is ¼ by 1/8

2´4 texture each texel is ½ by ¼

1´2 texture each texel is 1 by ½

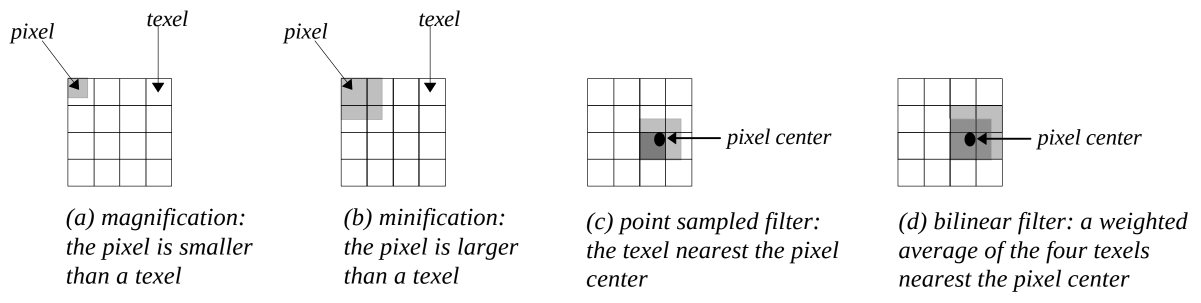
1´1 texture single texel degenerate case

Texture Filtering

All texture mapping capabilities of the graphics subsystem are handled in the TMU, which includes logic to support true-perspective texture mapping (dividing every pixel by *q*), per-pixel level-of-detail (LOD) mipmapping, and bilinear filtering. Additionally, TMU implements texture mapping techniques such as detail texture mapping, projected texture mapping, and trilinear filtering. While point sampled and bilinear filtering are single pass operations, single TMU systems require two passes for trilinear texture filtering. Multiple TMU systems support trilinear texture filtering as a single-pass operation. Note that regardless of the number of TMUs in a given graphics system, there is no performance difference between point-sampled and bilinear filtered texture-mapped rendering, and no performance penalty for per-pixel mipmapping or perspective correction.

Texture maps are square or rectangular, but after being mapped to a polygon or surface and transformed into screen coordinates, the individual texels of a texture map rarely correspond to screen pixels on a one-to-one basis. Depending on the transformations used and the texture mapping applied, a single pixel on the screen can correspond to anything from a tiny portion of a texel, resulting in magnification, to a large collection of texels, resulting in minification. In either case it is unclear exactly which texel values should be used and how they should be averaged or interpolated. Consequently, Glide allows an application to choose between two types of filtering: point sampling and bilinear interpolation.

Figure 9.4 Point sampling and bilinear filtering.
Glide supports two methods of choosing a texel within a texture map. If the pixel maps to less than one texel, as shown in diagram (a), texture magnification is called. If the pixel maps to more than one texel, as shown in diagram (b), then minification is required. The user can select between point-sampling and bilinear filtering during the minification or magnification. When using point sampling, the texel whose (s, t) coordinates are nearest the center of the pixel is chosen. Bilinear filtering computes a weighted average of the 2 by 2 array of texels that lie nearest the center of the pixel. The magnification and minification filters are independent: one can specify point sampling and the other bilinear filtering, or both can be the same.


pixel
texel
pixel
texel
pixel center
pixel center
(a) magnification: the pixel is smaller than a texel
(b) minification: the pixel is larger than a texel
(c) point sampled filter: the texel nearest the pixel center
(d) bilinear filter: a weighted average of the four texels nearest the pixel center

Magnification of a texture map occurs when a texture map is “blown up” on screen (see Figure 9.4(a)). For example, if a 64×64 texture map is rendered onto a polygon that covers 128×128 pixels on the screen, an average of four pixels will cover each texel in the texture map, causing noticeable blockiness. The graphics hardware supports bilinear interpolation of texels that greatly reduces the blockiness and pixelization of texture magnification.

Minification of a texture map occurs when a texture map is compressed on screen (see Figure 9.4(b)). For example, if a 64×64 texture map is rendered onto a polygon that only covers 16×16 pixels on the screen, an average of 16 texels will cover each pixel on the screen. This leads to disturbing artifacts known as “texture aliasing”. The graphics hardware remedies this problem by supporting both mipmapping and filtering.

If a graphics subsystem is performing *point sampled* filtering, the texel with coordinates nearest the center of the pixel being rendered is used to generate the color output on the screen (see Figure 9.4(c)). Point sampling, also known as nearest neighbor sampling, may result in pixelization and blockiness during magnification and “texture jerking” during minification.

One way of reducing the blockiness of point sampling is by linearly interpolating between the colors of the texels that are adjacent to the source pixel, which results in a much smoother image than point sampling (see Figure 9.4(d)). *Bilinear interpolation* is performed by the graphics hardware with no incurred additional performance overhead.

Minification and magnification filtering are controlled by the Glide function **grTexFilterMode()** and are independently selectable.

```
void grTexFilterMode( GrChipID_t tmu,
                      GrTextureFilterMode_t minFilterMode,
                      GrTextureFilterMode_t magFilterMode
                    )
```

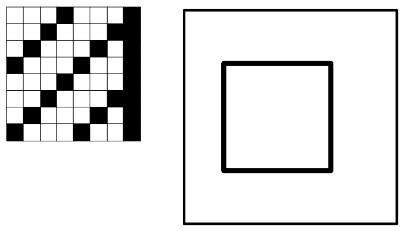
The first argument, *tmu*, selects the texture mapping unit that the filter selections apply to. Valid values are `GR_TMU0`, `GR_TMU1`, and `GR_TMU2`. The minification filter, *minFilterMode*, can be either `GR_TEXTUREFILTER_POINT_SAMPLED` or `GR_TEXTUREFILTER_BILINEAR`, as can the magnification filter, *magFilterMode*. The magnification filter is used when the LOD calculated for a pixel indicates that the pixel covers less than one texel. Otherwise, the minification filter is used.

Texture Clamping

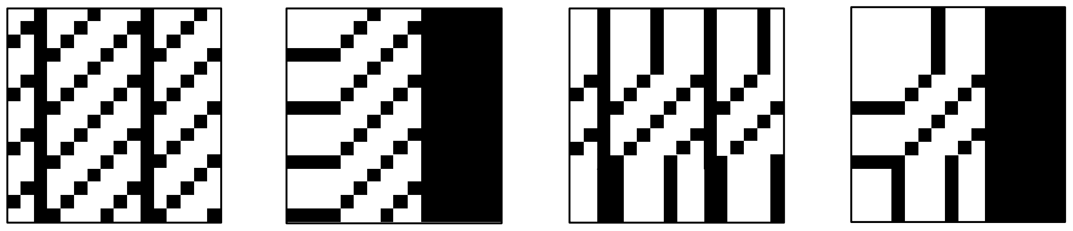
When texture *s* and *t* coordinates have overflowed during a texture mapped rendering operation, the hardware can either clamp the coordinates to a maximum value or, alternatively, wrap them around. This choice is up to the developer depending on whether tiled or non-tiled texture mapping is desired. Texture clamping also allows for interesting effects, for example, out of range *s* and *t* coordinates can be passed with a very small texture in a large polygon. Such an approach will effectively place the texture somewhere in the interior of the polygon with the rest of the polygon rendered with the border color of the texture. This can potentially save texture memory if small composite textures are used on a predominantly monotone surface, e.g., a window on the side of a space ship.

Figure 9.5 Texture clamping.
The texture clamp mode specifies what to do when texture coordinates are outside the range of the texture map. If wrapping is enabled, then texture maps will tile, i.e., values greater than 255 will wrap around to 0. If clamping is enabled, then texture map indices will be clamped to 0 and 255. Both modes should always be set to `GR_TEXTURECLAMP_CLAMP` when using projected textures.

Glide 3.0 introduces a texture clamp mode extension, `GR_TEXTURECLAMP_MIRROR_EXT`, that is available if the TEXMIRROR extension is supported. See Chapter 13 for details and an expanded version of this figure.



The texture on the left is to be mapped onto the rectangle, with the texture origin in the interior of the rectangle. The clamp mode settings for s and t affect the final result, as shown below.



Note that *s* and *t* coordinates may be individually wrapped or clamped, as shown in Figure 9.5.

```
void grTexClampMode( GrChipID_t tmu,
                     GrTextureClampMode_t sClampMode,
                     GrTextureClampMode_t tClampMode
                    )
```

The first argument, *tmu*, selects the TMU in which the mipmap resides and may be `GR_TMU0`, `GR_TMU1`, or `GR_TMU2`. The other two arguments set the clamping mode for *s* and *t* individually; they may be set to `GR_TEXTURECLAMP_CLAMP`, `GR_TEXTURECLAMP_WRAP`, or, if supported, `GR_TEXTURECLAMP_MIRROR_EXT` (see the discussion on the TEXMIRROR extension in Chapter 13). If wrapping is enabled, texture maps will tile: values greater than 255 will wrap around to 0. If clamping is enabled, texture map indices will be clamped to 0 and 255. Both modes should always be set to `GR_TEXTURECLAMP_CLAMP` when using projected textures.

Mipmapping

A mipmap is an ordered set of texture maps representing the same texture; each texture map has lower resolution than the previous one, and is typically derived by filtering and averaging down its predecessor. LOD0 is the name given to the texture with the highest resolution in the mipmap, where LOD stands for “level of detail”. The LOD1 texture, if defined, is half as high and half as wide, and defines one-quarter as many texels as LOD0. There can be up to nine texture maps in a mipmap.

PORTING NOTE

The GR_LOD and GR_ASPECT constants have been redefined: the value now represents the logarithm, base 2, of the largest dimension. In order to call attention to code that used them, the names have been changed as well, adding "LOG2_", as shown in Table 9.2 above.

Some code that used the old constants will require modification. For example, a Glide 2.x `for` loop that decrements a counter to access larger mipmap levels will increment the counter in Glide 3.0. Any tables indexed by mipmap level or aspect ratio must also be examined to see if changes are required.

Figure 9.6 gives a graphical representation of a complete mipmap. The texture maps can be square or rectangular, but each one in the mipmap must have the same aspect ratio. See Table 9.2.

The next chapter will describe Glide functions that manage texture memory and load textures and mipmaps. In this chapter, we will assume that the proper textures are already loaded; we will focus on the texel selection and texture combine capabilities.

Table 9.2 Texture sizes and shapes.

A mipmap can be composed of up to nine textures (the LOD names are shown in column 1) and can be square or rectangular (the aspect ratios are listed in row 1). All textures within a mipmap must have the same aspect ratio. The shaded entries in the table below have degenerate aspect ratios: one or both dimensions have been reduced to one texel.

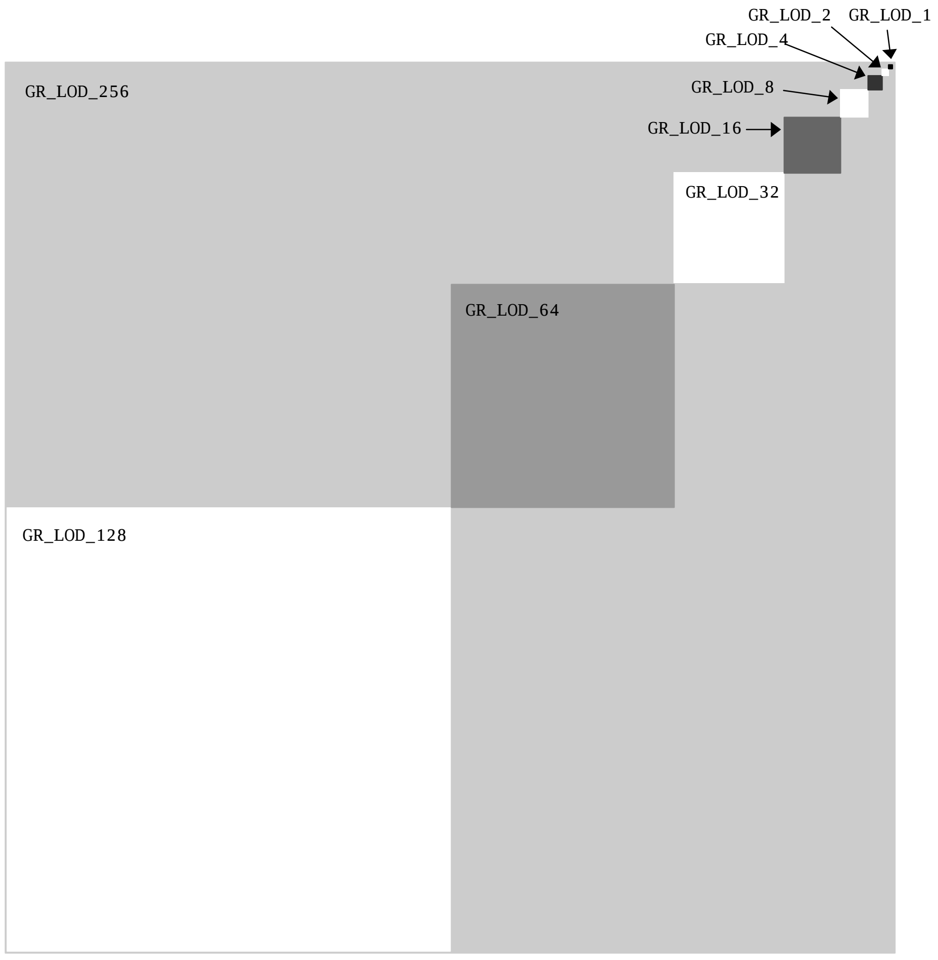
	GR_ASPECT_LOG2_1x1	GR_ASPECT_LOG2_2x1 *or* GR_ASPECT_LOG2_1x2	GR_ASPECT_LOG2_4x1 *or* GR_ASPECT_LOG2_1x4	GR_ASPECT_LOG2_8x1 *or* GR_ASPECT_LOG2_1x8
GR_LOD_LOG2_256	256×256	256×128 *or* 128×256	256×64 *or* 64×256	256×32 *or* 32×256
GR_LOD_LOG2_128	128×128	128×64 *or* 64×128	128×32 *or* 32×128	128×16 *or* 16×128
GR_LOD_LOG2_64	64×64	64×32 *or* 32×64	64×16 *or* 16×64	64×8 *or* 8×64
GR_LOD_LOG2_32	32×32	32×16 *or* 16×32	32×8 *or* 8×32	32×4 *or* 4×32
GR_LOD_LOG2_16	16×16	16×8 *or* 8×16	16×4 *or* 4×16	16×2 *or* 2×16
GR_LOD_LOG2_8	8×8	8×4 *or* 4×8	8×2 *or* 2×8	8×1 *or* 1×8
GR_LOD_LOG2_4	4×4	4×2 *or* 2×4	4×1 *or* 1×4	4×1 *or* 1×4
GR_LOD_LOG2_2	2×2	2×1 *or* 1×2	2×1 *or* 1×2	2×1 *or* 1×2
GR_LOD_LOG2_1	1×1	1×1	1×1	1×1

PORTING NOTE

The `GR_LOD` and `GR_ASPECT` constants have been redefined: the value now represents the logarithm, base 2, of the largest dimension. In order to call attention to code that used them, the names have been changed as well, adding “`LOG2_`”, as shown in Table 9.2 above.

Some code that used the old constants will require modification. For example, a Glide 2.x `for` loop that decrements a counter to access larger mipmap levels will increment the counter in Glide 3.0. Any tables indexed by mipmap level or aspect ratio must also be examined to see if changes are required.

Figure 9.6 Mipmaps.
A mipmap is an ordered set of texture maps representing the same texture. Each texture map in the set has lower resolution than the previous one, and is typically derived by filtering and averaging down its predecessor. `GR_LOD_LOG2_256` *is the name given to the texture with the highest resolution in the mipmap, where LOD stands for "level of detail". The* `GR_LOD_LOG2_128` *texture is half as high and half as wide, and defines one-quarter as many texels as its predecessor, and so on. The mipmap can contain up to nine texture maps, as shown. The texel addresses range from (0,0) to (256,256) in window coordinates, or from (0,0) to (1,1) in clip coordinates, in all nine textures, as described earlier in the chapter.*



The hardware computes an LOD for every pixel. The integer part of the LOD is used to choose one (or two) of the textures in the current mipmap; the fractional part is used to blend two mipmap levels if desired.

- *Nearest mipmapping*. The mipmap level is chosen based on which mipmap is nearest to a pixel's LOD. Nearest mipmapping may suffer from a visual artifact known as "mipmap banding" that manifests itself as visible bands between LOD levels appearing in a texture mapped image.
- *Nearest dithered mipmapping*. To offset the effects of mipmap banding, the hardware can dither between adjacent texture maps within a mipmap. This technique, known as nearest dithered mipmapping, alleviates the effects of mipmap banding to a great extent, at the cost of performance degradation for larger texture maps.

void **grTexMipMapMode(** *GrChipID_t tmu*, *GrMipMapMode_t mode*, *FxBool LODblend* **)**

Mipmapping style is controlled by **grTexMipMapMode()**. The first argument, *tmu*, designates the TMU to modify. The second argument, *mode*, selects the mipmapping style; valid values are `GR_MIPMAP_DISABLE`, `GR_MIPMAP_NEAREST`, and `GR_MIPMAP_NEAREST_DITHER`. The final argument, *LODblend*, enables or disables blending between levels of detail in the mipmap. `GR_MIPMAP_NEAREST` should be used when *LODblend* is `FXTRUE`.

Using dithered mipmapping with bilinear filtering results in images almost indistinguishable from images rendered with trilinear filtering techniques. On the down side, dithering of the mipmap levels reduces the peak fill rate by approximately 20% to 30%, depending on the scene being rendered. Since the presence or absence of mipmap dithering is not very noticeable, it is very hard to determine the cause of the performance loss. Therefore, Glide disallows this mode by default. An application may explicitly allow the use of dithered mipmapping by issuing a **grEnable(**`GR_ALLOW_MIPMAP_DITHER`**)** command (see Chapter 12).

If you are considering using dithered mipmapping, measure performance with and without it. The trade-off is that there may be visible mipmap bands, which can be eliminated by using trilinear mipmapping. On multiple TMU boards this is a one-pass operation, otherwise it requires two passes. Alternatively, dithered mipmapping can be allowed but disabled for most polygons and enabled only for those polygons that require it.

If there is no performance difference with and without dithered mipmapping, but the image quality did not improve with dithered mipmapping, don’t use it. As you enhance or extend your program, you run the risk of creating a situation in which performance loss due to dithered mipmapping could occur. It is best to selectively enable dithered mipmapping just for the polygons that require it.

Mipmap Blending

To reduce the effects of mipmap banding the hardware can perform a weighted blend between adjacent mipmap levels. This blend is a single pass operation on two TMU configurations and a two-pass operation on a single TMU configurations.

Mipmap blending can be performed independently of the type of minification and magnification filtering being performed. Since mipmap blending is actually a form of texture combining, it is controlled by proper set up of the texture combine function.

Trilinear Filtering

The combination of bilinear filtering, mipmapping, and mipmap blending is generally known as *trilinear mipmapping*. Trilinear mipmapping provides maximum visual quality by performing inter- and intra-mipmap blending. However, trilinear mipmapping is a two-pass operation on graphics subsystems with a single TMU. Nearest dithered mipmapping results in nearly the same visual quality as trilinear texture mapping, however, it is always a single pass operation and thus achieves consistent performance across a wider range of hardware.

LOD Bias

LOD bias affects the point at which mipmapping levels change. Increasing values for LOD bias makes the overall images blurrier or smoother. Decreasing values make the overall images sharper. Selection of LOD bias is a qualitative judgment that is application and texture dependent. LOD bias can be any value in the range [–8.0..7.75]. However, the hardware will snap LOD bias to the nearest quarter. There is no “best” setting for the LOD bias; it is a very subjective control. Some textures look better if sharper than “normal,” while others look better blurred.

The LOD bias is controlled with the function **grTexLodBiasValue()**. The first argument, *tmu*, identifies the TMU to modify; valid values are `GR_TMU0`, `GR_TMU1`, and `GR_TMU2`. The second argument, *bias*, is a signed floating point value in the range [–8..7.75].

void **grTexLodBiasValue(** *GrChipID_t tmu*, float *bias* **)**

grTexLodBiasValue() changes the current LOD bias value, which allows an application to maintain fine grain control over the effects of mipmapping, specifically when mipmap levels change. The LOD bias value is added to the LOD calculated for a pixel and the result determines which mipmap level to use. An LOD of *n* is calculated when a pixel covers approximately 2^{2n} texels. For example, when a pixel covers approximately one texel, the LOD is 0; when a pixel covers four texels, the LOD is 1; when a pixel covers 16 texels, the LOD is 2. Smaller LOD values make increasingly sharper images that may suffer from aliasing and moiré effects. Larger LOD values make increasingly smooth images that may suffer from becoming too blurry. The default LOD bias value is 0.0.

During some special effects, an LOD bias may help image quality. If an application is not performing texture mapping with trilinear filtering or dithered mipmapping, then an LOD bias of +.5 generally improves image quality by rounding to the nearest LOD. If an application is performing dithered mipmapping (i.e. **grTexMipMapMode()** is `GR_MIPMAP_NEAREST_DITHER`), then an LOD bias of 0.0 or +.25 generally improves image quality. An LOD bias value of 0.0 is usually best with trilinear filtering.

Combining Textures

The graphics hardware can combine multiple textures together simultaneously. This allows for interesting effects including detail texturing, projected texturing, and trilinear texture mapping. Combining two textures requires a single pass with two TMUs or two passes with a single TMU. Combining two textures is controlled with the function **grTexCombine()**.

Each TMU selects an appropriate texel for the current rendering mode and filters it (point sampled or bilinear, as determined by a mipmap’s associated filtering mode or the most recent call to **grTexFilterMode()**), then passes the texel on to the texture combine unit. The texture combine unit combines the filtered texel with the incoming texel from the other TMUs, according to the user-selectable formula defined by the most recent **grTexCombine()** function. The simplest combine function is a simple pass-through that implements decal texture mapping. However, more elaborate texture mapping combinations can be used to implement useful effects such as trilinear mipmapping, composite texturing, and projected textures.

```
void grTexCombine( GrChipID_t tmu,
                   GrCombineFunction_t rgbFunction,
                   GrCombineFactor_t rgbFactor,
                   GrCombineFunction_t alphaFunction,
                   GrCombineFactor_t alphaFactor,
                   FxBool rgbInvert,
                   FxBool alphaInvert
                  )
```

The first argument names the TMU to which the rest of the arguments apply. Valid values are `GR_TMU0`, `GR_TMU1`, and `GR_TMU2`. The next two arguments, *rgbFunction* and *rgbFactor*, describe the combining function and scale factor for the red, green, and blue components produced by the texel selection circuitry of *tmu*. Similarly, *alphaFunction* and *alphaFactor* define the combining function and scale factor for the alpha component. Table 9.3 lists the possible combining functions; the scale factors are detailed in Table 9.4. In both tables, c_{local} and a_{local} represent the color components generated by indexing and filtering from the mipmap stored on *tmu*; c_{other} and a_{other} represent the incoming color components from the neighboring TMU (refer to Figure 9.1).

The texture combine units compute the function specified by the *rgbFunction* and *alphaFunction* combine functions and the *rgbFactor* and *alphaFactor* combine scale factors on the local filtered texel and the filtered texel from the upstream TMU. The result is clamped to [0..255], and then a bit-wise inversion may be applied, controlled by the *rgbInvert* and *alphaInvert* parameters. Inverting the bits in an 8-bit color component is the same as computing (255 – *c*).

grTexCombine() also keeps track of required vertex parameters for the rendering routines. `GR_COMBINE_FACTOR_NONE` is provided to indicate that no parameters are required. Currently it is the same as `GR_COMBINE_FACTOR_ZERO`.

