Glide 2.2 Programming Guide

Programming the 3Dfx Interactive Glide Rasterization Library 2.2

Document Release 014
08 March 1997

3Dfx Interactive, Inc.
4435 Fortran Avenue
San Jose, CA 94134

Table of Contents

List of Figures

List of Tables

List of Examples

Chapter 1 An Introduction to Glide

Voodoo Graphics is the first video subsystem that enables personal computers and low cost video game platforms to host true 3D entertainment applications. Optimized for real−time texture−mapped 3D images, the Voodoo 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.

real−time, true−perspective, texture−mapped rendering with lighting support at low cost

simplified software porting: only the inner rasterization loop is affected

lowest cost solution designed expressly for use in the entertainment markets

patent pending techniques to reduce texture memory requirements

The Glide Rasterization Library is a set of low level rendering functions that serve as a software ˆmicro−layer˜ to the Voodoo Graphics family of 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 Voodoo 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 Voodoo Graphics hardware.

By abstracting the low level details of interfacing with the Voodoo 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 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 Voodoo Graphics hardware. Other components of the SDK include the Game Controller Interface (GCI) Library and the Texture Utility Software (TexUS).

Glide is *not* a full featured graphics API such as OpenGL , PHIGS, or the Autodesk CDK : it does not provide high level 3D graphics operations such as transformations, display list management, or light source shading. Glide specifically implements only those operations that are natively supported by the Voodoo Graphics hardware. In general, Glide does not implement any functions that do not directly access a Voodoo Graphics subsystem´s memory or registers.

The Glide Utility Library contains utility routines 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.

The Glide library can be linked with an application with or without debugging aids. The debug version has error checking and parameter validation, which may cause performance degradation. When an application is initially developed and debugged it should use the debugging version of Glide. After development is complete the release build of Glide is employed for optimum performance.

Voodoo Graphics

The Voodoo Graphics subsystem 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. 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 Voodoo Graphics subsystem can utilize scan−line interleaving to achieve the highest possible rendering performance.

Glide and the Voodoo 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 ˆnarrow channel˜ **YAB** compression scheme that maps 24−bit RGB values to a 8−bit **YAB** format with little loss in precision.

Depth buffering . Voodoo Graphics supports 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 Voodoo Graphics subsystem allows a floating point representation of the $1/w$ parameter to be used as the depth component.

Figure 1.1 Voodoo Graphics system configurations.

The Pixelfx chip interfaces with the host computer, the linear frame buffer, and the display monitor, and implements basic 3D primitives including Gouraud shading, alpha blending, depth buffering, dithering, and fog. The TMU (located on the Texelfx chip) implements texture mapping, including 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, full−speed, detail texture mapping, projected texture mapping, or trilinear filtering.

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

(d) For the highest possible rendering performance, multiple Voodoo Graphics subsystems can be chained together utilizing scan−line 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 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 Voodoo Graphics 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 . Voodoo Graphics 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 dither ing: Numeric operations are performed on 24−bit colors within the Voodoo 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 Voodoo 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 unit , and the special effects unit. The basic architecture is illustrated in .

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 pixel source is one of four things: a texture value , an iterated RGB A 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 effect s unit further modifies the pixel with alpha and depth testing , fog , and alpha blending operations. The final 24−bit color value is then dither ed to 16 bits and written to the frame buffer.

About This Manual

The *Glide 2.2 Programming Guide* attempts to introduce a knowledgeable graphics programmer to the capabilities of the Voodoo Graphics subsystem 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 function. The audience for this manual is the application programmer who just took delivery on a Voodoo Graphics subsystem 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 , , describes data types, data formats, and programming model used in Glide and the Voodoo Graphics subsystem.

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

Chapter , , 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 , , describes the functions that control the Voodoo Graphics 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 , , describes the various ways to utilize the alpha channel: alpha blending, alpha buffering, and alpha testing.

Chapter , , presents the two flavors of depth buffers.

Chapter , , describes other special rendering effects that can be produced in the pixel pipeline: atmospheric effects like fog, haze, and smoke; transparent objects implemented with chroma−keying; stippling; color dithering; alpha masking.

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

Chapter , , describes the Glide functions that provide a path for reading and writing the frame buffer directly.

Chapter , , and Chapter , *Glide Utilities* , 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 Voodoo Graphics hardware.

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 comprehensive *Index* .

Other Documentation

Available from 3Dfx Interactive, Inc. :

Glide 2.2 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, ˆPHIGS+ Functional Description − Revision 3.0˜. *Computer Graphics* , 22(3), p. 125−218

SUTH74 Sutherland, I. E. and G. W. Hodgman, ˆReentrant 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., ˆPyramidal 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

Chapter 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 *GrVertex* structure that holds the coordinates and parameters that define a vertex

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 **grDrawPolygon()** are both parts of the **grDraw** family. Glide uses the mixed caps convention for function names. When function names appear in the text of this manual, they will be shown in bold face type. Actual function names end with `()´; function family names do not.

The internal name for the Voodoo Graphics subsystem is ˆSST−1˜or ˆSST˜. Some function names, type definitions, and constants within Glide reflect this internal name, which is easier to type than Voodoo Graphics. For example, **grSstOpen()** 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 will be shown in Courier font when they appear in the text of this manual.

C specifications for functions and data types will be 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* will be italicized in both the C specifications and the text.

Code segments use Courier font.

The State Machine Model

Glide is state based: rendering ˆmodes˜ 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 will be 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 ˆcurrent 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 ˆcurrent 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.

[redacted]

[redacted]

grGlideGetState() makes a copy of the current state of the current Voodoo Graphics subsystem in a *GrState* structure *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.

Specifying Vertices

Voodoo Graphics 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 in the *GrVertex* data structure, shown below and defined in `glide.h`. Up to ten parameters can be used to specify a point:

the geometric coordinates (x, y, z, w) where x and y indicate a screen location, z indicates depth, and w is the homogeneous coordinate

the color components (r, g, b, a)

the texture coordinates (s, t)

Note that the *GrVertex* structure has a spot for z, but actually uses its reciprocal (ooz, for ˆone over z˜). Similarly, $1/w$ is stored in the variable oow. And, s/w and t/w are stored in the structure (as sow and tow) rather than s and t, because the scaled values are the ones actually used by the Voodoo Graphics system. These values need to be computed only once for each vertex, regardless of how many triangles include the vertex.

The *GrVertex* structure also includes a small array of *GrTmuVertex* data structures, one for each TMU present in the system, and each of the array elements contains private oow, sow, and tow variables. Each TMU and the Pixel*fx* chip each have their own copy of $1/w$, s/w, and t/w. Normally, they will all be the same. However, projected textures have a different w value than non−projected textures. Projected textures iterate q/w where w is the homogeneous distance from the eye and q is the homogeneous distance from the projected source.

[redacted]

[redacted]

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. lists some typical rendering operations and the vertex parameters they use.

Table 2.1 Vertex parameter requirements depend on the rendering function being performed.

The x and y coordinates must be specified for every vertex, regardless of the rendering function being performed. The other parameters stored in the GrVertex structure are optional and need to be supplied only if required for the desired computation. The table below details the values required by the rendering functions implemented by Glide and the Voodoo Graphics hardware.

rendering function	*required variables*	*expected values*	*see Chapter*
all vertices, all rendering functions	x, y	˜2048 to +2047	*4*
Gouraud shading	r, g, b	0 to 255.0	*5*
alpha blending/testing	a	0 to 255.0	*6*

non−projected texture mapping	*tmuvtx[0] .oow*, *tmuvtx[0] .sow*, *tmuvtx[0] .tow*	$1/w$ where w is in the range [1..65535] s/w where s is in the range [0..255.0] t/w where t is in the range [0..255.0]	*9*
projected texture mapping	*tmuvtx[0] .oow*, *tmuvtx[0] .sow*, *tmuvtx[0] .tow* *tmuvtx[1] .oow*, *tmuvtx[1] .sow*, *tmuvtx[1] .tow*	$1/w$ where $1/w$ is in the range [˜4096..61439] s/w where *s/w* is in the range [˜32768..32767] t/w where *t/w* is in the range [˜32768..32767] q/w where q/w is in the range [˜4096..61439] s/w where *s/w* is in the range [˜32768..32767] t/w where *t/w* is in the range [˜32768..32767]	*9*
linear z buffering	*ooz*	$1/z$ where $1/z$ is in the range [0, 65535]	
w buffering	*oow*	$1/w$ where w is in the range [1..65535]	
fog with iterated alpha	*a*	[0..255.0]	*8*
fog with iterated z	*ooz*	$1/z$ where $1/z$ is in the range [0, 65535]	*8*
fog with table	*oow*	$1/w$ where w is in the range [1..65535]	*8*

Numerical Data

The Voodoo Graphics hardware can accept vertex data in either fixed point or floating point formats. However, Glide provides only a floating point interface, since RISC and Pentium processors are optimized for floating point calculations. If you are porting a fixed point application to the Voodoo Graphics system, plan to convert all your data to floating point representation as part of the porting process.

The *GrVertex* structure contains single−precision, IEEE 754 32−bit floating point values.

Geometric Coordinates

The *x and y coordinates* are specified in pixel units in the range [˜2048..2047]. The pixel coordinate (0.5, 0.5) represents the exact center of the first visible pixel on the screen.

The *ooz coordinate* should be assigned a value that is linear in screen space. That is, it should be a linear function of $1/w$ that can be scaled and translated such that it increases or decreases with distance from the viewer. The valid range for *ooz* values is [0..65535]. To minimize *z* aliasing this range should be mapped to the smallest possible range of eye coordinates. For example, if *w* eye coordinates are within the range [2..15] and $1/w$ is in the range [1/2..1/15] then the mapping would be approximately

$$1/z = 151214.6/w ˜ 10080.9$$

where *w* is eye *w* and *ooz* is the value iterated in the Voodoo Graphics subsystem.

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. The range of *w* is [1..65535].

Glide and Voodoo Graphics actually use the reciprocal of the homogeneous coordinate, $1/w$. The valid range for $1/w$ is [˜4096..61439]. Normally, the homogeneous coordinate is clipped to a positive range of [1, *far*] and so its reciprocal is in the range [1..1/*far*]. Negative values should be avoided.

Each TMU and the Pixel*fx* chip each have their own $1/w$. Normally, the values in all the chips will be the same. However, projected textures have a different *w* value than non–projected textures. Projected textures iterate q/w where *w* is the homogeneous distance from the eye and *q* is the homogeneous distance from the projected source. In this case, q/w has a valid range of [˜–4096..61439].

The $1/w$ value in Pixel*fx* is used only for fog calculations and *w* buffering, and is not used for texture mapping. It can be scaled differently than the $1/w$ values sent to the TMUs. The fog table spans a range in $1/w$ from [1/65535..1]. If *w* buffering is enabled, the *w* buffer spans a range in $1/w$ from [1/65528..1]. Therefore, scale the $1/w$ value in Pixel*fx* such that the range [1/65535..1] encompasses all that is interesting in the scene.

Colors

The *color components* are in the range [0..255] where 0 is black and 255 is maximum intensity. Colors should be clamped to this range.

Glide supports four different color byte orderings: RGBA, ARGB, BGRA, and ABGR. Color byte ordering determines how linear frame buffer writes and color arguments passed to the constant color functions (**see Chapter 5**) are interpreted. Color ordering is established when Glide and the Voodoo Graphics system are initialized (see Chapter).

When the terms ˆRGB˜ and ˆRGBA˜ appear in this manual, they typically refer to any color system that represents red, green, blue, and optionally, alpha, as separate components, regardless of the byte order or component width. The exceptions will be clearly recognizable as discussions about specific color resolution and format.

