3D RAGE™ Windows® 95 Programmer's Guide

Technical Reference Manuals

P/N: SDK-C02700 Rev. 1.30

ATI Technologies Inc.
33 Commerce Valley Drive East
Thornhill, Ontario
Canada L3T 7N6

Developer Support:	905-882-2600 ext.6000
Offices:	905-882-2600
Fax:	905-882-2620
BBS:	905-764-9404

P/N: SDK-C02700

Revision: 1.30

Record of Revisions

Release	Date	Description of Changes
0.01	Jan. 96	Preliminary Release.
0.02	Jan. 96	Added Preface; update Intro.
0.03	Feb. 96	Added SML Appendix; general updates.
0.04	Mar. 96	General updates.
0.05	Apr. 96	Typographical corrections; PDF created.
0.06	Jul. 96	Added “Programming with ATI3DCIF” chapter; general updates.
1.00	Oct. 96	Added “3D RAGE II ATI3DCIF Programming” chapter; general updates for 3D RAGE II.
1.10	Apr. 97	Added “RAGE Pro ATI3DCIF Programming” chapter; general updates for RAGE PRO additions; Removed Appendix A and DOS init function.
1.20	May 97	General updates for (RAGE PRO): VQ compression and u32CIFCaps2 flags.
1.30	Jul. 97	Added C3D_CODEBOOKENTRY and C3D_TLVERTEX plus general updates

System Publications Index

Technical Reference Manuals

- ATI 3D RAGE Windows 95 Programmer’s Guide
  (SDK-C02700)

Table of Contents

Preface

ATI Technologies Inc. is pleased to present this Software Developers Kit for the 3D RAGE, ATI's new 2D, 3D and video graphics accelerator. Current market acceptance indicates that there will be millions of 3D RAGE-powered PCs and Power Macs installed in 1996-7. ATI is a unique vendor in this marketplace in that we design and manufacture our own chips and boards, as well as develop the accompanying software. This "one-stop shopping" benefit is one of the many reasons that ATI is a major chip supplier to OEMs and motherboard builders, as well as a being a preferred vendor of retail and OEM add-in graphics boards. Consequently, ATI's products are actively supported by the development community.

This brief section provides company background and a review of ATI's product line. The 3D RAGE will be described in detail in the next section.

ATI Company Background

ATI Technologies Inc. was founded in 1986 and now has 700 employees worldwide, with offices in Toronto, Munich, San Jose and Boston. ATI's fiscal year 1995 revenues were $359 million (Canadian, over $250 million U.S.), an increase of 55% over the previous year. For the 1996 fiscal year quarter ending November 30, 1995, ATI's revenues were almost $120 million (Canadian), another record quarter.

ATI's current product line is centered around the *mach64* family of graphics controllers, including:

- ATI264-CT, a value priced 2D graphics accelerator
- ATI-264VT, a richly featured video and 2D graphics accelerator
- 3D RAGE (ATI-264GT) an integrated 2D, 3D and video graphics accelerator

The entire *mach64* family is pin-compatible, allowing OEMs to easily upgrade motherboard implementations from one chip to the next. In addition to the graphics accelerators, ATI's product line now includes multimedia products offering video-in, video conferencing, TV tuner capability and MPEG decoding. This broad product offering and upgrade capability has solidified relationships with PC OEMs world wide. ATI expects to ship over 8 million graphics chips in 1996. With additional fabrication capacity coming on-line by the end of 1996, ATI is planning to ship 15 million chips in 1997.

ATI Developer Support

ATI appreciates the support our products have received from the development community to date and we continue to improve our developer support program. Our headquarters-based Developer Relations group is adding staff and resources. In addition, ATI will be setting up regional Developer Conferences to promote support for the 3D RAGE and get additional feedback from the development community. Please contact the Developer Relations group so that your input to our product planning process can continue. Developer Relations may be reached at:

Developer Relations

ATI Technolgies Inc.
33 Commerce Valley Drive East
Thornhill, Ontario
Canada, L3T 7N6
Tel: 905-882-2600, x 6000, 8:30 a.m. to 6:00 p.m. Eastern Time
Fax: 905-882-9339
Email: devrel@atitech.ca

Introduction

The ATI 3D RAGE family of chips are highly integrated graphics accelerators with superior support for 3D and motion video. It is ideal for gaming, consumer PCs, and multimedia workstations. The 3D RAGE will lead the implementation of 3D functionality across the ATI product line over the next two years.

The core hardware accelerated 3D features of the 3D RAGE include:

- seven texture filtering modes
- perspectively correct texture mapping
- video textures
- palettized textures (in the 3D RAGE II and later)
- Gouraud shading
- alpha blending
- fog effects
- dithering of limited colors (8 and 16 bpp)
- 16 bit z-buffer (in the 3D RAGE II and later)
- complete 2D and video feature set
- texture compositing (RAGE Pro and later)
- vector quantization (VQ) compression (RAGE Pro and later)

This rich feature set was determined by researching the needs of the development community. These are the most common 3D features utilized in 3D games, web content and other applications. ATI also reviewed the needs of the major 3D API (Application Programming Interface) developers, such as Microsoft with their RealityLab and Direct3D APIs, Apple Computer with their QuickDraw3D API, Intel with their 3DR API, and others.

For accessing accelerated 3D features on the 3D RAGE under Windows 95, ATI provides a proprietary interface called ATI3DCIF (ATI 3D C InterFace). This low-level interface provides a set of functions for executing and managing 3D rendering operations such as primitive drawing, texture mapping, color shading, and color blending. Under Windows 95, applications may use Microsoft's DirectDraw to create the double buffer and texture map surfaces required by ATI3DCIF.

ATI3DCIF is a 3D rendering interface. It does not perform 3D geometric transformations. It also does not implement any 2D blitting operations, as these services are provided by the GDI and DirectDraw under Windows 95.

NOTE: ATI3DCIF cannot be used with Direct3D within the same application (although ATI3DCIF may be, and usually is, used with DirectDraw).

3D RAGE PRO

The 3D RAGE PRO is the third generation chipset in the 3D RAGE product line. It is a highly integrated 64-bit graphics accelerator featuring full 2D acceleration, TV quality motion video and superior 3D acceleration. It incorporates comprehensive support for Intel's Accelerated Graphics Port (AGP) including 66 or 133 MHz fully pipelined operation with sideband.

The 3D RAGE PRO provides the following features:

General Features

- PCI version 2.1 with full bus mastering and scatter / gather support.
- Bi-endian support for compliance on a variety of processor platforms.
- Fast response to host commands:
  - 128-level command FIFO
  - 32-bit wide memory-mapped registers
  - Programmable flat or paged memory model with linear frame buffer access
- Triple 8-bit palette DAC with gamma correction for true WYSIWYG color. Pixel rates up to 220MHz.
- Supports DRAM, EDO DRAM, SDRAM and SGRAM at up to 100MHz memory clock providing bandwidths up to 800MB/sec across a 64-bit interface.
- Supports WRAM and 128-bit external DAC for ultra-high end configurations
- Flexible graphics memory configurations: 1MB up to 8MB; 256Kx4/8/16/32, 512Kx32; dual CAS.
- Memory upgrade via industry-standard SGRAM SO-DIMM, for reduced board area and higher memory speeds.
- DDC1 and DDC2B+ for plug and play monitors.
- Power management for full VESA DPMS and EPA Energy Star compliance.
- Integrated hardware diagnostic tests performed automatically upon initialization.
- High quality components through built-in SCAN, CRC and chip diagnostics
- Single chip solution in 0.35mm, 3.3V CMOS technology, with multiple package options.
- Comprehensive HDKs, SDKs and utilities augmented by full engineering support.
- Complete local language support (contact ATI for current list).

3D Acceleration

- Integrated 1 million triangles set-up engine, which reduces CPU and bus bandwidth requirements and dramatically improves performance of small 3D primitives
- 4KB on-chip texture cache, which dramatically improves large triangle performance.
- Complete 3D primitive support: points, lines, triangles, and quadrilaterals in lists and strips
- Hidden surface removal using 16-bit z-buffering

- Edge anti-aliasing
- Sub-pixel and sub-texel accuracy
- Gouraud, specular, flat, and solid shaded polygons
- Perspectively correct mip-mapped texturing with chroma-key support
- Single pass bi- and tri-linear texture filtering for vastly improved bi- and tri-linear performance
- Texture compositing
- Special effects such as alpha blending, fog, video textures, texture lighting, reflections, shadows, spotlights, LOD biasing and texture morphing
- Dithering support in 8bpp and 16bpp for near 24bpp quality in less memory
- Texture compression of up to 8:1 using vector quantization
- Filtered horizontal/vertical RGB scaler for high-quality stretching of 3D display
- Extensive 3D mode support: RGBA32, RGBA16, RGB16, RGB8, ARGB4444, YUV444, YUV422
- Compressed texture modes: YUV422, CLUT4 (CI4), CLUT8 (CI8), VQ

2D Acceleration

- Hardware acceleration of Bitblt, Line Draw, Polygon / Rectangle Fill, Bit Masking, Monochrome Expansion, Panning/Scrolling, Scissoring, full ROP support and h/w cursor (up to 64x64x2).
- Game acceleration including support for Microsoft's DirectDraw: Double Buffering, Virtual Sprites, Transparent Blit, Masked Blit and Context Chaining.
- Acceleration in 4/8/16/24/32 bpp modes. Packed pixel support (24bpp) enables true color in 1MB configurations.

Motion Video Acceleration

- Smooth video scaling and enhanced YUV to RGB color space conversion for full-screen / full-speed video playback.
- Front and back end scalers support multi-stream video for video conferencing and other applications.
- Filtered horizontal/vertical, up/down, scaling enhances playback quality.
- Enhanced line buffer allows vertical filtering of native MPEG-2 size (720x480) images.
- DVD / MPEG-2 decode assist provides dramatically improved frame rate without incurring cost of dedicated hardware.
- Special filter circuitry eliminates video artifacts caused by displaying interlaced video on non-interlaced displays.
- Intercast capable video capture interface.
- Bi-directional bus mastering engine with planar YUV to packed format converter for superior MPEG2 and video conferencing.
- Hardware mirroring for flipping video images in video conferencing systems.
- Supports graphics and video keying for effective overlay of video and graphics.
- YUV to RGB color space converter with support for both packed and planar YUV:
  - YUV422, YUV410, YUV420
  - RGB32, RGB16/15, RGB8, Mono

ATI Multimedia Channel

- 16-bit, bi-directional video port allows direct connection to popular video upgrades such as:

- video capture / video conferencing
- Hardware MPEG2 / DVD player
- TV tuner with Intercast support

Manual Contents

This manual serves as a programming guide to the ATI3DCIF. It is composed of the following chapters:

Chapter 1: Overview

This chapter presents an overview of the ATI3DCIF programming model. It describes the main components of the 3D graphics pipeline on the 3D RAGE. The chapter also describes the function groups in the interface and how they are used within the programming model.

Chapter 2: Programming with ATI3DCIF

This chapter describes the basic operations for setting up and using ATI3DCIF.

Chapter 3: 3D RAGE II ATI3DCIF Programming

This chapter covers programming issues which only apply to the ATI 3D RAGE II graphics accelerator.

Chapter 4: RAGE Pro ATI3DCIF Programming

This chapter covers programming issues which only apply to the ATI RAGE Pro graphics accelerator.

Chapter 5: ATI3DCIF API Reference

This chapter provides a comprehensive reference of the ATI3DCIF functions and data types.

Chapter 6: 3D RAGE /ATI3DCIF Porting and Performance Notes

This chapter covers ATI3DCIF and 3D RAGE porting and performance issues. It provides guidelines for porting existing applications to the 3D RAGE and discusses factors which affect image quality and performance.

SDK System Requirements

The following are the system requirements for this SDK:

- 486/Pentium system
- 32-bit PCI Local Bus 2.1
- 3D RAGE accelerator graphics board
- Microsoft DirectX Games Development kit (DirectX 2 for RAGESDK Beta 6 and greater)
- Microsoft Visual C/C++ 4.0 (for Windows 95 example projects)

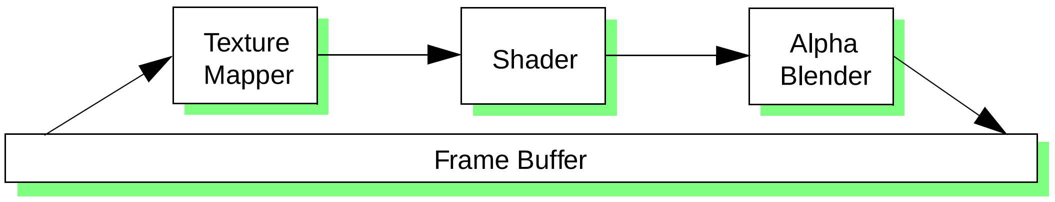
Chapter 1
Overview

Introduction

ATI3DCIF is an ATI proprietary programming interface that exposes the 3D hardware rasterization functionality of the ATI 3D RAGE graphics accelerator. The focus of ATI3DCIF is to provide client applications with an interface for accelerating 3D rendering operations. Therefore, ATI3DCIF is a rendering interface, and as such, does not perform any 3D geometric transformations, 2D rendering operations, or memory management. Applications may use Microsoft’s DirectX or the GDI to perform memory management and 2D blitting operations.

3D Drawing Operations

ATI3DCIF accelerates the drawing of quadrangles, triangles, and lines. The 3D RAGE supports a rich set of orthogonal 3D drawing operations for texture mapping, shading, and alpha blending. The hardware can be conceived as a three-stage graphics pipeline with a Texture Mapper, Shader, and Alpha Blender stage, where each stage is dedicated to carrying out one of the rendering operations.



When all three components are on, the Shader is applied to texels fetched from the frame buffer by the Texture Mapper to simulate texture lighting effects, and the resulting pixels are passed on to the Alpha Blender. The operation of each element is described below.

Texture Mapper

Texels from the frame buffer are read into the pixel pipeline and filtered by one of the programmable filtering algorithms. The resulting texel may optionally be compared against a chroma key. If the texel matches the chroma key, it is discarded and the destination is left unaltere; otherwise, the texel is passed on to the Shader stage of the pipe.