Table 9.3 Texture combine functions.
*The **rgbFunction** and **alphaFunction** arguments to **grTexCombine()** can take on any of the values listed in the first column. The second and third columns show the computed color or alpha value for each choice. c_{local} and a_{local} represent the color components generated by indexing and filtering from the mipmap stored on **tmu**; c_{other} and a_{other} represent the incoming color components from the neighboring TMU (refer to Figure 9.1).*

texture combine function (prefixed with `GR_COMBINE_FUNCTION_`*)*	*computed color if specified as **rgbFunction***	*computed alpha if specified as **alphaFunction***
`ZERO`	0	0
`LOCAL`	c_{local}	a_{local}
`LOCAL_ALPHA`	a_{local}	a_{local}
`SCALE_OTHER` `BLEND_OTHER`	$f * c_{other}$	$f * a_{other}$
`SCALE_OTHER_ADD_LOCAL`	$f * c_{other} + c_{local}$	$f * a_{other} + a_{local}$
`SCALE_OTHER_ADD_LOCAL_ALPHA`	$f * c_{other} + a_{local}$	$f * a_{other} + a_{local}$
`SCALE_OTHER_MINUS_LOCAL`	$f * (c_{other} - c_{local})$	$f * (a_{other} - a_{local})$
`SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL` `BLEND`	$f * (c_{other} - c_{local}) + c_{local}$ $\equiv f * c_{other} + (1 - f) * c_{local}$	$f * (a_{other} - a_{local}) + a_{local}$ $\equiv f * a_{other} + (1 - f) * a_{local}$
`SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL_ALPHA`	$f * (c_{other} - c_{local}) + a_{local}$	$f * (a_{other} - a_{local}) + a_{local}$
`SCALE_MINUS_LOCAL_ADD_LOCAL` `BLEND_LOCAL`	$f * (- c_{local}) + c_{local}$ $\equiv (1 - f) * c_{local}$	$f * (- a_{local}) + a_{local}$ $\equiv (1 - f) * a_{local}$
`SCALE_MINUS_LOCAL_ADD_LOCAL_ALPHA`	$f * (- c_{local}) + a_{local}$	$f * (-a_{local}) + a_{local}$

Table 9.4 Scale factors for texture color generation.
*The **rgbFactor** and **alphaFactor** arguments to **grTexCombine()** can take on any of the values listed in the first column. The second and third columns show the scale factor that will be used. c_{local} and a_{local} represent the color components generated by indexing and filtering from the mipmap stored on **tmu**; c_{other} and a_{other} represent the incoming color components from the neighboring TMU (refer to Figure 9.1).*

*If `GR_COMBINE_FACTOR_DETAIL_FACTOR` or `GR_COMBINE_FACTOR_ONE_MINUS_DETAIL_FACTOR` is specified, the scale factor employs the detail blend factor, called b in the table. See the discussion of **grTexDetailControl()** in the next section for more information.*

If `GR_COMBINE_FACTOR_LOD_FRACTION` or `GR_COMBINE_FACTOR_ONE_MINUS_LOD_FRACTION` is specified, the scale factor employs the fractional part of the computed LOD, called l in the table. See the discussion about computing an LOD earlier in this chapter for more information.

texture combine factor (prefixed with `GR_COMBINE_FACTOR_`*)*	*scale factor **f** if specified as **rgbFactor***	*scale factor **f** if specified as **alphaFactor***
`NONE`	*unspecified*	*unspecified*
`ZERO`	0	0
`LOCAL`	c_{local} / 255	a_{local} / 255
`OTHER_ALPHA`	a_{other} / 255	a_{other} / 255
`LOCAL_ALPHA`	a_{local} / 255	a_{local} / 255
`DETAIL_FACTOR`	β	β
`LOD_FRACTION`	λ	λ
`ONE`	1	1
`ONE_MINUS_LOCAL`	$1 - c_{local}$ / 255	$1 - a_{local}$ / 255
`ONE_MINUS_OTHER_ALPHA`	$1 - a_{other}$ / 255	$1 - a_{other}$ / 255
`ONE_MINUS_LOCAL_ALPHA`	$1 - a_{local}$ / 255	$1 - a_{local}$ / 255
`ONE_MINUS_DETAIL_FACTOR`	$1 - \beta$	$1 - \beta$

`ONE_MINUS_LOD_FRACTION`	$1 - \lambda$	$1 - \lambda$

Examples of Configuring the Texture Pipeline

The following code examples illustrate how to configure the texture pipeline for different texture mapping effects. The examples all assume that appropriate textures have been loaded and the addressing mechanism in the TMU points to the right place. This process is described in detail in the next chapter; the examples are repeated there, with the texture loading segments filled in. The examples also assume that **grColorCombine()** and/or **grAlphaCombine()** utilize texture mapping by setting the scale factor to `GR_COMBINE_FACTOR_TEXTURE_ALPHA` or `GR_COMBINE_FACTOR_ONE_MINUS_TEXTURE_ALPHA`.

The examples in this chapter attempt to cover most of the texture mapping techniques of interest. Table 9.5 shows the principle texture mapping algorithms and describes the implementation in terms of available TMUs. We show examples utilizing one or two TMUs, mipmaps split across two TMUs, and a two-pass application.

Table 9.5 The number of TMUs affects texture mapping functionality.
The number of texture mapping units determines the performance of advanced texture mapping rendering. The table below describes the number of passes required to implement the texture mapping techniques supported by the graphics subsystem. Note that in a system with three TMUs, only the most complicated algorithm (trilinear filtering with mipmapping, projected, and detail textures) requires more than one pass.

texture mapping functionality	*performance*		
	one TMU	*two TMUs*	*three TMUs*
Point sampling with mipmapping	one pass	one pass	one pass
Bilinear filtering with mipmapping	one pass	one pass	one pass
Bilinear filtering with mipmapping and projected textures	two pass	one pass	one pass
Bilinear filtering with mipmapping and detail textures	two pass	one pass	one pass
Bilinear filtering with mipmapping, projected and detail textures	*not supported*	two pass	one pass
Trilinear filtering with mipmapping	two pass	one pass	one pass
Trilinear filtering with mipmapping and projected textures	*not supported*	two pass	one pass
Trilinear filtering with mipmapping and detail textures	*not supported*	two pass	one pass
Trilinear filtering with mipmapping, projected, and detail textures	*not supported*	two pass	two pass

Configuring the Texture Pipeline for Decal Texture Mapping

The simplest texture mapping technique is decal mapping, which applies a texture to a polygon without modification. The first two entries in Table 9.5 are decal mapping, differing only in the choice of minification and magnification filters. Decal mapping is a single pass operation on all 3Dfx Interactive graphics accelerator configurations.

Example 9.1 Setting up simple (decal) texture mapping.
The following code sets up the texture pipeline so that a texel is placed into the pixel pipeline without modification. The code assumes that there is a single TMU, that a texture has already been loaded into texture

memory with the texture base address pointing to it, and that the color combine unit is configured to use the texture color and/or alpha value.

```
grTexCombine( GR_TMU0, GR_COMBINE_FUNCTION_LOCAL,
GR_COMBINE_FACTOR_NONE,
              GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
              FXFALSE, FXFALSE );
```

Configuring the Texture Pipeline for Projected Texture Mapping

Interesting spotlight effects are possible by multiplying two texture maps against each other. For example, one texture map can be an intensity map (e.g., a spotlight) and the other can be a source texture. Recall that the texture RGBA values from the "upstream" TMU1 become the *other* input to the "downstream" TMU0. In Example 9.2, the spotlight texture is upstream, the source texture is downstream and the resulting $RGBA_{texture} = RGBA_{spotlight} \times RGBA_{source}$.

Example 9.2 Applying a modulated (projected) texture.

The code segment below assumes that the texture maps have already been loaded: an intensity map for the spotlight in TMU0 and a source texture in TMU1. The resulting texture RGBA is a product of the texels chosen from the two textures. The color combine unit must be configured to use the output from the texture pipeline.

```
grTexCombine( GR_TMU0,
              GR_COMBINE_FUNCTION_SCALE_OTHER, GR_COMBINE_FACTOR_LOCAL,
              GR_COMBINE_FUNCTION_SCALE_OTHER, GR_COMBINE_FACTOR_LOCAL,
              FXFALSE, FXFALSE );

grTexCombine( GR_TMU1,
              GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
              GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
              FXFALSE, FXFALSE );
```

Configuring the Texture Pipeline for Trilinear Texture Mapping

When doing standard mipmapping, noticeable banding can occur because of the visible differences in mipmap levels. One way around this is to blend two separate textures within a mipmap based on the LOD (level of detail) fraction bits. This is known as mipmap blending which, in conjunction with bilinear filtering, is referred to as trilinear texture mapping. To perform trilinear texture mapping the application must download a texture specifically for use with trilinear mipmapping and then use this texture only for blended mipmapping operations.

When using texture combining to implement mipmap blending (i.e., trilinear texture mapping), mipmaps must be created specifically for trilinear texture mapping on each Texel*fx* chip. The odd levels must be downloaded to one chip, and the even levels must be downloaded to another chip. The mipmaps must have the trilinear variable set to `FXTRUE` (see Chapter 10). The texture combine unit on the downstream TMU is set differently, depending on whether it holds the even or the odd LODs. The upstream TMU always uses decal mapping.

If a texture will be used for both trilinear filtering and another combine operation (but not simultaneously), it must be allocated and downloaded twice, once with *LODblend* set to `FXTRUE` and the other time with *LODblend* set to `FXFALSE`.

Example 9.3 Using trilinear filtering: mipmap blending with bilinear filtering.
The first code segment shows the texture combine unit configuration for trilinear mipmapping when the even LODs are stored in TMU0 and the odd ones are in TMU1. As usual, the code assumes that the textures are loaded, the TMU base registers are pointing to them, and the color combine unit is configured to make use of the resulting RGBA value.

```
grTexCombine( GR_TMU0,
              GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
              GR_COMBINE_FACTOR_LOD_FRACTION,
              GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
              GR_COMBINE_FACTOR_LOD_FRACTION,
              FXFALSE, FXFALSE);

grTexCombine( GR_TMU1, GR_COMBINE_FUNCTION_LOCAL,
GR_COMBINE_FACTOR_NONE,
              GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
              FXFALSE, FXFALSE );
```

This second code segment gives the proper ***grTexCombine()*** *configuration when the situation is reversed: the odd LODs in the mipmap are on TMU0 while the even ones are upstream on TMU1. Note the difference: the setting of the* ***rgbInvert*** *and* ***alphaInvert*** *parameters. We make the same assumptions as above.*

```
grTexCombine( GR_TMU0,
              GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
              GR_COMBINE_FACTOR_ONE_MINUS_LOD_FRACTION,
              GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
              GR_COMBINE_FACTOR_ONE_MINUS_LOD_FRACTION,
              FXFALSE, FXFALSE);

grTexCombine( GR_TMU1, GR_COMBINE_FUNCTION_LOCAL,
GR_COMBINE_FACTOR_NONE,
              GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
              FXFALSE, FXFALSE );
```

Configuring the Texture Pipeline for Composite Texturing

When a bilinear-filtered texture-mapped surface is viewed closely, the resulting image may be blurry and overly soft. A technique known as composite texturing can remedy this blurriness. Composite texturing blends two textures together based on their LOD values. One texture represents the overall texture look, and the other texture represents the details that should be seen when the texture is viewed closely. For example, brick can be represented with a tiled brick pattern. As the viewer moves closer to the wall, pits and cracks in the bricks could begin to appear by blending a separate “pits and cracks” texture into the brick based on the LOD value.

The Glide function **grTexDetailControl()** manages the various parameters involved when performing composite texture mapping.

void **grTexDetailControl(** *GrChipID_t tmu*, int *detailBias*, *FxU8 detailScale*, float *detailMax* **)**

The first argument specifies the TMU to modify; valid values are `GR_TMU0`, `GR_TMU1`, and `GR_TMU2`. The second argument, *detailBias*, controls where the blending between the two textures begins and is an integer in the range [–32..31]. The *detailScale* argument controls the steepness of the blend; valid values are [0..7]. The scale is computed as 2^{detail_scale}. The *detailMax* argument specifies the maximum blending that will occur and is in the range [0..1].

Detail texturing refers to the effect where the blend between two textures in a texture combine unit is a function of the LOD calculated for each pixel. **grTexDetailControl()** controls how the detail blending factor, β, is computed from LOD. The *detailBias* parameter controls where the blending begins; the *detailScale* parameter controls how fast the detail shows up; and the *detailMax* parameter controls the maximum blending that occurs.

$$\beta = \min(\ \mathit{detailMax},\ \max(\ 0,\ (\mathit{detailBias} – \mathrm{LOD}) \ll \mathit{detailScale}\ )\ /\ 255.0\ )$$

where LOD is the calculated LOD before **grTexLodBiasValue()** is added. The detail blending factor is utilized by calling **grTexCombine()** with an *rgbFunction* of `GR_COMBINE_FUNCTION_BLEND` and an *rgbFactor* of `GR_COMBINE_FACTOR_DETAIL_FACTOR` to compute:

$$c_{out} = \beta(c_{detail\ texture}) + (1–\beta)(c_{main\_texture})$$

An LOD of *n* is calculated when a pixel covers approximately 2^{2n} texels. For example, when a pixel covers approximately one texel, the LOD is 0; when a pixel covers four texels, the LOD is 1; when a pixel covers 16 texels, the LOD is 2.

Detail blending occurs in the downstream TMU. Since the detail texture and main texture typically have very different computed LODs, the detail texturing control settings depend on which texture is in the downstream TMU.

Example 9.4 Creating a composite texture.
The code segment below creates a composite texture by adding details to the primary texture as the viewer approaches. The primary texture is loaded onto TMU0 while the detail texture is upstream on TMU1. The scale factor `GR_COMBINE_FACTOR_DETAIL_FACTOR` creates the composite on TMU0, while TMU1 does decal mapping.

```
grTexCombine( GR_TMU0,
              GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
              GR_COMBINE_FACTOR_DETAIL_FACTOR,
              GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
              GR_COMBINE_FACTOR_DETAIL_FACTOR,
              FXFALSE, FXFALSE);

grTexCombine( GR_TMU1, GR_COMBINE_FUNCTION_LOCAL,
GR_COMBINE_FACTOR_NONE,
              GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
              FXFALSE, FXFALSE );
```

10. Managing Texture Memory

In This Chapter

In the last chapter, the routines that control texel selection and texture combining on the TMU were presented. The discussion assumed that appropriate textures had already been loaded into the texture memory. This chapter describes the multitude of texture formats that Glide supports and the routines that download texture maps and manage texture memory.

You will learn about:

▼ the texture formats supported by Glide, including special formats for compressed textures and a color palette.

▼ how to allocate memory for all or part of a mipmap.

▼ how to download all or part of a mipmap.

▼ how to designate a specific texture map as the texel source.

▼ how to split a mipmap across two TMUs.

▼ how to download and access a fragmented mipmap, one in which successive LODs occupy non-contiguous texture memory.

▼ how to download a color palette or a narrow channel decompression table.

▼ how to download a texture map from a file.

Texture Map Formats

Texture memory is a valuable and limited resource. Glide supports a multitude of texture formats in order to help the application programmer use texture memory wisely. Each format encodes the color information for each texel in a different way; most compress it in some manner. Texels have either 8 or 16 bits, depending on the texture format, and are expanded to 32 bits before being sent to the texture combine unit.

Glide uses symbolic names for the texture formats; the name describes the form of encoding for the color information and the precision. For example:

- Texture formats `GR_TEXFMT_RGB_332` and `GR_TEXFMT_ARGB_8332` use three bits each for *red* and *green* and two bits for *blue*. An 8-bit *alpha* is included in the latter.
- Texture formats `GR_TEXFMT_RGB_565`, `GR_TEXFMT_ARGB_1555`, and `GR_TEXFMT_ARGB_4444` provide three different ways to compress three or four 8-bit color component values into 16 bits. The first format discards *alpha* and uses five bits for *red* and *blue*, and six bits for *green*. The second one uses five bits each for *red*, *green*, and *blue*, and saves the extra bit for *alpha*. The third format treats all four components equally, using four bits for each.

- Texture formats `GR_TEXFMT_INTENSITY_8`, `GR_TEXFMT_ALPHA_INTENSITY_44`, and `GR_TEXFMT_ALPHA_INTENSITY_88` contain an intensity value rather than color components and can model monochrome lighting effects. Example 9.2 in the previous chapter uses an intensity texture in combination with another to produce a modulated texture.
- Texture format `GR_TEXFMT_ALPHA_8` contains only an 8-bit *alpha* value. When the texel is expanded to a 32-bit ARGB form, the *alpha* value is used for *red*, *green,* and *blue* as well.
- Texture formats `GR_TEXFMT_YIQ_422` and `GR_TEXFMT_AYIQ_8422` use a narrow channel compression technique to encode the color information. Each TMU has storage for two distinct decompression tables that translate the encoded information into 32-bit colors. Narrow channel compression is described in detail below.
- Texture formats `GR_TEXFMT_P_8` and `GR_TEXFMT_AP_88` implement a color palette, described below. Each TMU has room for one 256-entry color palette.

Table 10.1 shows all thirteen texture formats, detailing the format of a texel and the expansion to 32 bits for each texture format.

Narrow Channel Compression

The 3Dfx Interactive graphics accelerators provide a form of *narrow channel compression* that uses a **YAB** color space based on intensity/chrominance information. The compression is based on an algorithm that compresses a 24-bit RGB value to an 8-bit **YAB** format with little loss in precision. This **YAB** compression algorithm is especially suited to texture mapping, as textures typically contain very similar color components. The algorithm is performed by the host CPU, and **YAB** compressed textures are passed to SST-1. The advantages of using compressed textures are increased effective texture storage space and lower bandwidth requirements to perform texture filtering.

The **YAB** color space is represented with eight bits per pixel, and, like the `GR_TEXTFMT_RGB_332` representation (see Table 10.1), it allocates specific fields in those eight bits to specific components: four bits for **Y** and two bits each for **A** and **B**. For example, if the mapping from RGB to **YAB** is accomplished by the following linear matrix transformation,

$$\mathbf{Y} = 0.299*red + 0.587*green + 0.114*blue$$

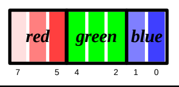
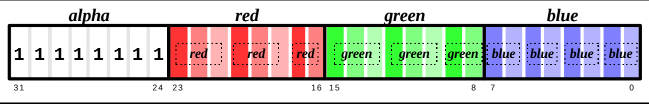
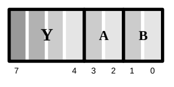
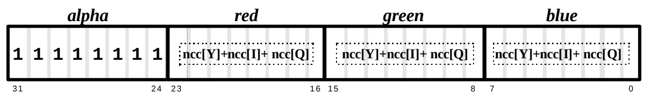
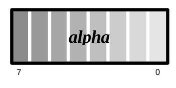
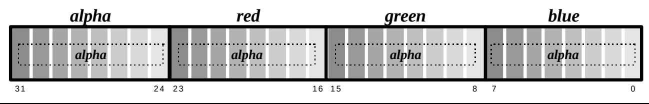
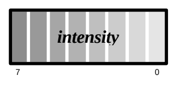
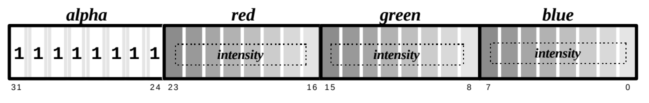
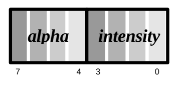
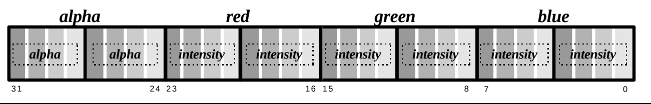
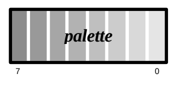
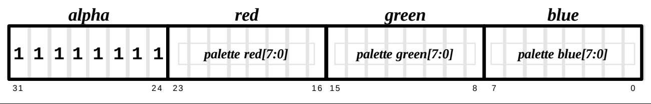
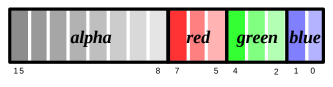
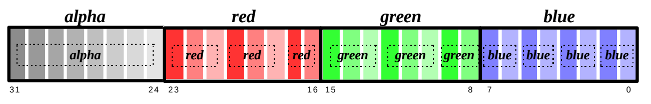
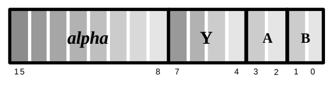
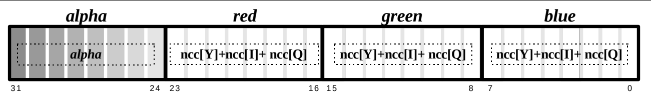
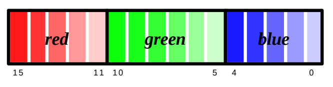
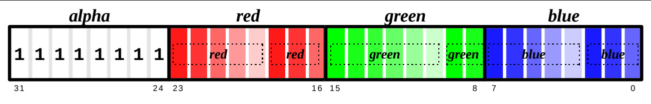
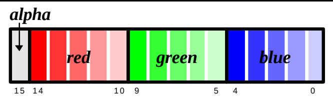
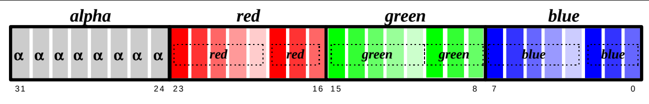
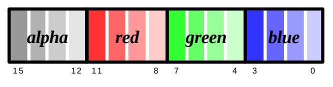
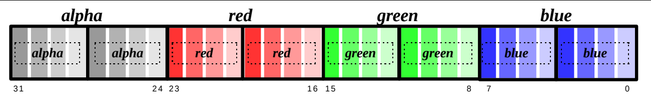
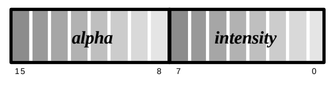
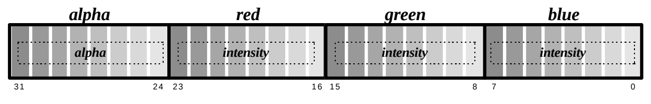
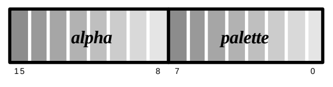
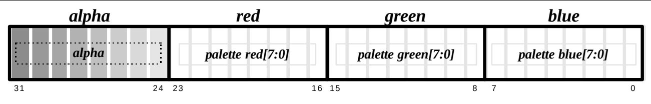
$$\mathbf{A} = 0.596*red + 0.275*green + 0.321*blue$$

$$\mathbf{B} = 0.212*red + 0.523*green + 0.311*blue \qquad \textit{Equation Set 1}$$

it is called ***YIQ** compression*. Two Glide texture formats utilize **YIQ** compression: `GR_TEXTFMT_YIQ_422` and `GR_TEXTFMT_AYIQ_8422`.

Compression is achieved by quantizing the **Y**, **A**, and **B** space more coarsely than the RGB space (by allocating fewer bits to each channel in **YAB** space) without degrading the quality of the image substantially. Also, instead of allocating the same number of bits to each channel (as is done when compressing RGB values directly), we can allocate more bits to channels carrying more information, and fewer bits otherwise. For example, when the image is represented in **YIQ** space with the equations above, it is possible to allocate only 16 distinct values to **Y**, which carries the intensity variations in the image, and only 4 distinct values for the **I** and **Q** channels, which carry the hue information. Hence, the original 24-bit RGB image can be represented in **YIQ** space with only eight bits of information, reducing the space requirements for the texture by a factor of three.

Table 10.1 Texture formats.
The table below shows the available texture formats and describes how texture data is expanded into 32-bit RGBA color. It also shows how 32-bit RGBA texture information is derived from the **YAB** *compression texture formats. This is detailed in the* **Narrow Channel Compression** *section in this chapter.*

symbolic name (prefixed with `GR_TEXFMT_`*)*	*compressed form in texture memory*	*expanded 32-bit ARGB form*
RGB_332 *8-bit RGB* *(3-3-2)*		
YIQ_422 *8-bit* ***YIQ*** *(4-2-2)*		
ALPHA_8 *8-bit Alpha*		
INTENSITY_8 *8-bit Intensity*		
ALPHA_INTENSITY_44 *8-bit Alpha and Intensity* *(4-4)*		
P_8 *8-bit Palette*		
ARGB_8332 *16-bit ARGB* *(8-3-3-2)*		
AYIQ_8422 *16-bit A****YIQ*** *(8-4-2-2)*		
RGB_565 *16-bit RGB* *(5-6-5)*		
ARGB_1555 *16-bit ARGB* *(1-5-5-5)*		
ARGB_4444 *16-bit ARGB* *(4-4-4-4)*		
ALPHA_INTENSITY_88 *16-bit Alpha and Intensity (8-8)*		
AP_88 *16-bit Alpha and Palette* *(8-8)*		

The decompression from **YIQ** to RGB is the inverse of the compression equations above. The RGB values can be recovered as follows:

$$\textit{red} = \mathbf{Y} + 0.95\bullet\mathbf{A} + 0.62\bullet\mathbf{B}$$

$$\textit{blue} = \mathbf{Y} - 0.28\bullet\mathbf{A} - 0.64\bullet\mathbf{B}$$

$$\textit{green} = \mathbf{Y} - 1.11\bullet\mathbf{A} + 1.73\bullet\mathbf{B}$$

Equation Set 2

Implementing these equations in hardware as formulated above is expensive: the **YAB** components must be scaled and two multipliers per component are needed. In addition, when compressed textures are used in conjunction with bilinear filtering, 24 multipliers are needed, since four texels must be made available simultaneously. But, by rewriting the equations as vectors (shown below) and building a small lookup table with pre-computed RGB values, the need for multipliers is eliminated, at least in the decompression circuitry.

$$(\textit{red}, \textit{green}, \textit{blue}) = (\mathbf{Y}, \mathbf{Y}, \mathbf{Y}) + (0.95\bullet\mathbf{A}, -0.28\bullet\mathbf{A}, -1.11\bullet\mathbf{A}) + (0.62\bullet\mathbf{B}, -0.64\bullet\mathbf{B}, -1.73\bullet\mathbf{B})$$

Equation 3

The four entries in the lookup table for **A**, then, represent the values of red, green, and blue calculated for four distinct values of **A**: –256, –85, 85, and 255. And the four entries in the lookup table for **B** represent the RGB values calculated for four distinct values of **B**. **Y** is implemented with a lookup table as well, but with sixteen distinct entries. Note that the quantized values of **Y**, **A**, and **B** can be any four values and don’t necessarily have to be evenly spaced or cover the full range of values.