Texture Coordinates

Glide uses texture coordinates in the range [˘32768..32767] and refers to them as (s,t) pairs, similar to the naming convention of OpenGL. A texture contains texels with (s,t) coordinates in the range [0..255.0]; the texture may be replicated many times to cover a surface by mapping the texture coordinates modulo 256 to a texel in the texture. The Voodoo Graphics subsystem iterates s/w and t/w, so s and t must be divided by w (or multiplied by oow) before storing them in the *GrVertex* structure.

The w term iterated by the SST−1 is actually $1/w$ or the reciprocal of the homogeneous coordinate. The valid range for $1/w$ is [˘4096..61439]. Normally, the homogeneous coordinate is clipped to a positive range of [$1..far$] and so its reciprocal is in the range [$1..1/far$]. Negative values should be avoided. Each TMU has its own s, t, and w values. Normally, they will be the same as the w in the Pixel*fx*. However, in certain cases they will be different. For example, projected textures have a different w value than non−projected textures. Projected textures iterate q/w where w is the homogeneous distance from the eye and q is the homogeneous distance from the projected source. In this case, q/w has a valid range of [˘4096..61439].

Mipmapping [WILL83] is a method of organizing several pre−filtered texture maps into a single logical entity used for anti−aliased texture mapping. The term mipmap is sometimes used to describe a pyramidal organization of gradually smaller, filtered sub−textures or an individual texture map within such an organization. Glide adopts the original convention that defines the term mipmap to mean the entire group of textures that comprise a single pyramidal data structure. Individual textures within a mipmap are referred to as mipmap *levels* .

Chapter 3 Getting Started

In This Chapter

You will learn how to:

initialize Glide

configure and initialize the hardware

manage multiple Voodoo Graphics subsystems

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 Voodoo Graphics subsystem. Two routines, **grSstQueryHardware()** and **grSstQueryBoards()** detect the presence of Voodoo Graphics subsystems. Three functions, **grGlideInit()**, **grSstSelect()**, and **grSstOpen()**, initialize Glide and the hardware and must be called, in the order listed, before calling any other Glide routines (except **grSstQueryHardware()** and **grSstQueryBoards()**). Failing to do this will cause the system to operate in an undefined (and, most likely, undesirable) state.

[redacted]

[redacted]

grSstQueryBoards() determines the number of installed Voodoo Graphics subsystems and stores this number in *hwConfig−>num_sst*. No other information is stored in the structure at this time; **grSstQueryHardware()** can be called after **grGlideInit()** to fill in the rest of the structure. **grSstQueryBoards()** is the only Glide routine that can be called before **grGlideInit()**; it does not change the state of any hardware, nor does it render any graphics.

grSstQueryHardware() detects the presence of one or more Voodoo Graphics subsystems and determines how they are configured. It should be called immediately after **grGlideInit()** but before any other Glide functions.

grSstQueryHardware() returns a Boolean value: `FXTRUE` indicates that at least one Voodoo Graphics subsystem was found. The argument, *hwConfig*, is a pointer to a structure that will be filled in with information about the number and configuration of the Voodoo Graphics subsystems it found.

Note that when two Voodoo Graphics subsystems are configured as a single scan−line interleaved system they are viewed by Glide and an application as a single subsystem.

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

[redacted]

The next function called to initialize the system is **grSstSelect()**, which makes a specific Voodoo Graphics subsystem ˆcurrent˜. It must be called after **grSstQueryHardware()** and **grGlideInit()** but before **grSstOpen()**.

[redacted]

The argument is the ordinal number of the subsystem that will be made active and must be in the range [0..*hwconfig.num_sst*] where *hwConfig* is the structure that holds the system configuration information returned by the preceding call to **grSstQueryHardware()**. If *whichSST* is outside the proper range of values and the debugging version of Glide is used, a run−time error will be 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, **grSstOpen()**, initializes the currently active Voodoo 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 0.

grSstOpen() should be called once per installed Voodoo Graphics subsystem (note that scan−line interleaved subsystems are treated as a single Voodoo Graphics subsystem) and must be executed *after* **grGlideInit()**, **grSstQueryHardware()** and **grSstSelect()**. It returns FXTRUE if the initialization was successful and FXFALSE otherwise.

[redacted]

The arguments to **grSstOpen()** configure the frame buffer. The screen resolution and refresh rate are specified in the first 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.

The third 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 ˆbyte 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 3.1. 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 3.1 Frame buffer color formats.
Glide supports four different color byte orderings: RGBA, ARGB, BGRA, and ABGR. Color byte ordering determines how linear frame buffer writes and color arguments passed to the constant color functions 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 fourth parameter to **grSstOpen()** 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 (a la IBM VGA). GR_ORIGIN_LOWER_LEFT places the screen space origin at the lower left corner with positive *y* going up (a la SGI GL). Figure 3.1 shows the two possibilities for locating the origin.

Figure 3.1 Locating the origin.
The Voodoo Graphics hardware allows the origin to be in the upper left or lower left corner of the screen. The choice of coordinate system must be made when first initializing Glide and a Voodoo Graphics subsystem by passing the appropriate parameter to **grSstOpen()**.

The Voodoo Graphics hardware operates on 32−bit ARGB color values internally. However, these values are eventually dithered to 16−bit RGB (5:6:5) by the hardware. Hardware dithering provides good display quality while at the same time reducing memory consumption by a third, allowing for less expensive hardware implementations. The fifth argument to **grSstOpen()**, *smoothMode*, enables or disables an optional 24−bit smoothing filter that can be applied during video refresh. Enabling smoothing reduces dithering artifacts but may result in a slightly blurrier image.

The final argument to **grSstOpen()**, *numBuffers*, specifies double or triple buffering and is an integer value, either 2 or 3. If there is not enough memory to support the desired resolution (e.g. 800×600 triple buffered on a 2MB system) then an error will occur.

Example . The Glide initialization sequence.
This code fragment demonstrates the four Glide functions that must be called, in order, to properly initialize both the software and the hardware subsystems. The parameters to **grSstOpen()** *establish a double buffered frame buffer with 640 ×480 screen resolution and a 60Hz refresh rate. Colors are stored as RGBA, smoothing techniques will be applied to reduce dither ing artifacts, and the origin is in the lower left corner.*

```
GrHwConfiguration hwconfig;

grGlideInit(void);
if (grSstQueryHardware(hwconfig)) {
   grSstSelect(0);
                  grSstOpen(  GR_RESOLUTION_640x480, GR_REFRESH_60HZ,
               GR_COLORFORMAT_RGBA, GR_ORIGIN_LOWER_LEFT,
               GR_SMOOTHING_ENABLE, 2);
};
else printf(ˆERROR: no Voodoo Graphics!\n˜);
```

Driving Multiple Systems

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

Selecting Voodoo Graphics Units

At any given moment, only a single Voodoo Graphics subsystem 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 Voodoo Graphics subsystem. Note that the global Glide state is bound to each Voodoo Graphics independently. So, to set the constant color in each Voodoo Graphics 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 . Setting a state variable in all Voodoo Graphics subsystems.
Each Voodoo Graphics subsystem has its own version of the Glide state variables, including a constant color value that will be used to clear the screen. The constant color is zero by default. The code fragment below cycles through all the Voodoo Graphics units found by a previous call to **grSstQueryHardware()***, setting the constant color to black.*

```
GrHwConfiguration hwconfig;

for ( i = 0; i < hwconfig.num_sst; i++ )
{
   grSstSelect( i );
   grConstantColorValue( ~0 ); /* only affects SST ˆi˜ */
}
```

Opening Multiple Voodoo Graphics Units

grSstOpen() must be called once for each Voodoo Graphics subsystem that will be used. Note that two Voodoo Graphics subsystems linked together in a scan−line interleaving configuration are treated in software as a single Voodoo Graphics subsystem.

Scan−line Interleaved Voodoo Graphics Units

Two Voodoo Graphics subsystems can be wired together in a configuration known as *scan−line interleaving* , which effectively doubles rasterization performance. From an application´s perspective, two Voodoo Graphics subsystems in a scan−line interleaved configuration are treated as if a single Voodoo Graphics subsystem is installed in the system, including during Voodoo Graphics selection, initialization, state management, texture download, etc.

Shutting Down

After an application has completed using Glide and the Voodoo 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 Voodoo Graphics subsystems previously opened with **grSstOpen()**. It should be called only when an application is finished using Glide, and should not be executed unless **grGlideInit()** and **grSstOpen()** have already been called.

shows a minimal Glide program: it executes the four function calls that initialize the Voodoo Graphics subsystem and then terminates.

Example . A minimal Glide program.

The complete program below includes the Glide initialization and termination procedure and nothing else.

```
#include <glide.h>

GrHwConfiguration hw;

void main(void)
{
   grGlideInit(void);
   if (! grSstQueryHardware(hw)) printf(ˆERROR: no Voodoo Graphics!\n˜);
   grSstSelect(0);
   grSstOpen( GR_RESOLUTION_640x480, GR_REFRESH_60HZ, GR_COLORFORMAT_RGBA
              GR_ORIGIN_LOWER_LEFT, GR_SMOOTHING_ENABLE, 2);
   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 ˆframe 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.

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 Voodoo Graphics 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), it is assumed that the auxiliary buffer will be used for depth buffering. Similarly, **grSstOpen()** 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 3.2 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	[redacted]	[redacted]	[redacted]
4 Mbytes	[redacted]	[redacted]	[redacted]

Logical Layout of the Linear Frame Buffer

The frame buffer is logically organized as a 1024×1024 array of 16 or 32−bit values, regardless of the amount of memory installed on the board, and is shown in . *Scan−line size is independent of screen resolution and is always 1024 pixels wide* . The data format within the frame buffer is programmable and is described in detail in Chapter .

Figure 3.2 Logical layout of the linear frame buffer.
The frame buffer is logically organized as a 1024 ×1024 array 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 buffer ing). 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, `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). 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.

[redacted]

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 **grLfbBypassMode()** is enabled (see Chapter). 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 will be 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 in the Voodoo Graphics subsystem after *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.

[redacted]

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 Voodoo 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 Voodoo Graphics subsystem using the **grBufferNumPending()** function to determine the number of buffers waiting to be viewed, although this is generally not necessary.

[redacted]

grBufferNumPending() returns the number of queued buffer swap requests. 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, **grBufferNumPending()** 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 **grSstVRetraceOn()** and **grSstVideoLine()** functions. **grSstVRetraceOn()** returns FXTRUE if the monitor is in vertical retrace and FXFALSE otherwise.

[redacted]

grSstVideoLine() returns the current line number of the display beam. This number is 0 during vertical retrace and increases as the display beam progresses down the screen. There are a small number of video lines that are not displayed at the top of the screen: the vertical backporch. Thus, **grSstVideoLine()** returns a small positive number when the display beam is at the top of the screen; as the beam goes off the bottom of the screen, the line number may exceed the number returned by **grSstScreenHeight()**.

[redacted]

The Glide 2.1 release was the first release to include **grSstVideoLine()**. Earlier versions used **grSstVRetraceTicks()**, now obsolete.

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.

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 Voodoo Graphics 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, therefore the performance cost of clearing the buffer is 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 will be 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 constants GR_ZDEPTHVALUE_NEAREST, GR_ZDEPTHVALUE_FARTHEST, GR_WDEPTHVALUE_NEAREST, GR_WDEPTHVALUE_FARTHEST, or a direct representation of a value in the depth buffer. See Chapter for more details.

[redacted]

Any buffers that are enabled are automatically and simultaneously cleared by **grBufferClear()**. For example, if depth buffering is enabled (with **grDepthBufferMode()**, described in Chapter), the depth buffer will be cleared to *depth*. If alpha buffering is enabled (with **grAlphaBlendFunction()**, described in Chapter 6), the alpha buffer will be cleared to *alpha*. And if writes to the display buffer are enabled (with **grColorMask()**, described in Chapter 5), then it will be 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.

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.

Chapter 4

Chapter 5 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

cull back–facing polygons from the scene

draw an anti–aliased point, line, triangle, or convex polygon

The *GrVertex* Structure

The *GrVertex* structure, first presented in Chapter 2, collects together all the parameters that define a vertex. In this chapter, only the *x* and *y* coordinates will be discussed; the other parameters are called into play in later chapters.

[redacted]

[redacted]

The *x* and *y* coordinates are 32–bit floating point values that represent the position of the vertex in screen space. While the Voodoo Graphics hardware renders only triangles, Glide provides functions to draw points, lines, and polygons as well as triangles.

When a point, line, triangle, or polygon is rendered, its appearance will reflect the current state of the rendering pipeline. That is, if texture mapping is enabled, then the point, line, triangle, or polygon will be texture mapped. Similarly, alpha blending, fogging, color, and lighting effects, chroma–keying, and other special effects will contribute to the appearance of any and all geometric shapes drawn while they are enabled.

Clipping

The Voodoo Graphics hardware supports per–pixel clipping to an arbitrary rectangle 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()**, described in the last chapter, will clear.

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.

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 will be 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* .

[redacted]

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 Voodoo Graphics subsystem´s 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 Voodoo Graphics subsystem´s clip window should be used to prevent stray pixels that appear from imprecise geometric clipping. Note that if **grLfbBypassMode()** is enabled, 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 with vertices a, b, and c to the screen.

[redacted]

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 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 ˆright edge˜ of triangles **A**, **H**, **G**, and **F**, and hence outside. It is part of the ˆtop 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 outside the triangle. Only in triangle **C** does the vertex lie on two ˆinside˜ edges and thus lies inside the triangle.

Clipping a Triangle

Recall from the clipping window discussion above that the hardware clipping implemented by Voodoo Graphics is at the end of the rendering pipeline: a pixel will incur all the rendering cost only to be discarded just before being written to the frame buffer. An alternative solution is to use host bandwidth to clip the triangle and process only the pixels that will be displayed. The Glide Utility Library provides just such a function. **guDrawTriangleWithClip()** uses Sutherland−Hodgman clipping [SUTH74] to clip the triangle to the rectangle specified by **grClipWindow()** and then draws the resulting polygon.

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 inside the triangle while dotted edges are outside. The top row of diagrams are drawn with the origin in the lower left corner. The bottom row represent the other possibility: the origin is in the upper left corner. The two pairs of diagrams are mirror images of each other.*

Points

The Glide function **grDrawPoint()** renders a single point to the screen. The point will be 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()** send three vertices per point to the hardware and iterates along three edges; only one linear frame buffer write per point is required.)

Example . 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 point 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.*