Shader

The Shader generates colors that can be applied to texels supplied by the Texture Mapper. If the Texture Mapper is off, these colors are passed to the Alpha Blender stage unaltered. The mechanism used to generate colors by the Shader is programmable, and allows for different shading methods such as solid, flat, and Gouraud shading. If texture mapping is on, the method used to mix texels with the colors generated by the Shader (texture lighting) is also programmable. The resulting pixels from this stage are passed on to the Alpha Blending stage.

Alpha Blender

The alpha blender simply blends the output of the Shader stage with the pixel at the destination address in the frame buffer. Various source and destination alpha blending methods are available on the 3D RAGE. Alpha blending may be used to achieve special effects such as translucency and transparency.

The ATI3DCIF interface is composed of the following functional groups:

- **Library Initialization functions**
- **Texture Management functions**
- **Context Management and Rendering functions**

Library Initialization functions allow the client application to load and unload the ATI3DCIF module. Library initialization must be performed prior to using any other library functions. and library termination must be performed after the client applications has finished using the interface to free resources.

Texture Management functions are used by ATI3DCIF to manage the use of textures. Textures are registered with ATI3DCIF and given a unique handle. This handles is used to select the texture for rasterization during the texture mapping process.

Context Management functions allow the client application to create and modify a rendering context. A rendering context represents the collection of rendering states currently set for the 3D RAGE accelerator. The Rendering functions allow the application to render line, triangle, or quadrilateral primitives in primitive lists or strips.

Chapter 2
Programming with ATI3DCIF

Basic ATI3DCIF Operations

This section describes basic operations for setting up and using ATI3DCIF. It covers ATI3DCIF initialization and describes the fundamentals of the ATI3DCIF rendering model, introducing concepts such as the rendering context and 3D rendering blocks. Finally, it demonstrates how an application can retrieve ATI3DCIF module and graphics subsystem information.

Initializing ATI3DCIF

The first step to use ATI3DCIF is to load and initialize the ATI3DCIF.DLL module. This is done by calling *ATI3DCIF_Init*. This DLL must be loaded before any ATI3DCIF functions are called. Before the application terminates, it must unload ATI3DCIF.DLL by calling *ATI3DCIF_Term*.

Creating a Rendering Context

After the ATI3DCIF module is loaded, the application must create a rendering context to use the 3D features of the 3D RAGE. A rendering context is created by calling *ATI3DCIF_ContextCreate*. When a rendering context is created, the RAGE's 3D rendering components are configured into a default state. This rendering context is identified by a unique handle, and the application must pass this handle to ATI3DCIF functions whenever referencing or modifying the rendering context. For example, to change the context shading mode, the application must call *ATI3DCIF_ContextSetState* with the context handle as the first argument, and the appropriate state modification flag and data as the other arguments.

When the application no longer needs the rendering context, the former may destroy the latter by calling *ATI3DCIF_ContextDestroy*. An application must destroy the rendering context before terminating, and before calling *ATI3DCIF_Term* to unload the ATI3DCI module.

Rendering 3D Primitives

When an application is ready to render 3D primitives, it must set the 3D RAGE into a 3D operating mode. While in this mode, the application will not be able to perform 2D operations, such as blitting, rectangle fills, or page flipping. The application should perform all 3D rendering operations while in 3D mode and switch back to 2D mode to perform 2D operations.

To switch to 3D mode, the application must call *ATI3DCIF_RenderBegin*. While in this mode, primitive lists and strips may be rendered by calling *ATI3DCIF_RenderPrimList* or *ATI3DCIF_RenderPrimStrip* respectively. To switch back to 2D rendering mode after rendering all the 3D primitives, the application must call *ATI3DCIF_RenderEnd*.

When switching between 2D and 3D modes, *ATI3DCIF_RenderBegin* saves the state of the 2D engine, and *ATI3DCIF_RenderEnd* restores the state of the 2D engine. To minimize the overhead incurred while saving and restoring, the applications should minimize the number of *ATI3DCIF_RenderBegin* – *ATI3DCIF_RenderEnd* blocks (3D rendering blocks) in each frame update. Ideally, there should be only one 3D rendering block per frame update, and all 3D primitive lists or strips should be rendered within this block. To accomplish this, the application may need to reorganize the order in which rendering operations are performed. 2D and 3D operations should be separated, and all 3D operations should be performed within one 3D rendering block whenever possible.

The following example demonstrates the recommended method for mixing 2D and 3D rendering operations during frame updates:

`Example 1: recommended`

```
// 2D rendering operations (e.g. background rectangle fills, bitmap blits, etc.)
...
// now switch to 3D mode and draw the 3D primitives. hRC is a rendering
// context created by calling ATI3DCIF_ContextCreate
ATI3DCIF_RenderBegin (hRC);

// render 3D primitives
ATI3DCIF_RenderPrimList (PrimList1, PrimList1NumVerts);

ATI3DCIF_RenderPrimList (PrimList2, PrimList2NumVerts);

ATI3DCIF_RenderPrimStrip (PrimStrip1, PrimStrip1NumVerts);

...

// now switch back to 2D mode
ATI3DCIF_RenderEnd ();

// perform 2D operations (such as page flipping, etc.) ...
```

Here are a couple of examples showing non-optimal and incorrect ways to perform the same operations:

`Example 2: not recommended`

```
// This example shows unnecessary ATI3DCIF_RenderBegin and ATI3DCIF_RenderEnd
// calls which incur unwanted overhead. Since no 2D operations are being
// performed in between the primitive rendering calls, all these calls can be
// lumped within one ATI3DCIF_RenderBegin - ATI3DCIF_RenderEnd block.

// 2D rendering operations (e.g. background rectangle fills, bitmap blits, etc.)
...
ATI3DCIF_RenderBegin (hRC);
ATI3DCIF_RenderPrimList (PrimList1, PrimList1NumVerts);
ATI3DCIF_RenderEnd ();

ATI3DCIF_RenderBegin (hRC);
ATI3DCIF_RenderPrimList (PrimList2, PrimList2NumVerts);
ATI3DCIF_RenderEnd ();
```

```
ATI3DCIF_RenderBegin (hRC);
ATI3DCIF_RenderPrimStrip (PrimStrip1, PrimStrip1NumVerts);
ATI3DCIF_RenderEnd ();

// perform 2D operations (such as page flipping, etc.) ...
```

Example 3: not recommended

```
// This example demonstrates a situation which may be optimized by reordering
// 2D and 3D rendering operations such that all 3D operation are performed
// within one 3D rendering block, as in Example 1.

// 2D rendering operation (e.g. blit, color fill, etc.)
...
ATI3DCIF_RenderBegin (hRC);
ATI3DCIF_RenderPrimList (PrimList1, PrimList1NumVerts);
ATI3DCIF_RenderEnd ();

// 2D rendering operation (e.g. blit, color fill, etc.)
...
ATI3DCIF_RenderBegin (hRC);
ATI3DCIF_RenderPrimList (PrimList2, PrimList2NumVerts);
ATI3DCIF_RenderEnd ();

// 2D rendering operation (e.g. blit, color fill, etc.)
...
ATI3DCIF_RenderBegin (hRC);
ATI3DCIF_RenderPrimStrip (PrimStrip1, PrimStrip1NumVerts);
ATI3DCIF_RenderEnd ();
```

Example 4: incorrect

```
// This example demonstrates the incorrect way to mix 2D and 3D rendering
// operations.

ATI3DCIF_RenderBegin (hRC);

ATI3DCIF_RenderPrimList (PrimList1, PrimList1NumVerts);

// perform a 2D operation such as a rect fill or 2D blit
...// ERROR: should not perform 2D operations within a 3D rendering
   // block!!!
   // If doing the 2D operation at this point is unavoidable, exit the
   // 3D rendering block by calling ATI3DCIF_RenderEnd, perform the 2D
   // operation, and begin a new 3D rendering block by calling
   // ATI3DCIF_RenderBegin

ATI3DCIF_RenderPrimList (PrimList2, PrimList2NumVerts);

ATI3DCIF_RenderPrimStrip (PrimStrip1, PrimStrip1NumVerts);

...
ATI3DCIF_RenderEnd ();
```

Modifying the Rendering Context

Rendering states within the rendering context may be modified by calling *ATI3DCIF_ContextSetState*. The first argument to this function is the handle of the rendering context which will be modified. The second argument is a *C3D_ERSID* enumeration type specifying which rendering state or property will be modified. The last argument is a C3D_UINT32 which has a different meaning depending on which state is being modified. Typically, it is a pointer to an ATI3DCIF enumeration or structure type specifying the new state or property.

ATI3DCIF_ContextSetState needs to save and restore the state of the 2D engine on entry and exit in the same manner as the *ATI3DCIF_RenderBegin* and *ATI3DCIF_RenderEnd* functions. If *ATI3DCIF_ContextSetState* is called outside of a 3D rendering block, it will explicitly save and restore the 2D engine state, and the application will incur overhead. However, if it is called within a 3D rendering block, the save and restore operation will not be performed, and no additional overhead will be incurred. Therefore, the application should attempt to group all *ATI3DCIF_ContextSetState* calls within a 3D rendering block.

The following example shows the recommended method for calling *ATI3DCIF_ContextSetState*:

Example 5: recommended

```
// This example changes the context shading mode from the default smooth
// (gouraud) state to flat

C3D_ESHADE eshade = C3D_ESH_FLAT;

// switch to 3D mode
ATI3DCIF_RenderBegin (hRC);

// change the rendering context shading mode
ATI3DCIF_ContextSetState (hRC, C3D_ERS_SHADE_MODE, &eshade);

// render a primitive list using flat shading
ATI3DCIF_RenderPrimList (PrimList, PrimListNumVerts);

// restore smooth shading
eshade = C3D_ESH_SMOOTH;
ATI3DCIF_ContextSetState (hRC, C3D_ERS_SHADE_MODE, &eshade);

// now switch back to 2D mode
ATI3DCIF_RenderEnd ();
```

Example 6: not recommended

```
// This example also changes the context shading mode from the default smooth
// gouraud) state to flat. However, ATI3DCIF_ContextSetState calls are made
// outside of the 3D rendering block. As a result, the state of the 2D engines
// will be saved and restored with each of these calls in addition to the
// ATI3DCIF_RenderBegin and ATI3DCIF_RenderEnd calls.

C3D_ESHADE eshade = C3D_ESH_FLAT;
```

```
// change the shading mode to flat
ATI3DCIF_ContextSetState (hRC, C3D_ERS_SHADE_MODE, &eshade);

// switch to 3D mode
ATI3DCIF_RenderBegin (hRC);

// render a primitive list using flat shading
ATI3DCIF_RenderPrimList (PrimList, PrimListNumVerts);

// now switch back to 2D mode
ATI3DCIF_RenderEnd ();

// restore smooth shading
eshade = C3D_ESH_SMOOTH;
ATI3DCIF_ContextSetState (hRC, C3D_ERS_SHADE_MODE, &eshade);
```

Getting ATI3DCIF Module and Graphics Subsystem Information

An application may retrieve information about ATI3DCIF capabilities and the graphics subsystem by calling *ATI3DCIF_GetInfo*. This function takes a pointer to a *C3D_3DCIFINFO* structure as its only argument. Upon return, this structure contains information about the module and the graphics subsystem, including the ASIC identification number, ASIC revision, pointer to the frame buffer, and total RAM on the accelerator card.

The syntax for the *C3D_3DCIFINFO* structure is as following:

```
typedef struct {
   C3D_UINT32 u32Size;          // size of struct must be initialized by client
   C3D_UINT32 u32FrameBuffBase; // Host pointer to frame buffer base
   C3D_UINT32 u32OffScreenHeap; // Host pointer to off-screen heap
   C3D_UINT32 u32OffScreenSize; // size of off-screen heap
   C3D_UINT32 u32TotalRAM;      // total amount of RAM on the card
   C3D_UINT32 u32ASICID;        // ASIC Id. code
   C3D_UINT32 u32ASICRevision;  // ASIC revision
   C3D_UINT32 u32CIFCaps1;      // ATI3DCIF capabilities field 1
   C3D_UINT32 u32CIFCaps2;      // ATI3DCIF capabilities field 2
   C3D_UINT32 u32CIFCaps3;      // ATI3DCIF capabilities field 3
   C3D_UINT32 u32CIFCaps4;      // ATI3DCIF capabilities field 4
   C3D_UINT32 u32CIFCaps5;      // ATI3DCIF capabilities field 5
} C3D_3DCIFINFO, * PC3D_3DCIFINFO;
```

u32Size must be set to the size of the *C3D_3DCIFINFO* before calling *ATI3DCIF_GetInfo*; otherwise, *ATI3DCIF_GetInfo* will return a C3D_EC_BADPARAM error. On return, u32FrameBuffBase should contain a host pointer to the base of the frame buffer. u32OffScreenHeap should contain a host pointer to the start of the off-screen video memory heap. u32OffScreenSize should specify the size of the off-screen heap. The total amount of RAM on the card should be in u32TotalRAM. u32ASICID should hold the RAGE ASIC ID code, and u32ASICRevision the ASIC revision code. u32CIFCaps1 should report the ATI3DCIF module's capabilities. u32CIFCaps2 to u32CIFCaps5 are capability fields (u32CIFCaps3 to u32CIFCaps5 are currently reserved for future use).

NOTE: In version 4.02.0217 of ATI3DCIF, the u32CIFCaps member was added to this structure. In version 4.03.0039 of ATI3DCIF, the u32CIFCaps member was renamed u32CIFCaps1, and four more capabilities fields, u32CIFCaps2 to u32CIFCaps5, were added to this structure. In version 4.03.2511 of ATI3DCIF, the u32CIFCaps2 member was defined to support additional capabilities under RAGE Pro. u32CIFCaps3 to u32CIFCaps5 are currently unused and are reserved for future use. The application must ensure that the ATI3DCIF module is version 4.03.0039 or greater to use the u32CIFCaps1 member, and version 4.03.2511 or greater to use the u32CIFCaps2 member. The ATI3DCIF.DLL version number may be determined by right clicking on the file under Windows Explorer, selecting Properties, and clicking on the Version tab. ATI3DCIF.DLL is located in the Windows 95 SYSTEM directory.