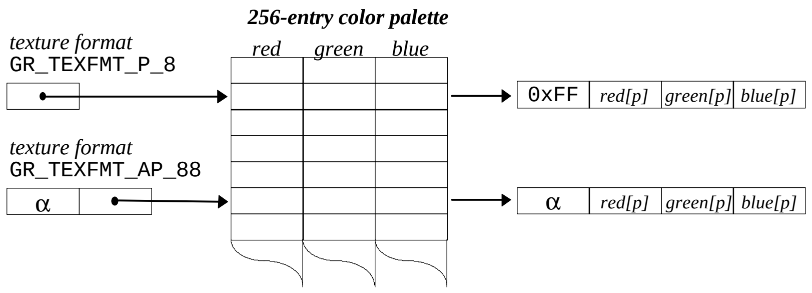
Note that the graphics hardware will work with any set of similar compression/decompression equations: the constants are contained in the table entries and the mechanics of the decompression are independent of them. The constants in the equations above are the ones used in **YIQ** space and were chosen to optimize the compression of flesh tones and backgrounds in photographs and videos. Most computer graphics textures, like terrain, sky, building facades, and so on, are not necessarily aligned along the orange-blue and purple-green axes of **YIQ** space and benefit from a different set of constants. The 3Dfx Interactive TexUS texture utility software provides routines for generating compressed textures using the **YIQ** equations shown above. It also provides a neural net program that can optimize the choice of factors in the equation for a given texture.

The Color Palette

An 8-bit color palette is implemented in all TMU chips after Revision 0. It is a 256-entry RGB table that is accessed during rendering by texture formats `GR_TEXFMT_P_8` and `GR_TEXFMT_AP_88` (see Table 10.1). These two texture formats store an 8-bit offset into the color palette for each texel in the texture map. During rendering, four texels are looked up simultaneously, each with an independent 8-bit address. The process of downloading NCC tables and color palettes is described later in this chapter.

Glide.30 introduces and color palette extension that provides an alternate palette format containing 6-bit ARGB entries instead of 8-bit RGB entries. It is described in Chapter 13.

Figure 10.1 The color palette.
TMU Revision 1 provides a color palette. The color palette holds 256 RGB colors that are retrieved during rendering, with a texture map utilizing one of the two palette texture formats: `GR_TEXFMT_P_8` or `GR_TEXFMT_AP_88`. The texel in these two formats is an offset into the color palette; `GR_TEXFMT_AP_88` appends an alpha value to the palette offset. In addition, see the discussion of the PALETTE6666 extension in Chapter 13.



Texture Memory

Each TMU has its own texture memory, which ranges in size from 2MB to 4MB depending on the system configuration. To download a texture into texture memory, one must complete the following steps:

STEP1: Determine how much memory is required for the texture.
STEP2: Determine the starting address and extent of free space. Is it adequate for the texture? Will a mipmap level straddle the 2Mbyte boundary in texture memory (thereby causing an error)?
STEP3: Download the texture.
STEP4: Identify the texture as the texel source for subsequent texture mapping operations.

Glide does no texture memory management; rather, it includes several functions that allow the application to manage it.

Computing the Size of a Mipmap

The Glide functions **grTexCalcMemRequired()** and **grTexTextureMemRequired()** determine the storage requirements of a mipmap. The size returned by these functions includes any bytes required to pad the texture to a hardware-specific alignment boundary, and may be added to the starting address of the texture to determine the next available location in texture memory.

Both routines use the texture format, aspect ratio, and range of LODs in the mipmap to compute the size. These values are arguments to **grTexCalcMemRequired()**; they are extracted from a *GrTexInfo* structure that is passed to **grTexTextureMemRequired()**. The other difference between the two routines is that **grTexTextureMemRequired()** has an *evenOdd* argument and can determine the memory requirements of a texture that will be split across two TMUs for trilinear filtering applications (see Example 9.3 in the previous chapter).

Table 10.2 Glide constants that specify arguments to grTex functions.
*The table below lists the constants used to name the values that can be specified as arguments to functions in the **grTex** family. The first column lists the argument names that are used in the function specifications. The second column gives the Glide type for the argument. The third column lists the constant name, and the fourth column gives a description.*

If the function argument is named	*and its type is*	*then these constants are valid values*	*and these are the consequences of choosing that value.*

tmu	*GrChipID_t*	`GR_TMU0`
		`GR_TMU1`
		`GR_TMU2`
			Selects the target TMU. The constant names it.

smallLOD
largeLOD
thisLOD
	GrLOD_t	`GR_LOD_LOG2_256`
		`GR_LOD_LOG2_128`
		`GR_LOD_LOG2_64`
		`GR_LOD_LOG2_32`
		`GR_LOD_LOG2_16`
		`GR_LOD_LOG2_8`
		`GR_LOD_LOG2_4`
		`GR_LOD_LOG2_2`
		`GR_LOD_LOG2_1`
			The number in the constant is the largest of the texture. The aspect ratio determines the smaller dimension.

aspectRatio	*GrAspectRatio_t*	`GR_ASPECT_LOG2_8x1`
		`GR_ASPECT_LOG2_4x1`
		`GR_ASPECT_LOG2_2x1`
		`GR_ASPECT_LOG2_1x1`
		`GR_ASPECT_LOG2_1x2`
		`GR_ASPECT_LOG2_1x4`
		`GR_ASPECT_LOG2_1x8`
			The constant sets the aspect ratio of the textures in a mipmap.

format	*GrTextureFormat_t*	`GR_TEXFMT_RGB_332`
		`GR_TEXFMT_YIQ_422`
		`GR_TEXFMT_ALPHA_8`
		`GR_TEXFMT_INTENSITY_8`
		`GR_TEXFMT_ALPHA_INTENSITY_44`
		`GR_TEXFMT_P_8`
		`GR_TEXFMT_ARGB_8332`
		`GR_TEXFMT_AYIQ_8422`
		`GR_TEXFMT_RGB_565`
		`GR_TEXFMT_ARGB_1555`
		`GR_TEXFMT_ARGB_4444`
		`GR_TEXFMT_ ALPHA_INTENSITY_88`
		`GR_TEXFMT_AP_88`
			See **Table** 10.1 for a description of the texture formats.

evenOdd	*FxU32*	`GR_MIPMAPLEVELMASK_EVEN`
		`GR_MIPMAPLEVELMASK_ODD`
		`GR_MIPMAPLEVELMASK_BOTH`
			Even LODs are `GR_LOD_LOG2_256`, `GR_LOD_LOG2_64`, `GR_LOD_LOG2_16`, `GR_LOD_LOG2_4`, and `GR_LOD_LOG2_1`.
			
			Odd LODs are `GR_LOD_LOG2_128`, `GR_LOD_LOG2_32`, `GR_LOD_LOG2_8`, and `GR_LOD_LOG2_2`.

range	*GrTexBaseRange_t*	`GR_TEXBASE_256`
		`GR_TEXBASE_128`
		`GR_TEXBASE_64`
		`GR_TEXBASE_32_TO_1`
			Specifies the base register when using more than one. A mipmap can be broken into four fragments. The number in the constant corresponds to the LOD number.

tableType
table
	GrTexTable_t	`GR_TEX_NCC0`
		`GR_TEX_NCC1`
		`GR_TEX_PALETTE`
			Each TMU can have two NCC tables and a palette. Load them one at a time with a general purpose routine.

mipmapMode
mode
	GrMipMapMode_t	`GR_MIPMAP_DISABLE`
		`GR_MIPMAP_NEAREST`
		`GR_MIPMAP_NEAREST_DITHER`
			Specifies the kind of mipmapping to perform.

```
FxU32 grTexCalcMemRequired(              GrLOD_t          smallLOD,
                           GrLOD_t                    largeLOD,
                           GrAspectRatio_t            aspectRatio,
                           GrTextureFormat_t          format
                          )
```

grTexCalcMemRequired() calculates and returns the amount of memory required by a mipmap of the specified LOD range, aspect ratio, and format. The first two arguments, *smallLOD* and *largeLOD*, define the range of LODs in the mipmap. The third argument, *aspectRatio*, specifies the aspect ratio of the mipmap and the fourth argument, *format*, gives the texture format. All four arguments are specified using Glide constants; the choices are listed in Table 10.2.

PORTING NOTE

Applications should make no assumptions about texture alignment. Specifically, applications should not assume that textures are aligned to 8-byte boundaries, as this could change in future 3Dfx chipsets. In Glide 3.0 the **grGet()** selector `GR_TEXTURE_ALIGN` has been added so that developers can write code that will automatically align textures correctly.

The memory requirements for the mipmap can be computed directly from these four parameters. The LOD range determines the length of the longest edge of each LOD. The aspect ratio provides a way to compute the length of the shorter edge of the LOD and, hence, the number of texels in the mipmap. The texture format determines the space requirements for one texel, which can be multiplied by the number of texels in order to compute the storage requirements for the mipmap. The two functions described here, **grTexCalcMemRequired()** and **grTexTextureMemRequired()**, will do the calculations.

Many of Glide's texture management routines make use of the *GrTexInfo* structure to collect the mipmap parameters together with the mipmap data.

PORTING NOTE

The *GrTexInfo* structure has been changed in Glide 3.0:

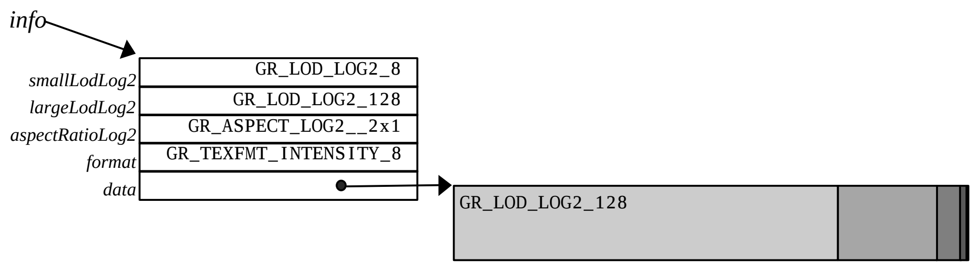
- *smallLodLog2* is the logarithm base 2 of the largest dimension of the lowest resolution mipmap. It replaces *smallLod*. The aspect ratio determines the smaller dimension.
- *largeLodLog2* is the logarithm base 2 of the largest dimension of the highest resolution mipmap. It replaces *largeLod*.
- *aspectRatioLog2* is the logarithm base 2 of the ratio of width to height. It replaces *aspectRatio*. If the aspect ratio is positive, then *s* will be the larger dimension of the mipmap; if it is negative, then *t* will be the larger dimension.

```
typedef struct {
    GrLOD_t               smallLodLog2;
    GrLOD_t               largeLodLog2;
    GrAspectRatio_t       aspectRatioLog2;
    GrTextureFormat_t     format;
    void                  *data;
} GrTexInfo;

FxU32 grTexTextureMemRequired( FxU32 evenOdd, GrTexInfo *info )
```

grTexTextureMemRequired() calculates and returns the number of bytes required to store the texture described in the structure pointed to by *info*. The number returned may be added to the starting address for a texture download to determine the next free location in texture memory.

The range of LODs in the mipmap is defined in the *info* structure. The other argument, *evenOdd*, indicates whether even, odd, or all LODs within the specified range should be used in computing the space requirements. For example, if the mipmap is used for trilinear filtering, the even LODs are downloaded and used on one TMU, and the odd LODs on another. *evenOdd* is specified symbolically: valid values are `GR_MIPMAPLEVELMASK_EVEN`, `GR_MIPMAPLEVELMASK_ODD`, and `GR_MIPMAPLEVELMASK_BOTH`. Figure 10.2 describes the *evenOdd* flag and even and odd LODs. In general, an LOD is even if its size is an even power of 2, and odd otherwise. Thus, the even LODs are `GR_LOD_LOG2_256`, `GR_LOD_LOG2_64`, `GR_LOD_LOG2_16`, `GR_LOD_LOG2_4`, and `GR_LOD_LOG2_1`. The other LODs are odd: `GR_LOD_LOG2_128`, `GR_LOD_LOG2_32`, `GR_LOD_LOG2_8`, and `GR_LOD_LOG2_2`.

Figure 10.2 The size of a mipmap depends on the setting of the evenOdd flag.
Suppose we have a GrTexInfo structure with data, as shown below.



*The size returned by **grTexTextureMemRequired()** depends on the value of the **evenOdd** flag, as shown below.*

LOD	*width*	*height*	*number of bytes*
`GR_LOD_LOG2_128`	128	64	2^{13} = 8192 bytes
`GR_LOD_LOG2_64`	64	32	2^{11} = 2048 bytes
`GR_LOD_LOG2_32`	32	16	2^{9} = 512 bytes
`GR_LOD_LOG2_16`	16	8	2^{7} = 128 bytes
`GR_LOD_LOG2_8`	8	4	2^{5} = 32 bytes

grTexTextureMemRequired(`GR_MIPMAPLEVELMASK_BOTH`, *info***)** returns the sum of the sizes of all 5 LODs.

8192 + 2048 + 512 + 128 + 32 = 10,912 bytes

grTexTextureMemRequired(`GR_MIPMAPLEVELMASK_ODD`, *info***)** returns the sum of the sizes of the odd LODs: `GR_LOD_LOG2_128`, `GR_LOD_LOG2_32`, and `GR_LOD_LOG2_8`.

8192 + 512 + 32 = 8,736 bytes

grTexTextureMemRequired(`GR_MIPMAPLEVELMASK_EVEN`, *info***)** returns the sum of the sizes of the even LODs: `GR_LOD_LOG2_64` and `GR_LOD_LOG2_16`.

2048 + 128 = 2,176 bytes

Querying for Available Memory

Two Glide functions, **grTexMinAddress()** and **grTexMaxAddress()** provide initial upper and lower bounds on texture memory for the specified TMU. They each have one argument, *tmu*, which selects the TMU on which to check the memory bounds.

FxU32 **grTexMinAddress(** *GrChipID_t tmu* **)**

FxU32 **grTexMaxAddress(** *GrChipID_t tmu* **)**

grTexMinAddress() and **grTexMaxAddress()** provide initial values for free space pointers in a Glide application. Be aware, however, that they always return the same values, regardless of whether any textures have been downloaded.

grTexMinAddress() returns the first location in texture memory into which a texture can be loaded.

grTexMaxAddress() returns the last possible appropriately aligned address that can be used as a starting address; only the smallest possible texture can be loaded there: the 1×1 texture `GR_LOD_LOG2_1`.

Texture memory management can be simple, sophisticated, or somewhere in between, depending on size and number of textures that will be loaded. The examples below show some straightforward techniques.

One important restriction must be mentioned: a mipmap level cannot straddle the 2Mbyte boundary in texture memory. That is, the addresses of the first and last words in the level must either both be greater or both be less than 2 Mbytes (2^{21}). One simple way to work around this limitation is to load complete mipmaps on one side or the other, depending on the fit, as shown in Example 10.2.

Example 10.1 Will the mipmap fit?
This code segment illustrates a simple scenario where a single mipmap is loaded into an empty texture memory on TMU0. Since this is the only texture that is loaded, there is no need to implement a free list.

```
FxU32 textureSize, startAddress;

textureSize = grTexCalcMemRequired(      GR_LOD_LOG2_1, GR_LOD_LOG2_256,
                                   GR_ASPECT_LOG2_1x1, GR_TEXFMT_ARGB_1555
);
startAddress = grTexMinAddress(GR_TMU0);

if (startAddress + textureSize <= grTexMaxAddress(GR_TMU0))
    download_the_texture;
```

Example 10.2 Setting up to load several mipmaps.
This code segment gets a little more real than the one above by keeping a pointer to the next available starting address for mipmaps. To get a starting address for a texture, call the subroutine.

```
#define TEXMEM_2MB_EDGE 2097152
FxU32 textureSize, nextTexture, lastTexture;
```

```
/* these two lines initialize the bounds and should be part        */
/* of the initialization code in the main program                  */
nextTexture = grTexMinAddress(GR_TMU0);
lastTexture = grTexMaxAddress(GR_TMU0)

long getStartAddress(FxU32 evenOdd, GrTexInfo *info)
{  long start;
   textureSize = grTexTextureMemRequired(evenOdd, info);
   start = nextTexture;

   /* check for 2MB edge and space past it if necessary */
   if ((start< TEXMEM_2MB_EDGE) && (start+textureSize> TEXMEM_2MB_EDGE))
      start = TEXMEM_2MB_EDGE

   nextTexture += textureSize;
   if (nextTexture <= lastTexture) return start;
   else {
      nextTexture = start;
      return -1;
   }
```

Downloading Mipmaps

Download a mipmap into texture memory with the function **grTexDownloadMipMap()**. Replace an individual mipmap level with **grTexDownloadMipMapLevel()**. Replace part of an LOD with **grTexDownloadMipMapLevelPartial()**.

The first argument to all three routines is *tmu*, which designates the target TMU for the load. Each of the three routines also provides a *startAddress* argument that specifies an offset into texture memory where the texture will be loaded, and an *evenOdd* argument that indicates which levels to load (specified as one of `GR_MIPMAPLEVELMASK_EVEN`, `GR_MIPMAPLEVELMASK_ODD`, or `GR_MIPMAPLEVELMASK_BOTH`). *startAddress* must lie between the values returned by **grTexMinAddress()** and **grTexMaxAddress()** and must be appropriately aligned.

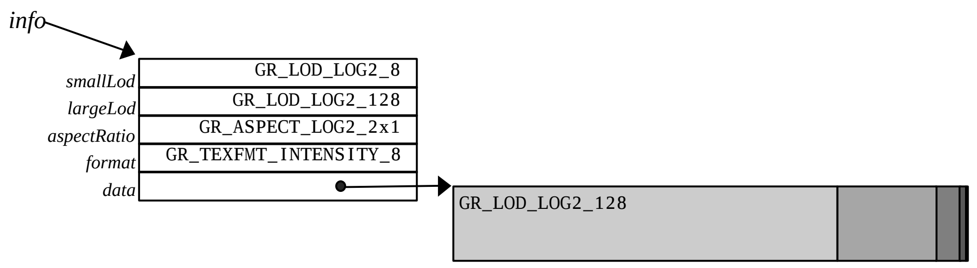
grTexDownloadMipMap() expects the mipmap parameters (aspect ratio, texture format, LOD range, and the texture data) in a *GrTexInfo* structure; the other two routines have arguments for each parameter.

Downloading All or Part of a Mipmap

Use **grTexDownloadMipMap()** to load a mipmap.

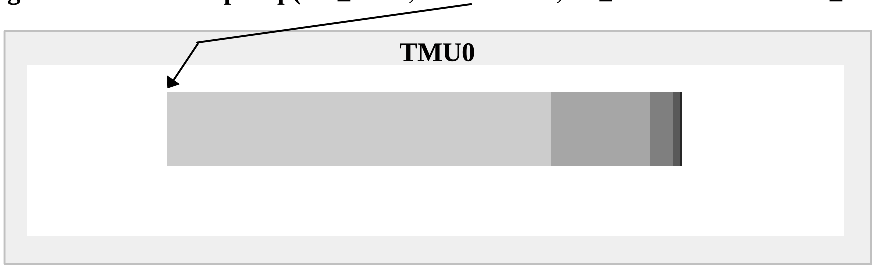
```
typedef struct {
    GrLOD_t              smallLodLog2;
    GrLOD_t              largeLodLog2;
    GrAspectRatio_t      aspectRatioLog2;
    GrTextureFormat_t    format;
    void                 *data;
}   GrTexInfo;

void grTexDownloadMipMap( GrChipID_t tmu, FxU32 startAddress, FxU32 evenOdd, GrTexInfo *info )
```

Figure 10.3 Downloading a mipmap.
Suppose we have a GrTexInfo structure with data, as shown below.



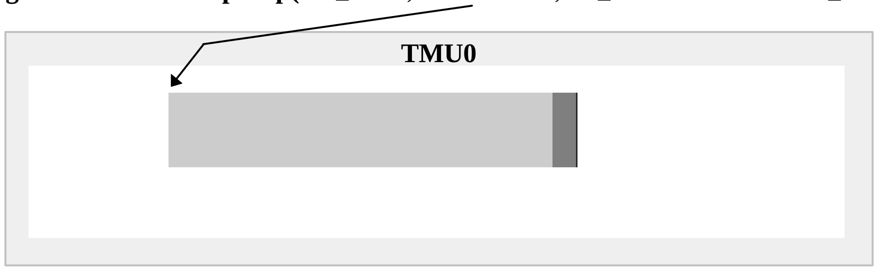
*The three drawings below show **startAddress** and its relationship to where and what textures are loaded, based on the **evenOdd** value. The first **grTexDownloadMipMap()** call loads all LODs between `GR_LOD_LOG2_128` and `GR_LOD_LOG2_8`.*

grTexDownloadMipMap(GR_TMU0, *startAddress*, GR_MIPMAPLEVELMASK_BOTH, *info* **)**



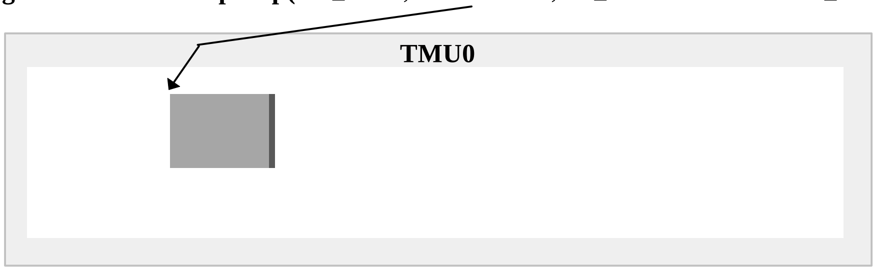
The second scenario loads only the odd LODs. Recall that the largest dimension of odd LODs is an odd power of two. In this case, `GR_LOD_LOG2_128`, `GR_LOD_LOG2_32`, and `GR_LOD_LOG2_8` are odd LODs.

grTexDownloadMipMap(GR_TMU0, *startAddress*, GR_MIPMAPLEVELMASK_ODD, *info* **)**



*The final scenario loads only the even LODs. Note that no modification is necessary to the values in the GrTexInfo structure pointed to by **info**. Glide will skip over the texture data for the odd LODs, only loading the even ones.*

grTexDownloadMipMap(GR_TMU0, *startAddress*, GR_MIPMAPLEVELMASK_EVEN, *info* **)**



Replacing a Single LOD

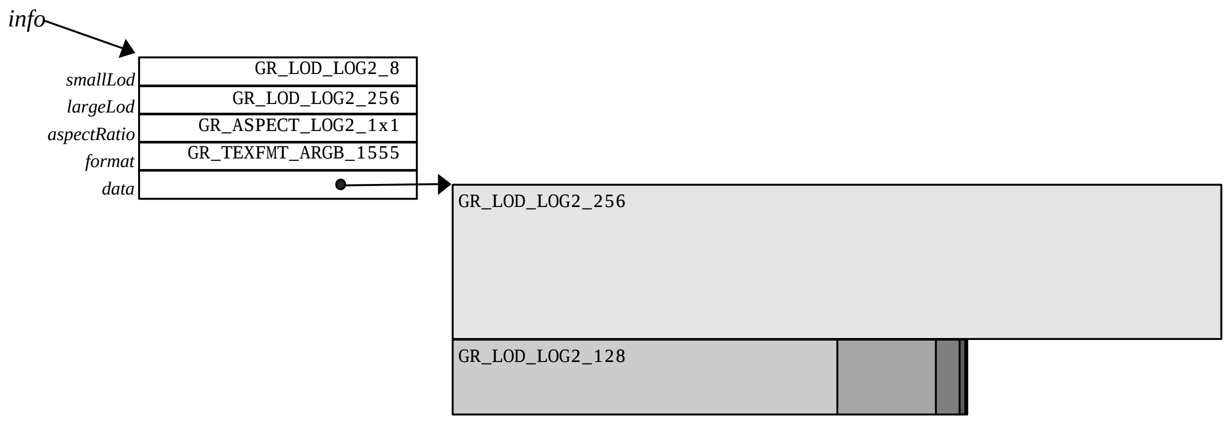
One form of simple texture memory management requires only that the application swap mipmaps with identical memory footprints (i.e., same format, dimensions, and mipmap levels) in and out of the same texture memory area. Texture replacement is a simple facility for doing texture map animation, and it is also a method of doing dynamic texture management: the local texture buffer is split into discrete texture regions that are updated as needed. To replace a mipmap, use the Glide function **grTexDownloadMipMap()** with new data. Alternatively, an application can swap out individual mipmap levels within a mipmap using **grTexDownloadMipMapLevel()**.

```
void grTexDownloadMipMapLevel( GrChipID_t          tmu,
                               FxU32               startAddress,
                               GrLOD_t             thisLOD,
                               GrLOD_t             largeLOD,
                               GrAspectRatio_t     aspectRatio,
                               GrTextureFormat_t   format,
                               FxU32               evenOdd,
                               void                *data
                             )
```

grTexDownloadMipMapLevel() replaces a single mipmap level in a previously downloaded mipmap that begins at *startAddress*. Argument *largeLOD* specifies the largest (and first) LOD in the downloaded mipmap; the *aspectRatio* and *format* locate the first texel of *thisLOD*. The *data* argument points to the first texel of the new LOD, as shown in Figure 10.4.