```
int n;
GrVertex p;

/* 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.

[redacted]

Convex Polygons

A polygon is a planar area enclosed by a closed loop of line segments, specified by their endpoints. While the Voodoo Graphics 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*.)

shows some examples of both valid and invalid polygons.

Figure 4.3 Polygons.

The convex polygons rendered by Glide are assumed to be planar in coordinate space. Two polygon rendering routines (**grDrawPlanarPolygon()** and **grDrawPlanarPolygonVertexList()**) requires that the rendering parameters (*r*, *g*, *b*, *a*, *ooz*, *oow*, *sow*, *tow*) be planar as well. None of the polygon rendering functions do any geometric clipping.

[redacted]

grDrawPlanarPolygon() and **grDrawPolygon()** both render a convex polygon with *nVerts* vertices. The second argument, *ilist*, is an array of indices into the list of vertices provided in the third argument. This level of indirection in specifying vertices is useful if you need to pre–process the list to do geometric clipping or hidden surface removal. The preprocessor can create the *ilist* for you rather than copying selected vertices to a new list.

grDrawPlanarPolygon() assumes that the vertex parameters for the polygon are planar. Parameter gradients will be calculated only once for the entire polygon, thus reduce the number of calculations significantly.

Another pair polygon rendering functions defined in Glide, **grDrawPlanarPolygonVertexList()** and **grDrawPolygonVertexList()**, are functionally equivalent to **grDrawPlanarPolygon ()** and **grDrawPolygon()** respectively. The difference between the two pairs of routines is the way the vertices are specified.

There is no level of indirection in **grDrawPlanarPolygonVertexList()** and **grDrawPolygonVertexList()**. The i^{th} vertex of the polygon passed to these routines is *vlist[i]*, assuming that $0 \le i < nVerts$, whereas the i^{th} vertex of a polygon passed to **grDrawPlanarPolygon ()** or **grDrawPolygon()** is *vlist[ilist[i]]*.

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 rendering primitive of the Voodoo Graphics hardware. shows a pair of triangles whose vertices have been labeled according to the rule given above.

Figure 4.4 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.

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 oriented triangle 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 .

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

Table 4.1 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 will be*
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 4.5(a). Anti−aliasing techniques reduce the jaggedness, as shown in Figure 4.5(b), by partially coloring neighboring pixels to simulate partial pixel coverage.

Figure 4.5 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–alias ed lines. These lines are examples of the general concepts; if you replicate this drawing on a Voodoo Graphics screen, the results may be different in detail.

Figure 4.6 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 4.6 Pixel coverage and lines.

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 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 will be 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 will be multiplied by the pixel coverage to obtain the final alpha value used for alpha blending.

Call a ***grAADraw*** *or* ***guAADraw*** *function* . The six functions are as shown below.

[redacted]

grAADrawPoint() renders the point as four pixels, each blended according to the computed pixel coverage.

Lines drawn with **grAADrawLine()** will be somewhat ˆfatter˜ than expected.

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

grAADrawPolygon() and **grAADrawPolygonVertexList()** draw convex polygons with anti−aliased edges.

guAADrawTriangleWithClip() performs 2D clipping on the specified triangle, and draws the resultant polygon with **grAADrawPolygonVertexList()**. All edges of the clipped triangle will be anti−aliased.

Example . 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 front to back or back to front. The first code shows the alpha blending factors is the scene is drawn 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);

/* draw the scene using the grAADraw 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);
```

Chapter 6 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 effect

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 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 **grSstOpen()** when the Glide and Voodoo Graphics systems are initialized (see Chapter).

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

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 in the *GrVertex* structure as individual color components, since the Voodoo Graphics system will iterate and compute slopes for each color individually.

Dithering

The Voodoo Graphics hardware represents color internally as 32−bit quadruplets in a format specified by the color format argument passed to **grSstOpen()** (see Chapter). 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.

[redacted]

grDitherMode() selects the form of dithering the Voodoo Graphics subsystem 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

*Note: Control of high level rendering functions is managed by three functions , **grColorCombine()**, **grAlphaCombine()** (see Chapter), and **grTexCombine()** (described in Chapter). 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 will be used to compute the color and selects among the input options.

[redacted]

Fourteen combining functions are defined in the *GrCombineFunction_t* enumerated type; one is selected with *func*, the first argument to **grColorCombine()**. 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 . Note that alpha values from the texture combine unit ($\alpha_{texture}$) or specified by **grAlphaCombine()** arguments (α_{local} and α_{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()**. $\mathbf{c}_{\mathbf{local}}$ and $\mathbf{c}_{\mathbf{other}}$ are specified by the third and fourth arguments. Some of the formulas specify an alpha value, $\alpha_{\mathbf{local}}$, 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	α_{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	$f * c_{other} + \alpha_{local}$
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_ALPHA	$f * (c_{other} - c_{local}) + \alpha_{local}$
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	$f * (- c_{local}) + \alpha_{local}$

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 ; its value is chosen from among the symbols list in the left hand column of the table below. The right hand column gives the scale factor that corresponds to each symbolic name. $\mathbf{c}_{\mathbf{local}}$ is specified by the third argument to **grColorCombine()**, $\alpha_{\mathbf{local}}$ and $\alpha_{\mathbf{other}}$ are defined in the **grAlphaCombine()** function described in the next chapter, and $\alpha_{\mathbf{texture}}$ comes from the texture combine unit, described in Chapter .

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	α_{other} / 255
GR_COMBINE_FACTOR_LOCAL_ALPHA	α_{local} / 255
GR_COMBINE_FACTOR_TEXTURE_ALPHA	$\alpha_{texture}$ / 255
GR_COMBINE_FACTOR_ONE	1
GR_COMBINE_FACTOR_ONE_MINUS_LOCAL	1 ˘ c_{local} / 255
GR_COMBINE_FACTOR_ONE_MINUS_OTHER_ALPHA	1 ˘ α_{other} / 255
GR_COMBINE_FACTOR_ONE_MINUS_LOCAL_ALPHA	1 ˘ α_{local} / 255
GR_COMBINE_FACTOR_ONE_MINUS_TEXTURE_ALPHA	1 ˘ $\alpha_{texture}$ / 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 *. Iterated colors are computed by iterating the colors specified in* GrVertex *structures passed to drawing functions. The texture color comes from the texture combine unit (see Chapter), 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 ; their values are chosen from among the symbols in the tables below. Iterated colors are computed by iterating the colors specified in GrVertex structures 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 . 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}.

```
GrVertex 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.*

```
GrVertex 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 . 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 is in the previous example.

```
GrVertex 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.

```
GrVertex 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 . Drawing a smooth shaded triangle.

In this example, a Gouraud shaded triangle will be 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.*

```
GrVertex 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.*

```
GrVertex 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 . Drawing a flat-shaded textured triangle.

The following code produces a textured flat shaded triangle using the constant color.

```
GrVertex 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 . 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 .*

```
GrVertex 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 . Drawing a smooth shaded triangle with specular lighting.

Specular lighting is modeled as the sum of the texture color and the iterated color.

```
GrVertex 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 . 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* α).

```
GrVertex 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 . 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* α) + *iterated RGB). Iterated alpha is chosen to be either* α_{local} *or* α_{other} *with a call to* **grAlphaCombine()** *that is not shown here. In the first code segment, iterated alpha is assumed to be available as* α_{local}.