The following table lists u32CIFCaps1 flags:

C3D_CAPS1_BASE	base line functionality
C3D_CAPS1_FOG	fog support
C3D_CAPS1_POINT	point primitive support
C3D_CAPS1_RECT	screen-aligned rectangle primitive support
C3D_CAPS1_Z_BUFFER	Z buffer support
C3D_CAPS1_CI4_TMAP	4 bit color index texture support
C3D_CAPS1_CI8_TMAP	8 bit color index texture support
C3D_CAPS1_LOAD_OBJECT	bus-master data loading support
C3D_CAPS1_DITHER_EN	dithering on/off support
C3D_CAPS1_ENH_PERSP	enhanced perspective levels available
C3D_CAPS1_SCISSOR	fixed origin clipping region support
C3D_CAPS1_PROFILE_IF	profile interface available

C3D_CAPS1_BASE represents the base line functionality available in versions 4.02.0217 and earlier of ATI3DCIF. All other capabilities were added after 4.02.0217.

NOTE: Z buffers, CI8 and CI4 textures are only available on the ATI 3D RAGE II graphics accelerator or later. RAGE II programming issues are covered in the next chapter.The **u32CIFCaps2** member was added to the C3D_3DCIFINFO structure to support additional capabilities under RAGE Pro. Please see *Chapter 4, RAGE PRO ATI3DCIF Programming*, for more information on this member.

ATI3DCIF Primitive Types

ATI3DCIF supports line, triangle, rectangle, point, and quadrilateral primitive types. The primitive type specifies the geometric interpretation of a vertex set during rasterization. For example, if the primitive type is set to triangle, subsequent calls to *ATI3DCIF_RenderPrimList* will interpret the list of vertices as triangles and will consume three vertices for each triangle drawn.

The primitive types are represented by the *C3D_EPRIM* enumeration type. The following is a list of its enumeration constants:

C3D_EPRIM_LINE	line primitive
C3D_EPRIM_TRI	triangle list or strip primitive
C3D_EPRIM_QUAD	quadrilateral list or strip primitive
C3D_EPRIM_RECT	screen aligned rectangle strip or list primitive
C3D_EPRIM_POINT	point list or strip primitive

The default primitive type set on rendering context creation is C3D_EPRIM_TRI. To modify the primitive type, call *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_PRIM_TYPE and the third argument set to the address of a *C3D_EPRIM* enumeration specifying the new primitive type.

NOTE: The C3D_EPRIM_RECT and C3D_EPRIM_POINT primitive types are not available on some earlier versions of ATI3DCIF. Applications should call ATI3DCIF_GetInfo and query the u32CIFCaps1 member of the C3D_3DCIFINFO structure to verify the availability of these primitive types.

Vertex Data Formats

ATI3DCIF offers a number of vertex data formats to represent vertices. The choice of which format to use depends on the vertex information needed. For example, if an application does not perform texture mapping, it can represent vertices in a format that does not hold texture coordinate data. Thevertex data type may be changed by calling ATI3DCIF_ContextSetState.

The following structures may be used to represent vertex data:

```
typedef struct {
    C3D_FLOAT32 x, y, z;                // FLOATING point type
} C3D_VF, * C3D_PVF;                    // identified by C3D_EV_VF

typedef struct {
    C3D_FLOAT32 x, y, z;                // FLOATING point type
    C3D_FLOAT32 r, g, b, a;             // identified by C3D_EV_VCF
} C3D_VCF, * C3D_PVCF;

typedef struct {
    C3D_FLOAT32 x, y, z;                // FLOATING point type
    C3D_FLOAT32 s, t, w;                // identified by C3D_EV_VTF
} C3D_VTF, * C3D_PVTF;

typedef struct {
    C3D_FLOAT32 x, y, z;                // FLOATING point type
    C3D_FLOAT32 s, t, w;                // identified by C3D_EV_VTCF
    C3D_FLOAT32 r, g, b, a;
} C3D_VTCF, * C3D_PVTCF;
```

The vertex data type set on context creation is *C3D_VTCF*. The vertex data type may be changed by calling *ATI3DCIF_ContextSetState*. The *C3D_EVERTEX* enumeration type may be used to specify the new vertex data type. *C3D_EVERTEX* includes the following enumeration constants:

C3D_EV_VF	vertex represented by C3D_VF structure
C3D_EV_VCF	vertex represented by C3D_VCF structure
C3D_EV_VTF	vertex represented by C3D_VTF structure
C3D_EV_VTCF	vertex represented by C3D_VTCF structure

The second argument to *ATI3DCIF_ContextSetState* should be set to C3D_ERS_VERTEX_TYPE. The third argument should be set to the address of a *C3D_EVERTEX* enumeration specifying the new vertex data format.

Shading Modes

Primitives can be rendered in solid, flat, or Gouraud shading under ATI3DCIF.

When solid shading is enabled, all primitives are rendered in the rendering context's current solid color. This color is set to black on context creation. It may be changed by calling *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_SOLID_CLR and the third argument set to the address of a *C3D_COLOR* structure specifying the new solid color. The *C3D_COLOR* structure has the following syntax:

```
typedef union {
    struct {
        unsigned r: 8;  // 8 red bits
        unsigned g: 8;  // 8 green bits
        unsigned b: 8;  // 8 blue bits
        unsigned a: 8;  // 8 alpha bits
    };
    C3D_UINT32 u32All;
} C3D_COLOR , * C3D_PCOLOR;
```

Regardless of the pixel bit-depth of the display mode, colors must always be entered in RGBA 8888 format in the *C3D_COLOR* structure. ATI3DCIF will perform the necessary conversions to render in the pixel format of the display mode.

The shading mode may be set by calling *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_SHADE_MODE. The third argument must be set to the address of a *C3D_ESHADE* enumeration type specifying the new shading mode. *C3D_ESHADE* includes the following constants:

C3D_ESH_NONE	the shading color is undefined
C3D_ESH_SOLID	shade using the solid color from the rendering context
C3D_ESH_FLAT	shade using the color of the last vertex in the primitive to color the primitive
C3D_ESH_SMOOTH	shade the primitive by linearly interpolating the color of its vertices from vertex to vertex

Flat shading renders the entire primitive in the color of the last vertex in the primitive. Gouraud shading interpolates the color of each vertex in the primitive from vertex to vertex, resulting in a smooth gradation of color across the face of the primitive. It is the default shading mode set on context creation.

Texture Mapping

The texture mapping operations supported by ATI3DCIF are perspective correction, texture lighting, texture filtering, texture transparency using either a chroma key in the RGB channel or a bit mask in the alpha channel, and mipmapping. All these operations are performed by the 3D RAGE hardware.

There are a number of requirements for texture mapping with the 3D RAGE. Textures must be placed in video memory. For this reason, ATI3DCIF does not support textures stored in system memory. Also, The width and height of each texture must be a power of two and cannot exceed 1024x1024.

Registering a Texture

The first step in using a texture is to load it into a region of video memory. The next step is to register the texture with ATI3DCIF. Registration provides ATI3DCIF with important information about the texture, such as its location in video memory, width, height, pixel format and bit depth, chroma key color, and whether it is a mipmap or not. When registered, the texture is assigned a unique handle which is used to select it during rendering operations.

A texture is registered by calling *ATI3DCIF_TextureReg*. The first argument is a pointer to a *C3D_TMAP* structure containing information about the texture that the client application must fill prior to calling this function. This structure specifies how the RAGE should interpret the texture stored in video memory. The second argument is a pointer to a *C3D_HTX* texture handle which will be set to a unique value by ATI3DCIF if the texture is successfully registered.

The C3D_TMAP structure has the following syntax:

```
typedef struct {
   C3D_UINT32      u32Size;            // size of structure
   C3D_BOOL        bMipMap;            // is texture a mip map
   C3D_PVOID       apvLevels[cu32MAX_TMAP_LEV]; // array of pointers to map
                                                //level
   C3D_UINT32      u32MaxMapXSizeLg2;  // log 2 X size of largest map
   C3D_UINT32      u32MaxMapYSizeLg2;  // log 2 Y size of largest map
   C3D_ETEXFMT     eTexFormat;         // texel format
   C3D_COLOR       clrTexChromaKey;    // specify texel transparency color
   C3D_HTXPAL      htxpalTexPalette;   // texture palette handle
} C3D_TMAP, * C3D_PTMAP;
```

u32Size should be set to the size of the *C3D_TMAP* structure. If this member is not set correctly, *ATI3DCIF_TextureReg* will return the C3D_EC_BADPARAM error code.

bMipMap is a BOOL specifying whether the texture is a mipmap or not. Setting it to TRUE will cause the RAGE to interpret the texture as a mipmap. Setting it to FALSE will cause the RAGE to interpret the texture as an ordinary texture map.

apvLevels is an array of host pointers to the individual maps which compose a mipmap texture. If bMipMap is set to TRUE, apvLevels contains one or more valid elements. The first element (index 0) points at the base map and subsequent elements point at sequentially smaller maps. If bMipMap is set to FALSE, only the first element in the array is valid, and the rest are ignored. When bMipMap is TRUE, the array contains n+1 pointers to mip levels, where n is equal to the log 2 of either the width or height of the base map, whichever is larger.

The following tables show how the apvLevels array should be initialized for a 256x128 and a 512x512 mipmap:

Example for a mipmap with a 256x128 base map

apvLevels[0]	address of 256x128 map
apvLevels[1]	address of 128x64 map
apvLevels[2]	address of 64x32 map
apvLevels[3]	address of 32x16 map
apvLevels[4]	address of 16x8 map
apvLevels[5]	address of 8x4 map
apvLevels[6]	address of 4x2 map
apvLevels[7]	address of 2x1 map
apvLevels[8]	address of 1x1 map

Example for a mipmap with a 512x512 base map

apvLevels[0]	address of 512x512 map
apvLevels[1]	address of 256x256 map
apvLevels[2]	address of 128x128 map
apvLevels[3]	address of 64x64 map
apvLevels[4]	address of 32x32 map
apvLevels[5]	address of 16x16 map
apvLevels[6]	address of 8x8 map
apvLevels[7]	address of 4x4 map
apvLevels[8]	address of 2x2 map
apvLevels[9]	address of 1x1 map

u32MaxMapXSizeLg2 must be set to the log 2 of the width of the largest map in the mipmap if bMipMap is TRUE, or to the log 2 of the texture's width if bMipMap is FALSE. u32MaxMapYSizeLg2 should similarly be set to the log 2 of the height of the largest map in the mipmap if bMipMap is TRUE, or to the log 2 of the texture's height if bMipMap is FALSE.

eTexFormat should be set to a *C3D_ETEXFMT* enumeration specifying the texel format. The RAGE supports the following texel formats:

C3D_ETF_CI4	4 bpp index into palette (pseudo color)
C3D_ETF_CI8	8 bpp index into palette (pseudo color)
C3D_ETF_RGB1555	1 bit Alpha, 5 bits Red, 5 bits Green, 5 bits Blue (16 bits total)
C3D_ETF_RGB565	0 bits Alpha, 5 bits Red, 6 bits Green, 5 bits Blue (16 bits total)
C3D_ETF_RGB8888	8 bits Alpha, 8 bits Red, 8 bits Green, 8 bits Blue (32 bits total)
C3D_ETF_RGB332	0 bits Alpha, 3 bits Red, 3 bits Green, 2 bits Blue (8 bits total)
C3D_ETF_Y8	8 bits Y (8 bits total)
C3D_ETF_YUV422	YUV 422 Packed (YUYV) (16 bits total)
C3D_ETF_RGB4444	4 bits Alpha, 4 bits Red, 4 bits Green, 4 bits Blue (16 bits total)

The YUV formats are especially suited for mapping motion video frames as textures onto polygons, allowing an application to use live or captured video as a texture source.

clrTexChromaKey is a *C3D_COLOR* structure specifying the texture transparency chroma key color in the texture. If texture transparency is enabled, any texel with the chroma key color will not be rendered onto the primitive (that is, the destination pixel at the screen location is not overwritten). Texture transparency is disabled by default on context creation, and must be enabled by calling *ATI3DCIF_ContextSetState*. This will be described in more detail later.

htxpalTexPalette is a *C3D_HTXPAL* handle to the texture palette associated with this texture if the texture format is C3D_ETF_CI4 or C3D_ETF_CI8. These two texture formats and texture palettes are only available on the 3D RAGE II graphics accelerator. 3D RAGE II programming issues are discussed in the next chapter.

The following example demonstrates the complete process for registering an ordinary texture map:

Example 7: Registering an ordinary texture

```
// This example demonstrates how to fill a C3D_TMAP structure to register a
// 128x128 RGB 565 ordinary texture.

C3D_TMAP Tmap;
void *lpTexture;
C3D_HTX htx;

// load a 128x128 texture into a region of off-screen memory and set lpTexture
// to point to this region
...
ZeroMemory (&TMap, sizeof (Tmap));// zero out the structure
TMap.u32Size = sizeof (TMap);
TMap.apvLevels[0] = lpTexure;// address of 128x128 texture
TMap.bMipMap = FALSE;// not a mipmap
TMap.u32MaxMapXSizeLg2 = 7;// log2 of 128
TMap.u32MaxMapYSizeLg2 = 7;// log2 of 128
TMap.eTexFormat = C3D_ETF_RGB565;// texel format is RGB 565
SET_CIF_COLOR (TMap.clrTexChromaKey, 0, 0, 0, 0); // black chroma key
```

```
// register the texture
if (ATI3DCIF_TextureReg (&TMap, &htx) != C3D_EC_OK)
{
// handle error
...
}
```

This example demonstrates how to register a mipmap texture:

Example 8: Registering a mipmap texture

```
// This example demonstrates how fill a C3D_TMAP structure to register a
// 256x128 RGB 565 mipmap.

C3D_TMAP Tmap;
void *lpTexture [cu32MAX_TMAP_LEV];
C3D_HTX htx;

// load the maps in a 256x128 mipmap texture into off-screen memory. Set each
// element of the lpTexture array to point to the address of each map, with
// index 0 pointing to the base map, and sequential indices pointing to
// sequentially smaller maps
...
ZeroMemory (&TMap, sizeof (TMap));
TMap.u32Size = sizeof (TMap);
TMap.apvLevels[0] = lpTexure [0];// address of 256x128 map
TMap.apvLevels[1] = lpTexure [1];// address of 128x64 map
TMap.apvLevels[2] = lpTexure [2];// address of 64x32 map
TMap.apvLevels[3] = lpTexure [3];// address of 32x16 map
TMap.apvLevels[4] = lpTexure [4];// address of 16x8 map
TMap.apvLevels[5] = lpTexure [5];// address of 8x4 map
TMap.apvLevels[6] = lpTexure [6];// address of 4x2 map
TMap.apvLevels[7] = lpTexure [7];// address of 2x1 map
TMap.apvLevels[8] = lpTexure [8];// address of 1x1 map
TMap.bMipMap = TRUE;// texture is a mipmap
TMap.u32MaxMapXSizeLg2 = 8;// log2 of 256
TMap.u32MaxMapYSizeLg2 = 7;// log2 of 128
TMap.eTexFormat = C3D_ETF_RGB565;// texel format is RGB 565
SET_CIF_COLOR (TMap.clrTexChromaKey, 0, 0, 0, 0);// black chroma key

// register the texture
if (ATI3DCIF_TextureReg (&TMap, &htx) != C3D_EC_OK)
{
// handle error
...
}
```

Applying a Texture

Mapping a texture on primitives takes two steps: (1) select the texture, and (2) enable texture mapping in the rendering context. Any primitives rendered with *ATI3DCIF_RenderPrimList* or *ATI3DCIF_RenderPrimStrip* will subsequently be textured until texture mapping is disabled again in the rendering context. By default, texture mapping is disabled when a rendering context is created.