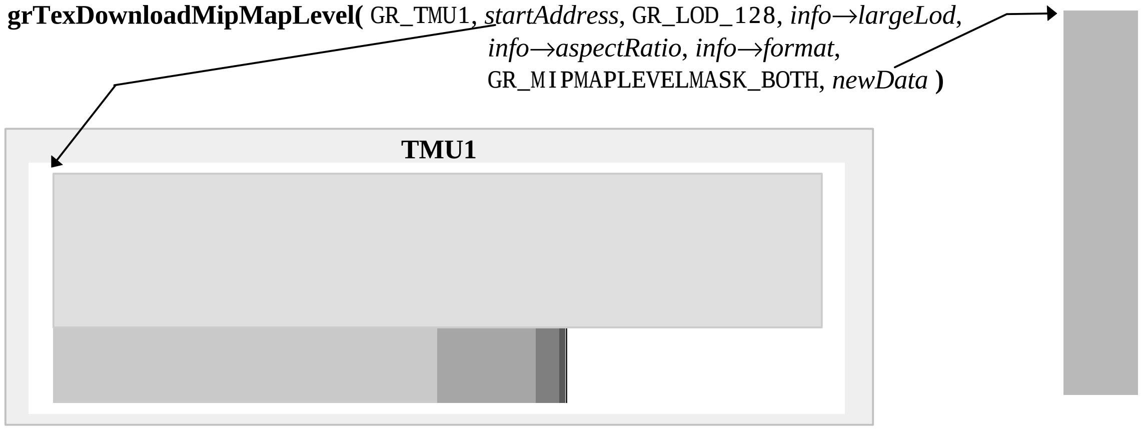
Figure 10.4 Replacing a single LOD.
Suppose a mipmap has been loaded into TMU1 with the following command and data.

grTexDownloadMipMap(`GR_TMU1`,*startAddress*,`GR_MIPMAPLEVELMASK_BOTH`, *info***)**



To replace `GR_LOD_LOG2_128`, use the following call to ***grTexDownloadMipMapLevel()****.*

grTexDownloadMipMapLevel(GR_TMU1, *startAddress*, GR_LOD_128, *info*→*largeLod*,
info→*aspectRatio*, *info*→*format*,
GR_MIPMAPLEVELMASK_BOTH, *newData* **)**



Replacing Part of an LOD

Applications that want to replace one of the large LODs in a mipmap, but also want to maintain a snappy frame rate, may opt to replace the LOD a few rows at a time with **grTexDownloadMipMapLevelPartial()**.

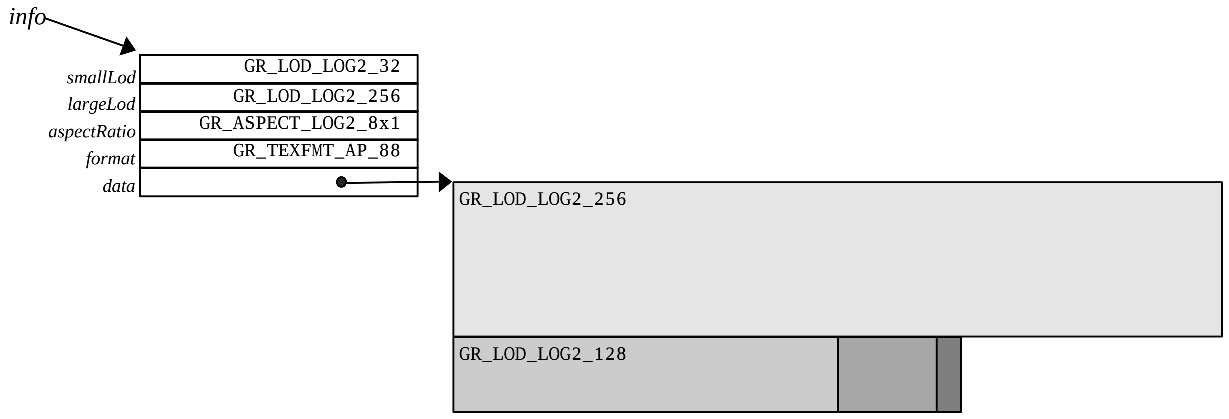
```
void grTexDownloadMipMapLevelPartial( GrChipID_t          tmu,
                                      FxU32               startAddress,
                                      GrLOD_t             thisLOD,
                                      GrLOD_t             largeLOD,
                                      GrAspectRatio_t     aspectRatio,
                                      GrTextureFormat_t   format,
                                      FxU32               evenOdd,
                                      void                *data,
                                      int                 firstRow,
                                      int                 lastRow
```

)

The first seven arguments to **grTexDownloadMipMapLevelPartial**() are the same as those to **grTexDownloadMipMapLevel**(): the *tmu* that the texture is loaded on, the starting address, the LOD that will be partially replaced, the largest LOD in the mipmap, the aspect ratio and texture format of the downloaded texture, and the *evenOdd* flag. The *data* argument points to a stream of texels that will overwrite those in texture memory, starting at the row *firstRow* in *thisLOD* and continuing through *lastRow*. To download one row of the texture, use the same value for *firstRow* and *lastRow*.

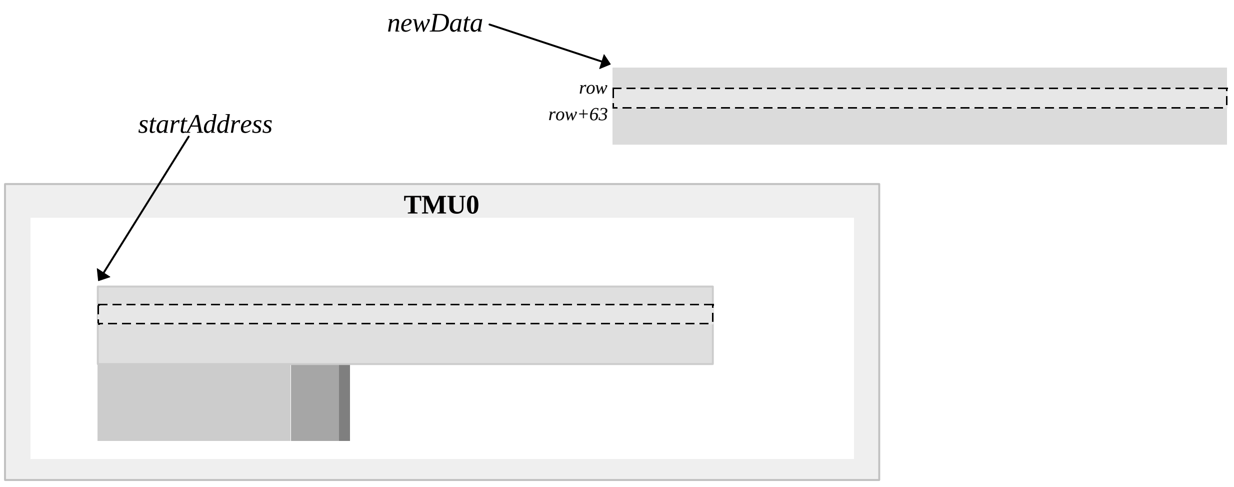
Figure 10.5 Replacing a few rows of an LOD.
Suppose a mipmap has been loaded into TMU0 with the following command and data.

grTexDownloadMipMap(`GR_TMU0`, *startAddress*, `GR_MIPMAPLEVELMASK_BOTH`, *info***)**



*To replace `GR_LOD_LOG2_256` in chunks, use a series of calls to **grTexDownloadMipMapLevelPartial()**:*

`for (row=0; row<256; row+=64)`
grTexDownloadMipMapLevel(`GR_TMU0`, *startAddress*, `GR_LOD_LOG2_256`, *info→largeLodLog2*, *info→aspectRatioLog2*, *info→format*, `GR_MIPMAPLEVELMASK_BOTH`, *newData*, *row*, *row + 63* **);**



Identifying a Mipmap as the Texel Source

The final step is to register the newly loaded mipmap with the TMU as the source for texels. The Glide function **grTexSource()** provides this service.

void **grTexSource(** *GrChipID_t tmu*, *FxU32 startAddress*, *FxU32 evenOdd*, *GrTexInfo *info* **)**

grTexSource() sets up the area of texture memory that is to be used as a source for subsequent texture mapping operations. The starting address, specified as argument *startAddress*, should be the same one that was used as an argument to **grTexDownloadMipMap()**, or the starting address used for the largest mipmap level when using **grTexDownloadMipMapLevel()**.

Here are the three examples from Chapter 9, with additional lines of code to download the appropriate textures.

Example 10.3 Downloading a texture for decal texture mapping.
The following code sets up the texture pipeline so that a texel is placed into the pixel pipeline without modification. The code assumes that the color combine unit is configured to use the texture color and/or alpha value.

```
FxU32 textureSize, startAddress;
GrTexInfo info;
FxU16 mipmap[256*256 + 128*128 + 64*64 + 32*32 + 256 + 64 + 16 + 4 +
1];

info.smallLodLog2 = GR_LOD_LOG2_1;
info.largeLodLog2 = GR_LOD_LOG2_256;
info.aspectRatioLog2 = GR_ASPECT_LOG2_1x1;
info.format = GR_TEXFMT_1555;
info.data = mipmap;

textureSize = grTexTextureMemRequired(GR_MIPMAPLEVELMASK_BOTH, &info);
startAddress = grTexMinAddress(GR_TMU0);
if ((startAddress + textureSize)> grTexMaxAddress(GR_TMU0)) {
              printf(“error: texture too big for TMU0\n”);
              exit();
}

grTexDownloadMipMap(GR_TMU0, startAddress, GR_MIPMAPLEVELMASK_BOTH,
&info);
grTexSource(GR_TMU0, startAddress, GR_MIPMAPLEVELMASK_BOTH, &info);

grTexCombine( GR_TMU0, GR_COMBINE_FUNCTION_LOCAL,
GR_COMBINE_FACTOR_NONE,
              GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
              FXFALSE, FXFALSE );
```

Example 10.4 Downloading two textures for modulated or composite texture mapping.
The code segment below loads an intensity map for a spotlight in TMU0 and a source texture in TMU1. The resulting texture RGBA is a product of the texels chosen from the two textures. The color combine unit must be configured to use the output from the texture pipeline.

```
FxU32 textureSize[2], startAddress[2];
GrTexInfo src, spot;
FxU16 srcdata[256*256 + 128*128 + 64*64 + 32*32 + 256 + 64 + 16 + 4 +
1];
FxU8 spotdata[256*256 + 128*128 + 64*64 + 32*32 + 256 + 64 + 16 + 4 +
1];

src.smallLodLog2 = spot.smallLodLog2 = GR_LOD_LOG2_1;
src.largeLodLog2 = spot.largeLodLog2 = GR_LOD_LOG2_256;
src.aspectRatioLog2 = spot.aspectRatioLog2 = GR_ASPECT_LOG2_1x1;
src.format = GR_TEXFMT_1555;
src.data = srcdata;
spot.format = GR_TEXFMT_INTENSITY_8;
spot.data = spotdata;

textureSize[0] = grTexTextureMemRequired(GR_MIPMAPLEVELMASK_BOTH,
&spot);
startAddress[0] = grTexMinAddress(GR_TMU0);
if ((startAddress[0] + textureSize[0])> grTexMaxAddress(GR_TMU0)) {
              printf(“error: spotlight texture too big for TMU0\n”);
              exit();
}

textureSize[1] = grTexTextureMemRequired(GR_MIPMAPLEVELMASK_BOTH,
&src);
startAddress[1] = grTexMinAddress(GR_TMU1);
if ((startAddress[1] + textureSize[1])> grTexMaxAddress(GR_TMU1)) {
              printf(“error: source texture too big for TMU1\n”);
              exit();
}


grTexDownloadMipMap(GR_TMU0,startAddress[0],GR_MIPMAPLEVELMASK_BOTH,
&spot);
grTexSource(GR_TMU0,startAddress[0],GR_MIPMAPLEVELMASK_BOTH, &spot);
grTexCombine(         GR_TMU0, GR_COMBINE_FUNCTION_SCALE_OTHER,
GR_COMBINE_FACTOR_LOCAL,
             GR_COMBINE_FUNCTION_SCALE_OTHER, GR_COMBINE_FACTOR_LOCAL,
             FXFALSE, FXFALSE );

grTexDownloadMipMap(GR_TMU1,startAddress[1],GR_MIPMAPLEVELMASK_BOTH,
&src);
grTexSource(GR_TMU1,startAddress[1],GR_MIPMAPLEVELMASK_BOTH, &src);
grTexCombine(                             GR_TMU1,
GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
             GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
             FXFALSE, FXFALSE );
```

Example 10.5 Splitting a texture across two TMUs for trilinear mipmapping.
The first code segment shows the texture combine unit configuration for trilinear mipmapping when the even LODs are stored in TMU0 and the odd ones are in TMU1. The code assumes that the color combine unit is configured to make use of the resulting RGBA value.

```
FxU32 textureSize[2], startAddress[2];
GrTexInfo tri;
FxU16 mipmap[256*256 + 128*128 + 64*64 + 32*32 + 256 + 64 + 16 + 4 +
1];

tri.smallLodLog2 = GR_LOD_LOG2_1;
tri.largeLodLog2 = GR_LOD_LOG2_256;
tri.aspectRatioLog2 = GR_ASPECT_LOG2_1x1;
tri.format = GR_TEXFMT_1555;
tri.data = mipmap;

textureSize[0] = grTexTextureMemRequired(GR_MIPMAPLEVELMASK_EVEN,
&tri);
startAddress[0] = grTexMinAddress(GR_TMU0);
if ((startAddress[0] + textureSize[0])> grTexMaxAddress(GR_TMU0)) {
              printf(“error: even LODs of texture too big for TMU0\n”);
              exit();
}

textureSize[1] = grTexTextureMemRequired(GR_MIPMAPLEVELMASK_ODD, &tri)
;
startAddress[1] = grTexMinAddress(GR_TMU1);
if ((startAddress[1] + textureSize[1])> grTexMaxAddress(GR_TMU1)) {
              printf(“error: odd LODs of texture too big for TMU1\n”);
              exit();
}


grTexDownloadMipMap(GR_TMU0,startAddress[0],GR_MIPMAPLEVELMASK_EVEN,
&tri) ;
grTexSource(GR_TMU0,startAddress[0],GR_MIPMAPLEVELMASK_EVEN, &tri);
grTexCombine(                             GR_TMU0,
             GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
             GR_COMBINE_FACTOR_LOD_FRACTION,
             GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
             GR_COMBINE_FACTOR_LOD_FRACTION,
             FXFALSE, FXFALSE);

grTexDownloadMipMap(GR_TMU1,startAddress[1],GR_MIPMAPLEVELMASK_ODD,
&tri);
grTexSource(GR_TMU1,startAddress[1],GR_MIPMAPLEVELMASK_ODD, &tri);
grTexCombine(                             GR_TMU1,
GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
             GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
             FXFALSE, FXFALSE );
```

This second code segment gives the proper ***grTexCombine()*** *configuration when the situation is reversed: the odd LODs in the mipmap are on TMU0 while the even ones are upstream on TMU1. Note the difference in the texture combine unit configuration: the setting of the* ***rgbInvert*** *and* ***alphaInvert*** *parameters.*

```
FxU32 textureSize[2], startAddress[2];
GrTexInfo tri;
FxU16 mipmap[256*256 + 128*128 + 64*64 + 32*32 + 256 + 64 + 16 + 4 +
1];

tri.smallLodLog2 = GR_LOD_LOG2_1;
tri.largeLodLog2 = GR_LOD_LOG2_256;
tri.aspectRatioLog2 = GR_ASPECT_LOG2_1x1;
tri.format = GR_TEXFMT_1555;
tri.data = mipmap;
```

```
textureSize[0] = grTexTextureMemRequired(GR_MIPMAPLEVELMASK_ODD, &tri);
startAddress[0] = grTexMinAddress(GR_TMU0);
if ((startAddress[0] + textureSize[0])> grTexMaxAddress(GR_TMU0)) {
              printf("error: even LODs of texture too big for TMU0\n");
              exit();
}

textureSize[1] = grTexTextureMemRequired(GR_MIPMAPLEVELMASK_EVEN,
&tri);
startAddress[1] = grTexMinAddress(GR_TMU1);
if ((startAddress[1] + textureSize[1])> grTexMaxAddress(GR_TMU1)) {
              printf("error: odd LODs of texture too big for TMU1\n");
              exit();
}


grTexDownloadMipMap(GR_TMU0,startAddress[0],GR_MIPMAPLEVELMASK_ODD,
&tri);
grTexSource(GR_TMU0,startAddress[0],GR_MIPMAPLEVELMASK_ODD, &tri);
grTexCombine(                             GR_TMU0,
             GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
             GR_COMBINE_FACTOR_ONE_MINUS_LOD_FRACTION,
             GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL,
             GR_COMBINE_FACTOR_ONE_MINUS_LOD_FRACTION,
             FXFALSE, FXFALSE);

grTexDownloadMipMap(GR_TMU1,startAddress[1],GR_MIPMAPLEVELMASK_EVEN,
&tri);
grTexSource(GR_TMU1,startAddress[1],GR_MIPMAPLEVELMASK_EVEN, &tri);
grTexCombine(                             GR_TMU1,
GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
             GR_COMBINE_FUNCTION_LOCAL, GR_COMBINE_FACTOR_NONE,
             FXFALSE, FXFALSE );
```

Loading a Mipmap into Fragmented Memory

Normally, mipmap levels are stored sequentially in texture memory. Multi-base addressing allows mipmap levels to be loaded into different texture memory locations. A mipmap can be split into four chunks (along pre-defined boundaries), each of which can be loaded in a different location in texture memory. Four different base addresses are specified for a multi-based texture, one each for `GR_LOD_LOG2_256`, `GR_LOD_LOG2_128`, and `GR_LOD_LOG2_64`, and one for textures `GR_LOD_LOG2_32` through `GR_LOD_LOG2_1`.

To use multi-base addressing, you must enable it with a call to **grTexMultibase()**, download the mipmap as four smaller mipmaps, and then set up the multi-base addressing by calling **grTexMultibaseAddress()** four times with the four starting addresses. See Example 10.6.

void **grTexMultibase(** *GrChipID_t tmu*, *FxBool enable* **)**

grTexMultibase() enables or disables multi-base addressing. Multi-base addressing must be enabled before downloading a multi-based texture, and before rendering using a multi-based texture. Multi-base addressing must be disabled before downloading or rendering from a texture with a single base address.

You must call **grTexMultibaseAddress()** once for each part of a fragmented texture with multiple base addresses. In each case, *startAddress* should point to the texture memory location for the corresponding mipmap level. All of the base addresses for a multi-based texture should be specified before downloading the texture or rendering from the texture.

void **grTexMultibaseAddress(** *GrChipID_t tmu*, *GrTexBaseRange_t range*, *FxU32 startAddress*, *FxU32 evenOdd*, *GrTexInfo *info* **)**

The first argument names the TMU on which the fragmented texture will be loaded. The second argument, *range*, tells which fragment this call is about, and is one of four Glide constants: `GR_TEXBASE_256`, `GR_TEXBASE_128`, `GR_TEXBASE_64`, or `GR_TEXBASE_32_TO_1`. The third argument, *startAddress*, is the starting address for this fragment. Note that **grTexMultibaseAddress()** should be called with a valid starting address before the fragment is downloaded.

The fourth argument, *evenOdd*, specifies whether the even, the odd, or all textures in the mipmap will be downloaded on this *tmu*. If a fragment is missing from the mipmap, or if a fragment will not be downloaded on this *tmu*, then **grTexMultibaseAddress()** need not be called for that fragment.

Calls to **grTexSource()** are equivalent to calls to **grTexMultibaseAddress()** with the *range* argument set to `GR_LOD_LOG2_256`.

Example 10.6 Using multiple texture base registers.
Suppose that `start` is an array of starting addresses that have been obtained from a memory management routine. (The memory management details are left as an exercise for the reader.) Further suppose that the block of texture memory pointed to by `start[0]` is large enough for `GR_LOD_LOG2_256`, that the block pointed to by `start[1]` is large enough for `GR_LOD_LOG2_128`, and so on. The array `mipmap` points to the four fragments. The `lod` array stores the four constants that identify the fragments for convenience in the `for` loop that sets up the multiple base registers and downloads the fragments.

```
int i;
GrTexInfo info;
FxU32 start[4];
FxU16 mipmap[4][];
GrTexBaseRange_t lod[4]=(                        GR_TEXBASE_256,
                         GR_TEXBASE_128, GR_TEXBASE_64, GR_TEXBASE_32_TO_1);


grTexMultibase(GR_TMU0, FX_TRUE);

for (i=0; i,4; i++) {
   info.smallLodLog2 = info.largeLodLog2 = lod[i];
   info.data = mipmap[i];
   grTexMultibaseAddress(GR_TMU0, lod[i], start[i], GR_MIPMAPLEVEL_BOTH,
&info);
   grTexDownloadMipMap(GR_TMU0, start[i], GR_MIPMAPLEVEL_BOTH, &info);
}
```

Downloading a Decompression Table or Color Palette

The texels in mipmaps that use texture formats `GR_TEXFMT_YIQ_422` and `GR_TEXFMT_AYIQ_8422` must be "decompressed" to 32-bit values before being filtered and combined in the TMU. Texels that are stored in texture formats `GR_TEXFMT_P_8` and `GR_TEXFMT_AP_88` must be looked up in a color palette to translate them to 32-bit color components. The translation tables must be downloaded to the same TMU as the textures that use them before texel selection can occur.

Glide maintains two NCC decompression tables and one 256-entry color palette. The NCC table or color palette must be downloaded before a texture that uses it can be used as the source for texels. Glide provides a routine that can download either a color palette or one of the two decompression tables.

void **grTexDownloadTable(** *GrTexTable_t tableType*, void **data* **)**

grTexDownloadTable() downloads either an NCC table or a 256-entry color palette. The first argument, *tableType*, describes the kind of table to be downloaded and is specified with one of these Glide constants: `GR_TEX_NCC0`, `GR_TEX_NCC1`, `GR_TEX_PALETTE`, or, if supported, `GR_TEX_PALETTE_6666_EXT` (see Chapter 13). The second argument points to the data for the table.

Part of a 256-entry color palette can be downloaded or replaced with the Glide function **grTexDownloadTablePartial()**.

void **grTexDownloadTablePartial(** *GrTexTable_t tableType*, void **data*, int *start*, int *end* **)**

Entries from *start* up to and including *end* are downloaded. To download one entry, use the same value for *start* and *end*. Partial downloads of NCC tables is not supported at this time.

The two table types are discussed separately in the paragraphs that follow. A downloading example is included for each kind.

PORTING NOTE

In Glide 2.x, **grTexDownloadTable()**, **grTexDownloadTablePartial()**, and **grTexNCCTable()** had an argument, *tmu*, that has been removed in Glide 3.0. All TMUs share the same NCC table or color palette. If one TMU is using a color palette, then none of the others can use an NCC table. Similarly, if TMU is using a compressed texture (and hence an NCC table for decompression), none of the other TMUs can use a texture that requires a color palette.

Decompression Tables

A texture can be compressed into a **YAB** texture with an appropriate decompression table with the help of the 3Dfx Interactive Texture Utility Software (TexUS). The compressed texture is stored as a 3Dfx texture map file (`.3DF`) that can then be loaded using the Glide Utility routine **gu3dfLoad()**, which is described later in this chapter. Space for two NCC tables is provided so that they can be swapped on a per-triangle basis when performing multi-pass rendering without interrupting the rendering process with table downloading.

Before a compressed texture can be used as the texel source, one of the two NCC tables must be designated as the source for decompression operations. The Glide function **grTexNCCTable()** should be called before any rendering operations using the compressed table are initiated.

void **grTexNCCTable(** *GrNCCTable_t table* **)**

grTexNCCTable() selects one of the two NCC tables as the current source for decompression operations. Valid values are `GR_TEXTABLE_NCC0` and `GR_TEXTABLE_NCC1`.

Example 10.7 Loading an NCC table.
*NCC tables are created by programs in the TexUS library and written to a `.3DF` file. This code segment uses **gu3dfLoad()**, described in the next section, to read the file into memory. Once in memory, the table is downloaded to NCC1 in TMU0. Once the table is loaded, a texture in one of the compressed formats can be downloaded and used as the texel source.*

```
Gu3dfInfo info;

gu3dfLoad("ncctable.3df", &info);
grTexDownloadTable( GU_TEX_NCC1, &info.table.nccTable);
grTexNCCTable( GR_TEXTABLE_NCC1);
```

Color Palettes

A color palette is an array of 256 ARGB colors, 8 bits for each component, and 32 bits per entry (refer back to Figure 10.1). The alpha component, in the high order 8 bits, is ignored. A second palette format is introduced in Glide 3.0 and may be used if the PALETTE6666 extension is supported. See Chapter 13 for more details.

Example 10.8 Loading a color palette.
The following code segment will create a random color palette and download it into TMU0. To use the palette, download a palletized texture (texture formats `GR_TEXFMT_P_8` or `GR_TEXFMT_AP_88`) and configure the texture and color combine units appropriately.

```
extern unsigned long lrand( void);
GuTexPalette palette;
int i, j;

// create a random 256-entry color palette
for (i=0; i<256; i++)
   palette.data[i] = 0x00FFFFFF & lrand();

grTexDownloadTable( GU_TEX_PALETTE, &palette);
```

Loading Mipmaps From Disk

TexUS (3Dfx Interactive's Texture Utility Software) programs create files in a 3DF file format. These files may contain mipmaps, decompression tables, or both. A pair of data types and a pair of functions provide access to `.3DF` files from Glide.

The data structures are shown below. *Gu3dfInfo* is the top level structure. It has a pointer to the mipmap data, and it stores the decompression table or palette if there is one. There is also a *Gu3dfHeader* structure that contains all the mipmap characteristics (LOD range, aspect ratio, format, dimensions) and the amount of memory the mipmap will require.

```
typedef struct {
      FxU32                   width, height;
      int                     small_lod, large_lod;
      GrAspectRatio_t         aspect_ratio;
      GrTextureFormat_t       format;
}     Gu3dfHeader;

typedef union {
      GuNccTable              nccTable;
      GuTexPalette            palette;
} GuTexTable;

typedef struct {
      Gu3dfHeader             header;
      GuTexTable              table;
      void                    *data;
      FxU32                   mem_required;
}     Gu3dfInfo;
```

The procedure for reading a `.3DF` file from Glide is shown in Example 10.9. The application first calls **gu3dfGetInfo()** to fill in the *Gu3dfInfo* structure pointed to by *info*.