```
GrVertex 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 α_{other} *in* **grAlphaCombine()**. *In that case the following* **grColorCombine()** *configuration is needed.*

```
GrVertex 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 $\alpha_{texture}$ 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.

[redacted]

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

Table 5.4 Overriding the local color when the high order bit of $\alpha_{texture}$ ***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* $\alpha_{texture}$ *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* $\alpha_{texture}$ *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 enabled 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. A gamma value can be passed to the hardware using the Glide function **grGammaCorrectionValue()**.

grGammaCorrectionValue() sets the gamma correction value used during video refresh. Gamma is a positive floating point value from 0.0 to 20.0. Typical values are 1.3 to 2.2. The default value is 1.0 (i.e. a linear ramp is used).

The displayed RGB value (RGB_{gamma}) is computed from the RGB value read from the frame buffer (RGB_{fb}) according to the following equation:

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

Chapter 7 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 associated with vertices are set in the *GrVertex* structure, along with the geometric coordinates and other parameters.

The Alpha Combine Unit

*Note: Control of high level rendering functions is managed by three functions , **grAlphaCombine()**, **grColorCombine()** (see Chapter) , and **grTexCombine()** (described in Chapter). While the three routines will be 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.

[redacted]

[redacted]

lists the possible values for *func*, the first argument to **grAlphaCombine()**. The *f* that appears in the formulas in is a scale factor that is chosen by the second argument, *factor*. lists the possible scale factors. α_{local} and α_{other} are chosen by the third and fourth arguments, *local* and *other*; the candidates are listed in . 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, and , use the α_{local} or α_{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()**. $\alpha_{\mathbf{local}}$ and $\alpha_{\mathbf{other}}$ are specified by the third and fourth arguments.

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

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 ; its value is chosen from among the symbols list in the left hand column of the table below. The right hand column gives the scale factor that corresponds to each symbolic name. $\alpha_{\mathbf{local}}$ and $\alpha_{\mathbf{other}}$ are defined by the third and fourth arguments to **grAlphaCombine()** and $\alpha_{\mathbf{texture}}$ comes from the texture combine unit, described in Chapter .

combine factor	*scale factor (**f**)*
GR_COMBINE_FACTOR_NONE	*unspecified*
GR_COMBINE_FACTOR_ZERO	0
GR_COMBINE_FACTOR_LOCAL	α_{local} / 255
GR_COMBINE_FACTOR_OTHER_ALPHA	α_{other} / 255
GR_COMBINE_FACTOR_LOCAL_ALPHA	α_{local} / 255
GR_COMBINE_FACTOR_TEXTURE_ALPHA	$\alpha_{texture}$ / 255
GR_COMBINE_FACTOR_ONE	1
GR_COMBINE_FACTOR_ONE_MINUS_LOCAL	1 ˘ α_{local} / 255
GR_COMBINE_FACTOR_ONE_MINUS_OTHER_ALPHA	1 ˘ α_{other} / 255
GR_COMBINE_FACTOR_ONE_MINUS_LOCAL_ALPHA	1 ˘ α_{local} / 255
GR_COMBINE_FACTOR_ONE_MINUS_TEXTURE_ALPHA	1 ˘ $\alpha_{texture}$ / 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 α_{local} and α_{other} values that appear in the alpha combine formulas delineated in 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 GrVertex 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 (α_{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 (α_{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 will be discarded, unless the auxiliary buffer is being used as an alpha buffer.

With alpha buffering enabled, the Voodoo 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.

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

In the anti−aliasing discussion in Chapter , 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 ˆjaggies˜.

The final example in this chapter, , 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 , routines to draw anti−aliased points, lines, triangles and polygons were presented. They use *alpha blending* to smooth the jagged edges. 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 ˆshows 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 will be used to scale the window color before adding it to the existing color.

The Voodoo 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:

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

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

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 α_{dst} will be 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()**.

[redacted]

The first two arguments specify blending factor 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 .

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 . Blending two images.

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 . 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 α_{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 will be 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, Voodoo Graphics supports only GR_BLEND_ZERO *and* GR_BLEND_ONE.

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	$\alpha_{src}/255$
GR_BLEND_ONE_MINUS_SRC_ALPHA	$1 - \alpha_{src}/255$
GR_BLEND_DST_ALPHA	$\alpha_{dst}/255$
GR_BLEND_ONE_MINUS_DST_ALPHA	$1 - \alpha_{dst}/255$
GR_BLEND_ALPHA_SATURATE	$\min(\alpha_{src}/255,\ 1 - \alpha_{dst}/255)$

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`	$\alpha_{src}/255$
`GR_BLEND_ONE_MINUS_SRC_ALPHA`	$1 - \alpha_{src}/255$
`GR_BLEND_DST_ALPHA`	$\alpha_{dst}/255$
`GR_BLEND_ONE_MINUS_DST_ALPHA`	$1 - \alpha_{dst}/255$

Example . 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. The example uses the alpha combine unit , alpha buffer ing, 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_ 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_
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 */
```

Chapter 8 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 $1/z$ and $1/w$ value (*ooz* and *oow* in the *GrVertex* structure) 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

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.

[redacted]

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 effecting 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 *w,* 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 effecting 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, both methods:

> clearing the buffers with **grBufferClear()** and then repainting the cockpit
> 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 **grSstQueryHardware()** to obtain the revision number).

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

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`.

[redacted]

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 ˆpass˜ 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 ˆpass˜ 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`	$\text{depth}_{new} \le \text{depth}_{old}$	*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`	$\text{depth}_{new} \ne \text{depth}_{old}$	*passes if the pixel´s depth value is not equal to the stored depth value*
`GR_CMP_GEQUAL`	$\text{depth}_{new} \ge \text{depth}_{old}$	*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 Voodoo Graphics hardware must instead interpolate $1/z$ values, which *are* linear in screen space. When linear z buffering is enabled, the Voodoo Graphics hardware interpolates a high precision fixed point $1/z$ value (provided by the application), but 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`.

If z buffering is enabled, Glide expects $1/z$ values to be passed in the *ooz* element of the *GrVertex* structure used by the various **grDraw** functions. The *ooz* values take the form $(\delta + \phi/z)$; the values of scalars δ and ϕ should be chosen so that *ooz* is in the range [0..65535] in order to use the full range of the z buffer.

Example . Configuring a z buffer.

The following code sequence configures Glide for z buffer ing:

```
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 Voodoo Graphics hardware can also derive a depth value from the $1/w$ 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 Voodoo 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`.

The value stored in the depth buffer when w buffering is enabled is actually not the $1/w$ value used during texture mapping, but an approximation of the reciprocal of $1/w$, in effect recovering w from $1/w$ during the depth buffering phase. This should be transparent to an application unless the application needs to read depth information back from the depth buffer.

Example . Configuring a w buffer.

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

```
grDepthBufferMode( GR_DEPTHBUFFER_WBUFFER );
grDepthBufferFunction( GR_CMP_LESS );  // larger W values are farther
away
grDepthMask( FXTRUE );
grBufferClear(0, 0, GR_WDEPTHVALUE_FARTHEST);
```

Establishing a Depth Bias

When depth buffering coplanar polygons (e.g. when one polygon is used as a ˆdetail˜ polygon on another), precision problems with coplanar polygons may result in ˆpoke through˜ artifacts if the vertices of the two polygons are not the same. To eliminate the artifacts, an application should apply a ˆdepth 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.

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 ˆnearest to viewer˜ and `0xFFFF` corresponds ˆfarthest from viewer˜, depth bias levels should be decremented on successive rendering of coplanar polygons. When the coplanar polygons have been rendered, the depth bias mode should be reset to 0.

Example . Using a depth bias.

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

```
/* Render the underlying polygon */
grDrawPolygon( /* base polygon´s parameters */ );

/* Render the composite polygon by first enabling depth bias */
grDepthBiasLevel( −1 );
grDrawPolygon( /* 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 some of all parts of other objects. If the viewpoint never changes, you can sort the polygons sort them 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 shows the depth buffer in action.

Example . 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);
}
```

Chapter 9 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

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 Voodoo 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 may improve performance in large simulations as some objects are too fogged to be visible and can be culled before rendering.

The fog operation blends the fog color (c_{fog}) with each rasterized pixel´s post−texturing color (c_{in}) using a blending factor $\boldsymbol{f}$. Factor $\boldsymbol{f}$ is retrieved from the high order bits of the iterated alpha value or from a user downloaded fog table indexed with the pixel´s $1/w$ component.

The post−fog color is computed as follows:

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

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

Fog is enabled and disabled with the **grFogMode()** function call. The *mode* argument can be one of three values: `GR_FOG_DISABLE`, `GR_FOG_WITH_ITERATED_ALPHA`, or

`GR_FOG_WITH_TABLE`. 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 fogging factor ***f*** is determined by *mode*. If mode is `GR_FOG_WITH_ITERATED_ALPHA`, then ***f*** is equal to the integer bits of iterated alpha. If *mode* is `GR_FOG_WITH_TABLE`, 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 *w* component. 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 I and is specified in the format defined in the *cFormat* parameter to **grSstOpen()** (see Chapter 3).

Fogging With Iterated Alpha

To fog with iterated alpha, the fog mode must be set to `GR_FOG_WITH_ITERATED_ALPHA`. In this mode the high order eight bits of the value produced by the alpha iterator are used as the fog blending factor ***f***. The fog equation becomes

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

presents a code segment that adds iterated alpha fog to a scene.

Example . Fogging with iterated alpha.

```
/* fog is white  color is ARGB format */
grFogColorValue( 0xFFFFFF);
grFogMode(GR_FOG_WITH_ITERATED_ALPHA);

/* vertices have alpha values that grow as the object gets more
indistinct */
draw_objects();
```

Fogging With A User Specified Fog Table

The application may supply a fog table to the hardware via the function **grFogTable()**. To enable fog table based fogging, the fog mode must be set to `GR_FOG_WITH_TABLE`. This fog table should consist of 64 density values of type *GrFog_t*, which is 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 *w* where world $w \cong 2^{i/4}$ where *i* is the index into the fog table and $(0 \le i < 64)$. To minimize ˆfog banding˜, the Voodoo

Graphics hardware linearly blends between adjacent fog levels within the fog table. *The difference between consecutive fog values must be less than 64.*

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`, 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 w component.

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

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

guFogTableIndexToW() returns the floating point eye−space w value associated with entry i in a fog table. Because fog table entries are non−linear in w, 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 $(1 - e^{-kw})$*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 . Creating a fog table.

The two code segments below each create a fog table. 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.

```
const GrFog_t fog[63];
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<63; i++) fog[i]= fog[i-1] + 7;
fog[63] = 255;
```

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

```
#define FOG_DENSITY .5
const GrFog_t fog[GR_FOG_TABLE_SIZE];
int i;

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

```
    }
    fog[GR_FOG_TABLE_SIZE] = 255;
```

Example . Fogging with 1/w and a fog table.

The code segment below assumes that a fog table has been defined. This is the default mode: all you have to do is load a fog table (see) and enable fog mode .

```
const GrFog_t fog[GR_TABLE_SIZE];
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);

/* draw the scene */
```

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

[redacted]

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

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

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 normalized (scaled) such that the last entry is maximum fog (255).

[redacted]

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

$$e^{-(density\ *w)\ (density\ *w)}$$

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 normalized (scaled) such that the last entry is maximum fog (255).

[redacted]

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

$$(w - nearW)/(farW - nearW)$$

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, not* in screen−space.

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). 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.

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

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

Example . 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 ˆblue˜ 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 ˆgreater than˜ comparison function, transparent texels will be rejected and the destination pixel will be 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.

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 . 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()**.

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 8.1 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 linear frame buffer bypass is enabled (see **grLfbBypassMode()** in Chapter).

Stenciling

Stenciling is not directly supported by the Voodoo 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.

Chapter 10 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 Voodoo 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 Voodoo 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 unit s.*

(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 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] and must be divided by w before storing them in a *GrVertex* structure as *oow* (one over w), *sow* (s over w) and *tow* (t over w). The large range for s and t allow a texture to be repeated many times across a polygon. A large number of fraction bits allow for precise s and t representation and iteration even when divided by a large w value.

Each TMU in the system maintains its own *oow*, *sow*, and *tow* variables. The *GrVertex* structure reflects this architecture by keeping *oow*, *sow*, and *tow* in an array, *tmuvtx*, that is indexed by the TMU number. Normally, they will all be the same. However, projected textures have a different w value than non–projected textures. Projected textures iterate q/w where w is the homogeneous distance from the eye and q is the homogeneous distance from the projected source.

[redacted]

[redacted]

By default, Glide assumes that all w coordinates (*oow*) in the *GrVertex* structure are identical, and that all s and t coordinates (*sow* and *tow*) are also identical.

These assumptions significantly reduce the amount of time spent computing gradients for s, t, and w, and transferring data to the graphics hardware. If these assumptions are false, however, the application must alert Glide that specific values in the *GrVertex* structure are different and that gradients need to be computed for these values. The **grHints()** routine is provided for this purpose.

grHints() informs Glide of special conditions regarding optimizations and operation. Each *hintType* controls a different optimization or mode of operation. The `GR_HINT_STWHINT` hint type controls *stw* parameter optimization and specify Glide´s source for the parameter values. Hints of a given type are ORed together into a *hintMask*.