ATI3DCIF will scale, orient, and apply perspective correction to the texture as it is rendered based on the texture coordinates set for the primitive vertices. These coordinates are represented by the s, t, and w members of the *C3D_VTCF*, and *C3D_VTF* vertex structures. The application is responsible for setting these coordinates correctly to map the texture in the manner intended.

A texture is selected by calling *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_TMAP_SELECT and the third argument set to the address of the *C3D_HTX* handle of the texture. This handle must have been initialized beforehand by calling *ATI3DCIF_TextureReg* to register the texture with ATI3DCIF. Texture mapping is enabled by calling *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_TMAP_EN and the third argument pointing to a BOOL data type set to TRUE. The following example illustrates this:

Example 9: applying a texture to a primitive

```
BOOL bTexEnable = FALSE;
C3D_HTX htx1, htx2;
C3D_HRC hRC;

// create a rendering context (handle hRC), and load and register two
// textures, initializing htx1 and htx2
...
// switch to 3D mode
ATI3DCIF_RenderBegin (hRC);

// enable texture mapping
bTexEnable = TRUE;
ATI3DCIF_ContextSetState (hRC, C3D_ERS_TMAP_EN, &bTexEnable);

// select the first texture
ATI3DCIF_ContextSetState (hRC, C3D_ERS_TMAP_SELECT, &htx1);

// render 3D primitives, mapping the first texture on each primitive
ATI3DCIF_RenderPrimList (PrimList1, PrimList1NumVerts);

// select the second texture
ATI3DCIF_ContextSetState (hRC, C3D_ERS_TMAP_SELECT, &htx2);

// render 3D primitives, mapping the second texture on each primitive
ATI3DCIF_RenderPrimList (PrimList2, PrimList2NumVerts);

// disable texture mapping
bTexEnable = FALSE;
ATI3DCIF_ContextSetState (hRC, C3D_ERS_TMAP_EN, &bTexEnable);

// render 3D primitives without texture mapping
ATI3DCIF_RenderPrimStrip (PrimStrip1, PrimStrip1NumVerts);

...
// now switch back to 2D mode
ATI3DCIF_RenderEnd ();
```

Notice that all *ATI3DCIF_ContextSetState* calls were made between *ATI3DCIF_RenderBegin* and *ATI3DCIF_RenderEnd*. Again, *ATI3DCIF_ContextSetState* will not incur overhead to save and restore the state of the 2D engine when called within a 3D rendering block.

Unregistering a Texture

Before the application terminates, it must unregister all registered textures by calling *ATI3DCIF_TextureUnreg* with the handle of the texture to be unregistered.

The following example illustrates how to unregister a texture:

Example 10: unregistering a texture

```
C3D_HTX htx;

// load, register and use a texture
...
// unregister the texture
if (ATI3DCIF_TextureUnreg (htx) != C3D_EC_OK)
{
// handle error
...
}
```

Setting Texture Filtering, Lighting, and Perspective Correction Levels

All rendering states in the rendering context related to texture mapping, such as texture filtering, lighting, and perspective correction level, may be modified by calling *ATI3DCIF_ContextSetState*.

To modify the texture filtering, call *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_TMAP_FILTER and the third argument pointing to a *C3D_ETEXFILTER* enumeration specifying the new filtering mode. ATI3DCIF allows different filtering to be performed on texture minification and magnification. For mipmapping, it supports filtering modes which filter within maps and blend the results of two maps.

The following table shows the *C3D_ETEXFILTER* enumeration constants:

C3D_ETFILT_MINPNT_MAGPNT	pick-nearest filtering on both minification and magnification
C3D_ETFILT_MINPNT_MAG2BY2	pick-nearest filtering on minification, bi-linear filtering on magnification
C3D_ETFILT_MIN2BY2_MAG2BY2	bi-linear filtering on both minification and magnification
C3D_ETFILT_MIPLIN_MAGPNT	1x1 blend between maps on minification (only applies to mipmaps), pick-nearest filtering on magnification
C3D_ETFILT_MIPLIN_MAG2BY2	1x1 blend between maps on minification (only applies to mipmaps), bi-linear filtering on magnification
C3D_ETFILT_MIPTRI_MAG2BY2	2x2 blend between maps, bi-linear filtering within each map on minification (only applies to mipmaps), bi-linear filtering on magnification

To modify the texture lighting, call *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_TMAP_LIGHT and the third argument pointing to a *C3D_ETLIGHT* enumeration specifying the new lighting mode. The following table shows the *C3D_ETLIGHT* enumeration constants:

C3D_ETL_NONE	the texture is not lighted: the texel color is applied directly.
C3D_ETL_MODULATE	the texture is lighted: the texel color is multiplied by the color of the primitive on which the texture is being mapped. The primitive color is determined by rendering context's current shading mode (gouraud, flat, etc.)
C3D_ETL_ALPHA_DECAL	the texture is lighted: the texel color is determined by the following equation: output texel = (texel color x texel alpha) + (primitive color x (1 - texel alpha)), where the texel alpha varies between 0 and 1

The perspective correction level is set by calling *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_TMAP_PERSP_COR, and the third argument pointing to a C3D_ETPERSPCOR enumeration specifying the new perspective correction level. ATI3DCIF provides seven perspective correction levels, from C3D_ETPC_NONE, which provides no correction, to C3D_ETPC_NINE, which provides full correction. The frame rate decreases with increased levels of perspective correction. No correction offers the best frame rate, while full correction offers the worst. The default level set on context creation is C3D_ETCP_THREE, which offers a good compromise between frame rate and the level of perspective correction.

Transparent Texture Mapping

ATI3DCIF provides two ways for a client application to perform texture transparency. The first method is to use a chroma key color in the texel's RGB data. The second method is to use a bit mask in the texel's alpha data (needless to say, this method only works for texel formats which have an alpha channel, such as RGB 4444, RGB 1555, and RGB 8888). Both methods are termed as texel operations, and either may be selected by calling *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_TMAP_TEXOP, and the third argument pointing to a *C3D_ETEXOP* enumeration specifying the operation.

To select chroma key texture transparency, *C3D_ETEXOP* should be set to C3D_ETEXOP_CHROMAKEY. The chroma key used will be the color set in the clrTexChromaKey member of the *C3D_TMAP* structure used to register the texture. Any texel matching this chroma key color will not be rendered on the primitive.

To use the alpha channel for transparency, *C3D_ETEXOP* should be set to C3D_ETEXOP_ALPHA_MASK. The data in the alpha channel is used as a bit field to decide which texel is rendered and which is transparent. If the least significant bit in the channel is set to 0, the texel is not drawn. If set to 1, the texel is drawn.

If C3D_ETEXOP is set to C3D_ETEXOP_ALPHA, the texture's alpha value is passed on to the alpha blender if alpha blending is enabled. Alpha blending is described later in this chapter.

Texture Coordinates

ATI3DCIF accepts homogeneous texture coordinates (s, t, and inverse w). Here is a simple illustration of how to calculate homogeneous texture coordinates from true texture coordinates (u, v):

Let (X, Y, Z) represent a point in camera coordinates (camera coordinates are defined as 3D world coordinates with the camera, or eye, located at the origin). Camera coordinates (X, Y, Z) are projected to screen coordinates (x, y, z) according to the following equations:

$$x = k \times X/w,$$

$$y = k \times Y/w,$$

$$z = ((k1 \times Z) + k2)/w$$

Where k, k1, and k2 are constants and w is proportional to Z. w should be positive. If the view direction is along the –Z axis and the view plane is at Z = –1, then w = –Z. The homogeneous texture coordinates to be sent to ATI3DCIF should be:

$$s = u/w$$

$$t = v/w$$

$$\text{inverse } w = 1/w$$

Alpha Blending

The RAGE's alpha blender may be used to combine source and destination pixels in accordance to source and destination blending factors which are functions of vertex alpha or RGB values. The equation used to determine the output destination pixel is the following:

output destination color = (source color x source blending factor) + (current destination color x destination blending factor)

ATI3DCIF allows the source and destination blending factors to be set by calling *ATI3DCIF_ContextSetState*. Source blending factors are represented by the *C3D_EASRC* enumeration type. Destination blending factors are represented by the *C3D_EADST* enumeration type. To set the source factor, call *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_ALPHA_SRC and the third set to the address of a *C3D_EASRC* enumeration specifying the blending factor. To set the destination blending factor, call *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_ALPHA_DST and the third set to the address of a *C3D_EADST* enumeration.

The following table lists the *C3D_EASRC* source blending factor enumeration constants:

C3D_EASRC_ZERO	blend factor is (0, 0, 0)
C3D_EASRC_ONE	blend factor is (1, 1, 1)
C3D_EASRC_DSTCLR	blend factor is (Rd, Gd, Bd)
C3D_EASRC_INVDSTCLR	blend factor is (1-Rd, 1-Gd, 1-Bd)
C3D_EASRC_SRCALPHA	blend factor is (As, As, As)
C3D_EASRC_INVSRCALPHA	blend factor is (1-As, 1-As, 1-As)

The following table lists the *C3D_EADST* destination blending factor enumeration constants:

C3D_EADST_ZERO	blend factor is (0, 0, 0)
C3D_EADST_ONE	blend factor is (1, 1, 1)
C3D_EADST_SRCCLR	blend factor is (Rs, Gs, Bs)
C3D_EADST_INVSRCCLR	blend factor is (1-Rs, 1-Gs, 1-Bs)
C3D_EADST_SRCALPHA	blend factor is (As, As, As)
C3D_EADST_INVSRCALPHA	blend factor is (1-As, 1-As, 1-As)

CAUTION: Enabling alpha blending adds another process to the graphics pipeline, causing rendering performance to decrease. For the trivial case of source blending factor = C3D_EASRC_ONE and destination blending factor = C3D_EADST_ZERO, the alpha blender is disabled and removed from the graphics pipeline. If the application does not intend to use alpha blending, it should disable this operation by forcing the source and destination blending factors to these states (which are set by default during context creation). It is NOT recommended that the application disable alpha blending by setting vertex alphas to trivial values, but leave the source and destination blending factors at states other than C3D_EASRC_ONE and C3D_EADST_ZERO. Otherwise, the alpha blender will be enabled and chained into the pixel data path in the graphics pipeline.

Please note that it is better to use ATI3DCIF's fog support rather than alpha blending to implement fog, as the fog support is much faster for this effect.

Applying Fog

Fog is applied through a two step process. The first step is to set a fog color, and the next step is to enable fogging. Once enabled, the fog blending factor for each vertex is determined from its alpha value. This does not cause a conflict between fogging and alpha blending because the two operations are mutually exclusive. The manner in which the fog color is applied to the primitive depends on the current shading mode. For example, to interpolate the fog across the primitive, the shading mode should be set to smooth.

To set the fog color, call *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_FG_CLR and the third argument pointing to a *C3D_COLOR* structure specifying the fog color. Fogging is enabled by calling *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_FOG_EN and the third argument pointing to a BOOL set to TRUE.

ATI3DCIF Viewport

ATI3DCIF renders primitives relative to a rectangular region called the viewport. Its top left corner defines the origin of the screen coordinate system in which primitives are drawn, and its width and height extents define the non-clipped rendering area. Parts of primitives which lie outside of this area are clipped by hardware during rendering.

The viewport can be moved to change the logical origin of the screen coordinate system. For example, setting its top, left corner to (10, 10) will cause a vertex with x, y coordinates (0, 0) to be rendered at screen location (10, 10). The viewport's width and height extents are always defined relative to its top left corner.

The viewport can be changed by calling *ATI3DCIF_ContextSetState*. The new origin and extents are specified by the *C3D_RECT* structure, which has the following syntax:

```
typedef struct {
    C3D_INT32 top;// top, left corner top coordinate
    C3D_INT32 left;// top, left corner left coordinate
    C3D_INT32 bottom;// height extent
    C3D_INT32 right;// width extent
} C3D_RECT , * C3D_PRECT;
```

The second argument to *ATI3DCIF_ContextSetState* should be set to C3D_ERS_SURF_VPORT. The third argument should contain the address of a *C3D_RECT* structure specifying the new viewport origin and extents.

CAUTION: Pixels are clipped at the edges of the viewport during the last stage of the rendering process. Prior to reaching this stage, these pixels are still processed through the graphics pipeline although they are not rendered. If vertex coordinates extend beyond the viewport boundaries by a large amount, the graphics engine will end up processing a large number of pixels which will never be rendered. This may have a detrimental effect on performance. For this reason, it may be necessary to pre-clip primitives in software if the clipping overhead proves to be less costly than the time spent processing unrendered pixels.