FxBool **gu3dfGetInfo(** const char **filename*, *Gu3dfInfo* **info* **)**

After an application has determined the characteristics of a `.3DF` mipmap, memory must be allocated for the mipmap and the address stored in the *info®data* pointer. Then **gu3dfLoad()** is invoked to load the mipmap from the file into memory. Note that the mipmap must be downloaded into a TMU before it can be used as a texel source.

FxBool **gu3dfLoad(** const char **filename*, *Gu3dfInfo* **info* **)**

Both **gu3dfGetInfo()** and **gu3dfLoad()** return `FXTRUE` if the file specified by *filename* exists and can be read; otherwise they return `FXFALSE`.

Example 10.9 Reading a `.3DF` file.

*The following code segment assumes that "`mipmap.3df`" contains a properly formatted 3DF file. The code calls **gu3dfGetInfo()** to determine memory requirements, allocates storage for the mipmap using the system subroutine malloc(), then reads the mipmap into the newly allocated memory by calling **gu3dfLoad()**.*

```
Gu3dfInfo fileInfo;

gu3dfGetInfo("mipmap.3df", &fileInfo);
fileInfo.data = malloc(fileInfo.mem_required);
gu3dfLoad("mipmap.3df", &fileInfo);
```

11. Accessing the Linear Frame Buffer

In This Chapter

The frame buffer on a graphics subsystem is directly accessible by software as a single linear address space. This address space is segmented into separate readable and writable areas, and each of these areas in turn can address any of the three hardware buffers: the front buffer, the back buffer, or the auxiliary buffer.

You will learn how to:

- ▼ calculate a pixel address.
- ▼ acquire an LFB (linear frame buffer) read or write pointer.
- ▼ read pixel data from the color, alpha, or depth buffer.
- ▼ write pixel data in a user-selectable format to the color alpha, or depth buffer.
- ▼ set constant values for direct writes to the depth and alpha buffers.
- ▼ enable and disable the pixel pipeline during direct LFB writes.

Acquiring an LFB Read or Write Pointer

When a Glide application desires direct access to a color or auxiliary buffer, it must lock that buffer in order to gain access to a pointer in the frame buffer data. This lock may assert a critical code section which affects process scheduling and precludes the use of GUI debuggers; therefore, time spent doing direct accesses should be minimized and the lock should be released as soon as possible.

```
FxBool grLfbLock( GrLock_t            type,
                  GrBuffer_t          buffer,
                  GrLfbWriteMode_t    writeMode,
                  GrOriginLocation_t  origin,
                  FxBool              pixelPipeline,
                  GrLfbInfo_t         *info
                )
```

An application may hold multiple simultaneous locks to various buffers, if the underlying hardware allows it. Application software should *always* check the return value of **grLfbLock()**: a lock may fail. A buffer is locked for reads or for writes, as specified in the *type* parameter.

type is a bit field created by ORing a read/write flag and an idle flag. The read/write flag can be either `GR_LFB_READ_ONLY` or `GR_LFB_WRITE_ONLY`. The idle flag can be either `GR_LFB_IDLE` or `GR_LFB_NOIDLE`. The default is `GR_LFB_IDLE`: the graphics subsystem is idle until the buffer is unlocked. `GR_LFB_NOIDLE` allows the pixel pipeline to continue operating during the lock: triangle rendering and buffer clearing operations may be interspersed with frame buffer accesses.

TAKE NOTE

Using `GR_LFB_NOIDLE` may interfere with sound generation.

The *buffer* parameter specifies which Glide buffer to lock; currently supported buffer designations are `GR_BUFFER_FRONTBUFFER`, `GR_BUFFER_BACKBUFFER`, and `GR_BUFFER_AUXBUFFER`.

If the graphics hardware supports multiple write formats to the linear frame buffer space, an application may request a particular write format with the *writeMode* parameter; valid values are listed below.

```
GR_LFBWRITEMODE_565                GR_LFBWRITEMODE_565_DEPTH
GR_LFBWRITEMODE_555                GR_LFBWRITEMODE_555_DEPTH
GR_LFBWRITEMODE_1555               GR_LFBWRITEMODE_1555_DEPTH
GR_LFBWRITEMODE_888                GR_LFBWRITEMODE_8888
GR_LFBWRITEMODE_ZA16               GR_LFBWRITEMODE_ANY
```

Use `GR_LFBWRITEMODE_ANY` when acquiring a read-only LFB pointer or when you want to use the existing data format. If the data format specified in *writeMode* is not supported on the target hardware, the lock will fail. Supported pixel formats are described in Table 11.2 and Table 11.3, later in this chapter.

If the application specifies `GR_LFB_WRITEMODE_ANY` and the lock succeeds, the destination pixel format is returned in *info.writeMode*. This default destination pixel format will always be the pixel format that most closely matches the true pixel storage format in the frame buffer. On Voodoo Graphics and Voodoo Rush, this will always be `GR_LFBWRITEMODE_565` for color buffers and `GR_LFBWRITEMODE_ZA16` for the auxiliary buffer. The *writeMode* argument is ignored for read-only locks.

Some 3Dfx hardware supports a user-specified *y* origin for LFB writes. An application may request a particular *y* origin by passing an *origin* argument other than `GR_ORIGIN_ANY`. If the *origin* specified is not supported on the target hardware, then the lock will fail. If the application specifies `GR_ORIGIN_ANY` and the lock succeeds, the LFB *y* origin is returned in *info.origin*. The default *y* origin for LFB writes is `GR_ORIGIN_UPPER_LEFT`; currently supported values are `GR_ORIGIN_UPPER_LEFT`, `GR_ORIGIN_LOWER_LEFT`, and `GR_ORIGIN_ANY`.

Some 3Dfx hardware allows linear frame buffer writes to be processed by the pixel pipeline before being written into the selected buffer. This feature is enabled by passing a value of `FXTRUE` in the *pixelPipeline* argument; **grLfbLock()** will fail if the underlying hardware is incapable of processing pixels through the pixel pipeline. When enabled, color, alpha, and depth data from the linear frame buffer write is processed as if it were generated by the triangle iterators. If the selected *writeMode* lacks depth information, then the depth value is derived from **grLfbConstantDepth**(). If the *writeMode* lacks alpha information, then the alpha value is derived from **grLfbConstantAlpha**(). Linear frame buffer writes through the pixel pipeline may not be enabled for auxiliary buffer locks. The *pixelPipeline* argument is ignored for read-only locks.

The final parameter to **grLfbLock()** is a structure of type *GrLfbInfo_t*. The *info.size* is used to provide backward compatibility for future revisions of **grLfbLock()** and must be initialized by the user to the size of the *GrLfbInfo_t* structure, as shown below. An unrecognized size will cause the lock to fail.

```
info.size = sizeof( GrLfbInfo_t );
```

Upon successful completion, the rest of the structure is filled in with information pertaining to the locked buffer. The *GrLfbInfo_t* structure is defined as:

```
typedef struct {
    int                       size;
    void                      *lfbPtr;
    FxU32                     strideInBytes;
    GrLfbWriteMode_t          writeMode;
    GrOriginLocation_t        origin;
} GrLfbInfo_t;
```

info.lfbPtr is assigned a valid linear pointer to be used for accessing the requested buffer. The access is either read-only or write-only; reading from a write pointer or writing to a read pointer will have undefined effects on the graphics subsystem. *info.strideInBytes* is assigned the byte distance between scanlines. As described above, *info.writeMode* and *info.origin* are filled in with values describing the settings in use in the currently selected buffer.

A successful call to **grLfbLock()** will cause the 3D graphics engine to idle. This is equivalent to calling **grFinish()** and may negatively impact the performance of some applications. Writes to the linear frame buffer should use **grLfbWriteRegion()**, described later in this chapter, to interleave ordered linear frame buffer copies into the 3D command stream as efficiently as possible.

When the application has completed its direct access transactions, the lock is relinquished by calling **grLfbUnlock()**, thus restoring 3D and GUI access to the buffer.

FxBool **grLfbUnlock(** *GrLock_t type*, *GrBuffer_t buffer* **)**

The two parameters, *type* and *buffer*, are identical to the first two arguments of the corresponding call to **grLfbLock()**. Note that after a successful call to **grLfbUnlock()**, accessing the *info.lfbPtr* used in the **grLfbUnlock()** call will have undefined results.

An application may not call any Glide routines other than **grLfbLock()** and **grLfbUnlock()** while any lock is active. Any such calls will result in undefined behavior.

Calculating a Pixel Address

The address of a particular pixel is computed from the (*x*,*y*) coordinates and the length of a scanline, a value that is returned in the *info* structure when **grLfbLock()** is successful. *info.strideInBytes* represents the number of bytes in a row or scanline. Thus,

$$address_{(x,y)} = y * info.strideInBytes + x$$

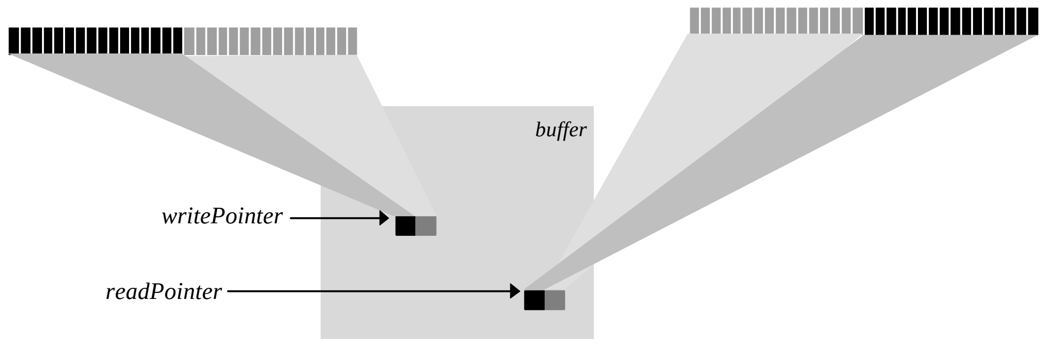
$$\text{address of the word containing } (x,y) = address_{(x,y)}/2 = (y * info.strideInBytes + x)/2$$

The location of the y origin, set in the call to **grSstWinOpen()** (see Chapter 3), determines the mapping of y addresses into frame buffer memory. When writing to the LFB, the location of the *y* origin set in **grSstWinOpen()** can be overridden, as described in the discussion of **grLfbLock()** that follows.

Reading from the LFB

To read data directly from the linear frame buffer, obtain a read-only LFB pointer by calling **grLfbLock()**, as described in the previous section. All data is read as two 16-bit pixels per 32-bit word. The default pixel ordering within the 32-bit read is 0xRRRRLLLL where the left pixel in the pair is in the lower 16-bits of the 32-bit word, as shown in Figure 11.1.

Figure 11.1 Reading from and writing to the LFB.
*When a 32-bit word is read using the read pointer acquired with a call to **grLfbLock()**, the bytes are swapped: the left most pixel is returned in the low-order half word. When a 32-bit word containing two pixels is written to the LFB, the left most pixel is in the high-order half word. Remember that.*



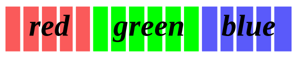
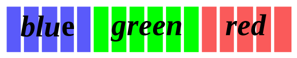
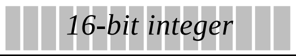
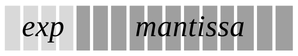
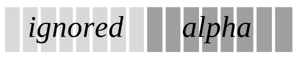
When a 32-bit word is read using the read pointer returned in *info.lfbPtr*, the target buffer determines how the data should be interpreted. If the locked buffer is a color buffer, the data should be interpreted as two RGB colors, each containing a 5-bit red value, a 6-bit green value, and a 5-bit blue value. If the locked buffer is a depth buffer, then the data contains two depth values, either 16-bit fixed point *z* values or 16-bit floating point *w* values, depending on **grDepthBufferMode()**. If the locked buffer is an alpha buffer, then the data contains two 8-bit alpha values, stored in the low order byte of each halfword. Table 11.1 shows the possible data formats.

The 16-bit floating point format for *w* is shown in Table 11.1. It has a 4-bit exponent and a 12-bit mantissa. Like IEEE floating point, a leading 1 value in the MSB of the mantissa is hidden. Note that the *w* floating point value is unsigned only. The *w* floating point format converts to a real number by using the equation:

$$1.\mathit{mantissa} * 2^{\mathit{exponent}}$$

Using this format the minimum depth value is 1.0 (floating point encoding: `0x0000`) and the maximum depth value is 65528.0 (floating point encoding: `0xFFFF`).

Table 11.1 Interpreting data read from the LFB.
When a 32-bit word is read using the read pointer acquired with a call to **grLfbLock()***, the target buffer determines how the data should be interpreted. If the locked buffer is a color buffer, the data should be interpreted as two RGB colors, each containing a 5-bit red value, a 6-bit green value, and a 5-bit blue value. If the locked buffer is a depth buffer, then the data contains two depth values, either 16-bit fixed point* ***z*** *values or 16-bit floating point* ***q*** *values, depending on* **grDepthBufferMode()***. If the locked buffer is an alpha buffer, then the data contains two 8-bit alpha values, stored in the low order byte of each halfword.*

buffer	*depth buffer mode*	*color format*	*physical layout of the data read*
`GR_BUFFER_FRONTBUFFER` `GR_BUFFER_BACKBUFFER` `GR_BUFFER_AUXBUFFER`	*ignored*	`GR_COLORFORMAT_ARGB` *or* `GR_COLORFORMAT_RGBA`	
		`GR_COLORFORMAT_ABGR` *or* `GR_COLORFORMAT_BGRA`	
`GR_BUFFER_AUXBUFFER`	`GR_DEPTHBUFFER_ZBUFFER`	*ignored*	
	`GR_DEPTHBUFFER_WBUFFER`	*ignored*	
`GR_BUFFER_AUXBUFFER`	*ignored*	*ignored*	

Example 11.1 Reading a pixel value from the LFB.
The following code segment reads 10 pixels from the color buffer currently being displayed, starting with the pixel at (100, 200), and stores them in the `pix[]` *array. The read pointer is initially set to the value returned in the* `info` *structure when the lock was initiated. A byte offset representing (100, 200) is calculated, converted to a word address, and added to the initial value to produce the starting address. The* **writeMode**, ***origin***, *and* **pixelPipeline** *arguments to* **grLfbLock()** *are ignored for read-only pointers.*

```
#define BYTESPERPIXEL 2

FxU16 pix[10];
GrLfbInfo_t info;
FxU32 *rptr;
int i;

/* get a read pointer */
if ( grLfbLock(GR_LFB_READ_ONLY, GR_LFB_FRONTBUFFER, GR_LFB_WRITEMODE_ANY,
               GR_ORIGIN_ANY, FXFALSE, &info)) {

   /* add in the word address of the first pixel */
   /* (compute byte offset for (100,200)/4   */
   rptr = info.lfbPtr
   rptr += ((*info.strideInBytes * 200) + 100*BYTESPERPIXEL)>>2;

   /*read two pixels at a time */
   for (i=0; i<10; rptr++) {
      pix[i++] = *rptr && 0xFFFF;
      pix[i++] = *rptr >>16;
   }
   grLfbUnlock( GR_LFB_READ_ONLY, GR_LFB_FRONTBUFFER );
}
```

Reading a Rectangle of Pixels from the LFB

The **grLfbReadRegion()** convenience function copies a rectangle of pixels from the frame buffer to user memory as efficiently as possible, performing the buffer locks and unlocks as needed. Note that this is the only way to read back from the frame buffer on Scanline Interleaved systems.

```
FxBool grLfbReadRegion( GrBuffer_t  src_buffer,
                        FxU32       src_x,
                        FxU32       src_y,
                        FxU32       src_width,
                        FxU32       src_height,
                        FxU32       dst_stride,
                        void        *dst_data
                       )
```

A *src_width* by *src_height* rectangle of pixels is copied from the buffer specified by *src_buffer*, starting at the location (*src_x*, *src_y*). The pixels are copied to user memory starting at *dst_data*, with a stride in bytes defined by *dst_stride*. The frame buffer *y* origin is always assumed to be at the upper left and the pixel data format is assumed to be `GR_LFBWRITEMODE_565` (see Table 11.2). The *dst_stride* must be greater than or equal to *src_width* * 2.

Writing to the LFB

To write directly to the linear frame buffer, obtain a write-only LFB pointer as described above. The call to **grLfbLock()** specifies a *writeMode* that defines the data format and a *y* origin location for the LFB writes. Both of these can be set to default to whatever conditions exist in the buffer. The *pixelPipeline* parameter enables or disables the pixel special effects pipeline.

The incoming pixel data can be interpreted in many different ways depending on the current linear frame buffer write mode and color ordering configuration. The source of depth, alpha, and color information is determined by a combination of the current linear frame buffer write mode and whether the pixel special effects pipeline is being bypassed or not. If the selected *writeMode* lacks depth information, then the value is derived from **grLfbConstantDepth()**. If the *writeMode* lacks alpha information, then the value is derived from **grLfbConstantAlpha()**. Linear frame buffer writes through the pixel pipeline may not be enabled for auxiliary buffer locks. The *pixelPipeline* argument is ignored for read only locks.

The procedure for writing to the LFB is as follows:

STEP1: If the pixel pipeline and depth buffering or alpha buffering are enabled, and if the desired *writeMode* is lacking depth or alpha values, set constant values for depth and/or alpha with **grLfbConstantDepth()** and **grLfbConstantAlpha()**.
STEP2: Call **grLfbLock()** to get a write pointer. Specify a write mode and *y* origin if desired. Bypass the pixel pipeline if desired.
STEP3: Write into the linear frame buffer using the write pointer.
STEP4: Disable LFB writing and free the buffer by calling **grLfbUnlock()**.

Each of these steps and the associated Glide functions are addressed in the remainder of this chapter, accompanied by examples of their use.

Setting LFB Write Parameters

Before you start writing data into the linear frame buffer, you need to do some set-up work.

- There are ten different formats for the data; you must choose one.
- A pixel can have red, green, blue, alpha, and depth components, but not all of the data formats provide values for all five components; you must set constant values for the ones that won't be provided by the data.
- The *y* origin can be different for LFB writes than it is for conventional rendering; set it if you want.

Linear Frame Buffer Write Modes

Data can be written into the LFB in one of several data formats or write modes.

When two 16-bit pixels are written to the hardware as a packed 32-bit value, the pixel located in the high 16-bits is written as the leftmost pixel, as shown in Figure 11.1. This is endian dependent, however, the `GLIDE_PLATFORM` compile time constant automatically allows Glide to configure itself for the proper endian characteristics. Incoming color data can be interpreted as either RGBA, ARGB, BGRA, or ABGR. This is determined by the *cFormat* parameter passed to **grSstWinOpen()** (see Table 3.2).

The write modes and resulting data formats are shown in Table 11.2 and Table 11.3.

Table 11.2 16-bit LFB data formats.
*Three of the LFB data formats write a minimum of 16 bits to the linear frame buffer. The first column in the table below gives the Glide constant for the write mode. The packing order of the color components is controlled by the **cFormat** argument to **grSstWinOpen()**. The third column shows the packing order for each write mode and each color format. Table 11.3 gives the layouts for the 32-bit LFB write formats.*

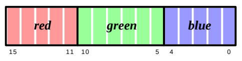
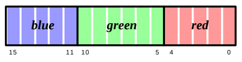
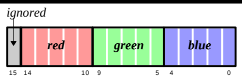
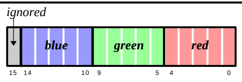
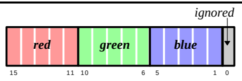
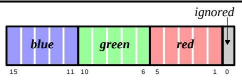
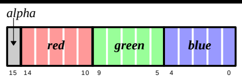
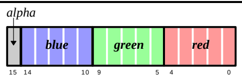
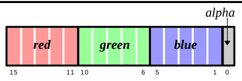
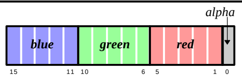
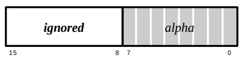
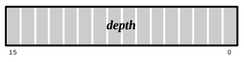
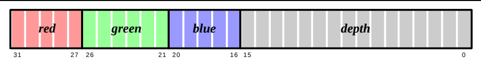
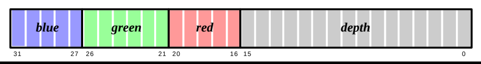
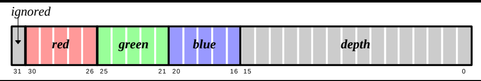
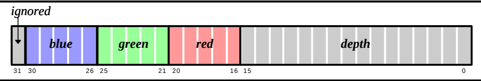
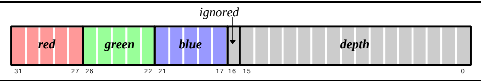
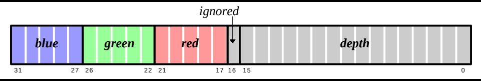
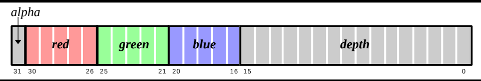
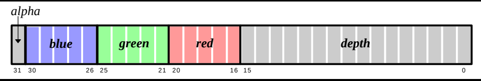
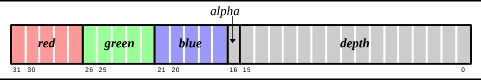
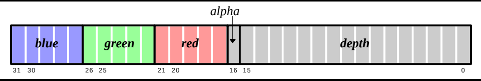
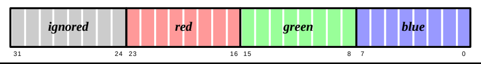
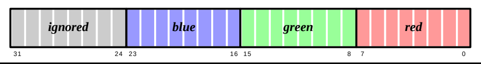
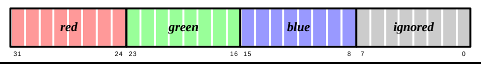
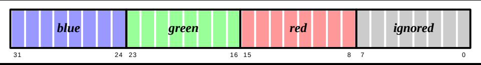
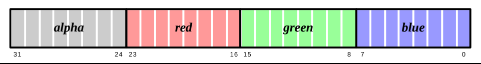
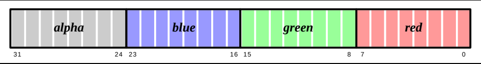
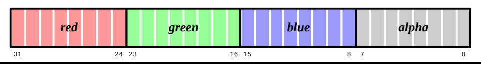
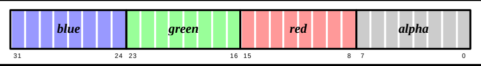
LFB write mode	*cFormat*	*physical layout of the color and depth components*
`GR_LFBWRITEMODE_565`	`GR_COLORFORMAT_ARGB` *or* `GR_COLORFORMAT_RGBA`	
	`GR_COLORFORMAT_ABGR` *or* `GR_COLORFORMAT_BGRA`	
`GR_LFBWRITEMODE_555`	`GR_COLORFORMAT_ARGB`	
	`GR_COLORFORMAT_ABGR`	
	`GR_COLORFORMAT_RGBA`	
	`GR_COLORFORMAT_BGRA`	
`GR_LFBWRITEMODE_1555`	`GR_COLORFORMAT_ARGB`	
	`GR_COLORFORMAT_ABGR`	
	`GR_COLORFORMAT_RGBA`	
	`GR_COLORFORMAT_BGRA`	
`GR_LFBWRITEMODE_ZA16` *with alpha buffering enabled*	*ignored*	
`GR_LFBWRITEMODE_ZA16` *with depth buffering enabled*	*ignored*	

Table 11.3 32-bit LFB data formats.
The LFB data formats shown below write a minimum of 32 bits to the linear frame buffer. The first column in the table below gives the Glide constant for the write mode. The packing order of the color components is controlled by the ***cFormat*** *argument to* ***grSstWinOpen()****. The third column shows the packing order for each write mode and each color format. Table 11.2 gives the layouts for the 16-bit LFB write formats.*

LFB write mode	*cFormat*	*physical layout of the color and depth components*
`GR_LFBWRITEMODE_565_DEPTH`	`GR_COLORFORMAT_ARGB` *or* `GR_COLORFORMAT_RGBA`	
	`GR_COLORFORMAT_ABGR` *or* `GR_COLORFORMAT_BGRA`	
`GR_LFBWRITEMODE_555_DEPTH`	`GR_COLORFORMAT_ARGB`	
	`GR_COLORFORMAT_ABGR`	
	`GR_COLORFORMAT_RGBA`	
	`GR_COLORFORMAT_BGRA`	
`GR_LFBWRITEMODE_1555_DEPTH`	`GR_COLORFORMAT_ARGB`	
	`GR_COLORFORMAT_ABGR`	
	`GR_COLORFORMAT_RGBA`	
	`GR_COLORFORMAT_BGRA`	
`GR_LFBWRITEMODE_888`	`GR_COLORFORMAT_ARGB`	
	`GR_COLORFORMAT_ABGR`	
	`GR_COLORFORMAT_RGBA`	
	`GR_COLORFORMAT_BGRA`	
`GR_LFBWRITEMODE_8888`	`GR_COLORFORMAT_ARGB`	
	`GR_COLORFORMAT_ABGR`	
	`GR_COLORFORMAT_RGBA`	
	`GR_COLORFORMAT_BGRA`	

Setting Constant Color, Alpha, and Depth Values

If a linear frame buffer write mode does not provide an alpha, depth, or color value, the necessary value is read from the appropriate constant alpha, color, or depth value. Pixel data written in `GR_LFBWRITEMODE_1555`, for example, contains no depth component, so depth information is pulled from the constant depth register set by **grLfbConstantDepth()**. Data written in `GR_LFBWRITEMODE_888` is missing alpha and depth components; the constant alpha register, set by **grLfbConstantAlpha()**, and the constant depth register are used.