There is an implicit ordering of TMUs within Glide, starting with TMU0, followed by TMU1, and TMU2. By default, Glide reads *sow* and *tow* values from the *GrVertex* structure for the first TMU that is active. Whenever s and t coordinates are read, they are transmitted to all subsequent TMUs. For example, if texturing is active in TMU1 but not active in TMU0, then *sow* and *tow* values are read from *tmuvtx[1]* and broadcast to TMU1 and TMU2. Once *sow* and *tow* values are read, they will not be read again unless a hint is specified. If one of the subsequent units has a unique or different parameter value, then a hint must be used. If a hint is specified, the parameter value will be read again and sent to the specified unit and all other units following it.

Hints are also used to help Glide find w coordinates. The rule for the w coordinate is very simple: the w coordinate is read from the *GrVertex* structure and broadcast to all TMUs unless a w hint is specified. If a w hint is specified and if w buffering or table−based fog is enabled, then *tmuvtx[].oow* structure corresponding to the TMU mentioned in *hintMask* is read and broadcast to all subsequent TMUs.

The *hintMask* for `GR_HINT_STWHINT` hints is created by ORing together the *stw* hints that are shown in .

Table 9.1 The stw hints.

The **grHints()** *function alerts Glide to situations that differ from the norm. The stw hints indicate that the* ***sow***, ***tow***, *and* ***oow*** *values in the* ***tmuvtx*** *arrays are not the same as the ones in the* GrVertex *itself. A* ***hintMask*** *is composed by ORing together a collection of the hints listed below.*

hint	*description*
GR_STWHINT_ST_DIFF_TMU0	s and t for TMU0 are different than previous values
GR_STWHINT_ST_DIFF_TMU1	s and t for TMU1 are different than previous values
GR_STWHINT_ST_DIFF_TMU2	s and t for TMU2 are different than previous values
GR_STWHINT_W_DIFF_TMU0	w for TMU0 is different than previous w values
GR_STWHINT_W_DIFF_TMU1	w for TMU1 is different than previous w values
GR_STWHINT_W_DIFF_TMU2	w for TMU2 is different than previous w values

Texel Coordinate Systems

All square texture maps have their origin at $(s,t) = (0,0)$ and their opposite corner at (255,255). This is true even for a 1×1 texture map. Note that these texture coordinates are *before* division by w. 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 an $2^n \times 2^n$ texture map (where $0 \leq n \leq 8$) 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 (255, n) where $(n+1)/256=t/s$. For example, if the texture is four times as wide as high, then n=63. Likewise, if the rectangular texture is taller than it is wide, the opposite corner is at (n, 255) and $n/255=s/t$. Therefore, the longer texture axis always has texture coordinates running from 0 to 255, while the shorter texture axis is proportionally smaller. Table 9.2 shows the texel coordinates of the first and last pixel for all supported aspect ratios and texture map dimensions.

All texture mapping capabilities of the Voodoo Graphics subsystem are handled in the TMU, which includes logic to support true−perspective texture mapping (dividing by w every pixel), 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 Voodoo 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.

Table 9.2 Mapping pixels to texture coordinates in texture maps.

The texel coordinates in a texture map always go from 0 to 255, regardless of the size of the texture map.

if the aspect ratio is	*and the texture map size is*	*the center of the first pixel is at*	*the center of the last pixel is at*
1:1	256×256	(.5, .5)	(255.5, 255.5)
(*a square texture*)	128×128	(1, 1)	(255, 255)
	64×64	(2, 2)	(254, 254)
	32×32	(4, 4)	(252, 252)
	16×16	(8, 8)	(248, 248)
	8×8	(16, 16)	(240, 240)
	4×4	(32, 32)	(224, 224)
	2×2	(64, 64)	(192, 192)
	1×1	(128, 128)	(128, 128)
2:1	256×128	(.5, 1)	(255.5, 255)
	128×64	(1, 2)	(255, 254)
	64×32	(2, 4)	(254, 252)
	32×16	(4, 8)	(252, 248)
	16×8	(8, 16)	(248, 240)
	8×4	(16, 32)	(240, 224)
	4×2	(32, 64)	(224, 192)
	2×1	(64, 128)	(192, 128)
	1×1	(128, 128)	(128, 128)
4:1	256×64	(.5, 2)	(255.5, 254)
	128×32	(1, 4)	(255, 252)
	64×16	(2, 8)	(254, 248)
	32×8	(4, 16)	(252, 240)
	16×4	(8, 32)	(248, 224)
	8×2	(16, 64)	(240, 192)
	4×1	(32, 128)	(224, 128)
	2×1	(64, 128)	(192, 128)
	1×1	(128, 128)	(128, 128)
8:1	256×32	(.5, 4)	(255.5, 252)
	128×16	(1, 8)	(255, 248)
	64×8	(2, 16)	(254, 240)
	32×4	(4, 32)	(252, 224)
	16×2	(8, 64)	(248, 192)
	8×1	(16, 128)	(240, 128)
	4×1	(32, 128)	(224, 128)
	2×1	(64, 128)	(192, 128)
	1×1	(128, 128)	(128, 128)
1:2	128×256	(1, .5)	(255, 255.5)
	64×128	(2, 1)	(254, 255)
	32×64	(4, 2)	(252, 254)
	16×32	(8, 4)	(248, 252)
	8×16	(16, 8)	(240, 248)
	4×8	(32, 16)	(224, 240)
	2×4	(64, 32)	(192, 224)
	1×2	(128, 64)	(128, 192)

	1×1	(128, 128)	(128, 128)
1:4	64×256	(2, .5)	(254, 255.5)
	32×128	(4, 1)	(252, 255)
	16×64	(8, 2)	(248, 254)
	8×32	(16, 4)	(240, 252)
	4×16	(32, 8)	(224, 248)
	2×8	(64, 16)	(192, 240)
	1×4	(128, 32)	(128, 224)
	1×2	(128, 64)	(128, 192)
	1×1	(128, 128)	(128, 128)
1:8	32×256	(4, .5)	(252, 255.5)
	16×128	(8, 1)	(248, 255)
	8×64	(16, 2)	(240, 254)
	4×32	(32, 4)	(224, 252)
	2×16	(64, 8)	(192, 248)
	1×8	(128, 16)	(128, 240)
	1×4	(128, 32)	(128, 224)
	1×2	(128, 64)	(128, 192)
	1×1	(128, 128)	(128, 128)

Texture Filtering

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 9.2 Point sampled 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.

Magnification of a texture map occurs when a texture map is ˆblown up˜ on screen (see Figure 9.2(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 Voodoo 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 9.2(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 ˆtexture aliasing˜. The Voodoo Graphics hardware remedies this problem by supporting both mipmapping and filtering.

If a Voodoo 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 9.2(c)). Point sampling, also known as nearest neighbor sampling, may result in pixelization and blockiness during magnification and ˆtexture 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 9.2(d)). *Bilinear interpolation* is performed by the Voodoo Graphics hardware with no incurred additional performance overhead.

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

[redacted]

[redacted]

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 9.3 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 will be clamped to 0 and 255. Both modes should always be set to `GR_TEXTURECLAMP_CLAMP` *when using projected textures.*

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

[redacted]

[redacted]

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 *either* `GR_TEXTURECLAMP_CLAMP` or `GR_TEXTURECLAMP_WRAP`. 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 ˆlevel 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. 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 .

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 9.3 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_1x1`	`GR_ASPECT_2x1` *or* `GR_ASPECT_1x2`	`GR_ASPECT_4x1` *or* `GR_ASPECT_1x4`	`GR_ASPECT_8x1` *or* `GR_ASPECT_1x8`

GR_LOD_256	256×256	256×128 *or* 128×256	256×64 *or* 64×256	256×32 *or* 32×256
GR_LOD_128	128×128	128×64 *or* 64×128	128×32 *or* 32×128	128×16 *or* 16×128
GR_LOD_64	64×64	64×32 *or* 32×64	64×16 *or* 16×64	64×8 *or* 8×64
GR_LOD_32	32×32	32×16 *or* 16×32	32×8 *or* 8×32	32×4 *or* 4×32
GR_LOD_16	16×16	16×8 *or* 8×16	16×4 *or* 4×16	16×2 *or* 2×16
GR_LOD_8	8×8	8×4 *or* 4×8	8×2 *or* 2×8	8×1 *or* 1×8
GR_LOD_4	4×4	4×2 *or* 2×4	4×1 *or* 1×4	
GR_LOD_2	2×2	2×1 *or* 1×2		
GR_LOD_1	1×1			

Figure 9.4 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_256` *is the name given to the texture with the highest resolution in the mipmap, where LOD stands for ˆlevel of detail˜. The* `GR_LOD_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 all nine textures, as described in Table 9.2.*

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 ˆmipmap 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.

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. To allow `GR_MIPMAP_NEAREST_DITHER` mode to be used , call **grHints()**.

grHints() informs Glide of special conditions regarding optimizations and operation. Each *hintType* controls a different optimization or mode of operation. Hints of a given type are ORed together into a *hintMask*. The default *hintMask* is `0x00`.

The GR_HINT_ALLOW_MIPMAP_DITHER hint type controls whether or not GR_MIPMAP_NEAREST_DITHER mode can be used. If *hintMask* is zero, then GR_MIPMAP_NEAREST_DITHER mode cannot be enabled with **grTexMipMapMode()**. This is the default. To allow GR_MIPMAP_NEAREST_DITHER mode to be used, specify a non−zero *hintMask* with the hint, as shown below

```
grHints( GR_HINT_ALLOW_MIPMAP_DITHER, 1 );
```

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 Voodoo 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

Glide allows an application to control an arbitrary factor known as *LOD bias*; it 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 ˆbest˜ setting for the LOD bias; it is a very subjective control. Some textures look better if sharper than ˆnormal˜, 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 *si*gned floating point value in the range [˜8..7.75].

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 Voodoo 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 passthru 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.

[redacted]

[redacted]

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 9.4 lists the possible combining functions; the scale factors are detailed in Table 9.5. In both tables, c_{local} and α_{local} represent the color components generated by indexing and filtering from the mipmap stored on *tmu*; c_{other} and α_{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 9.4 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 α_{local} represent the color components generated by indexing and filtering from the mipmap stored on* **tmu**; *c_{other} and α_{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}	α_{local}
LOCAL_ALPHA	α_{local}	α_{local}
SCALE_OTHER BLEND_OTHER	$f * c_{other}$	$f * \alpha_{other}$
SCALE_OTHER_ADD_LOCAL	$f * c_{other} + c_{local}$	$f * \alpha_{other} + \alpha_{local}$
SCALE_OTHER_ADD_LOCAL_ALPHA	$f * c_{other} + \alpha_{local}$	$f * \alpha_{other} + \alpha_{local}$
SCALE_OTHER_MINUS_LOCAL	$f * (c_{other}$ ˘ $c_{local})$	$f * (\alpha_{other}$ ˘ $\alpha_{local})$
SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL BLEND	$f * (c_{other}$ ˘ $c_{local}) + c_{local}$ $\equiv f * c_{other} + (1$ ˘ $f) * c_{local}$	$f * (\alpha_{other}$ ˘ $\alpha_{local}) + \alpha_{local}$ $\equiv f * \alpha_{other} + (1$ ˘ $f) * \alpha_{local}$
SCALE_OTHER_MINUS_LOCAL_ADD_LOCAL_ALP HA	$f * (c_{other}$ ˘ $c_{local}) + \alpha_{local}$	$f * (\alpha_{other}$ ˘ $\alpha_{local}) + \alpha_{local}$
SCALE_MINUS_LOCAL_ADD_LOCAL BLEND_LOCAL	$f * ($˘ $c_{local}) + c_{local}$ $\equiv (1$ ˘ $f) * c_{local}$	$f * ($˘ $\alpha_{local}) + \alpha_{local}$ $\equiv (1$ ˘ $f) * \alpha_{local}$
SCALE_MINUS_LOCAL_ADD_LOCAL_ALPHA	$f * ($˘ $c_{local}) + \alpha_{local}$	$f * ($˘$\alpha_{local}) + \alpha_{local}$

Table 9.5 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 α_{local} represent the color components generated by indexing and filtering from the mipmap stored on* **tmu***; c_{other} and α_{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* β *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* λ *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	α_{local} / 255
OTHER_ALPHA	α_{other} / 255	α_{other} / 255
LOCAL_ALPHA	α_{local} / 255	α_{local} / 255
DETAIL_FACTOR	β	β
LOD_FRACTION	λ	λ
ONE	1	1
ONE_MINUS_LOCAL	1 ˘ c_{local} / 255	1 ˘ α_{local} / 255
ONE_MINUS_OTHER_ALPHA	1 ˘ α_{other} / 255	1 ˘ α_{other} / 255
ONE_MINUS_LOCAL_ALPHA	1 ˘ α_{local} / 255	1 ˘ α_{local} / 255
ONE_MINUS_DETAIL_FACTOR	1 ˘ β	1 ˘ β
ONE_MINUS_LOD_FRACTION	1 ˘ λ	1 ˘ λ

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 will be 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 9.6 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 9.6 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 Voodoo 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	*Voodoo Graphics 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 9.6 are decal mapping, differing only in the choice of minification and magnification filters. Decal mapping is a single pass operation on all Voodoo Graphics configurations.