ATI3DCIF Clipping Scissors

The ATI3DCIF clipping scissors define a rectangular region outside of which primitives are clipped. The behavior and performance constraints are the same as for the viewport described in the last section, except the origin is always fixed at the top left corner of the drawing surface. The scissors are set by calling *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_SURF_SCISSOR and the third set to the address of a *C3D_RECT* structure defining the rectangular clipping region. On rendering context creation, the scissors are set to the rectangular region of the desktop.

NOTE: The scissors are not available on some earlier versions of ATI3DCIF. Applications should call *ATI3DCIF_GetInfo* and query the u32CIFCaps1 member of the *C3D_3DCIFINFO* structure to verify the availability of the scissors.

Chapter 3
3D RAGE II ATI3DCIF Programming

Introduction

This chapter covers programming topics which only apply to the ATI 3D RAGE II graphics accelerator. The 3D RAGE II is the next generation of 3D accelerators in ATI's RAGE family of graphics processors. It offers the same core functionality as the 3D RAGE and adds the following new features:

- CI8 palettized textures
- CI4 palettized textures
- Z buffering

The 3D RAGE II doubles 3D performance and improves 2D performance by 20% over the 3D RAGE. The new features in the 3D RAGE II are supported by extensions to ATI3DCIF added in version 4.02.0230 of the interface.

Determining ATI3DCIF Capabilities

Because of the feature differences between the 3D RAGE and 3D RAGE II, an application should query ATI3DCIF capabilities to determine if a desired feature is available. In version 4.02.0217 of ATI3DCIF, the u32CIFCaps member was added to the C3D_CIFINFO structure to enable querying. This field reports the capabilities available with the combination of the ATI3DCIF version in use and the RAGE accelerator in the graphics subsystem. The functionality provided by ATI3DCIF prior to version 4.02.0217 is considered the base-line functionality, and is represented by the flag C3D_CAPS_BASE. Features added to ATI3DCIF in vision 4.02.0217 and later are represented by capabilities flags which are bit-wise OR-ed together. For example, if fog and Z buffering are supported, u32CIFCaps will contain C3D_CAPS_FOG OR-ed with C3D_CAPS_Z_BUFFER.

In version 4.03.0039 of ATI3DCIF, four more capabilities fields were added to *C3D_3DCIFINFO*. u32CIFCaps was renamed u32CIFCaps1. The new fields are labeled u32CIFCaps2 to u32CIFCaps5. In version 4.03.2511 of ATI3DCIF, the u32CIFCaps2 member was defined to support additional capabilities under RAGE Pro. u32CIFCaps3 to u32CIFCaps5 are currently unused and are reserved for future use. The flags have also been modified to indicate which field they correspond to. For instance, C3D_CAPS_BASE has been changed to C3D_CAPS1_BASE.

NOTE: Because both the 3D RAGE and 3D RAGE II will have a large installed base, it is recommended that applications be designed to handle the feature set differences between the two accelerators rather than support only one. The ATI3DCIF capabilities fields may be used to query feature differences.

Palettized Textures

In addition to the texture formats supported by the 3D RAGE, the 3D RAGE II supports CI8 and CI4 palettized textures. In the CI8 format, each texel is an 8-bit packed value which represents an index into a 256 color palette. In the CI4 format, each texel is a 4-bit packed index into a 16 color palette. The four bit texel values must be byte aligned and may be packed in either the low or high nibble. It is possible to compress two CI4 textures into the space of one CI8 texture by loading one into the low nibble and the other into the high nibble of each byte.

As with the non-palettized textures supported by the 3D RAGE, the CI8 and CI4 textures must also be loaded into video memory. Their width and height must similarly be powers of two and cannot exceed 1024x1024.

The procedure for texture mapping CI8 and CI4 textures under ATI3DCIF is essentially the same as that for mapping non-palettized textures. Texture mapping with ATI3DCIF is covered in full detail in the section *Texture Mapping* in the previous chapter. The only difference is that a logical palette representing the texture's palette must be created in ATI3DCIF and attached to the texture before it is registered. Also, the palette must be destroyed after the texture is unregistered and prior to terminating ATI3DCIF.

A logical palette is created by calling *ATI3DCIF_TexturePaletteCreate*. The first argument is a C3D_ECI_TMAP_TYPE enum constant specifying the kind of palette to create. The C3D_ECI_TMAP_TYPE enum has the following syntax:

```
typedef enum {
    C3D_ECI_TMAP_TRUE_COLOR = 0,         //  no palette
    C3D_ECI_TMAP_4BIT_HI    = 1,         //  16 entry palette
    C3D_ECI_TMAP_4BIT_LOW   = 2,         //  16 entry palette
    C3D_ECI_TMAP_8BIT       = 3,         //  256 entry palette
    C3D_ECI_TMAP_NUM        = 4,         //  invalid enumeration
    C3D_ECI_TMAP_FORCE_U32  = C3D_FORCE_SIZE
} C3D_ECI_TMAP_TYPE;
```

The second argument is an array of *C3D_PALETTENTRY* structures specifying the color of each element in the palette. The *C3D_PALETTENTRY* struct has the following syntax:

```
typedef union {
    struct {
        unsigned r: 8;      // 8 red bits
        unsigned g: 8;      // 8 green bits
        unsigned b: 8;      // 8 blue bits
        unsigned flags: 8;  // flag bits - see above defines
    };
    C3D_UINT32 u32All;
} C3D_PALETTENTRY , * C3D_PPALETTENTRY;
```

For a CI8 texture, a 256 element *C3D_PALETTENTRY* must be specified. For a CI4 texture, the array must contain 16 elements. The r, g, and b members of *C3D_PALETTENTRY* specify the red, green, and blue components, respectively, of each palette entry. The flags member may be used to inhibit individual entries in the palette from loading. If it is set to C3D_LOAD_PALETTE_ENTRY, the physical palette entry at the corresponding index will be replaced with the specified color. If flags is set to C3D_NO_LOAD_PALETTE_ENTRY, the palette entry at the corresponding index will not be altered.

The last argument is a pointer to a *C3D_HTXPAL* palette handle. If the palette is created successfully, the handle will be set to a valid non-zero value. Otherwise, it will be set to NULL.

Once created, the palette must be attached to its associated texture. This is done by assigning its handle to the htxpalTexPalette member of the *C3D_TMAP* structure used to register the texture. To identify the texture format as a CI8, the eTexFormat member must be set to C3D_ETF_CI8. To identify the texture format as a CI4, eTexFormat must be set to C3D_ETF_CI4.

After the application has finished using the texture and has unregistered it, its palette should be destroyed. This is done by calling *ATI3DCIF_TexturePaletteDestroy* with the handle of the palette as the argument.

NOTE: If palettized textures contain their own unique palettes (i.e. each has a palette identified by a different handle), the physical palette is changed each time a different texture is selected. If textures are changed frequently, the fast rate of physical palette updates may cause visual artifacts on the screen. For best results, applications should use one palette for CI8 textures and 16 palettes for CI4 textures at most.

The following examples demonstrate several operations related to palettized textures:

Example 1: creating a palette

```
// This example demonstrates how to create a palette in ATI3DCIF. It assumes the
// texture being loaded is a 256 color Windows bitmap with a 256 element RGBQUAD
// array representing the palette.

    RGBQUAD rgbPalette [256];
    C3D_PALETTENTRY peTexturePalette [256];
    C3D_HTXPAL hTXPal;
    int I;

    // read bitmap palette RGB values from bitmap file into rgbPalette array
    ...

    // fill peTexturePalette array. Set flag to load all entries
    for (int i = 0; i < 256; i++)
    {
        peTexturePalette [i].r = rgbPalette [i].rgbRed ;
        peTexturePalette [i].g = rgbPalette [i].rgbGreen ;
        peTexturePalette [i].b = rgbPalette [i].rgbBlue ;
        peTexturePalette [i].flags = C3D_LOAD_PALETTE_ENTRY;
    }

    // create texture palette and get handle
    if (ATI3DCIF_TexturePaletteCreate (C3D_ECI_TMAP_8BIT, peTexturePalette,
       &hTXPal) != C3D_EC_OK)
    {
        // handle error
        ...
    }
```

Example 2: registering a palettized texture

```
    // In this example, a CI8 128x128 texture is registered.
```

```
C3D_HTXPAL hTXPal;
C3D_HTX hTX;
pTexAddress;

// load the texture into video memory. Let pTexAddress point
// to this location
...

// create a palette for the texture by calling ATI3DCIF_TexturePaletteCreate,
// initializing handle hTXPal
...

// fill a C3D_TMAP struct
ZeroMemory (&TMap, sizeof (TMap));
TMap.u32Size = sizeof (TMap);
TMap.apvLevels[0] = pTexAddress;
TMap.bMipMap = FALSE;
TMap.u32MaxMapXSizeLg2 = 7;
TMap.u32MaxMapYSizeLg2 = 7;
TMap.eTexFormat = C3D_ETF_CI8;
SET_CIF_COLOR (TMap.clrTexChromaKey, 0, 0, 0, 0);
TMap.htxpalTexPalette = hTXPal;

// register the texture
ecRetVal = ATI3DCIF_TextureReg (&TMap, &hTX);
if (ecRetVal != C3D_EC_OK)
{
    // destroy palette
    ATI3DCIF_TexturePaletteDestroy (hTXPal);
    // other error handling
    ...
}
```

Example 3: destroying a palette

```
// unregister the texture
ecRetVal = ATI3DCIF_TextureUnreg (hTX);
if (ecRetVal != C3D_EC_OK)
{
    // handle error
    ...
}

ecval = ATI3DCIF_TexturePaletteDestroy (hTXPal);
if (ecval != C3D_EC_OK)
{
    // handle error
    ...
}
```

Z Buffers

The 3D RAGE II supports Z buffers for sorting primitives by their z values while rendering. Z buffers must be allocated in video memory and must be aligned on eight byte boundaries. An application can ensure Z buffers are aligned properly by using DirectDraw surfaces for the buffers, as DirectDraw surfaces are aligned on eight byte boundaries. Z buffers on the 3D RAGE II are 16 bits deep. This gives a resolution of 2^{16} for z values. Applications should ensure that the range of z values used can be scaled within this resolution (for example 0 to 2^{16} - 1) to avoid inaccuracies due to truncation error.

Under ATI3DCIF, the Z buffer is always associated with the drawing surface. For example, in a double buffer configuration where ATI3DCIF is only rendering to the back surface, the Z buffer will always be associated with the back surface. Therefore, the Z buffer must have the same pitch in pixels and height in scan lines as the drawing surface.

To designate a memory region as a Z buffer, its starting address and pitch in pixels must be specified to ATI3DCIF. The starting address may be set by calling *ATI3DCIF_ContextSetState* with C3D_ERS_SURF_Z_PTR as the second argument and the address of a pointer containing the buffer's starting address as the third argument. To set the pitch, *ATI3DCIF_ContextSetState* must be called with the second argument set to C3D_ERS_SURF_Z_PITCH and the third set to the address of a C3D_UINT32 variable holding the pitch in pixels.

ATI3DCIF provides a variety of compare functions for testing z values. These functions are logical operations which determine whether a pixel is selected or rejected based on the way its z value compares with the buffered z value at that location. ATI3DCIF Z compare functions are represented by the *C3D_EZCMP* enumeration. The following table lists the *C3D_EZCMP* constants:

C3D_EZCMP_NEVER	Z compare never passes
C3D_EZCMP_LESS	Z compare passes if test z is less than buffered z
C3D_EZCMP_LEQUAL	Z compare passes if test z is less than or equal to buffered z
C3D_EZCMP_EQUAL	Z compare passes if test z is equal to buffered z
C3D_EZCMP_GEQUAL	Z compare passes if test z is greater than or equal to buffered z
C3D_EZCMP_GREATER	Z compare passes if test z is greater than buffered z
C3D_EZCMP_NOTEQUAL	Z compare passes if test z is not equal to buffered z
C3D_EZCMP_ALWAYS	Z compare always passes

The Z compare function may be set by calling *ATI3DCIF_ContextSetState* with the second argument set to `C3D_ERS_Z_CMP_FNC` and the third set to the address of a *C3D_EZCMP* enum specifying the compare function.

Z buffering may be enabled and disabled through the C3D_EZMODE enumeration. When Z buffering is enabled, it can be set to either update the contents of the Z buffer after performing Z compare tests, or simply do the compare tests without modifying the Z buffer. The C3D_EZMODE constants are listed in the following table:

C3D_EZMODE_OFF	Disable Z testing
C3D_EZMODE_TESTON	Test Z, do not update the Z buffer
C3D_EZMODE_TESTON_WRITEZ	Test Z, update the Z buffer

The Z test mode may be set by calling *ATI3DCIF_ContextSetState* with the second argument set to C3D_ERS_Z_MODE and the third set to the address of a C3D_EZMODE enum specifying the test mode. Applications should ensure that the Z buffer is initialized properly at the start of each frame update based on the manner in which frames are updated. For instance, some or all parts of the of Z buffer may have to be reset to zeros or ones (or whatever value is appropriate for the specific application) based on the compare function used.

The following example shows how to set up a Z buffer with ATI3DCIF:

Example 4: setting up a Z buffer

```
// This example demonstrates how to set up a Z buffer with the Z testing
// mode set to update the buffer after each test and the Z compare function
// set to greater than.

C3D_HRC hrc;
C3D_EZCMP eZCompFnc;
C3D_EZMODE eZMode;
void *lpSurface;
C3D_UINT32 ui32Pitch;

// create rendering context (handle in hrc), allocate Z buffer in video
// memory (address in lpSurface), and get pitch (value in pixels in
// ui32Pitch. Must match pitch of drawing surface associated with Z
// buffer)
...

// set address of Z buffer
if (ATI3DCIF_ContextSetState (hrc, C3D_ERS_SURF_Z_PTR, (C3D_PRSDATA)
    &lpSurface) != C3D_EC_OK)
{
    // handle error
    ...
}

// set pitch
if (ATI3DCIF_ContextSetState (hrc, C3D_ERS_SURF_Z_PITCH, (C3D_PRSDATA)
    &ui32Pitch) != C3D_EC_OK)
{
    // handle error
    ...
}

// set Z buffering mode
eZMode = C3D_EZMODE_TESTON_WRITEZ;
if (ATI3DCIF_ContextSetState, hrc, C3D_ERS_Z_MODE, (C3D_PRSDATA)
    &eZMode) != C3D_EC_OK)
```

```
{
    // handle error
    ...
}

// set Z buffer compare function
ezCompFnc = C3D_EZCMP_GREATER;
if (ATI3DCIF_ContextSetState (hrc, C3D_ERS_Z_CMP_FNC, (C3D_PRSDATA)
    &zCompFnc) != C3D_EC_OK)
{
    // handle error
    ...
}
```

This page intentionally left blank.