In `GR_LFBWRITEMODE_DEPTH_DEPTH` mode, color information is retrieved from the constant color register, set by **grConstantColorValue()** and described in Chapter 5. Note that the color set by **grConstantColorValue()** is written to the color buffer while the depth components in the LFB write are written to the depth buffer. If the pixel pipeline is enabled, only the depth information is written. Table 11.4 details the source of each component for each of the LFB write modes.

Table 11.4 Color, alpha, and depth sources.
The following table illustrates where the color, alpha, and depth values come from for each of the different write modes for LFB writes that go through the pixel pipeline.

Glide constant	*color source*	*alpha source*	*depth source*
`GR_LFBWRITEMODE_565`	incoming pixel	constant alpha[2]	constant depth[3]
`GR_LFBWRITEMODE_0555`	incoming pixel	constant alpha[2]	constant depth[3]
`GR_LFBWRITEMODE_1555`	incoming pixel	incoming pixel	constant depth[3]
`GR_LFBWRITEMODE_565_DEPTH`	incoming pixel	constant alpha[2]	incoming pixel
`GR_LFBWRITEMODE_0555_DEPTH`	incoming pixel	constant alpha[2]	incoming pixel
`GR_LFBWRITEMODE_1555_DEPTH`	incoming pixel	incoming pixel	incoming pixel
`GR_LFBWRITEMODE_888`	incoming pixel	constant alpha[2]	constant depth[3]
`GR_LFBWRITEMODE_8888`	incoming pixel	incoming pixel	constant depth[3]
`GR_LFBWRITEMODE_DEPTH_DEPTH`	constant color[1]	constant alpha[2]	incoming pixel

[1]*The constant color is set by* ***grConstantColorValue()*** *and only affects chroma-keying operations, not output.*

[2]*The constant alpha value is set by* ***grLfbConstantAlpha()*** *and is only used for alpha test operations, not output.*

[3]*The constant depth value is set by* ***grLfbConstantDepth()*** *and is only used for depth test operations, not output.*

Some linear frame buffer write modes, specifically `GR_LFBWRITEMODE_555`, `GR_LFBWRITEMODE_565`, `GR_LFBWRITEMODE_1555`, `GR_LFBWRITEMODE_888`, `GR_LFBWRITEMODE_8888`, and `GR_LFBWRITEMODE_ALPHA_ALPHA`, do not possess depth information. **grLfbConstantDepth()** specifies the depth value for these linear frame buffer write modes.

void **grLfbConstantDepth(** *FxU32 depth* **)**

This depth value is used for depth buffering and fog operations and is assumed to be in a format suitable for the current depth buffering mode. Table 11.1 describes the floating point format used for *w* buffering; *z* buffers use 16-bit fixed point values. The default constant depth value is `0`.

If a linear frame buffer format contains depth information, then the depth supplied with the linear frame buffer write is used, and the constant depth value set with **grLfbConstantDepth()** is ignored.

Some linear frame buffer write modes, specifically `GR_LFBWRITEMODE_555`, `GR_LFBWRITEMODE_888`, `GR_LFBWRITEMODE_555_DEPTH`, and `GR_LFBWRITEMODE_DEPTH_DEPTH`, do not contain alpha information. **grLfbConstantAlpha()** specifies the alpha value for these linear frame buffer write modes.

void **grLfbConstantAlpha(** *GrAlpha_t alpha* **)**

This alpha value is used if alpha testing and blending operations are performed during linear frame buffer writes. The default constant alpha value is `0xFF`.

If a linear frame buffer format contains alpha information, then the alpha supplied with the linear frame buffer write is used, and the constant alpha value set with **grLfbConstantAlpha()** is ignored.

Establishing a *y* Origin

The origin for linear frame buffer writes can be set separately from the origin for other rendering (points, lines, triangles, buffer clears, etc.). This is useful in cases where images have a different origin than graphics primitives, or where different images have different origins.

The *origin* argument to **grLfbLock()** is used to establish a separate *y* origin for LFB writes, either `GR_ORIGIN_UPPER_LEFT` or `GR_ORIGIN_LOWER_LEFT`.

Special Effects and Linear Frame Buffer Writes

Look back to Figure 1.2 in Chapter 1. The pixel pipeline is not bypassed when writing directly to the linear frame buffer, unless you disable it. In fact, writing to the linear frame buffer is functionally equivalent to sending individual pixels down the pixel pipeline. Effects such as depth buffering, fog, chroma-keying, and alpha blending are not automatically disabled during LFB writes. As a result, unexpected results can occur unless all special effects are disabled, or at least set to a known state.

Disabling All Special Effects

If "pure" unmodified writes to the frame buffer are desired (a la VGA direct access), two mechanisms can be used to effect this. The first technique is to save the global state by calling **grGlideGetState()**, then disable all special effects via **grDisableAllEffects()**. Special effects can then be re-enabled individually; subsequent writes are performed on the linear frame buffer with only the desired effects enabled. When raw access to the frame buffer is complete, a call to **grGlideSetState()** resets the graphics hardware to its previous state.

void **grGlideGetState(**void **state* **)**

void **grDisableAllEffects(** void **)**

void **grGlideSetState(** const void **state* **)**

The other option for unmodified writes is enabling a hardware special effects pipeline bypass by setting the *pixelPipeline* parameter to **grLfbLock()** to `FXFALSE`. This is useful when rendering overlays or text directly to the screen and the application does not wish to disable all current effects (such as fog, depth buffering, etc.) individually.

Note that if the pixel pipeline is bypassed, then *no* effects are enabled with the exception of dithering. This includes clipping to the **grClipWindow()**, so an application must be careful not to write outside of the visible display. The values of **grColorMask()** and **grDepthMask()** are also ignored when the pixel pipeline is bypassed.

Example 11.2 Enabling specific special effects.
The following code fragment illustrates how to save Glide's state, set certain special effects, then restore Glide's state.

```
GrState state;
GrLfbInfo_t info;

// Save the state
grGlideGetState( &state );

// Selectively enable some effects
grChromakeyMode( GR_CHROMAKEY_ENABLE );
grFogMode( GR_FOG_WITH_TABLE_ON_Q );

if ( grLfbLock(GR_WRITE_ONLY, GR_BUFFER_BACKBUFFER, GR_LFBWRITEMODE_ANY,
               GR_ORIGIN_ANY, FXTRUE, &info)) {

   // write some pixels using info.lfbPtr
   // ...

   grLfbUnlock(GR_WRITE_ONLY, GR_BUFFER_BACKBUFFER);
}

// Restore the state
grGlideSetState( &state );
```

What Happens When a Special Effect is Enabled During an LFB Write?

If *depth buffering* is enabled during linear frame buffer writes, incoming pixel depths are either retrieved from the incoming pixel or from the constant depth register, depending on the write mode. Note that this can lead to some very odd effects: rarely will an application wish to depth buffer values being written to the depth buffer. If depth buffering is not desired, then the application should disable it by calling **grDepthBufferMode()** with the parameter `GR_DEPTHBUFFER_DISABLE`. Note that *depth biasing* is disabled during linear frame buffer writes because of a resource conflict between depth biasing and linear frame buffer writes.

If *alpha testing* is enabled during linear frame buffer writes, incoming pixel alpha values are either retrieved from the incoming pixel or from the constant alpha register, depending on the write mode. If alpha testing is not desired, then the application should set the alpha test function to `GR_CMP_ALWAYS`.

If *alpha blending* is enabled during linear frame buffer writes, incoming pixel alpha values are either retrieved from the incoming pixel or from the constant alpha register, depending on the write mode. If alpha blending is not desired, then the application should call **grAlphaBlendFunction(**`GR_BLEND_ONE`, `GR_BLEND_ZERO`, `GR_BLEND_ONE`, `GR_BLEND_ZERO`**)**

All other effects, such as *chroma-keying* and *fog*, act the same in linear frame buffer write modes as in normal rendering operations and are disabled as described in Chapter 8.

It is possible to directly read from and write to the alpha/depth buffer for various special effects. To write directly to the alpha/depth buffer call **grLfbLock()** with a *buffer* parameter of `GR_BUFFER_AUXBUFFER`, and then use the newly acquired pointer. When writing to the depth buffer, incoming values must be in the correct format (16-bit floating point for *w* buffering or 16-bit integer for linear *z* buffering). The 16-bit floating point format used for *w* buffering is described in Table 11.1. Remember that if depth buffering is enabled and the application is writing directly to the depth buffer, unexpected results may occur since, in essence, the application is depth buffering writes to the depth buffer.

Example 11.3 Writing one 565 RGB pixel to the back buffer (RGB ordering).

```
FxU16 pixel = 0xFFFF; // White pixel
GrLfbInfo_t info;
FxU16 *ptr;

if ( grLfbLock(GR_WRITE_ONLY, GR_BUFFER_BACKBUFFER, GR_LFBWRITEMODE_565,
               GR_ORIGIN_ANY, FXTRUE, &info)) {
   ptr = info.lfbPtr;
   ptr[x + y*info.strideInBytes] = pixel;
   grLfbUnlock(GR_WRITE_ONLY, GR_BUFFER_BACKBUFFER);
}
```

Example 11.4 Writing two 565 RGB pixels to the back buffer (RGB color ordering).
The significant difference between this example and the last one is the type of the pointer `ptr` *that is used to access frame buffer memory.*

```
GrLfbInfo_t info;
FxU32 *ptr;
Fx16 whitePixel, blackPixel;
FxU32 pixel;

whitePixel = 0xFFFF;
blackPixel = 0x0000;

// This will make the black pixel the leftmost of the pair.
pixel = ( ( ( FxU32 ) blackPixel ) << 16 ) | whitePixel;

if ( grLfbLock(GR_WRITE_ONLY, GR_BUFFER_BACKBUFFER, GR_LFBWRITEMODE_565,
               GR_ORIGIN_ANY, FXTRUE, &info)) {
   ptr = info.lfbPtr;
   ptr[x + y*info.strideInBytes] = pixel;
   grLfbUnlock(GR_WRITE_ONLY, GR_BUFFER_BACKBUFFER);
}
```

Example 11.5 Writing one 888 RGB pixel to the back buffer (ARGB color ordering).

```
GrLfbInfo_t info;
FxU32 pixel = 0x00FF0000; // Red pixel

if ( grLfbLock(GR_WRITE_ONLY, GR_BUFFER_BACKBUFFER, GR_LFBWRITEMODE_888,
               GR_ORIGIN_ANY, FXTRUE, &info)) {
   info.lfbPtr[x + y* info.strideInBytes] = pixel;
   grLfbUnlock(GR_WRITE_ONLY, GR_BUFFER_BACKBUFFER);
}
```

Writing a Rectangle of Pixels into the LFB

The **grLfbWriteRegion()** convenience function copies a rectangle of pixels from a region of memory into the linear frame buffer as efficiently as possible. It performs the buffer locks and unlocks as needed.

```
FxBool grLfbWriteRegion( GrBuffer_t buffer,
                         FxU32 xStart,
                         FxU32 yStart,
                         GrLfbSrcFmt_t srcFormat,
                         FxU32 width,
                         FxU32 height,
                         FxBool pixelPipline,
                         FxI32 strideInBytes,
                         void *data
                        )
```

The first argument, *buffer*, specifies the buffer that the data will be copied into; the choices are `GR_BUFFER_FRONTBUFFER`, `GR_BUFFER_BACKBUFFER`, and `GR_BUFFER_AUXBUFFER`. The next two parameters, *xStart* and *yStart*, specify the starting coordinates in the buffer where the data will be written. The *y* origin is assumed to be in the upper left corner of the screen.

The *srcFormat* argument describes the format of the data; valid values are shown in Table 11.5. The *width* and *height* parameters give the dimensions, in pixels, of the rectangular region to be written to the LFB, and *strideInBytes* specifies how many bytes are in one row of the array. The *pixelPipeline* argument is a Boolean value. If set to `FXTRUE`, the data is sent through the pixel pipeline on its way to the LFB. The final argument, *data*, points to the pixel data in memory.

PORTING NOTE

Glide 3.0 introduces a new argument to **grLfbWriteRegion()**: *pixelPipeline*. If is is `FXTRUE`, LFB data is written through the 3D pixel pipe. Not all hardware supports pixel pipe writes (e.g. Voodoo Rush) or source formats; **grLfbWriteRegion()** will return `FXFALSE` if an invalid or unsupported operation is attempted.

Note that *strideInBytes* can be a negative number. If *data* points to the pixel closest to the origin, and *strideInBytes* is the length of a row in the array, then the sign of *strideInBytes* represents the location of the origin in the image pointed to by *data*. A negative *strideInBytes* is used if *data* points to the lower left corner, as shown in Figure 11.2.

Note also that not all hardware supports pixel pipe writes or source formats (e.g., Voodoo Rush). **grLfbWriteRegion()** will return `FXFALSE` if an invalid or unsupported operation is attempted.

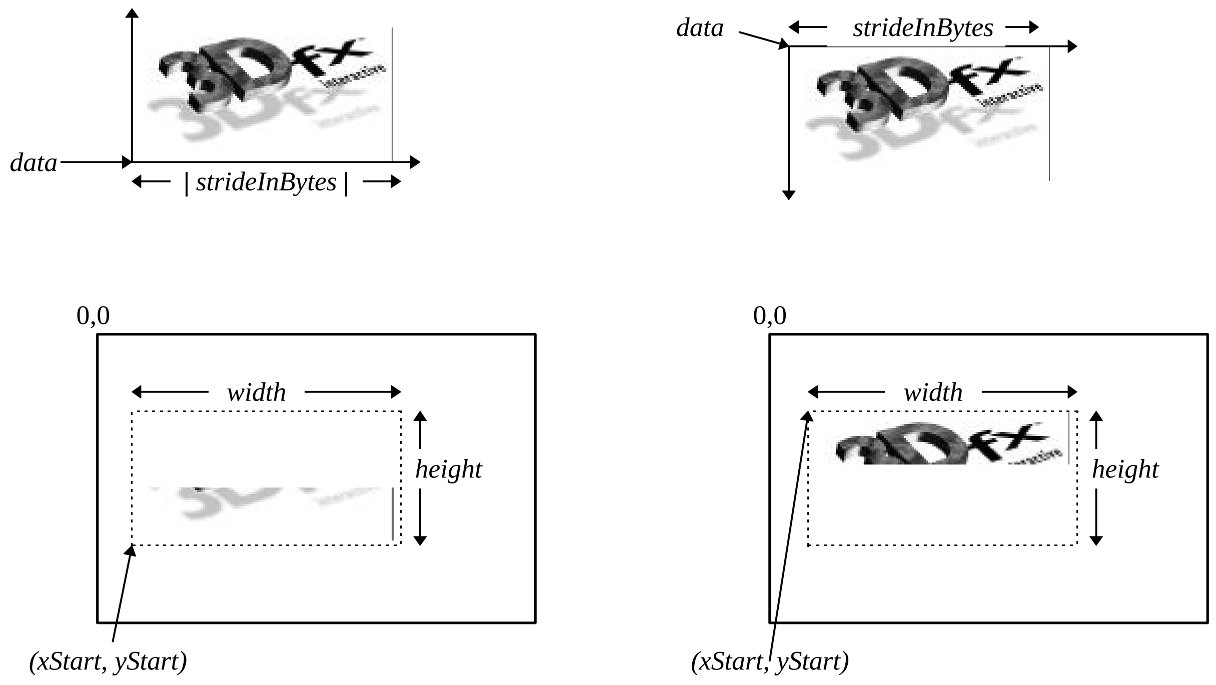
Table 11.5 Source data formats for the grLfbWriteRegion() routine.

source data format	*description*
`GR_LFB_SRC_FMT_565`	RGB 565 color image
`GR_LFB_SRC_FMT_555`	RGB 555 color image
`GR_LFB_SRC_FMT_1555`	RGB 1555 color image
`GR_LFB_SRC_FMT_888`	RGB 888 color image. Each pixel is padded to 32 bits with RGB in the low order 24 bits.
`GR_LFB_SRC_FMT_8888`	ARGB 8888 color image
`GR_LFB_SRC_FMT_565_DEPTH`	RGB 565 and 16-bit depth value packed into each 32-bit element of image
`GR_LFB_SRC_FMT_555_DEPTH`	RGB 555 and 16-bit depth value packed into each 32-bit element of image
`GR_LFB_SRC_FMT_1555_DEPTH`	RGB 1555 and 16-bit depth value packed into each 32-bit element of image
`GR_LFB_SRC_FMT_ZA16`	Two 16-bit depth or alpha values. Alpha values are stored into odd bytes.
`GR_LFB_SRC_FMT_RLE16`	A 16-bit RLE Encoded image: each pixel has a16-bit signed count and a 16-bit color. Negative counts are currently ignored.

Figure 11.2 Frame buffer writes: encoding the location of the origin as the sign of the strideInBytes.
If the image you want to write into the linear frame buffer is defined with the origin in the lower left corner, you can use a negative ***strideInBytes*** *to compute addresses, as shown in part (a) below. If the origin is in the upper left corner, use a positive* ***strideInBytes****, as shown in part (b). The bottom half of each diagram shows the pixel copy in progress.*

(a).
corner and is negative.

(b).
left corner and ***strideInBytes***



$$address_{(x,y)} = data + (x + y*strideInBytes)$$

Thus, a rectangle of *srcFormat* pixels pointed to by *data* and defined by *width*, *height*, and *strideInBytes* will be copied into *buffer* at the location (*xStart*, *yStart*). Note that not all 3Dfx graphics subsystems support all source image formats; **grLfbWriteRegion()** will fail if the source format is not supported.

12. *Housekeeping Routines*

In This Chapter

Glide provides a collection of routines that return information about the system, the software, and the scene being rendered.

You will learn how to:

- ▼ retrieve system configuration information: the current version of Glide, the number of SST subsystems present, the size of the display screen, fog table or gamma correction table, the minimum and maximum values for the depth buffer.
- ▼ answer system status questions: How full is the FIFO? How many pixels entering the pixel pipeline are actually drawn? What is the swap rate?
- ▼ make sure that all pending graphics commands have been executed.
- ▼ change the location of the *y* origin.
- ▼ enable and disable Glide operating modes.

Retrieving Configuration Information

The **grGet()** routine retrieves the values of selected Glide state variables that are numbers.

FxI32 **grGet(** *FxU32 pname, FxU32 plength, FxI32 *params* **)**

grGet() retrieves the values of selected Glide state variables. The first argument, *pname,* tells Glide which environmental parameters to return. The possible values are shown in Table 12.1. The other arguments describe the buffer in which the values are returned: *plength* is the length, in bytes, of the buffer and *params* is a pointer to it.

If successful, **grGet()** returns the number of bytes written into the *params* buffer. If **grGet()** fails, it returns 0; the contents of the *params* array are unchanged. Possible reasons for failure include invalid Glide context, an invalid *pname*, and NULL *params*.

PORTING NOTE

grGet() replaces a whole bunch of APIs:

Out with the old	*In with the new:*
grBufferNumPending()	**grGet(**`GR_PENDING_BUFFERSWAPS,`…**)**
grGlideGetVersion()	**grGetString(**`GR_VERSION,`…**)**
grSstIsBusy()	**grGet(**`GR_IS_BUSY,`…**)**
grSstPerfStats()	**grGet(**`GR_STATS_PIXEL_*,`…**)**
grSstQueryBoards()	**grGet(**`GR_NUM_BOARDS,`…**)**
grSstQueryHardware()	**grGet(*,** …**), grGetString(*,** …**)**
grSstScreenHeight()	**grGet(**`GR_VIEWPORT,`…**)**
grSstScreenWidth()	**grGet(**`GR_VIEWPORT,`…**)**

grSstStatus()	**grGet(**`GR_IS_BUSY,`…**), grGet(**`*,`…**)**
grSstVideoLine()	**grGet(**`GR_VIDEO_POS,`…**)**
grSstVRetraceOn()	**grGet(**`GR_VIDEO_POS,`…**)** *when 0 is returned*
Constants:	
`GR_WDEPTHVALUE_NEAREST`	**grGet(**`GR_WDEPTH_MIN_MAX,`…**)**
`GR_WDEPTHVALUE_FARTHEST`	**grGet(**`GR_WDEPTH_MIN_MAX,`…**)**
`GR_ZDEPTHVALUE_NEAREST`	**grGet(**`GR_ZDEPTH_MIN_MAX,`…**)**
`GR_ZDEPTHVALUE_FARTHEST`	**grGet(**`GR_ZDEPTH_MIN_MAX,`…**)**

Table 12.1 Selectors for grGet().
The pre-defined constants in the first column can be used as the first argument to ***grGet()****. The other three columns describe the data that will be used if the chosen selector is used.*

selector encoded in ***pname***	*number of values returned*	*number of bytes returned*	*description of value(s) returned in* ***params***
`GR_BITS_DEPTH`	1	4	The number of bits of depth (*z* or *q*) in the frame buffer.
`GR_BITS_RGBA`	4	16	The number of bits each of red, green, blue, alpha in the frame buffer. If there is no separate alpha buffer (e.g. on the SST-1 the depth buffer can be used as an alpha buffer), 0 is returned for the alpha bits.
`GR_BITS_GAMMA`	1	4	The number of bits for each channel in the gamma table. If gamma correction is not available, **grGet()** will fail, and the *params* array is unmodified.
`GR_FIFO_FULLNESS`	2	8	How full the FIFO is, as a percentage. The value is returned in two forms: 1.24 fixed point and a hardware-specific format.
`GR_FOG_TABLE_ENTRIES`	1	4	The number of entries in the hardware fog table. For Voodoo Graphics, the value is 64.
`GR_GAMMA_TABLE_ENTRIES`	1	4	The number of entries in the hardware gamma table. Returns `FXFALSE` if it is not possible to manipulate gamma (e.g. on a Macronix card, or in windowed mode).
`GR_GLIDE_STATE_SIZE`	1	4	Size of buffer, in bytes, needed to save Glide state. See **grGlideGetState()**.
`GR_GLIDE_VERTEXLAYOUT_SIZE`	1	4	Size of buffer, in bytes, needed to save the current vertex layout. See **grGlideGetVertexLayout()**.
`GR_IS_BUSY`	1	4	Returns `FXFALSE` if idle, `FXTRUE` if busy.
`GR_LFB_PIXEL_PIPE`	1	4	Returns `FXTRUE` if LFB writes can go through the 3D pixel pipe, `FXFALSE` otherwise.
`GR_MAX_TEXTURE_SIZE`	1	4	The width of the largest texture supported on this configuration (e.g. Voodoo Graphics returns 256).
`GR_MAX_TEXTURE_ASPECT_RATIO`	1	4	The logarithm base 2 of the maximum aspect ratio supported for power-of-two, mipmap-able textures (e.g. Voodoo Graphics returns 3).
`GR_MEMORY_FB`	1	4	The total number of bytes per Pixel*fx* chip if a non-UMA configuration is used, else 0. In non-UMA configurations, the total FB memory is `GR_MEMORY_FB` * `GR_NUM_FB`.
`GR_MEMORY_TMU`	1	4	The total number of bytes per Texel*fx* chip if a non-UMA configuration is used, else `FXFALSE`. In non-UMA configurations, the total usable texture memory is `GR_MEMORY_TMU` * `GR_NUM_TMU`.
`GR_MEMORY_UMA`	1	4	The total number of bytes if a UMA configuration, else 0.

Table 12.1 Selectors for grGet(). (continued)
*The pre-defined constants in the first column can be used as the first argument to **grGet()**. The other three columns describe the data that will be used if the chosen selector is used.*

*selector encoded in **pname***	*number of values returned*	*number of bytes returned*	*description of value(s) returned in **params***
`GR_NON_POWER_OF_TWO_TEXTURES`	1	4	Returns `FXTRUE` if this configuration supports textures with arbitrary width and height (up to the maximum). Note that only power-of-two textures may be mipmapped. *Not implemented in the initial release of Glide 3.0.*
`GR_NUM_BOARDS`	1	4	The number of installed boards supported by Glide. Valid before a call to **grSstWinOpen()**.
`GR_NUM_FB`	1	4	The number of Pixel*fx* chips present. This number will always be 1 except for SLI configurations.
`GR_NUM_SWAP_HISTORY_BUFFER`	1	4	Number of entries in the swap history buffer. Each entry is 4 bytes long.
`GR_NUM_TMU`	1	4	The number of Texel*fx* chips per Pixel*fx* chip. For integrated chips, the number of TMUs is returned.
`GR_PENDING_BUFFERSWAPS`	1	4	The number of buffer swaps pending.
`GR_REVISION_FB`	1	4	The revision of the Pixel*fx* chip(s).
`GR_REVISION_TMU`	1	4	The revision of the Texel*fx* chip(s).
`GR_STATS_LINES`	1	4	The number of lines drawn.
`GR_STATS_PIXELS_AFUNC_FAIL`	1	4	The number of pixels that failed the alpha function test.
`GR_STATS_PIXELS_CHROMA_FAIL`	1	4	The number of pixels that failed the chroma key (or range) test.
`GR_STATS_PIXELS_DEPTHFUNC_FAIL`	1	4	The number of pixels that failed the depth buffer test.
`GR_STATS_PIXELS_IN`	1	4	The number of pixels that went into the pixel pipe.
`GR_STATS_PIXELS_OUT`	1	4	The number of pixels that went out of the pixel pipe.
`GR_STATS_POINTS`	1	4	The number of points drawn.
`GR_STATS_TRIANGLES_IN`	1	4	The number of triangles received.
`GR_STATS_TRIANGLES_OUT`	1	4	The number of triangles drawn.
`GR_SWAP_HISTORY`	1	4	The *swapHistory* buffer contents. The *i*th 4-byte entry counts the number of vertical syncs between the (current frame – *i)*th frame and its predecessor. If *swapHistory* is not implemented (e.g. on Voodoo Graphics and Voodoo Rush), **grGet()** will fail, and the *params* array is unmodified. *Use **grGet(**`GR_NUM_SWAP_HISTORY_BUFFER,`**…)** to determine the number of entries in the buffer.*
`GR_SUPPORTS_PASSTHRU`	1	4	Returns FXTRUE if pass through mode is supported. See **grEnable**().