Example . 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 ˆupstream˜ TMU1 become the *other* input to the ˆdownstream˜ TMU0. In , the spotlight texture is upstream, the source texture is downstream and the resulting $RGBA_{texture} = RGBA_{spotlight} \times RGBA_{source}$.

Example . 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, and 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 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 . 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 ˆpits 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.

The first argument specifies the TMU to modify; valid values are `GR_TMU0`, `GR_TMU1`, and `GR_TMU2`. The second argument, *detail Bias*, 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 *detail Bias* parameter controls where the blending begins; the *detailS cale* parameter controls how fast the detail shows up; and the *detailM ax* parameter controls the maximum blending that occurs.

$$\beta = \min(\, detailMax\ ,\ \max(\, 0,\ (detail\ Bias ˜ \mathrm{LOD}) << detailScale\ ) \,/\, 255.0\,)$$

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:

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

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 . 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 );
```

Example . Creating a composite texture, part II.

multiply RGB, add alpha ; show how to compose the AI88 texture

normal on tmu1
composed one on tmu0
RGB factor local, function scale other add local alpha

A factor zero, invert

Chapter 11 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 puts the horse back in front of the cart by describing the multitude of texture formats that Glide supports and the routines to 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 can model monochrome lighting effects. 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.

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 Voodoo Graphics system provides 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), 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,

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

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

$$\mathbf{B} = 0.212*red + 0.523*green + 0.311*blue$$

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:

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

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

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

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.

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

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 Voodoo 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 *(not implemented in TMU Revision 0)*

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). 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.

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.*

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:

Determine how much memory is required for the texture.
Determine the starting address and extent of free space. Is it adequate for the texture?
Download the texture.
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. Textures must start on an 8−byte boundary in memory. The size returned by these functions includes any bytes required to pad the texture to an 8−byte 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 determines the memory requirements of a texture that will be split across two TMUs for trilinear filtering applications (see 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_256 GR_LOD_128 GR_LOD_64 GR_LOD_32 GR_LOD_16 GR_LOD_8 GR_LOD_4 GR_LOD_2 GR_LOD_1	*The number in the constant is the largest of the texture . The aspect ratio determines the smaller dimension .*
aspectRatio	*GrAspectRatio_t*	GR_ASPECT_8x1 GR_ASPECT_4x1 GR_ASPECT_2x1 GR_ASPECT_1x1 GR_ASPECT_1x2 GR_ASPECT_1x4 GR_ASPECT_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_4 4 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 for a description of the texture format s .*
evenOdd	*FxU32*	GR_MIPMAPLEVELMASK_EVEN GR_MIPMAPLEVELMASK_ODD GR_MIPMAPLEVELMASK_BOTH	*Even LODs are GR_LOD_256, GR_LOD_64, GR_LOD_16, GR_LOD_4, and GR_LOD_1.* *Odd LODs are GR_LOD_128, GR_LOD_32, GR_LOD_8, and GR_LOD_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.*

[redacted]

[redacted]

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 .

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.

[redacted]

[redacted]

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 will be 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`. 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_256`, `GR_LOD_64`, `GR_LOD_16`, `GR_LOD_4`, and `GR_LOD_1`. The other LODs are odd: `GR_LOD_128`, `GR_LOD_32`, `GR_LOD_8`, and `GR_LOD_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.*

[redacted]			
GR_LOD_128	*128*	*64*	$2^{13} = 8192$ *bytes*
GR_LOD_64	*64*	*32*	$2^{11} = 2048$ *bytes*
GR_LOD_32	*32*	*16*	$2^{9} = 512$ *bytes*
GR_LOD_16	*16*	*8*	$2^{7} = 128$ *bytes*
GR_LOD_8	*8*	*4*	$2^{5} = 32$ *bytes*

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

$$8192 + 2048 + 512 + 128 + 32 = 10{,}912 \text{ bytes}$$

- **grTexTextureMemRequired(**GR_MIPMAPLEVELMASK_ODD, *info***)** returns the sum of the sizes of the odd LODs: GR_LOD_128, GR_LOD_32, and GR_LOD_8.

$$8192 + 512 + 32 = 8{,}736 \text{ bytes}$$

- **grTexTextureMemRequired(**GR_MIPMAPLEVELMASK_EVEN, *info***)** returns the sum of the sizes of the even LODs: GR_LOD_64 and GR_LOD_16.

$$2048 + 128 = 2{,}176 \text{ bytes}$$

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.

[redacted]

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 8-byte 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_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.

Example . Will the mipmap fit?

This code segment illustrates a simple scenario where a single mipmap will be loaded into an empty texture memory on TMU0. Since this is the only texture that will ever be loaded, there is no need to implement a free list.

```
FxU32 textureSize, startAddress;

textureSize = grTexCalcMemRequired(     GR_LOD_1, GR_LOD256,
GR_ASPECT_1x1,
                                   GR_TEXFMT_ARGB_1555 );
startAddress = grTexMinAddress(GR_TMU0);

if (startAddress + textureSize <= grTexMaxAddress(GR_TMU0))
    download_the_texture;
```

Example . 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.

```
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;
   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 be between the values returned by **grTexMinAddress()** and **grTexMaxAddress()** and must be 8–byte 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.

[redacted]

[redacted]

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_128` *and* `GR_LOD_8`.

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_128`, `GR_LOD_32`, *and* `GR_LOD_8` *are odd LODs.*

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.*

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()**.

[redacted]

[redacted]

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 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_128`*, use the following call to* **grTexDownloadMipMapLevel()**.

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()**.

[redacted]

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_256` *in chunks, use a series of calls to*
grTexDownloadMipMapLevelPartial()*:*

```
for (row=0; row<256; row+=64)
        grTexDownloadMipMapLevel( GR_TMU0, startAddress , GR_LOD_256,
info→largeLod ,
                info→aspectRatio , 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.

[redacted]

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 , with additional lines of code to download the appropriate textures.

Example . 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.smallLod = GR_LOD_1;
info.largeLod = GR_LOD_256;
info.aspectRatio = GR_ASPECT_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(ˆerror: 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 . 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.smallLod = spot.smallLod = GR_LOD_1;
src.largeLod = spot.largeLod = GR_LOD_256;
src.aspectRatio = spot.aspectRatio = GR_ASPECT_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(ˆerror: 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(ˆerror: 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 . 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.smallLod = GR_LOD_1;
tri.largeLod = GR_LOD_256;
tri.aspectRatio = GR_ASPECT_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(ˆerror: 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(ˆerror: 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.smallLod = GR_LOD_1;
tri.largeLod = GR_LOD_256;
```

```
tri.aspectRatio = GR_ASPECT_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(ˆerror: 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(ˆerror: 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_256`, `GR_LOD_128`, and `GR_LOD_64`, and one for textures `GR_LOD_32` through `GR_LOD_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. provides an example.

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.

[redacted]

[redacted]

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_256`.

Example . 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_256`*, that the block pointed to by* `start[1]` *is large enough for* `GR_LOD_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.smallLod = info.largeLod = 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 ˆdecompressed˜ 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.

Each TMU has room for 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.

[redacted]

grTexDownloadTable() downloads either an NCC table or a 256−entry color palette to a TMU. The first argument names the TMU on which the table will be loaded. The second argument, *tableType*, describes the kind of table that will be downloaded and is specified with one of three Glide constants: GR_TEX_NCC0, GR_TEX_NCC1, or GR_TEX_PALETTE. The third argument points to the data for the table, which must be of type *GuNccTable* or *GuTexPalette.*

Part of a 256−entry color palette can be downloaded or replaced with the Glide function **grTexDownloadTablePartial()**.

[redacted]

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.

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.

Glide represents NCC decompression tables with the *GuNccTable* data structure, shown below.

[redacted]

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.

[redacted]

grTexNCCTable() selects one of the two NCC tables on *tmu* as the current source for decompression operations. Valid values are `GR_TEXTABLE_NCC0` and `GR_TEXTABLE_NCC1`.

Example . 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 NCC 1 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(ˆncctable.3df˜, &info);
grTexDownloadTable(GR_TMU0, GU_TEX_NCC1, &info.table.nccTable);
grTexNCCTable(GR_TMU0, GR_TEXTABLE_NCC1);
```

Color Palettes

A color palette is an array of 256 ARGB colors, 8 bits for each component, 32 bits per entry (refer back to). The alpha component, in the high order 8 bits, is ignored. It is defined using the Glide structure *GuTexPalette*, shown below.

[redacted]

[redacted]

Example . 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(GR_TMU0, 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 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.

[redacted]

[redacted]

The procedure for reading a `.3DF` file from Glide is shown in . The application first calls **gu3dfGetInfo()** to fill in the *Gu3dfInfo* structure pointed to by *info*.

[redacted]

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.

[redacted]

Both **gu3dfGetInfo()** and **gu3dfLoad()** return `FXTRUE` if the file specified by *filename* exists and can be read; otherwise they return `FXFALSE`.