Chapter 4
RAGE PRO ATI3DCIF Programming

Introduction

The following features have been added to ATI3DCIF to support the RAGE PRO:

- Determining capabilities
- Texture compositing
- Texture clamping
- LOD biasing
- Specular lighting
- Destination Alpha testing
- Vector Quantization (VQ) compression
- TL Vertex type (C3D_TLVERTEX)

Determining Capabilities

In version 4.03.2511 of ATI3DCIF, the **u32CIFCaps2** member was added to the C3D_CIFINFO structure to support additional capabilities under RAGE PRO, adding to the capabilites specified in *Getting ATI3DCIF Module and Graphics Subsystem Information* in Chapter 2.

The following table lists u32CIFCaps2 flags:

C3D_CAPS2_TEXTURE_COMPOSITE	second texture and composite blend factor support
C3D_CAPS2_TEXTURE_CLAMP	clamp texture coordinates to 1.0 enable/disable
C3D_CAPS2_DESTINATION_ALPHA_BLEND	extended alpha blending modes supported
C3D_CAPS2_TEXURE_TILING	texture tiling support

Texture Compositing

Texture compositing is the process of combining two textures into one composite texture. This process may be used to apply a light map to a texture or to dissolve from one texture into another. Texture compositing allows an application to get more mileage out of its textures by combining and modifying them in unlimited ways. For example, a single light map may be applied to several textures, eliminating the need to create unique, modified versions of the same textures.

Texture compositing is possible only when texture mapping is enabled. To composite two textures, a primary texture must be selected and texture mapping must be enabled by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_TMAP_EN and the state data pointing to a BOOL variable set to TRUE. The primary texture is selected by calling ATI3DCIF_ContextSetState with the state flag C3D_ERS_TMAP_SELECT and the state data pointing to the texture's *C3D_HTX* handle. The handle must be obtained beforehand by loading and registering the texture using *ATI3DCIF_TextureReg* as described in Chapter 2, *Texture Mapping*. Once texture mapping is enabled, texture compositing may be turned on by calling ATI3DCIF_ContextSetState with the state flag C3D_ERS_COMPOSITE_EN and the state data pointing to a BOOL variable set to TRUE. The composite texture must be loaded and registered in the same manner as the primary texture. It is selected into the rendering context by calling ATI3DCIF_ContextSetState with the state flag C3D_ERS_COMPOSITE_SELECT and the state data pointing to its C3D_HTX handle. Applications must ensure that the primary texture is selected when compositing, otherwise behavior is undefined.

The two texture maps may have different dimensions. They may also have different texel formats, but with two restrictions:

1. if one texture is in a YUV format, then the other texture must be in a YUV format, and

2. if one texture is in an eight-bit format (CI8, CI4, RGB332, or Y8), then the other texture must be in an eight-bit format, although it may be in a different eight-bit format.

The secondary composite texture's filtering mode must be set separately than the primary's. Its filtering mode is set by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_COMPOSITE_FILTER and the state data pointing to a *C3D_ETEXFILTER* enum specifying the filtering mode. Only four of the six C3D_ETEXFILTER modes apply to the secondary texture. These are C3D_ETFILT_MINPNT_MAGPNT, C3D_ETFILT_MINPNT_MAG2BY2, C3D_ETFILT_MIN2BY2_MAGPNT, and C3D_ETFILT_MIN2BY2_MAG2BY2. For more on texture filtering modes, see the sub-section *Setting Texture Filtering, Lighting, and Perspective Correction Levels* in the section *Texture Mapping* in Chapter 2.

There are three possible texture composting functions in ATI3DCIF: blend, modulation, and specular-addition. These states are represented by the *C3D_ETEXCOMPFCN* enum, which has the following syntax:

```
typedef enum {
    C3D_ETEXCOMPFCN_BLEND     = 0,  // use blend factor set in
                                     // C3D_ERS_COMPOSITE_FACTOR
    C3D_ETEXCOMPFCN_MOD       = 1,   // modulate the two textures
    C3D_ETEXCOMPFCN_ADD_SPEC  = 2,
    C3D_ETEXCOMPFCN_MAX       = 3,
    C3D_ETEXCOMPFCN_FORCE_U32 = C3D_FORCE_SIZE
} C3D_ETEXCOMPFCN, * C3D_PETEXCOMPFCN;
```

The following is a description of the composite functions:

Blend

This is the default texture composite function. The two textures are combined by a blending factor according to the following equation:

final texel = (primary texel x (1 - (blending factor/16))) + (secondary texel x blending factor/16)

The blending factor may be any integer value between 0 and 15, giving 16 blending levels. It is set by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_COMPOSITE_FACTOR and the state data set to the blend factor. The default blending factor on context creation is 8. By progressively increasing or decreasing the blending factor, the two texture maps may be gradually dissolved into one another.

An alternative to setting the blending factor explicitly through an *ATI3DCIF_ContextSetState* call is to extract the blending factor from the alpha channel of the composite texture. This state can be enabled by calling ATI3DCIF_ContextSetState with the state flag C3D_ERS_COMPOSITE_FACTOR_ALPHA and the state data pointing to a BOOL variable set to TRUE. This state can also be set as the default for the composite texture by setting the bAlphaBlend member of the *C3D_TMAP* structure to TRUE when registering the texture. Once set, the state can be toggled on or off by calling ATI3DCIF_ContextSetState with the C3D_ERS_COMPOSITE_FACTOR_ALPHA flag.

Modulation

The two textures are combined according to the following equation:

final texel = primary texel x secondary texel

The two textures are simply multiplied together for the final texel. There is no blending factor affecting the modulation.

Specular-Addition

The two textures are combined according to the following equation:

final texel = (primary texel x diffuse color) + secondary texel

Texture Clamping

Texture clamping allows a texture's s and t coordinates to be clamped at 1.0, which causes the texel at coordinate 1.0 to be replicated towards the edge of the primitive. The effect is that of smearing, which can be used to extend the edges of the texture to fill in the gaps when tiling is not desired.

To clamp the s coordinate, the bClampS member of the *C3D_TMAP* structure must be set to TRUE before the texture is registered. Similarly, to clamp the t coordinate, the bClmapT member must be set to TRUE.

LOD Biasing

LOD (Level of Detail) biasing controls level switching during mipmapping. The LOD bias is an integer value between 0 and 15 which modifies the threshold stride at which the switch is made to the next smallest map. The threshold stride is determined according to the following equation:

threshold stride = 1 + (LOD bias/16)

This equation bounds the threshold stride between 1 and 1 15/16.

The LOD bias is set by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_LOD_BIAS_LEVEL and the state data set to the LOD bias value. By default, the LOD bias is 0, setting the threshold stride to 1.

Specular Lighting

Specular lighting is the process of applying an additive highlight to a primitive. This is done by specifying a specular color in the vertex description. Specular lighting must be enabled before it can be applied. It can be enabled by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_SPECULAR_EN and the state data pointing to a BOOL variable set to TRUE. The desired color is set into the *.specular* member of the *C3D_TLVERTEX* structure. Specular color may be referenced in this structure as either a C3D_UINT32 or as individual C3D_UINT8 color components: b, g, r and a.

Destination Alpha Testing

Destination alpha testing is a mechanism for selectively writing source data to the destination buffer. This process compares the alpha value from one of six selectable sources to a reference value maintained in the rendering context. If the comparison passes, the RGB data of the primitive being rendered is written to the destination RGB channels, and the alpha from the selected source is written to the destination alpha channel. Otherwise, the destination remains unchanged. Conceptually, the process is similar to z-buffering in that a decision is made to render a pixel based on the comparison of two values. But unlike z-buffering, the alpha comparison reference is not the target destination pixel’s alpha value, but a single reference value in the rendering context which is compared against for all pixels. Also, as mentioned, there are six sources of alpha test values, as opposed to one source of z data (the vertex z value).

The depth of the alpha channel varies depending on the pixel format. The alpha channel may be as narrow as one bit, as in the ARGB1555 format, or as wide as eight bits, as in the ARGB8888 format.

The alpha write source is selected by calling ATI3DCIF_ContextSetState with the state flag C3D_ERS_ALPHA_DST_WRITE_SELECT and the state data pointing to a *C3D_EASEL* enum specifying the source.

The six alpha write sources represented by the C3D_EASEL enum has the following syntax:

```
typedef enum {
    C3D_EASEL_ZERO          = 0, // write all bits 0
    C3D_EASEL_ONE           = 1, // write all bits 1
    C3D_EASEL_SRCALPHA      = 4, // write As
    C3D_EASEL_INVSRCALPHA   = 5, // write 1-As
    C3D_EASEL_DSTALPHA      = 6, // write Ad
    C3D_EASEL_INVDSTALPHA   = 7, // write 1-Ad
    C3D_EASEL_FORCE_U32     = C3D_FORCE_SIZE
} C3D_EASEL, *C3D_PEASEL;
```

As represents the source primitive alpha channel, and *Ad* represents the destination alpha channel. A note on using C3D_EASEL_DSTALPHA and C3D_EASEL_INVDSTALPHA — the alpha data in the destination alpha channel may be different for each buffer when double buffering, which is in contrast to z-buffering, where both buffers usually share a single z-buffer and therefore reference the same z values.

If C3D_EASEL_SRCALPHA or C3D_EASEL_INVSRCALPHA is selected, the alpha write data is read from the source primitive. If the primitive data is a texel which does not have an alpha channel, an alpha value of 0xff is used for comparison and alpha writing.

The alpha value from the selected write source is compared to a reference value maintained in the rendering context. This reference value may be set by calling ATI3DCIF_ContextSetState with the state flag C3D_ERS_ALPHA_DST_REFERENCE and the state data pointing to a C3D_UINT32 variable indicating the desired reference value. The reference value default is 0.

The alpha compare functions are represented by the *C3D_EACMP* enum, which has the following syntax:

```
typedef enum {
    C3D_EACMP_NEVER     = 0,
    C3D_EACMP_LESS      = 1,
    C3D_EACMP_LEQUAL    = 2,
    C3D_EACMP_EQUAL     = 3,
    C3D_EACMP_GEQUAL    = 4,
    C3D_EACMP_GREATER   = 5,
    C3D_EACMP_NOTEQUAL  = 6,
    C3D_EACMP_ALWAYS    = 7,
    C3D_EACMP_MAX       = 8,
    C3D_EACMP_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EACMP, * C3D_PEACMP;
```

These functions are similar to the z compare functions described in Chapter 3. The alpha compare function is set by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_ALPHA_DST_TEST_FNC and the state data pointing to a *C3D_EACMP* enum specifying the function. The default compare function is C3D_EACMP_ALWAYS. To enable or disable destination alpha testing, ATI3DCIF_ContextSetState should be called with the state flag C3D_ERS_ALPHA_DST_TEST_ENABLE and the state data pointing to a BOOL variable indicating the enable state.

Vector Quantization (VQ) Compression

VQ textures are very similar to normal paletted textures in that they both have image data and a palette. VQ texture are decompressed by the RAGE PRO at run time. Space savings and performance savings can be signicant. For example, a 128x64 texture will fit into the RAGE PRO's cache, minimizing the number of cache misses. This technique also will reduce the memory bandwidth that is required for texture fetches, and allow more textures to be stored in local memory.

To create a VQ Texture, a palette must be created using *ATI3DCIF_TexturePaletteCreate*. The first argument is an enumeration constant that defines the type of palette to create and should be set to C3D_ECI_TMAP_VQ. The second argument is an array of 256 entries specifying the code book, which has the following syntax:

```
typedef struct {
    C3D_UINT16 ul; // upper-left of 2x2 block
    C3D_UINT16 ur; // upper-right of 2x2 block
    C3D_UINT16 ll; // lower-left of 2x2 block
    C3D_UINT16 lr; // upper-right of 2x2 block
} C3D_CODEBOOKENTRY, *C3D_PCODEBOOKENTRY;
```

When finished with the texture, the palette should be destroyed by calling the *ATI3DCIF_TexturePaletteDestroy* function, with the palette's handle as an argument.

It is possible to create mipmapped VQ textures. The same code book will be used for all the mipmap levels. Textures should be pre-compressed for better performance.

The following examples demonstrate several operations related to VQ textures:

Example 1: Creating a Palette

```
C3D_CODEBOOK cbCodeBook[256];
C3D_HTXPAL hTXPal;

// cbTempCodeBook: code book from read in from a VQ Texture file
// copy it into a code book structure for (int j = 0; j < 256; j++)
{
    cbCodeBook[j].ul = cbTempCodeBook[j].ul;
    cbCodeBook[j].ur = cbTempCodeBook[j].ur;
    cbCodeBook[j].ll = cbTempCodeBook[j].ll;
    cbCodeBook[j].lr = cbTempCodeBook[j].lr;
}

if (ATI3DCIF_TexturePaletteCreate(C3D_ECI_TMAP_VQ, cbCodeBook, &hTXPal)
    != C3D_EC_OK
{
    // handle error
    ...
}
```

Example 2: Creating a VQ Texture

```
// this example registers a 64x64 VQ Texture
C3D_HTXPAL hTXPal;
C3D_HTX hTX;
pIndexMapAddress;

// pIndexMapAddress, this is a pointer to the index map of the VQ Texture
// read in from a VQ Texture file
// create the CODEBOOK structure and create the palette using
// ATI3DCIF_TexturePaletteCreate to receive a valid handle
ZeroMemory(&TMap, sizeof(TMap));
TMap.u32Size = sizeof(TMap);
TMap.apvLevels[0] = pIndexMapAddress;
TMap.bMipMap = FALSE;
TMap.u32MaxMapXSizeLg2 = 6;
TMap.u32MaxMapYSizeLg2 = 6;
TMap.eTexFormat = C3D_ETF_VQ;
SET_CIF_COLOR(TMap.clrTexChromaKey, 0, 0, 0, 0);
TMap.htxpalTexPalette= hTXPal;

// register the texture
ecRetVal = ATI3DCIF_TextureReg (&TMap, &hTX);
```

```
if (ecRetVal != C3D_EC_OK)
{
    // destroy palette
    ATI3DCIF_TexturePaletteDestroy(hTXPal);
    // other error handling
    ...
}
```

Example 3: Destroying the Palette

```
// unregister texture
ecRetVal = ATI3DCIF_TextureUnreg(hTX);
if (ecRetVal != C3D_EC_OK)
{
    // handle error
    ...
}

ecRetVal = ATI3DCIF_TexturePaletteDestroy (hTXPal);
if (ecRetVal != C3D_EC_OK)
{
    // handle error
    ...
}
```

TL Vertex Type (C3D_TLVERTEX)

A new *C3D_TLVERTEX* vertex type has been implemented. This vertex type is highly portable and faster than older vertex types on the RAGE PRO family of accelerators. The portable nature of the C3D_TLVERTEX vertex type helps eliminate the need for copying/reformatting of vertex data when porting applications to the ATI3DCIF driver interface.