Table 12.1 Selectors for grGet(). (continued)
The pre-defined constants in the first column can be used as the first argument to ***grGet()***. *The other three columns describe the data that will be used if the chosen selector is used.*

selector encoded in ***pname***	*number of values returned*	*number of bytes returned*	*description of value(s) returned in* ***params***
`GR_TEXTURE_ALIGN`	1	4	Alignment boundary for textures. For example, if textures must be 16-byte aligned, `0x10` would be returned.
`GR_VIDEO_POSITION`	2	8	Vertical and horizontal beam location. Vertical retrace is indicated by *y* == 0.
`GR_VIEWPORT`	4	16	*x*, *y*, width, height of the viewport.
`GR_WDEPTH_MIN_MAX`	2	8	Minimum and maximum allowable *w* buffer values.
`GR_ZDEPTH_MIN_MAX`	2	8	Minimum and maximum allowable *z* buffer values.

The **grGetString()** routine returns environmental parameters that are character strings.

const char ***grGetString(** *FxU32 name* **)**

grGetString() returns a pointer to the string selected by the *name* argument, or NULL if *name* is invalid.

Table 12.2 Selectors for grGetString().

selector specified in ***name***	*description*
`GR_EXTENSION`	Returns a space-separated list of Glide extension names (the extension names themselves do not contain spaces). If no extensions are supported, a single space “ “ is returned.
`GR_HARDWARE`	Returns one of “Voodoo Graphics”, “Voodoo Rush”, “Voodoo2”, or “Banshee”. Other types may be added in the future.
`GR_RENDERER`	“Glide”.
`GR_VENDOR`	The vendor, “3Dfx Interactive”.
`GR_VERSION`	The Glide version. For example, “3.0-alpha”.

Completing Graphics Commands

When a Glide user issues a command that provides data or state to the hardware, the command is queued and will be executed some time later, in the order issued. Two commands allow the user to force the completion of outstanding commands.

void **grFinish(** void **)**

Calling **grFinish()** forces all previously issued Glide commands to complete: it does not return until all effects from previous commands are fully realized on the screen. **grFinish()** should be used judiciously as it can have severe performance impacts if called to frequently.

void **grFlush(** void **)**

Calling **grFlush()** forces all previously issued commands to begin execution, guaranteeing they will complete in finite time. However, they may not all be completed upon return. Use **grFlush()** to guarantee command completion upon return.

Glide 3.0 is the first release to support **grFlush()**. It is a no-op in current hardware because commands are not buffered (they are FIFOed, and the FIFO is guaranteed to drain). Future hardware designs may utilize a buffer rather than a FIFO; in that case, this command will become necessary. Developers interested in writing upward-compatible software should start using them now.

PORTING NOTE

The Glide 2.x routine **grSstIdle()** has been replaced by **grFinish()**.

Monitoring System Performance

The graphics hardware maintains a set of counters that collect statistics about the fate of pixels as they move through the pixel pipeline. Glide returns the current values of these counters with **grGet()**; the counters can be reset by calling **grReset()**, described below.

In order to account for every pixel counted and saved in *pixelsOut*, one must use the following equation:

$$pixelsOut = LfbWritePixels + bufferClearPixels + (pixelsIn – depthFuncFail – chromaFail – aFuncFail)$$

The pixel counters are accessed with **grGet()** selectors similar to the variable names used in the equations: `GR_STATS_PIXELS_OUT`, `GR_STATS_PIXELS_IN`, `GR_STATS_DEPTHFUNC_FAIL`, `GR_STATS_CHROMA_FAIL`, and `GR_STATS_AFUNC_FAIL`. *bufferClearPixels* represents the number of pixels written as a result of calls to **grBufferClear()** and can be calculated as:

bufferClearPixels = (# of times the buffer was cleared)* (clip window width) * (clip window height)

In addition to the pixel statistics, **grGet()** will return the number of points drawn (`GR_STATS_POINTS`), the number of lines drawn (`GR_STATS_LINES`), the number of triangles started (`GR_STATS_TRIANGLES_IN`) and the number of triangles actually drawn (`GR_STATS_TRIANGLES_OUT`).

grGet() does not wait for the system to be idle, and hence does not include statistics for commands that are still in the FIFO. Call **grFinish()** to empty the FIFO.

The counters are reset by calling **grReset()** with the appropriate selector. The hardware counters are only 24-bits wide, so regular calls to **grReset()** are required to avoid overflow.

void **grReset(** *FxU32 what* **)**

grReset() resets statistic counters. The argument *what* is one of the selectors listed in Table 12.3.

PORTING NOTE

The Glide 2.x routine **grSstResetPerfStats()** has been replaced by **grReset(**`GR_STATS_PIXELS,…`**)**.

Table 12.3 Selectors for grReset().

***what** selector*	*description*
`GR_STATS_PIXELS`	Reset all the pixel statistic counters.
`GR_STATS_POINTS`	Reset all the point statistic counters.
`GR_STATS_LINES`	Reset all the line statistic counters.
`GR_STATS_TRIANGLES`	Reset all the triangle statistic counters.
`GR_VERTEX_PARAMETERS`	Reset all **grVertexLayout()** parameter offsets to zero and all modes to `GR_PARAM_DISABLE`.

Changing the *y* Origin

The location of the *y* origin is initially established as part of the **grSstWinOpen()** call in the Glide initialization sequence. The initial setting can be overridden later on by calling **grSstOrigin()**.

void **grSstOrigin(** *GrOriginLocation_t origin* **)**

The argument, *origin*, specifies the direction of the *y* coordinate axis. `GR_ORIGIN_UPPER_LEFT` places the screen space origin at the upper left corner of the screen with positive *y* going down. `GR_ORIGIN_LOWER_LEFT` places the screen space origin at the lower left corner of the screen with positive *y* going up.

Enabling Glide Operating Modes

Several operating modes can be selectively enabled and disabled by the application programmer:

- *Anti-aliasing*. Vertices must be sorted by depth. When enabled, points, lines, and triangles are anti-aliased. This mode is ignored when drawing strips and fans.
- *Shameless plug*. When enabled, the 3Dfx Interactive power shield logo is blended into each frame drawn. Good for trade shows.
- *Video smoothing*. When enabled and with hardware support,
- *Allow nearest dithered mipmapping*. When enabled, the application is allowed to enable nearest dithered mipmapping, a technique that alleviates the effects of mipmap banding at the cost of performance degradation for larger texture maps. Use it only if you can live with the poor performance. Note that you must actually enable nearest dithered mipmapping by calling **grTexMipMapMode()**.

Use **grEnable()** and **grDisable()** to select these operating modes.

void **grEnable(** *GrEnableMode_t mode***)**
void **grDisable(***GrEnableMode_t mode***)**

The single argument to both routines is one of the mode selectors shown in Table 12.4.

PORTING NOTE

Most of the functionality of the old **grSstOpen()** command was implemented in **grSstWinOpen()**. The *smoothing_mode* argument, however, has been replaced by **grReset(**`GR_VIDEO_SMOOTHING,…`**)**.

The old grHints(`GR_HINT_ALLOW_MIPMAP_DITHER`**)** functionality is now implemented as grEna**ble(**`GR_ALLOW_MIPMAP_DITHER`**).**

Table 12.4 Glide operating modes.

mode	*description*	*default*
`GR _AA_ORDERED`	An anti-aliasing method that requires objects to be sorted by depth. This mode applies to all primitives except strips and fans.	*disabled*
`GR_ALLOW_MIPMAP_DITHER`	Allow `GR_MIPMAP_NEAREST_DITHER` mode. By default, this mode cannot be enabled with **grTexMipMapMode()** because of the performance impact. Note that this does not actually set mipmap dithering; **grTexMipMapMode()** must still be called.	*disabled*
`GR_PASSTHRU`	Pass through mode. When enabled, the graphics frame buffer will displayed. When disabled, the VGA frame buffer is displayed. (This feature replaces the now-obsolete **grSstControl()** API). Pass through mode is not supported by all hardware configurations. Use **grGet(**`GR_SUPPORTS_PASSTHRU,…`**)** to determine whether or not pass through mode is supported on the current system.	*depends on system configuration*
`GR_SHAMELESS_PLUG`	The 3Dfx power shield shameless plug is blended into each displayed frame if the mode is enabled.	*disabled*
`GR_VIDEO_SMOOTHING`	Video smoothing mode. If the hardware does not support video smoothing, this mode is a no-op.	*enabled*

Glide Utilities

Glide 3.0 defines six utility commands in the glideutl.h header file. Four help generate fog tables and are described in Chapter 8. The other two define and read files of frame buffer data and are described in Chapter 11.

PORTING NOTE

Glide 3.0 make a long list of utility routines disappear: most of these are remnants of Glide 1.0 that have no possible use any more:

guAADrawTriangleWithClip()
guAlphaSource()
guColorCombineFunction()
guDrawTriangleWithClip()
guDrawPolygonVertexListWithClip()
guEncodeRLE16()
guEndianSwapBytes()
guEndianSwapWords()
guTexAllocateMemory()
guTexChangeAttributes()
guTexCombineFunction()
guTexGetCurrentMipMap()
guTexGetMipMapInfo()
guTexMemQueryAvail()
guTexMemReset()
guTexDownloadMipMap()
guTexDownloadMipMapLevel()
guTexSource()
guTexCreateColorMipMap()
guFbReadRegion()
guFbWriteRegion()

13. Glide Extensions

In This Chapter

Glide 3.0 introduces a mechanism for adding hardware, operating system, and application specific extensions to the Glide Library. A Glide application calls **grGetString()** to determine if a given extension is available on the current system configuration. If it is, **grGetProcAddress()** returns an entry point. By convention, extension procedure names end with "**_EXT**".

In this chapter, you will discover:

- ▼ a mechanism for identifying and executing extensions.
- ▼ a chroma-ranging extension that allows a range of color values (instead of a single value) to be used as the chroma-key.
- ▼ an extension that implements chroma-ranging on texels.
- ▼ an extension that allows a fog coordinate to be included in each vertex.
- ▼ an extension that allows an ARGB color palette to be used.
- ▼ an extension that allows textures to be mirrored as they are repetitively applied.

Using Extensions

Calling the procedure **grGetString(**`GR_EXTENSION,…`**)** returns a space-delimited list of the names of extensions that are available for the current system configuration. In general, newer hardware (like Voodoo^2), supports all of the extensions while older hardware (like Voodoo Graphics and Voodoo Rush) support none of them. A single space is returned if no extensions are supported by the current hardware.

Some of the extensions increase the available modes for existing commands. Others introduce new commands; these are shown in the last column of Table 13.1. To access one of the procedures associated with a supported extension, use **grGetProcAddress()** to retrieve a pointer to it. Table 13.1 lists the extensions that may be present in Glide 3.0.

GrProc **grGetProcAddress(** char **procName* **)**

Table 13.1 Extension and procedure names.
This table lists the names of Glide extensions. If the current system configuration supports the extension, its name is included in the string returned by ***grGetString(**`GR_EXTENSION,…`**)**. *If an extension is supported, entry points for the procedures that implement it can be accessed through calls to* ***grGetProcAddress()***. *These extensions are not available on systems with Voodoo Graphics and Voodoo Rush hardware.*

extension name	*description*	*associated procedure names*

CHROMARANGE	*Chroma-range feature in the pixel pipeline is supported.*	**grChromaRangeModeExt( )**
		grChromaRangeExt()

TEXCHROMA	*Chroma-range feature in the texture mapping unit is supported.*	**grTexChromaModeExt()**
		grChromaRangeExt()

FOGCOORD	`GR_FOG_PARAM` *vertex parameter in* **grVertexLayout()** *is supported.*	
PALETTE6666	`GR_TEXTABLE_PALETTE_6666` *format is supported.*	
TEXMIRROR	`GR_TEXTURECLAMP_MIRROR_EXT` *mode in* **grTexClampMode()** *is supported.*	

The Chroma-Range Extension

Chapter 8 described the chroma-key operation: a way to screen out all pixels that match a designated color. Glide 3.0 introduces *chroma-ranging*, a generalization of the single chroma-key color to a range of colors. The chroma-range extension is available only with hardware support. Use **grGetString(**`GR_EXTENSION,…`**)** and search for the sub-string "CHROMARANGE" to query for availability of this extension. If the chroma-range extension is present, the entry points may be retrieved via **grGetProcAddress()**.

Two routines implement the chroma-range extension: **grChromaRangeModeExt()** enables and disables the mode and **grChromaRangeExt()** establishes the chroma-range and the match criteria.

void **grChromaRangeModeExt(** *GrChromakeyMode_t mode* **)**

grChromaRangeModeExt() enables and disables chroma-range checking. The mode argument can be one of two values: `GR_CHROMARANGE_DISABLE` or `GR_CHROMARANGE_ENABLE`.

Chroma-keying must be enabled (using **grChromakeyMode()**) before **grChromaRangeModeExt()** is executed, and it will remain enabled after chroma-ranging is disabled. You can disable both modes by disabling chroma-keying.

void **grChromaRangeExt(** *GrColor_t color0, GrColor_t color1, FxU32 mode* **)**

grChromaRangeExt() sets the global chroma-range reference values as order-insensitive packed RGB values. The color format for *color0* and *color1* should be the same one as specified in the *cFormat* parameter to **grSstWinOpen()** (see Chapter 3). The order in which chroma-range values are specified for a particular color component is irrelevant, i.e. the { *color0*, *color1* } pairs {(130,36,87), (150,38,92)} and {(150,36,92), (130,38,87)} are equivalent.

The *mode* argument determines the match criteria for the chroma test. Only one value for *mode* is supported in Glide 3.0: `GR_CHROMARANGE_RGB_ALL`. In this *mode*, the { *color0*, *color1* } pair defines an inclusive range, i.e., the range falling between the minimum and maximum pair values. If all components of the incoming pixel color fall within their respective ranges, the chroma test succeeds and the pixel is invalidated.

The chroma-range comparison uses the *other* color specified in the configuration of the color combine unit, and is performed between colors with 24-bit.

Chroma-Ranges and Texels

If **grGetString(**`GR_EXTENSION`**)** returns the sub-string "TEXCHROMA", then chroma-ranges for each TMU can be specified. The entry points for the two routines that implement the TEXCHROMA extension, **grTexChromaModeExt()** and **grTexChromaRangeExt()**, may be retrieved via **grGetProcAddress()**.

void **grTexChromaModeExt(** *GrChipID_t tmu, GrChromakeyMode_t mode* **)**

grTexChromaModeExt() enables or disables chroma-ranging in the designated *tmu* (`GR_TMU0`, `GR_TMU1`, or `GR_TMU`). The *mode* argument is either `GR_TEXCHROMA_ENABLE` or `GR_TEXCHROMA_DISABLE`.

```
void grTexChromaRangeExt( GrChipID_t              tmu,
                          GrColor_t               color0,
                          GrColor_t               color1,
                          GrTexChromakeyMode_t    mode
                        )
```

grTexChromaRangeExt() sets the chroma-range values for the TMU specified by *tmu*. The *color0* and *color1* arguments behave like those for pixel chroma-ranging, described in the previous section.

The *mode* parameter determines the way the color ranges are used in the texel chroma test. Only one value is currently supported, `GR_TEXCHROMARANGE_RGB_ALL_EXT`. In this mode, each color component pair defines an inclusive range, i.e., the range falling between the minimum and maximum pair values. If ***all*** components of the incoming pixel color fall within their ranges, the chroma test succeeds and the pixel is invalidated.

The FOGCOORD Extension

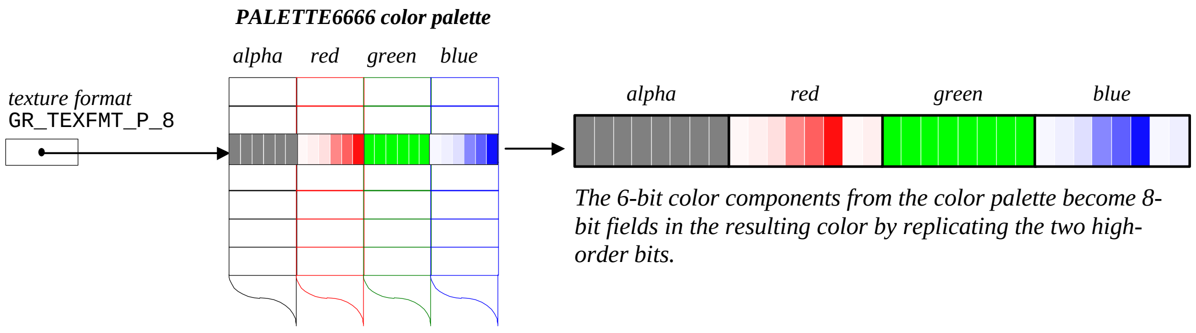
If the FOGCOORD extension is supported, the `GR_PARAM_FOG_EXT` parameter can be declared as part of a vertex layout (see **grVertexLayout()** in Chapter 2). The fog coordinate is used to index a fog table in `GR_FOG_WITH_FOGTABLE_ON_FOGCOORD_EXT` mode. See Chapter 8 for more details.

The PALETTE6666 Extension

Glide 3.0 introduces another color palette format with the PALETTE6666 extension. The new format stores a 24-bit ARGB color (6 bits per component) in the palette rather than the 24-bit RGB value of th standard palette. The PALETTE6666 format is used in conjunction with texture format `GR_TEXFMT_P_8` (see Table 10.1 for more information about texture formats).

Figure 13.1 The PALETTE666 color palette.
The PALETTE6666 color palette holds 256 ARGB colors. Each entry is 24 bits wide, with 6 bits allocated to each of the color components. A palette entry is retrieved when rendering with a texture map utilizing texture format `GR_TEXFMT_P_8`. The texel is an offset into the color palette.

The resulting color is a 32-bit quantity with each 6-bit color component expanded to 8 bits as shown below.


PALETTE6666 color palette
alpha
red
green
blue
texture format
GR_TEXFMT_P_8
alpha
red
green
blue
The 6-bit color components from the color palette become 8-bit fields in the resulting color by replicating the two high-order bits.

The TEXMIRROR Extension

If the TEXMIRROR extension is supported, another texture clamping mode is available: `GR_TEXTURECLAMP_MIRROR_EXT`. Figure 13.2 shows the effect of repetitively applying the texture with mirroring in both the s and t directions. Figure 13.3 shows how mirror mode interacts with the other two texture clamping modes, described in Chapter 9.

Figure 13.2 A GR_TEXTURECLAMP_MIRROR_EXT example.
The illustrations below show a texture mapping with three different ranges for s and t and the texture clamping mode for both coordinates set to `GR_TEXTURECLAMP_MIRROR_EXT`. Clip coordinate space is used for this example.

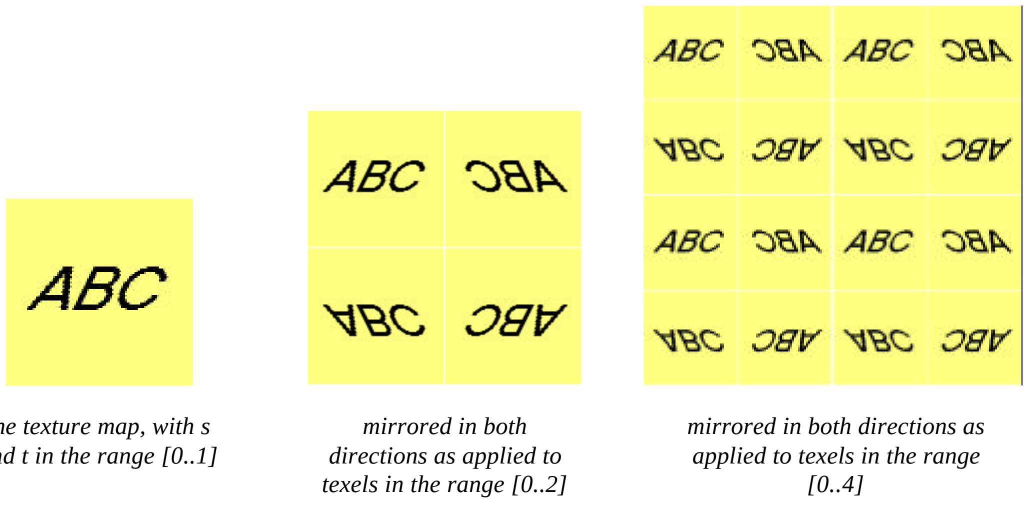
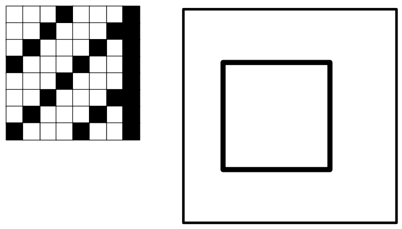
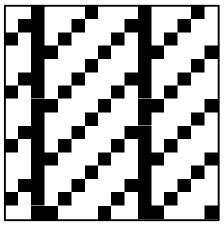

the texture map, with s and t in the range [0..1]
mirrored in both directions as applied to texels in the range [0..2]
mirrored in both directions as applied to texels in the range [0..4]

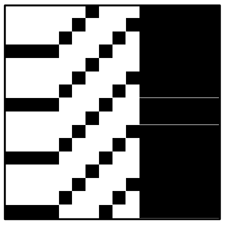
Figure 13.3 Texture clamping.
The texture clamp mode specifies what to do when texture coordinates are outside the range of the texture map. If wrapping is enabled, then texture maps will tile, i.e., values greater than 255 will wrap around to 0. If clamping is enabled, then texture map indices are clamped to 0 and 255. Both modes should always be set to `GR_TEXTURECLAMP_CLAMP` *when using projected textures.*

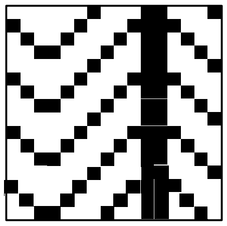
Glide 3.0 introduces a texture clamp mode extension, `GR_TEXTURECLAMP_MIRROR_EXT`*, that is available if the TEXMIRROR extension is supported. See Chapter 13 for details and an expanded version of this figure.*

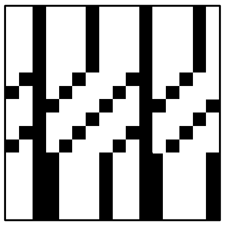


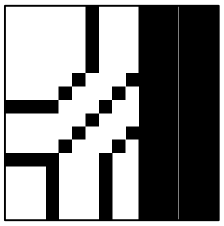
The texture on the left is to be mapped onto the rectangle, with the texture origin in the interior of the rectangle. The clamp mode settings for s and t affect the final result, as shown below.

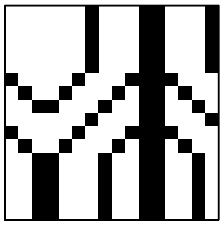

wrap both *s* and *t*

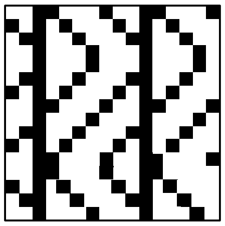

clamp *s*, wrap *t*

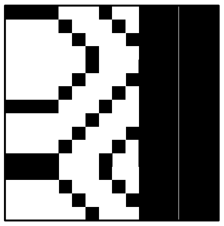

mirror *s*, wrap *t*

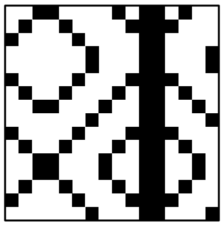

wrap *s*, clamp *t*


clamp both *s* and *t*


mirror *s*, clamp *t*


wrap *s*, mirror *t*


clamp *s*, mirror *t*


mirror both *s* and *t*

14. Programming Tips and Techniques

In This Chapter

This chapter is a collection of short programming tips. You will read about:

▼ avoiding redundant state changes.

▼ minimizing screen clears.

▼ controlling texture aliasing artifacts with an LOD bias.

▼ precision compression artifacts that can arise when *z* buffering.

▼ state coherency and contention between processes.

Avoiding Redundant State Setting

If an application depth sorts all the polygons in a scene, the arbitrary order in which polygons are rendered can potentially cause an immense amount of redundant state information to be passed to the hardware. This is a difficult problem to solve, however, the following guidelines should assist when attempting to efficiently maintain state:

- Use material libraries to clump together attributes into "materials". Change states en masse whenever a new material becomes current, but only change the current material when necessary.
- Use intelligent object rendering code that renders similar triangles (in terms of state attributes) together to minimize unnecessary state updates.

Avoiding Screen Clears by Rendering Background Polygons

If an application does not need to clear the alpha or depth buffers, it can forego clearing the display buffer by rendering large background polygons first. For example, a flight simulator will typically render large sky and ground polygons that will effectively cover the entire screen, removing the need to clear the display buffer.