Example . Reading a `.3DF` ***file.***

The following code segment assumes that ˆ `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(ˆmipmap.3df˜, &fileInfo);
fileInfo.data = malloc(fileInfo.mem_required);
gu3dfLoad(ˆmipmap.3df˜, &fileInfo);
```

Chapter 12 Accessing the Linear Frame Buffer

In This Chapter

The frame buffer on a Voodoo 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

[redacted]

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 the frame buffer data. This lock may assert a critical code section which effects 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.

[redacted]

[redacted]

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()** and take into account the possibility that a lock may fail. A buffer is locked for reads or for writes, as specified in the *type* parameter. Valid types are `GR_LFB_READ_ONLY` and `GR_LFB_WRITE_ONLY`.

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 pixels formats are described in and , later in this chapter.

If the application specifies GR_LFB_WRITEMODE_ANY and the lock succeeds, the destination pixel format will be 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 will be 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 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 will be 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 will be filled in with information pertaining to the locked buffer. The *GrLfbInfo_t* structure is defined as:

[redacted]

[redacted]

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 scan lines. 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 **grSstIdle()** 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.

[redacted]

[redacted]

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 scan line, a value that is returned in the *info* structure when **grLfbLock()** is successful. *info.strideInBytes* represents the number of bytes in a row or scan line. Thus,

$$address_{(x,y)} = y * info.strideInBytes + x$$

$$address\ of\ the\ word\ containing\ (x,y) = address_{(x,y)}/2 = (y * info.strideInBytes + x)/2$$

The interpretation of y (does it count down from the upper left corner or up from the lower left corner?) is dependent upon the choice of y origin location set in the call to **grSstOpen()** (see Chapter). When writing to the LFB, the location of the y origin set in **grSstOpen()** 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 ***grLfbGetReadPtr()****, 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 . 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:

$$1.mantissa \quad * \ 2^{exponent}$$

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* ***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.*

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 . 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* ***writeMode****,* ***origin****, and* ***pixelPipeline*** *arguments to* ***grLfbLock()*** *are ignored for read−only pointers.*

```
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 */
   rptr = info.lfbPtr
   rptr += ((200<<10) + 100)>>1;

   /*read two pixels at a time */
   for (i=0; i<10; rptr++) {
      pix[i++] = *info.lfbPtr && 0xFFFF;
      pix[i++] = *info.lfbPtr >>16;
   }
   grLfbUnlock( GR_LFB_READ_ONLY, GR_LFB_FRONTBUFFER );
}
```

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.

[redacted]

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:

If the pixel pipeline will be enabled during LFB writes, depth buffering or alpha buffering is enabled, and the desired *writeMode* is lacking depth or alpha values, set constant values for depth and/or alpha with **grLfbConstantDepth()** and **grLfbConstantAlpha().**

Call **grLfbLock()** to get a write pointer. Specify a write mode and *y* origin if desired. Bypass the pixel pipeline if desired.

Write into the linear frame buffer using the write pointer.

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 **grSstOpen()** (see).

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* ***grSstOpen()****. 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* ***grSstOpen()****. 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()** will be 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 will be 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.

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.

[redacted]

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 ˆpure˜ 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.

[redacted]

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 . 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 );

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?

The discussion in this section centers on what will happen and how to individually disable or circumvent the special effects.

If *depth buffer ing* 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 bias ing* 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 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 . 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 . 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 . 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*1024] = 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.

[redacted]

[redacted]

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 final argument, *data*, points to the pixel data in memory.

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.

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).

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.

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 padded to 32−bits with RGB in 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.

Chapter 13 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 additional system configuration information: the current version of Glide, the number of SST subsystems present, the size of the display screen

check the system status

utilize two display monitors

monitor system performance by leaning the fate of pixels in the pixel pipeline

Retrieving Configuration Information

The first three chapters of this manual present some routines that retrieve and use system configuration information. The remaining routines are presented here.

Which Glide Release?

When your customer service representative asks you which version of Glide you are using, you might whip up a little program that calls **grGlideGetVersion()**.

[redacted]

A null−terminated string that describes the Glide version is returned in *version*. For example, the string ˆGlide Version 2.2˜ is returned by the Glide software described in this manual.

How Big a Screen?

grSstScreenHeight() and **grSstScreenWidth()** the height and width in pixels, respectively, of the current SST display buffer.

[redacted]

Checking System Status

Three Glide routines help you determine the status of the Voodoo Graphics hardware.

[redacted]

grSstIdle() blocks until the Voodoo Graphics subsystem is idle. The system is busy when either the hardware FIFO is not empty or the graphics engine is busy.

The other routine, **grSstIsBusy(),** is non–blocking. It returns `FXTRUE` if the Voodoo Graphics subsystem is busy, and `FXFALSE` otherwise.

You can also look at the contents of the status register in the Voodoo Graphics system by calling **grSstStatus().**

[redacted]

grSstStatus() returns a 32–bit unsigned integer containing the contents of the status register. The bits within this register are defined in .

Figure 12.1 The Voodoo Graphics status register.

bit	*description*
5:0	PCI FIFO free space (`0x3F`=FIFO empty)
6	Vertical retrace (0=vertical retrace active; 1=vertical retrace inactive).
7	Pixel*fx* graphics engine busy (0=engine idle; 1=engine busy)
8	TMU busy (0=engine idle; 1=engine busy)
9	Voodoo Graphics busy (0=idle; 1=busy)
11:10	Displayed buffer (0=buffer 0; 1=buffer 1; 2=auxiliary buffer; 3=reserved)
27:12	Memory FIFO free space (`0xFFFF`=FIFO empty)
30:28	Number of swap buffer commands pending
31	PCI interrupt generated (*not implemented*)

Utilizing Two Displays

grSstPassthruMode() should be called when switching between the VGA and the Voodoo Graphics display for things like an attract mode, introductory video clips, etc. Use this routine instead of initializing and shutting down Glide.

[redacted]

mode The new passthru mode.

grSstPassthruMode() either shows the VGA or shows the Voodoo Graphics display depending on the value of *mode* : `GR_PASSTHRU_SHOW_VGA` or `GR_PASSTHRU_SHOW_SST1`.

Monitoring System Performance

The Voodoo Graphics hardware maintains a set of five counters that collect statistics about the fate of pixels as they move through the pixel pipeline. Glide provides access to these counters through the *GrSstPerfStats_t* structure and **grSstPerfStats()**.

[redacted]

In order to account for every pixel counted and saved in *pixelsOut*, one must use the following equation:

$$pixelsOut = LfbWritePixels + bufferClearPixels + (pixelsIn - zFuncFail - chromaFail - aFuncFail)$$

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)

grSstPerfStats() does not wait for the system to be idle, and hence does not include statistics for commands that are still in the FIFO. Call **grSstIdle()** to empty the FIFO.

All five counters are reset whenever **grSstResetPerfStats()** is called. The hardware counters are only 24–bits wide, so regular calls to **grSstResetPerfStats()** are required to avoid overflow. Alternatively, counter overflows can be detected and accounted for without calling **grSstResetPerfStats()**.

[redacted]

Chapter 14 Glide Utilities

In This Chapter

The Glide Utility Library is a set of utility functions that are built on top of the lower−level Glide routines presented in the preceding chapters. Many are convenience routines that provide higher−level services: functional descriptions of color, alpha, and texture combine functions, texture memory management services, and so on. Some of the Glide Utility Library functions have already been described: the routines to clip and draw triangles in Chapter , the functions that create fog tables in Chapter , or the functions that download mipmaps and decompression tables from `.3DF` files in Chapter .

In this chapter, you will discover

a different way to configure the color combine, alpha combine, and texture combine units

a different way to manage texture memory

a higher level way to read and write the linear frame buffer

A Higher Level Color Combine Function

Chapter 4 introduced the **grColorCombine()** function; it provides a low−level way to configure the color combine unit. The **guColorCombineFunction()** provides a higher level mechanism for controlling common rendering modes without manipulating individual registers within the hardware.

The argument, *function*, specifies one of fourteen color combine functions. lists the Glide constants that define the color combine function and the effects than can be achieved with that function. The default color combine function is undefined, so an application must set the color combine function before executing any rendering commands. Refer to the **guColorCombineFunction()** page in the *Glide 2.2 Reference Manual* for more information.

Table 13.1 Color combine functions.

function	*effect*
GR_COLORCOMBINE_ZERO	`0x00` (black) for each component
GR_COLORCOMBINE_ITRGB	Gouraud shading
GR_COLORCOMBINE_DECAL_TEXTURE	texture
GR_COLORCOMBINE_TEXTURE_TIMES_CCRGB	flat shaded texture using the constant color set by **grConstantColorValue()** as the shading value
GR_COLORCOMBINE_TEXTURE_TIMES_ITRGB	Gouraud shaded texture
GR_COLORCOMBINE_TEXTURE_TIMES_ITRGB_ADD_ALPHA	Gouraud shaded texture + alpha
GR_COLORCOMBINE_TEXTURE_TIMES_ALPHA	texture * alpha
GR_COLORCOMBINE_TEXTURE_ADD_ITRGB	texture + iterated RGB
GR_COLORCOMBINE_TEXTURE_SUB_ITRGB	texture ˘ iterated RGB
GR_COLORCOMBINE_CCRGB	flat shading using the constant color set by **grConstantColorValue()**
GR_COLORCOMBINE_CCRGB_BLEND_ITRGB_ON_TEXALPHA	blend between constant color and iterated RGB using an alpha texture, where alpha of 0 and 1 correspond to constant color and iterated RGB respectively
GR_COLORCOMBINE_DIFF_SPEC_A	*texture* * α + *iterated* RGB
GR_COLORCOMBINE_DIFF_SPEC_B	*texture* * *iterated* RGB + α
GR_COLORCOMBINE_ONE	`0xFF` (white) for each component

A Higher Level Alpha Combine Function

guAlphaSource() is a higher level interface to the Voodoo Graphics alpha combine unit than **grAlphaCombine()**, which was presented in Chapter .

The alpha combine unit has two configurable inputs and one output. The output of the alpha combine unit gets fed into the alpha testing and blending units. The selection of the α_{local} input is important because it is used in the color combine unit.

The following table describes how α_{local} and output alpha are computed based on the mode:

Table 13.2 Alpha combine unit modes.

mode	*output*	α_{local}
GR_ALPHASOURCE_CC_ALPHA	*constant color* α♦	*constant color* α♦
GR_ALPHASOURCE_ITERATED_ALPHA	*iterated vertex* α	*iterated vertex* α
GR_ALPHASOURCE_TEXTURE_ALPHA	*texture* α♠	*none*
GR_ALPHASOURCE_TEXTURE_ALPHA_TIMES_ITERATED_ALPHA	*texture* α * *iterated* α♠	*iterated vertex* α

♦ *Constant color* α is the value passed to **grConstantColorValue()**.

♠ If *texture* has no alpha component, *texture* α is 255.

A Higher Level Texture Combine Function

Configuring the Glide texture pipeline consists of setting up a mipmap source and configuring the texture combine function on each TMU. Mipmap sources are established by downloading mipmaps and naming them as the current texel source.

Chapter 9 presented a collection of Glide functions that configure the texture combine units; Chapter 10 talked about managing texture memory and downloading mipmaps. Most of the memory management details were left to the application.

The Glide Utilities Library includes a set of higher level routines that configure the texture combine unit on a functional level and that provides increased memory management functionality.

guTexCombineFunction() specifies the function used when combining textures on *tmu* with incoming textures from the neighboring TMU. Texture combining operations allow for interesting effects such as detail and projected texturing as well as the trilinear filtering of LOD blending.

The following table describes the available texture combine functions and their effects. c_{local} represents the color components generated by indexing and filtering from the mipmap stored on *tmu* and c_{other} represents the incoming color components from the neighboring TMU. Typically, the texture combine function on the neighboring TMU operates in `GR_TEXTURECOMBINE_DECAL` (just pass the texel through) mode.

Table 13.3 Texture combine functions.

texture combine function	*result*	*effect*
GR_TEXTURECOMBINE_ZERO	*0*	`0x00` per component
GR_TEXTURECOMBINE_DECAL	c_{local}	decal texture

GR_TEXTURECOMBINE_OTHER	c_{other}	pass through
GR_TEXTURECOMBINE_ADD	$c_{other} + c_{local}$	additive texture
GR_TEXTURECOMBINE_MULTIPLY	$c_{other} + c_{local}$	modulated texture
GR_TEXTURECOMBINE_SUBTRACT	c_{other} ˇ c_{local}	subtractive texture
GR_TEXTURECOMBINE_DETAIL	*blend* (c_{other}, c_{local})	composite textures with composite on selected TMU
GR_TEXTURECOMBINE_DETAIL_OTHER	*blend* (c_{other}, c_{local})	composite textures with composite on neighboring TMU
GR_TEXTURECOMBINE_TRILINEAR_ODD	*blend* (c_{other}, c_{local})	LOD blended textures with odd levels on selected TMU
GR_TEXTURECOMBINE_TRILINEAR_EVEN	*blend* (c_{other}, c_{local})	LOD blended textures with even levels on selected TMU
GR_TEXTURECOMBINE_ONE	*255*	`0xFF` per component

---

guTexCombineFunction() also keeps track of which TMUs require texture coordinates for the rendering routines. Many combine functions that simultaneously use both c_{local} and c_{other} can be computed with two passes on a single TMU system by using the frame buffer to store intermediate results and the alpha blender to combine the two partial results.

Allocating Texture Memory

[redacted]

Before downloading a mipmap, an application must first allocate some memory for it. This is done using **guTexAllocateMemory()**, which returns a handle to an allocated mipmap storage area within a specific TMU.

[redacted]

[redacted]

The arguments are similar to those for routines in Chapter 10; refer to for a summary of the possible values. The memory will be allocated on *tmu*. The *height* and *width* of the largest mipmap level are specified, and, in combination with *oddEvenMask*, *format*, *smallLOD*, *largeLOD*, and *aspect ratio*, are used to compute the amount of memory required.

oddEvenMask is used to selectively download LOD levels when LOD blending is to be used. Correct usage is to allocate and download the even levels onto one TMU, and the odd levels onto another, both with the *LODblend* parameter set to `FXTRUE`. Then the texture combine mode for the lower numbered TMU is set to `GR_TEXTURECOMBINE_TRILINEAR_ODD` or `GR_TEXTURECOMBINE_TRILINEAR_EVEN` depending on whether the odd levels or the even levels were downloaded to it.

All the parameters that define the texel selection process when this mipmap is downloaded are also included in the call: *mipmapMode*, *sClampMode*, *tClampMode, minFilterMode, magFilterMode, LODbias*, and *LODblend.* See Chapter 9 for more details.

If memory could not be allocated, a value of `GR_NULL_MIPMAP_HANDLE` is returned. Mipmap handles cannot be shared across multiple Voodoo Graphics subsystems, i.e. a texture must be allocated and downloaded multiple times if an application wishes to use it across multiple Voodoo Graphics subsystems.

The amount of unallocated texture memory can be determined with the Glide function **guTexMemQueryAvail()**, which returns the amount of unallocated memory in bytes for the specified TMU in the currently active Voodoo Graphics subsystem. Only memory that was allocated with **guTexAllocateMemory()** is taken into account.

[redacted]

Resetting Texture Memory

Instead of elaborate memory recovery mechanisms, it is sometimes easier to download textures until there is no more room, then clear texture memory and start over with new textures. For example, a game with obvious breaks between levels can avoid complex texture memory management by clearing out texture memory whenever the player enters a new ˆlevel˜. To reset the texture memory call **guTexMemReset()**. After **guTexMemReset()** is called all texture map handles associated with the reset Voodoo Graphics subsystem are invalidated.

[redacted]

guTexMemReset() frees up all allocated texture memory. This allows for a simple form of texture memory management; all texture memory is allocated at once, then freed en masse. While simple, this form of memory management prevents some of the complexity associated with standard memory management techniques, such as garbage collection and memory fragmentation and compaction.

Downloading Textures

After the memory has been allocated, **guTexDownloadMipMap()** and **guTexDownloadMipMapLevel()** can be used to download complete mipmaps or individual levels, respectively.

[redacted]

guTexDownloadMipMap() downloads an entire mipmap to an area of texture memory previously allocated with **guTexAllocateMemory()** and pointed to by *mipmapID*. The data to be downloaded must have the pixel format and aspect ratio associated with *mipmapID*. If the texture uses an NCC decompression table , *NCCtable* is a pointer to it. This is only valid for 8−bit compressed textures loaded with **gu3dfLoad()**.

[redacted]

guTexDownloadMipMapLevel() downloads a single mipmap level within a mipmap to an area of texture memory previously allocated with **guTexAllocateMemory()** and updates **src* to point to the next mipmap level. The data to be downloaded must be of the same pixel format and aspect ratio as *mipmapID* and must be of the correct size for *lod*, the LOD that is being downloaded.

guTexSource makes current a mipmap for the TMU it resides on. Each TMU has one current mipmap. In systems with multiple TMUs, multiple mipmap sources are combined by the texture combine function and the output of the final combine is passed on to the pixel shading pipeline. By default, all the TMUs have null texture handles associated with them.

[redacted]

Changing Mipmap Attributes

When a mipmap is made current, all of its attributes take effect. Some of these attributes can be temporarily overridden with **grTexClampMode()**, **grTexFilterMode()**, **grTexLodBiasValue()**, and **grTexMipMapMode()**. Note, however, that these routines do not change the mipmap´s attribute, only the current mode of the rendering hardware.

guTexChangeAttributes() changes some of the attributes of a mipmap. This allows a section of texture memory to be reused without resetting all of texture memory. Upon success, FXTRUE is returned, else FXFALSE is returned.

For projected textures, the clamp modes, *sClampMode* and *tClampMode*, should always be set to `GR_TEXTURECLAMP_CLAMP`.

[redacted]

Retrieving Information about Mipmaps

guTexGetCurrentMipMap() returns the handle of the currently active mipmap on a selected TMU. Each TMU has one currently active mipmap. Mipmaps are made current with **guTexSource()**.

[redacted]

guTexGetMipMapInfo() allows an application to retrieve information about a mipmap.

[redacted]

Chapter .

Chapter 15 Programming Tips and Techniques

In This Chapter

This chapter is a collection of short programming tips. You will read about:

snapping vertex coordinates to a grid to avoid anomalies when rendering very small triangles

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

Floating Point Vertex Snapping and Area Calculations

Glide´s rasterization primitives, such as **grDrawTriangle()**, perform area calculations in order to determine parameter gradients, facedness, etc. A potential inconsistency may arise between Glide´s and the Voodoo Graphics hardware´s perception of area and vertex values when Glide´s floating point values change upon conversion to the hardware´s fixed point <12.4> representation. This typically only occurs with very small triangles, however in certain cases this may cause the hardware to begin rendering outside of a triangle and in the wrong direction, leading to anomalies such as long horizontal stripes on the screen and very long rendering times.

To avoid this problem, software should ˆsnap˜ vertices to .0625 resolution before passing them to Glide, but after they have been projected. On most processors, snapping can be performed by adding a large number (2^{19}) to the vertices then subtracting this same large number, which in effect normalizes the value to a known range and precision.

Example . Snapping coordinates to .0625 resolution.