The C3D_TLVERTEX vertex type has the following syntax:

```
typedef struct {
    union {
          C3D_FLOAT32  sx;     // screen X
          C3D_FLOAT32  x;
    };
    union {
          C3D_FLOAT32  sy;     // screen Y
          C3D_FLOAT32  y;
    };
    union {
          C3D_FLOAT32  sz;     // screen Z
          C3D_FLOAT32  z;
    };
    union {
          C3D_FLOAT32  rhw;    // reciprocal of the homogenious W
          C3D_FLOAT32  w;
    };
    union {
          C3D_UINT32   color;  // diffuse color
```

```
                struct {
                      C3D_UINT8b;
                      C3D_UINT8g;
                      C3D_UINT8r;
                      C3D_UINT8a;
                };
          };
          union {
                C3D_UINT32  specular;// specular color
                struct {
                      C3D_UINT8spec_b;
                      C3D_UINT8spec_g;
                      C3D_UINT8spec_r;
                      C3D_UINT8spec_a;
                };
          };
          union {
                C3D_FLOAT32 tu;    // texture U
                C3D_FLOAT32 s;
          };
          union {
                C3D_FLOAT32 tv;    // texture V
                C3D_FLOAT32 t;
          };
          struct {
                C3D_FLOAT32 reserved1;
                C3D_FLOAT32 reserved2;
                C3D_FLOAT32 reserved3;
          } composite;
      } C3D_TLVERTEX;
```

In keeping with current industry standards, note that the texture coordinates for the C3D_TLVERTEX are now non-homogenous u and v (*.tu* and *.tv*) coordinates. In the *C3D_VTCF* and older vertex types, texture coordinates were specified as homogenous s and t (i.e., s = u/w and t = v/w).

The current vertex type may be changed by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_VERTEX_TYPE and the state data pointing to a *C3D_EVERTEX* enum specifying the vertex type. This must be done to choose the C3D_TLVERTEX vertex type, because the default is C3D_VTCF.

The C3D_VTF, C3D_VCF and C3D_VF vertex types are no longer supported by RAGE PRO. Support for the C3D_VTCF vertex type has been maintained, however.

Chapter 5
ATI3DCIF API Reference

Introduction

This chapter describes the ATI3DCIF functions and data types supported under Windows 95.

Windows 95 Functions

ATI3DCIF_ContextCreate

Version

1.0

Syntax

```
C3D_HRC DLLEXPORT WINAPI ATI3DCIF_ContextCreate (
    void);
```

Arguments

None.

Return Value

A *C3D_HRC* handle identifying the rendering context if successful, otherwise NULL.

Description

This function creates an ATI3DCIF rendering context. If successful, it returns a *C3D_HRC* handle which uniquely identifies the rendering context. The handle is used in subsequent ATI3DCIF functions which reference the context.

Prior to creating the rendering context, the application must load and initialize the ATI3DCIF module by calling *ATI3DCIF_Init*. Before terminating, the application must destroy the rendering context by calling *ATI3DCIF_ContextDestroy* to free system resources.

See Also

ATI3DCIF_Init, *ATI3DCIF_ContextDestroy*

ATI3DCIF_ContextDestroy

Version

1.0

Syntax

```
void DLLEXPORT WINAPI ATI3DCIF_ContextDestroy (
    C3D_HRC hRC);
```

Arguments

hRC	*C3D_HRC* handle to rendering context

Return Value

None.

Description

This function destroys the rendering context identified by hRC. The context must have been created by a previous call to *ATI3DCIF_ContextCreate*. An application must destroy the rendering context before terminating to free system resources.

See Also

ATI3DCIF_ContextCreate

ATI3DCIF_ContextSetState

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_ContextSetState (
    C3D_HRC hRC,
    C3D_ERSID eRStateID,
    C3D_PRSDATA pRStateData);
```

Arguments

hRC	*C3D_HRC* handle to rendering context
eRStateID	*C3D_ERSID* enumeration specifying state to set
pRStateData	*C3D_PRSDATA* pointer to new state data

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function modifies a rendering state in the context identified by hRC. The state to be modified is specified by eRStateID. pRStateData points to a data object containing new state information. The data type addressed by pRStateData depends on eRStateID. The following table lists the type of object pointed to by pRStateData for each eRStateID constant, as well as the default states set on context creation.

eRStateID	**pRStateData**	**Context Default**
C3D_ERS_FG_CLR	pointer to a *C3D_COLOR* structure specifying the fog color	{0, 0, 0, 0}
C3D_ERS_VERTEX_TYPE	pointer to *C3D_EVERTEX* enumeration specifying vertex type	C3D_EV_VTCF
C3D_ERS_PRIM_TYPE	pointer to a *C3D_EPRIM* enumeration specifying primitive type	C3D_EPRIM_TRI
C3D_ERS_SOLID_CLR	pointer to a *C3D_COLOR* structure specifying solid color	{0, 0, 0, 0}
C3D_ERS_SHADE_MODE	pointer to *C3D_ESHADE* enumeration specifying shading mode	C3D_ESH_SMOOTH
C3D_ERS_TMAP_EN	pointer to a BOOL enabling or disabling texture mapping.	FALSE
C3D_ERS_TMAP_SELECT	pointer to a *C3D_HTX* handle specifying texture	NULL

(continued on next page)

eRStateID	**pRStateData**	**Context Default**
C3D_ERS_TMAP_LIGHT	pointer to *C3D_ETLIGHT* enumeration specifying texture lighting mode	C3D_ETL_NONE
C3D_ERS_TMAP_FILTER	pointer to a *C3D_ETEXFILTER* specifying texture filtering mode	C3D_ETFILT_MINPNT_MAG2BY2
C3D_ERS_TMAP_PERSP_COR	pointer to a *C3D_ETPERSPCOR* enumeration specifying texture perspective correction level	C3D_ETPC_THREE
C3D_ERS_TMAP_TEXOP	pointer to a *C3D_ETEXOP* enumeration specifying texel rendering operation	C3D_ETEXOP_NONE
C3D_ERS_ALPHA_SRC	pointer to a *C3D_EASRC* enumeration specifying source alpha blend mode	C3D_EASRC_ONE
C3D_ERS_ALPHA_DST	pointer to a *C3D_EADST* enumeration specifying destination alpha blend mode	C3D_EADST_ZERO
C3D_ERS_ SURF_DRAW_PTR	pointer to a C3D_PVOID pointer specifying the address of the drawing surface region. The address must be an integer multiple of 8 bytes	set to address of on-screen desktop region on rendering context creation
C3D_ERS_ SURF_DRAW_PITCH	pointer to a C3D_UINT32 specifying the pitch of the drawing surface region in pixels. The pitch must be an integer multiple of 8 pixels	set to pitch of on-screen desktop region on rendering context creation
C3D_ERS_ SURF_DRAW_PF	pointer to a *C3D_EPIXFMT* enumeration specifying the drawing surface region pixel format	set to pixel format of on-screen desktop region on rendering context creation
C3D_ERS_SURF_VPORT	pointer to a *C3D_RECT* structure specifying a rectangular clipping region on the drawing surface. Primitives will not be rendered outside of this viewport rectangle	set to coordinates of visible desktop rectangular region on rendering context creation
C3D_ERS_FOG_EN	pointer to a BOOL enabling or disabling fog	FALSE
C3D_ERS_DITHER_EN	pointer to a BOOL enabling or disabling dither	TRUE
C3D_ERS_Z_CMP_FNC	pointer to a *C3D_EZCMP* enumeration specifying the Z compare function.	C3D_EZCMP_ALWAYS
C3D_ERS_Z_MODE	pointer to a *C3D_EZMODE* enumeration specifying the Z testing mode	C3D_EZMODE_OFF

(continued on next page)

eRStateID	pRStateData	Context Default
C3D_ERS_SURF_Z_PTR	pointer to a C3D_PVOID pointer specifying the address of the Z buffer	NULL
C3D_ERS_SURF_Z_PITCH	pointer to a C3D_UINT32 specifying the pitch of the Z buffer in pixels	set to pitch of on-screen desktop region on rendering context creation
C3D_ERS_SURF_SCISSOR	pointer to a *C3D_RECT* structure specifying a rectangular clipping region on the drawing surface. Primitives will not be rendered outside of this scissors rectangle. The scissors region differs from the viewport region in that its origin is always fixed at the top left corner of the surface	set to coordinates of visible desktop rectangular region on rendering context creation
C3D_ERS_COMPOSITE_EN	pointer to a BOOL enabling or disabling texture compositing	FALSE
C3D_ERS_COMPOSITE_SELECT	pointer to a *C3D_HTX* handle specifying the secondary composite texture	NULL
C3D_ERS_COMPOSITE_FNC	pointer to a *C3D_ETEXCOMPFCN* enumeration specifying the texture compositing function	C3D_ETEXCOMPFCN _BLEND
C3D_ERS_COMPOSITE_FACTOR	pointer to a C3D_UNIT32 specifying blend factor if C3D_ETEXCOMPFCN_BLEND compositing function selected. This value must be an integer between 0 and 15	8
C3D_ERS_COMPOSITE_FILTER	pointer to a *C3D_ETEXFILTER* specifying the filtering mode for the secondary composite texture. The only texturing modes supported for the secondary filter are: C3D_ETFILT_MINPNT_MAGPNT, C3D_ETFILT_MINPNT_MAG2BY2, C3D_ETFILT_MIN2BY2_MAGPNT and C3D_ETFILT_MIN2BY2_MAG2BY2	C3D_ETFILT_MIN2BY2 _MAG2BY2
C3D_ERS_COMPOSITE_FACTOR _ALPHA	pointer to a BOOL enabling blend factor to be taken from composite texture's alpha channel for blend texture compositing function	FALSE

(continued on next page)

eRStateID	**pRStateData**	**Context Default**
C3D_ERS_LOD_BIAS_LEVEL	pointer to a C3D_UNIT32 specifying the LOD bias. This value must be an integer between 0 and 15	0
C3D_ERS_ALPHA_DST_TEST_ENABLE	pointer to a BOOL enabling or disabling destination alpha testing	FALSE
C3D_ERS_ALPHA_DST_TEST_FNC	pointer to a *C3D_EACMP* enumeration specifying destination alpha test compare function	C3D_EACMP_ALWAYS
C3D_ERS_ALPHA_DST_WRITE_SELECT	pointer to a *C3D_EASEL* enumeration specifying the alpha source for destination alpha test alpha write	C3D_EASEL_ZERO
C3D_ERS_ALPHA_DST_REFERENCE	pointer to a C3D_UNIT32 specifying reference alpha value for destination alpha testing	0
C3D_ERS_SPECULAR	pointer to a BOOL enabling or disabling specular lighting	FALSE

The data types that may be addressed by pRStateData are described in detail in other sections of this reference.

NOTE: z-buffers are only supported in the 3D RAGE II graphics accelerator or later.

See Also

C3D_EACMP, *C3D_EADST*, *C3D_EASEL*, *C3D_EASRC*, *C3D_EPIXFMT*, *C3D_EPRIM*, *C3D_ESHADE*, *C3D_ETEXCOMPFCN*, *C3D_ETEXFILTER*, *C3D_ETLIGHT*, *C3D_ETPERSPCOR*, *C3D_ETEXOP*, *C3D_EVERTEX*, *C3D_COLOR*, *C3D_HTX*, *C3D_EZCMP*, *C3D_EZMODE*

ATI3DCIF_GetInfo

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_GetInfo (
    C3D_P3DCIFINFO pCIFInfo);
```

Arguments

pCIFInfo	pointer to a *C3D_3DCIFINFO* structure to be initialized with ATI3DCIF module information.

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function returns information about the graphics subsystem and the ATI 3D RAGE graphics accelerator in the *C3D_3DCIFINFO* structure pointed to by pCIFInfo. Prior to calling this function, the u32Size member of the C3D_3DCIFINFO structure must be set to the size of the structure. Otherwise, the function will fail.

See Also

C3D_3DCIFINFO

ATI3DCIF_Init

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_Init (
    void);
```

Arguments

None.

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function loads and initializes the ATI3DCIF module. This function must be called before any other ATI3DCIF functions are called. Otherwise, the ATI3DCIF functions will fail.

Before the application terminates, it must terminate and unload the ATI3DCIF driver by calling *ATI3DCIF_Term*.

See Also

ATI3DCIF_Term

ATI3DCIF_RenderBegin

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_RenderBegin (
    C3D_HRC hRC);
```

Arguments

hRC	*C3D_HRC* handle to rendering context

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function prepares the hardware to draw using the context identified by hRC. It must be called prior to any other ATI3DCIF_Renderxxx functions. Typically, it is called at the beginning of each frame update before rendering primitives with *ATI3DCIF_RenderPrimList* or *ATI3DCIF_RenderPrimStrip*.

After completing rendering operations, the application should call *ATI3DCIF_RenderEnd* to end the 3D hardware drawing operations. This will free the graphics hardware for 2D operations.

See Also

ATI3DCIF_RenderEnd, *ATI3DCIF_RenderPrimList*, *ATI3DCIF_RenderPrimStrip*.