Using LOD Bias To Control Texture Aliasing

LOD calculations computed for mipmapping can be biased to finely control the point at which mipmap levels are crossed. The LOD bias for a texture is specified by calling **grTexLodBiasValue()**. For bilinear, blended, mipmapped, non-mipmap dithered, non-mipmap-interpolated textures, an LOD bias value of 0.5 is typically sufficient. For bilinear, blended, mipmapped, mipmap interpolated textures, an LOD bias value of –3/8 is typically sufficient.

However, the choice of an LOD bias value is highly dependent on the frequency of a texture. If textures are fairly high in frequency, then a larger LOD bias may be required to reduce texture aliasing artifacts.

Linear *z* Buffering and Coordinate System Ranges

The graphics hardware supports linear *z* buffering by storing the 16-bit whole part of any *z* values passed to the hardware. A side effect of this is that the precision of the *z* buffer tends to be concentrated very close to the viewer. Therefore *z* buffer "poke through" may occur as a result of the compression of precision close to the viewer.

State Coherency and Contention Between Processes

Neither the graphics hardware nor Glide handle resource contention management in multithreaded or multitasking environments. Thus, an application that has multiple threads or processes accessing Glide and/or the graphics hardware must maintain state coherency and perform context management manually using some form of mutual exclusion management.

Appendix A

A Sample Program

```
/*
** Copyright (c) 1995-8, 3Dfx Interactive, Inc.
** All Rights Reserved.
**
** This is UNPUBLISHED PROPRIETARY SOURCE CODE of 3Dfx Interactive, Inc.;
** the contents of this file may not be disclosed to third parties, copied
or
** duplicated in any form, in whole or in part, without the prior written
** permission of 3Dfx Interactive, Inc.
**
** RESTRICTED RIGHTS LEGEND:
** Use, duplication or disclosure by the Government is subject to
restrictions
** as set forth in subdivision (c)(1)(ii) of the Rights in Technical Data
** and Computer Software clause at DFARS 252.227-7013, and/or in similar or
** successor clauses in the FAR, DOD or NASA FAR Supplement. Unpublished -
** rights reserved under the Copyright Laws of the United States.
**
** $Id: test05.c,v 1.1 1995/06/30 06:47:04 garymct Exp $
*/
#ifdef __DOS__
#include <conio.h>
#endif
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <glide.h>

typedef struct {
   float x, y;
   float r, g, b, a;
} myVertex;

FxI32 numBoards=0;
GrContext_t win;

void main( void )
{  float color = 255.0;

   puts( "\nTEST05:" );
   puts( "renders a Gouraud-shaded triangle" );
#ifdef __DOS__
   puts( "press a key to continue" );
   getch();
#endif

   grGlideInit();
   grGet(GR_NUM_BOARDS, 1, &numboards) ;

   if ( numBoards==0 )
      grErrorSetCallback( "main: grGet(GR_NUM_BOARDS) returned 0!", FXTRUE )
;


   /* Select SST 0 and open up the hardware */
   grSstSelect( 0 ) ;
   if ( !(win=grSstWinOpen( NULL, GR_RESOLUTION_640x480, GR_REFRESH_60Hz,
                              GR_COLORFORMAT_ABGR, GR_ORIGIN_LOWER_LEFT, 2, 0
)) )
      grErrorSetCallback( "main: grSstWinOpen failed!", FXTRUE );


   /# establish the vertex layout */
   grCoordinateSpace(GR_WINDOW_COORDS);
   grVertexLayout(GR_PARAM_XY, 0, GR_PARAM_ENABLE);
   grVertexLayout(GR_PARAM_RGB, 8, GR_PARAM_ENABLE);
   grVertexLayout(GR_PARAM_A, 20
   while ( 1 ) {
      myVertex vtx1, vtx2, vtx3;

      grBufferClear( 0, 0, GR_WDEPTHVALUE_FARTHEST );
      guColorCombineFunction( GR_COLORCOMBINE_ITRGB );

      vtx1.x = 160;
      vtx1.y = 120;
      vtx1.r = color;
      vtx1.g = 0;
      vtx1.b = 0;
      vtx1.a = 0;
      vtx2.x = 480;
      vtx2.y = 180;
      vtx2.r = 0;
      vtx2.g = color;
      vtx2.b = 0;
      vtx2.a = 128;
      vtx3.x = 320;
      vtx3.y = 360;
      vtx3.r = 0;
      vtx3.g = 0;
      vtx3.b = color;
      vtx3.a = 255;
      grDrawTriangle( &vtx1, &vtx2, &vtx3 );

      grBufferSwap( 1 );
#ifdef __DOS__
      getch();
      break;
#endif

   }
   grGlideShutdown();
}
```

Appendix B. Glide State Constants

This following table shows the Glide constants that define values for modes, functions, and other Glide state variables.

if the Glide type is	*and the argument name is something like*	*then these constants are valid values for the argument*	*and these are the consequences of choosing that value.*
FxU32	*evenOdd* *oddEvenMask*	GR_MIPMAPLEVELMASK_EVEN GR_MIPMAPLEVELMASK_ODD GR_MIPMAPLEVELMASK_BOTH	Selects mipmaps for loading. Even LODs are GR_LOD_LOG2_256, GR_LOD_LOG2_64, GR_LOD_LOG2_16, GR_LOD_LOG2_4, and GR_LOD_LOG2_1. Odd LODs are GR_LOD_LOG2_128, GR_LOD_LOG2_32, GR_LOD_LOG2_8, and GR_LOD_LOG2_2
GrAlphaBlendFnc_t	*rgbSrcFactor* *rgbDestFactor* *alphaSrcFactor* *alphaDestFactor*	GR_BLEND_ZERO GR_BLEND_SRC_ALPHA GR_BLEND_SRC_COLOR GR_BLEND_DST_COLOR GR_BLEND_DST_ALPHA GR_BLEND_ONE GR_BLEND_ONE_MINUS_SRC_ALPHA GR_BLEND_ONE_MINUS_SRC_COLOR GR_BLEND_ONE_MINUS_DST_COLOR GR_BLEND_ONE_MINUS_DST_ALPHA GR_BLEND_RESERVED_8 GR_BLEND_RESERVED_9 GR_BLEND_RESERVED_A GR_BLEND_RESERVED_B GR_BLEND_RESERVED_C GR_BLEND_RESERVED_D GR_BLEND_RESERVED_E GR_BLEND_ALPHA_SATURATE GR_BLEND_PREFOG_COLOR	sets alpha blending factors
GrAspectRatio_t	*aspectRatio*	GR_ASPECT_LOG2_8x1 GR_ASPECT_LOG2_4x1 GR_ASPECT_LOG2_2x1 GR_ASPECT_LOG2_1x1 GR_ASPECT_LOG2_1x2 GR_ASPECT_LOG2_1x4 GR_ASPECT_LOG2_1x8	sets the aspect ratio of the textures in a mipmap
GrBuffer_t	*buffer*	GR_BUFFER_FRONTBUFFER GR_BUFFER_BACKBUFFER GR_BUFFER_AUXBUFFER GR_BUFFER_DEPTHBUFFER GR_BUFFER_ALPHABUFFER GR_BUFFER_TRIPLEBUFFER	
GrChipID_t	*tmu*	GR_TMU0 GR_TMU1 GR_TMU2	Selects the target TMU. The constant names it.
GrChromakeyMode_t	*mode*	GR_CHROMAKEY_DISABLE GR_CHROMAKEY_ENABLE	
GrCmpFnc_t	*func*	GR_CMP_NEVER GR_CMP_LESS GR_CMP_EQUAL GR_CMP_LEQUAL GR_CMP_GREATER GR_CMP_NOTEQUAL GR_CMP_GEQUAL GR_CMP_ALWAYS	
GrColorFormat_t	*cFormat*	GR_COLORFORMAT_ARGB GR_COLORFORMAT_ABGR GR_COLORFORMAT_RGBA GR_COLORFORMAT_BGRA	

if the Glide type is	*and the argument name is something like*	*then these constants are valid values for the argument*	*and these are the consequences of choosing that value*

GrCombineFactor_t	*factor*
	rgbFactor
	alphaFactor
		`GR_COMBINE_FACTOR_ZERO`
		`GR_COMBINE_FACTOR_NONE`
		`GR_COMBINE_FACTOR_LOCAL`
		`GR_COMBINE_FACTOR_OTHER_ALPHA`
		`GR_COMBINE_FACTOR_LOCAL_ALPHA`
		`GR_COMBINE_FACTOR_TEXTURE_ALPHA`
		`GR_COMBINE_FACTOR_DETAIL_FACTOR`
		`GR_COMBINE_FACTOR_LOD_FRACTION`
		`GR_COMBINE_FACTOR_ONE`
		`GR_COMBINE_FACTOR_ONE_MINUS_LOCAL`
		`GR_COMBINE_FACTOR_ONE_MINUS_OTHER_ALPHA`
		`GR_COMBINE_FACTOR_ONE_MINUS_LOCAL_ALPHA`
		`GR_COMBINE_FACTOR_ONE_MINUS_TEXTURE_ALPHA`
		`GR_COMBINE_FACTOR_ONE_MINUS_DETAIL_FACTOR`
		`GR_COMBINE_FACTOR_ONE_MINUS_LOD_FRACTION`
			chooses a combine factor for the color combine, alpha combine, or texture combine units

GrCombineFunction_t	*factor*
	rgbFunction
	alphaFunction
		`GR_COMBINE_FUNCTION_ZERO`
		`GR_COMBINE_FUNCTION_NONE`
		`GR_COMBINE_FUNCTION_LOCAL`
		`GR_COMBINE_FUNCTION_LOCAL_ALPHA`
		`GR_COMBINE_FUNCTION_SCALE_OTHER`
		`GR_COMBINE_FUNCTION_BLEND_OTHER`
		`GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL`
		`GR_COMBINE_FUNCTION_SCALE_OTHER_ADD_LOCAL_ALPHA`
		`GR_COMBINE_FUNCTION_SCALE_OTHER_MINUS_LOCAL`
		`GR_COMBINE_FUNCTION_SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL`
		`GR_COMBINE_FUNCTION_BLEND`
		`GR_COMBINE_FUNCTION_SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL_ALPHA`
		`GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL`
		`GR_COMBINE_FUNCTION_BLEND_LOCAL`
		`GR_COMBINE_FUNCTION_SCALE_MINUS_LOCAL_ADD_LOCAL_ALPHA`
			chooses a combining function for the color combine, alpha combine, or texture combine units

GrCombineLocal_t	*local*	`GR_COMBINE_LOCAL_ITERATED`
		`GR_COMBINE_LOCAL_CONSTANT`
		`GR_COMBINE_LOCAL_NONE`
		`GR_COMBINE_LOCAL_DEPTH`
			chooses a local alpha or RGB source for color, alpha, or texture combine units

GrCombineOther_t	*other*	`GR_COMBINE_OTHER_ITERATED`
		`GR_COMBINE_OTHER_TEXTURE`
		`GR_COMBINE_OTHER_CONSTANT`
		`GR_COMBINE_OTHER_NONE`
			chooses an alpha or RGB source for the “other” value in the color, alpha, or texture combine units

GrCullMode_t	*mode*	`GR_CULL_DISABLE`
		`GR_CULL_NEGATIVE`
		`GR_CULL_POSITIVE`
			Do back-facing polygons have negative or positive area?

GrDepthBufferMode_t	*mode*	`GR_DEPTHBUFFER_DISABLE`
		`GR_DEPTHBUFFER_ZBUFFER`
		`GR_DEPTHBUFFER_WBUFFER`
		`GR_DEPTHBUFFER_ZBUFFER_COMPARE_TO_BIAS`
		`GR_DEPTHBUFFER_WBUFFER_COMPARE_TO_BIAS`
			chooses a depth buffering algorithm

GrDitherMode_t	*mode*	`GR_DITHER_DISABLE`
		`GR_DITHER_2x2`
		`GR_DITHER_4x4`
			Wanna dither?

GrFogMode_t	*mode*	`GR_FOG_DISABLE`
		`GR_FOG_WITH_ TABLE_ON_Q`
		`GR_FOG_WITH_TABLE_ON_FOGCOORD_EXT`
		`GR_FOG_MULT2`
		`GR_FOG_ADD2`
			enables and characterizes fogging

GrLfbWriteMode_t	*mode*	`GR_LFBWRITEMODE_565`
		`GR_LFBWRITEMODE_555`
		`GR_LFBWRITEMODE_1555`
		`GR_LFBWRITEMODE_888`
		`GR_LFBWRITEMODE_8888`
		`GR_LFBWRITEMODE_565_DEPTH`
		`GR_LFBWRITEMODE_555_DEPTH`
		`GR_LFBWRITEMODE_1555_DEPTH`
		`GR_LFBWRITEMODE_DEPTH_DEPTH`
		`GR_LFBWRITEMODE_ALPHA_ALPHA`	

GrLOD_t	*smallLOD*
	largeLOD
	thisLOD
		`GR_LOD_LOG2_256`
		`GR_LOD_LOG2_128`
		`GR_LOD_LOG2_64`
		`GR_LOD_LOG2_32`
		`GR_LOD_LOG2_16`
		`GR_LOD_LOG2_8`
		`GR_LOD_LOG2_4`
		`GR_LOD_LOG2_2`
		`GR_LOD_LOG2_1`
			Specifies the largest dimension of the texture. The aspect ratio determines the smaller dimension.

GrMipMapMode_t	*mipmapMode*
	mode
		`GR_MIPMAP_DISABLE`
		`GR_MIPMAP_NEAREST`
		`GR_MIPMAP_NEAREST_DITHER`
			specifies the kind of mipmapping to perform

GrNCCTable_t	*table*	`GR_NCCTABLE_NCC0` `GR_NCCTABLE_NCC1`	chooses an NCC table for use in decompressing texels

If the Glide type is	*and the argument name is something like*	*then these constants are valid values for the argument*	and these are the consequences of choosing that value
GrOriginLocation_t	*locateOrigin* *origin*	`GR_ORIGIN_UPPER_LEFT` `GR_ORIGIN_LOWER_LEFT`	sets location of origin
GrSmoothingMode_t	*smoothMode*	`GR_SMOOTHING_DISABLE` `GR_SMOOTHING_ENABLE`	enables/disables 24-smoothing filter

GrTexBaseRange_t	*range*	`GR_TEXBASE_256`
		`GR_TEXBASE_128`
		`GR_TEXBASE_64`
		`GR_TEXBASE_32_TO_1`
			Specifies which base register when using more than one. A mipmap can be broken into four fragments. The number in the constant corresponds to the LOD number.

GrTexTable_t	*tableType*
	table
		`GR_TEX_NCC0`
		`GR_TEX_NCC1`
		`GR_TEX_PALETTE`
			Each TMU can have two NCC tables and a palette. Load them one at a time with a general purpose routine.

GrTextureClampMode_t	*sClampMode* *tClampMode*	`GR_TEXTURECLAMP_WRAP` `GR_TEXTURECLAMP_CLAMP`	Clamp or wrap at the edges of a texture?
GrTextureFilterMode_t	*minFilterMode* *magFilterMode*	`GR_TEXTUREFILTER_POINT_SAMPLED` `GR_TEXTUREFILTER_BILINEAR`	chooses minification and magnification filters

GrTextureFormat_t	*format*	`GR_TEXFMT_RGB_332`
		`GR_TEXFMT_YIQ_422`
		`GR_TEXFMT_ALPHA_8`
		`GR_TEXFMT_INTENSITY_8`
		`GR_TEXFMT_ALPHA_INTENSITY_44`
		`GR_TEXFMT_P_8`
		`GR_TEXFMT_ARGB_8332`
		`GR_TEXFMT_AYIQ_8422`
		`GR_TEXFMT_RGB_565`
		`GR_TEXFMT_ARGB_1555`
		`GR_TEXFMT_ARGB_4444`
		`GR_TEXFMT_ ALPHA_INTENSITY_88`
		`GR_TEXFMT_AP_88`
			see **Table** 10.1 for a description of the texture formats

The types below are used in three Glide Utilities Library functions that present higher level views of the texture, color, and alpha combine units.

if the Glide type is	*and the argument name is something like*	*then these constants are valid values for the argument*	*and these are the consequences of choosing that value*

GrAlphaSource_t	*mode*	`GR_ALPHASOURCE_CC_ALPHA`
		`GR_ALPHASOURCE_ITERATED_ALPHA`
		`GR_ALPHASOURCE_TEXTURE_ALPHA`
		`GR_ALPHASOURCE_TEXTURE_ALPHA_TIMES_ITERATED_ALPHA`
			chooses an alpha source for alpha and color combing

GrColorCombineFnc_t	*function*	`GR_COLORCOMBINE_ZERO`
		`GR_COLORCOMBINE_CCRGB`
		`GR_COLORCOMBINE_ITRGB`
		`GR_COLORCOMBINE_ITRGB_DELTA0`
		`GR_COLORCOMBINE_DECAL_TEXTURE`
		`GR_COLORCOMBINE_TEXTURE_TIMES_CCRGB`
		`GR_COLORCOMBINE_TEXTURE_TIMES_ITRGB`
		`GR_COLORCOMBINE_TEXTURE_TIMES_ITRGB_DELTA0`
		`GR_COLORCOMBINE_TEXTURE_TIMES_ITRGB_ADD_ALPHA`
		`GR_COLORCOMBINE_TEXTURE_TIMES_ALPHA`
		`GR_COLORCOMBINE_TEXTURE_TIMES_ALPHA_ADD_ITRGB`
		`GR_COLORCOMBINE_TEXTURE_ADD_ITRGB`
		`GR_COLORCOMBINE_TEXTURE_SUB_ITRGB`
		`GR_COLORCOMBINE_CCRGB_BLEND_ITRGB_ON_TEXALPHA`
		`GR_COLORCOMBINE_DIFF_SPEC_A`
		`GR_COLORCOMBINE_DIFF_SPEC_B`
		`GR_COLORCOMBINE_ONE`
			chooses a color combining function

GrTextureCombineFnc_t	*function*	`GR_TEXTURECOMBINE_ZERO`
		`GR_TEXTURECOMBINE_DECAL`
		`GR_TEXTURECOMBINE_OTHER`
		`GR_TEXTURECOMBINE_ADD`
		`GR_TEXTURECOMBINE_MULTIPLY`
		`GR_TEXTURECOMBINE_SUBTRACT`
		`GR_TEXTURECOMBINE_DETAIL`
		`GR_TEXTURECOMBINE_DETAIL_OTHER`
		`GR_TEXTURECOMBINE_TRILINEAR_ODD`
		`GR_TEXTURECOMBINE_TRILINEAR_EVEN`
		`GR_TEXTURECOMBINE_ONE`
			chooses a texture combining function

Glossary

aliasing	Rendering artifacts that occur when a continuous function is discretely sampled or sub-sampled. Two common types of aliasing are polygonal aliasing and texture aliasing. Polygonal aliasing is a rendering artifact that occurs when rasterization applies color to a pixel without considering how much of the pixel is covered by the triangle. Along the edges of the triangle, only a portion of the pixel is likely to be covered by the triangle. An aliased triangle will have jagged edges. Texture aliasing is a rendering artifact that occurs when a texture map is not sampled frequently enough or when the texel area covered by a pixel is not accounted for. *See* ***anti-aliasing***.
alpha	The A in an RGBA color. The alpha component is never displayed. It is a multiplier used to describe transparency and controls the blending of overlapping colors. *See* ***blending***.
ambient light	One of the components of a lighting model. Ambient light seems to come from all directions rather than from a specific source. Back lighting in a room is an example. It scatters in all directions after striking a surface, as does diffuse light. *See* ***diffuse***, ***emitted***, *and* ***specular light***.
animation	Generating and displaying a scene as the viewpoint and/or objects change position to give the illusion of motion.
anti-aliasing	Techniques for eliminating aliasing. For polygonal aliasing, a rendering technique that accounts for fractional coverage of a pixel when assigning it a color, thereby reducing or eliminating the jagged edges that characterize an aliased rendering. For texture aliasing, a rendering technique that accounts for the areas of texels covered by a pixel. *See* ***aliasing***.
API	Application program interface.
ASIC	Application-specific integrated circuit.
back face culling	The process of eliminating back facing triangles. A triangle has two sides, front and back, with only one side visible at a time. The sign of the area of the triangle determines which side is visible and can be used to eliminate back facing triangles before they are rendered.
bilinear filtering	A technique for choosing the texel color to apply to a pixel during texture mapping. The weighted average of the four texels nearest the pixel center is used.
blending	When two triangles overlap in screen space, a decision must be made about the color of the pixels in the overlapping area. Blending is a

	technique for reducing the two colors to one, usually as a linear interpolation of the two candidates.
chroma-key	A technique for removing pixels of a specific color, used to implement a “blue screen”.
clamp	Forcing a value to lie within a specified range of values.
clipping	Elimination of those portions of a scene that are outside the clipping rectangle defined by calling **grClipWindow()**.
depth bias	A constant that is added to the calculated depth of a pixel.
depth buffer	One possible use of the auxiliary buffer. It stores a depth value for each pixel. Subsequent pixels can be accepted or discarded based on a depth test.
diffuse light	One of the components of a lighting model. Diffuse light comes from a single source, but it is scattered equally in all directions when it strikes a surface. *See* ***ambient****,* ***emitted****, and* ***specular light****.*
dithering	A technique for increasing the perceived range of colors in an image by applying a pattern to surrounding pixels to modify their color values. When viewed from a distance, these colors appear to blend into an intermediate color that can’t be represented directly. Dithering is similar to the half-toning used in black and white publications to produce shades of gray.
double buffering	Using two color buffers: a scene is rendered in one buffer while the previously rendered scene in the other buffer is displayed. When the rendering is complete, the two buffers are swapped and the rendering of the next scene can begin in the buffer that is no longer being displayed. *See* ***single buffering****,* ***triple buffering****, and* ***frame buffer****.*
EDO DRAM	Extended-data-out dynamic random access memory.
emitted light	One of the components of a lighting model. Emitted light comes from an object and is unaffected by other light sources. Lamps, headlights, and candles are examples. *See* ***ambient****,* ***diffuse****, and* ***specular light****.*
FBI	Frame buffer interface.
FIFO	First in, first out. A list data structure in which new entries are added at the end of the list.
flat shading	Coloring a triangle with a single, constant color. *See* ***Gouraud shading****.*
fog	A rendering technique that simulates atmospheric effects such as haze, fog, and smog by fading object colors to a background color based on distance from the viewer.
frame buffer	The memory used to hold pixels. In an SST system, the frame buffer is accessed by the FBI chip and can be used for up to three color buffers. In single or double buffer mode, the auxiliary buffer can optionally be used as an alpha buffer or a depth buffer.

Gouraud shading	Colors are assigned to the vertices of a triangle and linearly interpolated across the triangle to produce a smooth variation in color. Also called *smooth shading*. *See* ***flat shading***.
homogeneous coordinates	(x, y, z, w). The w coordinate is a scaled positive depth value used during perspective projection, perspective texture mapping, and depth buffering. Some graphics systems do not use homogeneous coordinates; in these instances the z depth value can be used in lieu of the w coordinate, assuming that the z value is positively increasing into the screen.
LOD	Level of detail. *See* ***mipmap***.
magnification	If a texture-mapped screen pixel is smaller than a texel, magnification techniques are used. *See* ***mipmap*** *and* ***minification***.
minification	If a texture-mapped screen pixel is larger than a texel, minification techniques are used. *See* ***mipmap*** *and* ***magnification***.
mipmap	A pyramidal organization of gradually smaller, filtered sub-textures or an individual texture map within the set, that is used for anti-aliased texture mapping.
PCI system bus	The bus in a PC that connects the host CPU and the peripheral devices, including the SST-1 board.
pixel	Picture element.
point sampling	In the context of SST-1 texture mapping, choosing the texel nearest the pixel center.
rendering	The process of converting triangles into bits in the frame buffer, applying texture mapping, alpha blending, depth buffering, etc. Rendering is what SST-1 does.
RGBA	Red, green, blue, and alpha.
single buffering	Rendering into the color buffer as it is being displayed.
specular light	One of the components of a lighting model. Specular light comes from a specific direction and bounces off surfaces in a preferred direction as well. It models the shininess of a surface. *See* ***ambient***, ***diffuse***, *and* ***emitted light***.
subpixel correction	Adjusting the vertex parameter values (x, y, z, w, s, t, *red*, *green*, *blue*, and *alpha*) to lie at the center of the pixel rather than somewhere else. The result is very accurate rendering.
texel	Texture element.
texture	A one- or two-dimensional image that is used to modify the color of a triangle and add realism to the scene. You might map a brick texture onto a set of triangles that represents a wall, for example.
texture coordinates	(s, t). Texture coordinates can be specified over any range of values. However, the SST-1 hardware expects texture coordinates in the range $[-2^{16}..2^{16}-1]$ where [0..256] represents one replication of a texture map.

texture mapping	The process of applying a texture to a triangle.
texture memory	Memory used for storing textures. On an SST graphics system, this memory is part of TMU.
TMU	Texture Mapping Unit.
triangle	The SST-1 system's rendering primitive.
trilinear filtering	A technique for blending texels between two levels of detail to avoid mipmap banding.
triple buffering	One possible use of the auxiliary buffer. Three drawing buffers are in use, one being displayed, one waiting to be displayed, and one being rendered into.
vertex	One of the corners of a triangle. It has *x* and *y* coordinates and a set of attributes: an RGBA color, a *z* value indicating depth, *s* and *t* coordinates for texture mapping, and a *w* coordinate for perspective correction.

Index

Bold face page numbers indicate an example of use.