```
const float vertex_snapper = ( float ) ( 3L << 18 );

vertex.x += vertex_snapper;
vertex.x -= vertex_snapper;
vertex.y += vertex_snapper;
vertex.y -= vertex_snapper;
```

The only caveat is that an Intel FPU must be configured to operate in 24−bit precision so that temporaries are not immediately promoted to a higher precision internal to the

FPU. This is accomplished by masking off the precision control bits in the floating point control world. The assembly code in performs this function.

Example . Masking off precision control bits on Intel processors.

```
finit                   ; initialize the FPU
fwait                   ; wait for operation to complete
fstcw [memvar]          ; store FPU control word to memvar
fwait                   ; wait for operation to complete
mov eax, [memvar]       ; move memvar to a register
and eax, 0fffffcffh     ; mask off precision bits to set to 24-bit
precision
mov [memvar], eax       ; save control word to memory
fldcw [memvar]          ; load control word back to FPU
fwait                   ; wait for operation to complete
```

The same effect can be realized by multiplying by 16, casting to a `long` to truncate off trailing bits, then dividing by 16.0 to reconvert back to floating point, as shown in . This is not an ideal solution, but it is portable and simple to implement. Note that this solution is very inefficient and should never be used in production code.

Example . A portable way to snap coordinates to .0625 resolution.
Note that this solution is very inefficient and should never be used in production code.

```
long tmp;

tmp = vertex.x * 16;     // increase by 4 bits, truncate off the rest
tmp = vertex.y * 16;     // increase by 4 bits, truncate off the rest
vertex.x = tmp / 16.0;   // remove extra 4 bits, convert back to float
vertex.y = tmp / 16.0;   // remove extra 4 bits, convert back to float
```

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 ˆmaterials˜. 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 Voodoo 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 ˆpoke through˜ may occur as a result of the compression of precision close to the viewer.

State Coherency and Contention Between Processes

Neither the Voodoo 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 Voodoo Graphics hardware must maintain state coherency and perform context management manually using some form of mutual exclusion management.

A Sample Program

```
/*
** Copyright (c) 1995, 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>

GrHwConfiguration hwconfig;

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();

   if ( !grSstQueryHardware( &hwconfig ) )
      grErrorSetCallback( "main: grSstQueryHardware failed!", FXTRUE );


   /* Select SST 0 and open up the hardware */
   grSstSelect( 0 );
   if ( !grSstOpen( GR_RESOLUTION_640x480, GR_REFRESH_60Hz,
      GR_COLORFORMAT_ABGR, GR_ORIGIN_LOWER_LEFT,
      GR_SMOOTHING_ENABLE, 2 ) )
      grErrorSetCallback( "main: grSstOpen failed!", FXTRUE );


   while ( 1 ) {
      GrVertex 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	*Which mipmaps will be loaded. Even LODs are* GR_LOD_256, GR_LOD_64, GR_LOD_16, GR_LOD_4, *and* GR_LOD_1. *Odd LODs are* GR_LOD_128, GR_LOD_32, GR_LOD_8, *and* GR_LOD_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	*Set alpha blending factors.*
GrAspectRatio_t	*aspectRatio*	GR_ASPECT_8x1 GR_ASPECT_4x1 GR_ASPECT_2x1 GR_ASPECT_1x1 GR_ASPECT_1x2 GR_ASPECT_1x4 GR_ASPECT_1x8	*The constant 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	*Select 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	*Choose 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	*Choose 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	*Choose 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	*Choose an alpha or RGB source for the ˆother˜ 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	*What kind of depth buffer will you have?*
GrDitherMode_t	*mode*	GR_DITHER_DISABLE GR_DITHER_2x2 GR_DITHER_4x4	*Wanna dither?*
GrFogMode_t	*mode*	GR_FOG_DISABLE GR_FOG_WITH_ITERATED_ALPHA GR_FOG_WITH_TABLE GR_FOG_MULT2 GR_FOG_ADD2	*Enable and characterize fogging.*
GrLfbBypassMode_t	*mode*	GR_LFBBYPASS_DISABLE GR_LFBBYPASS_ENABLE	*Bypass the pixel pipeline on LFB writes?*
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_256 GR_LOD_128 GR_LOD_64 GR_LOD_32 GR_LOD_16 GR_LOD_8 GR_LOD_4 GR_LOD_2 GR_LOD_1	*The number in the constant is the largest of the texture. The aspect ratio determines the smaller dimension.*
GrMipMapMode_t	*mipmapMode* *mode*	GR_MIPMAP_DISABLE GR_MIPMAP_NEAREST GR_MIPMAP_NEAREST_DITHER	*The kind of mipmapping to perform.*

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.*
GrNCCTable_t	*table*	GR_NCCTABLE_NCC0 GR_NCCTABLE_NCC1	*Choose an NCC table for use in decompressing texels.*

GrOriginLocation_t	*locateOrigin* *origin*	GR_ORIGIN_UPPER_LEFT GR_ORIGIN_LOWER_LEFT	*Set location of origin.*
GrSmoothingMode_t	*smoothMode*	GR_SMOOTHING_DISABLE GR_SMOOTHING_ENABLE	*Enable/disable 24−smoothing filter.*
GrTexBaseRange_t	*range*	GR_TEXBASE_256 GR_TEXBASE_128 GR_TEXBASE_64 GR_TEXBASE_32_TO_1	*Specify 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	*Choose 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 for a description of the texture format s.*

The types below are used in three Glide Utilities Library functions that presents 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	*Choose 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	*Choose 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	*Choose 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 ˆblue 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 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..255] 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.

A

B

C

D

E

F

G

H

I

L

M

N

O

P

R

S

T

W