ATI3DCIF_RenderEnd

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_RenderEnd (
    void);
```

Arguments

None.

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function terminates 3D hardware rendering operations initiated by a prior call to *ATI3DCIF_RenderBegin*. After completing rendering operations, for instance at the end of frame updates, the application should call ATI3DCIF_RenderEnd to end the 3D hardware drawing operations. This will free up the graphics hardware to resume 2D operations.

See Also

ATI3DCIF_RenderBegin, *ATI3DCIF_RenderPrimList*, *ATI3DCIF_RenderPrimStrip*.

ATI3DCIF_RenderPrimList

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_RenderPrimList (
    C3D_VLIST vList,
    C3D_UINT32 u32NumVert);
```

Arguments

vList	array of pointers to primitive list vertex structures
u32NumVert	number of vertices in the primitive

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function draws a primitive list using the current primitive type of the rendering context. By default, the rendering context is initialized to draw triangle primitives. The primitive type may be modified by calling *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_PRIM_TYPE and pRStateData set to the address of a *C3D_EPRIM* enumeration specifying the new primitive type. vList is an array of pointers to vertex structures representing the vertices in the primitive list. The default rendering context vertex structure is *C3D_VTCF*. This may be changed by calling ATI3DCIF_ContextSetState with eRStateID set to C3D_ERS_VERTEX_TYPE, and pRStateData set to the address of a *C3D_EVERTEX* enumeration specifying the new vertex structure type. u32NumVerts specifies the number of vertices in the primitive list.

See Also

ATI3DCIF_ContextCreate

ATI3DCIF_RenderPrimStrip

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_RenderPrimStrip (
    C3D_VSTRIP vStrip,
    C3D_UINT32 u32NumVert);
```

Arguments

vStrip	array of primitive strip vertex structures
u32NumVert	number of vertices in the primitive

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function draws a primitive strip using the current primitive type of the rendering context. By default, the rendering context is initialized to draw triangle primitives. The primitive type may be modified by calling *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_PRIM_TYPE and pRStateData set to the address of a *C3D_EPRIM* enumeration specifying the new primitive type. vStrip is an array of vertex structures representing the vertices in the primitive strip. The default rendering context vertex structure is *C3D_VTCF*. This may be changed by calling ATI3DCIF_ContextSetState with eRStateID set to C3D_ERS_VERTEX_TYPE, and pRStateData set to the address of a *C3D_EVERTEX* enumeration specifying the new vertex structure type. u32NumVerts specifies the number of vertices in the primitive strip.

See Also

ATI3DCIF_ContextCreate

ATI3DCIF_RenderSwitch

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_RenderSwitch (
    C3D_HRC hNewRC);
```

Arguments

hNewRC	*C3D_HRC* handle of rendering context to switch to

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

NOTE: This function is not yet implemented.

This function switches 3D rendering operations to the rendering context identified by hNewRC. It is the functional equivalent of calling *ATI3DCIF_RenderEnd* for the existing context followed by *ATI3DCIF_RenderBegin* with the handle of the new context. This function is only valid while in a 3D rendering state, that is, while a rendering operation has been initiated by calling ATI3DCIF_RenderBegin.

See Also

ATI3DCIF_RenderBegin, *ATI3DCIF_RenderEnd*

ATI3DCIF_Term

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_Term (
    void);
```

Arguments

None.

Return Value

TRUE if successful, otherwise FALSE.

Description

This function terminates and unloads the ATI3DCIF module. The module must have been loaded previously by a call to *ATI3DCIF_Init*. An application must unload the ATI3DCIF module before terminating to free system resources.

See Also

ATI3DCIF_Init

ATI3DCIF_TexturePaletteCreate

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_TexturePaletteCreate
    C3D_ECI_TMAP_TYPE epalette,
    C3D_PPALETTENTRY pclrPalette,
    C3D_PHTXPAL phtpalCreated);
```

Arguments

epalette	kind of palette to create (CI4, CI8, or VQ code book)
pclrPalette	array of 16 or 256 *C3D_PALETTENTRY* structures, or 256-entry code book (C3D_PCODEBOOKENTRY)
phtpalCreated	palette handle to be initialized

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function creates a 16 or 256 entry logical texture palette within ATI3DCIF. The handle obtained by calling this function is assigned to the htxpalTexPalette member of a CI4 or CI8 texture's *C3D_TMAP* structure before the texture is registered. 16 entry palettes are used with CI4 textures and 256 entry palettes with CI8 textures. epalette specifies what kind of palette to create. The colors of each entry in the palette are specified by the pclrPalette array passed to this function. If successful, the *C3D_HTXPAL* handle addressed by phtpalCreated will be set to a valid value. Otherwise, it is set to NULL.

After the application has finished using the texture and has unregistered it, the palette should be destroyed by calling *ATI3DCIF_TexturePaletteDestroy*.

NOTE: Texture palettes, C3D_ETF_CI4, and C3D_ETF_CI8 texel formats are only available with the RAGE II or later graphics accelerators. All other formats are available with both the 3D RAGE and 3D RAGE II accelerators.

See Also

ATI3DCIF_TexturePaletteDestroy

ATI3DCIF_TexturePaletteDestroy

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_Texture
    C3D_HTXPAL htxpalToDestroy);
```

Arguments

htxpalToDestroy	handle of texture palette to destroy

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function destroys a logical texture palette created by calling *ATI3DCIF_TexturePaletteCreate*. A texture palette must be destroyed after the texture it is assigned to has been unregistered, and before terminating ATI3DCIF.

NOTE: Texture palettes are only available with the 3D RAGE II graphics accelerator.

See Also

ATI3DCIF_TexturePaletteCreate

ATI3DCIF_TextureReg

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_TextureReg (
    C3D_PTMAP ptmapToReg,
    C3D_PHTX phtxTMap);
```

Arguments

ptmapToReg	pointer to a *C3D_TMAP* structure describing the texture
phtxTMap	pointer to *C3D_HTX* to be initialized

Return Value

A C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function registers a texture with the ATI3DCIF module. ptmapToReg points to a *C3D_TMAP* structure providing texture information required by the ATI3DCIF module. If successful, this function initializes the *C3D_HTX* handle pointed to by phtxTMap with a unique value identifying the texture. Otherwise, the handle is set to NULL.

To map the texture, the application must (1) select the texture by calling *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_TMAP_SELECT and pRStateData pointing to the texture's *C3D_HTX* handle, and (2) enable texture mapping by calling *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_TMAP_EN and pRStateData pointing to a BOOL set to TRUE.

See Also

ATI3DCIF_ContextSetState, *C3D_TMAP*

ATI3DCIF_TextureUnreg

Version

1.0

Syntax

```
C3D_EC DLLEXPORT WINAPI ATI3DCIF_TextureUneg (
    C3D_HTX htxToUnreg);
```

Arguments

htxToUnreg	*C3D_HTX* handle of texture to unregister

Return Value

C3D_EC_OK if successful, otherwise a *C3D_EC* error code.

Description

This function unregisters the texture map identified by htxToUnreg. This texture must have been registered with the ATI3DCIF module by a previous call to *ATI3DCIF_TextureReg*.

See Also

ATI3DCIF_TextureReg

ATI3DCIF Data Types

ATI3DCIF Fundamental Data Types

C3D_BOOL	unsigned int
C3D_INT32	int
C3D_UINT32	unsigned int
C3D_UINT16	unsigned short
C3D_UINT8	unsigned char
C3D_FLOAT32	float
C3D_PBOOL	unsigned int *
C3D_PINT32	int *
C3D_PUINT32	unsigned int *
C3D_PUINT16	unsigned short *
C3D_PUINT8	unsigned char *
C3D_PFLOAT32	float *
C3D_PVOID	void *

C3D_3DCIFINFO

Version

1.0

Syntax

```
typedef struct {
     C3D_UINT32 u32Size;
     C3D_UINT32 u32FrameBuffBase;
     C3D_UINT32 u32OffScreenHeap;
     C3D_UINT32 u32OffScreenSize;
     C3D_UINT32 u32TotalRAM;
     C3D_UINT32 u32ASICID;
     C3D_UINT32 u32ASICRevision;
     C3D_UINT32 u32CIFCaps1;
     C3D_UINT32 u32CIFCaps2;
     C3D_UINT32 u32CIFCaps3;
     C3D_UINT32 u32CIFCaps4;
     C3D_UINT32 u32CIFCaps5;
} C3D_3DCIFINFO, * C3D_P3DCIFINFO;
```

Members

u32Size	size of C3D_3DCIFINFO structure
u32FrameBuffBase	host pointer to the base of the frame buffer
u32OffScreenHeap	host pointer to the off-screen heap
u32OffScreenSize	size of the off-screen heap
u32TotalRAM	total amount of video RAM on the graphics board
u32ASICID	RAGE ASIC ID
u32ASICRevision	RAGE ASIC revision
u32CIFCaps1	ATI3DCIF module capabilities, field 1
u32CIFCaps2	ATI3DCIF module capabilities, field 2 (RAGE PRO)
u32CIFCaps3	ATI3DCIF module capabilities, field 3 (reserved for future use)
u32CIFCaps4	ATI3DCIF module capabilities, field 4 (reserved for future use)
u32CIFCaps5	ATI3DCIF module capabilities, field 5 (reserved for future use)

Description

This structure is used by the *ATI3DCIF_GetInfo* function to retrieve information about the graphics subsystem and the ATI 3D RAGE graphics accelerator. Prior to calling ATI3DCIF_GetInfo, the client application must set the u32Size member to the size of this structure. Otherwise, ATI3DCIF_GetInfo will fail.

In version 4.02.0217 of ATI3DCIF, the u32CIFCaps member was added to this structure. In version 4.03.0039 of ATI3DCIF, the u32CIFCaps member was renamed u32CIFCaps1, and four more capabilities fields, u32CIFCaps2 to u32CIFCaps5, were added to this structure. In version

4.03.2511 of ATI3DCIF, the u32CIFCaps2 member was defined to support additional capabilities under RAGE PRO. u32CIFCaps3 to u32CIFCaps5 are currently unused and are reserved for future use. The application must ensure that the ATI3DCIF module is version 4.03.0039 or greater to use the u32CIFCaps1 member, and version 4.03.2511 or greater to use the u32CIFCaps2 member. The ATI3DCIF.DLL version number may be determined by right-clicking on the file under Windows Explorer, selecting Properties, and clicking on the Version tab. ATI3DCIF.DLL is located in the Windows 95 SYSTEM directory.

The following table lists **u32CIFCaps1** flags:

C3D_CAPS1_BASE	baseline functionality
C3D_CAPS1_FOG	fog support
C3D_CAPS1_POINT	point primitive support
C3D_CAPS1_RECT	screen-aligned rectangle primitive support
C3D_CAPS1_Z_BUFFER	Z buffer support
C3D_CAPS1_CI4_TMAP	4 bit color index texture support
C3D_CAPS1_CI8_TMAP	8 bit color index texture support
C3D_CAPS1_LOAD_OBJECT	bus-master data loading support
C3D_CAPS1_DITHER_EN	dithering on/off support
C3D_CAPS1_ENH_PERSP	enhanced perspective levels available
C3D_CAPS1_SCISSOR	fixed origin clipping region support
C3D_CAPS1_PROFILE_IF	profile interface available

C3D_CAPS1_BASE represents the baseline functionality available in versions 4.02.0217 and earlier of ATI3DCIF. All other capabilities were added after version 4.02.0217.

The following table lists **u32CIFCaps2** flags:

C3D_CAPS2_TEXTURE_COMPOSITE	second texture and composite blend factor support
C3D_CAPS2_TEXTURE_CLAMP	clamp texture coordinates to 1.0 enable/disable
C3D_CAPS2_DESTINATION_ALPHA_BLEND	extended alpha blending modes supported
C3D_CAPS2_TEXURE_TILING	texture tiling support

See Also

ATI3DCIF_GetInfo

C3D_CODEBOOKENTRY

Version

1.0

Syntax

```
typedef struct {
    C3D_UNIT16 ul;
    C3D_UNIT16 ur;
    C3D_UNIT16 ll;
    C3D_UNIT16 lr;
} C3D_CODEBOOKENTRY, * C3D_PCODEBOOKENTRY;
```

Members

ul	upper-left of 2x2 block
ur	upper-right of 2x2 block
ll	lower-left of 2x2 block
lr	lower-right of 2x2 block

Description

This structure is used to specify a single code book entry for VQ compressed textures. An array of 256 of these entries will be passed into *ATI3DCIF_TexturePaletteCreate*, along with the C3D_ECI_TMAP_VQ enumeration constant, to specify that a VQ texture be created, Upon successful creation, a valid codebook handle will be returned.

When finished with the texture, the codebook should be destroyed by calling the *ATI3DCIF_TexturePaletteDestroy* function, with the codebook's handle as an argument.

See Also

None

C3D_COLOR

Version

1.0

Syntax

```
typedef union
    struct {
          unsigned r: 8;
          unsigned g: 8;
          unsigned b: 8;
          unsigned a: 8;
    } ;
    C3D_UINT32 u32All
} C3D_COLOR, * C3D_PCOLOR;
```

Members

r	red color component
g	green color component
b	blue color component
a	alpha color component

Description

This structure is used to specify the RGBA colors when setting the background and solid colors of the rendering context and the texel transparency chroma key color in the *C3D_TMAP* structure. The C3D_TMAP structure is used to provide texture information to the ATI3DCIF module when registering a texture map.

On context creation, the solid color is set to black (RGBA = {0, 0, 0, 0}). To modify the solid color, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_SOLID_CLR and pRStateData set to the address of a C3D_COLOR structure specifying the new color.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*, *C3D_TMAP*

C3D_EACMP

Version

1.0

Syntax

```
typedef enum {
    C3D_EACMP_NEVER = 0,
    C3D_EACMP_LESS = 1,
    C3D_EACMP_LEQUAL = 2,
    C3D_EACMP_EQUAL = 3,
    C3D_EACMP_GEQUAL = 4,
    C3D_EACMP_GREATER = 5,
    C3D_EACMP_NOTEQUAL = 6,
    C3D_EACMP_ALWAYS = 7,
    C3D_EACMP_MAX = 8,
    C3D_EACMP_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EACMP, * C3D_PEACMP;
```

Constants

C3D_EACMP_NEVER	alpha compare never passes
C3D_EACMP_LESS	alpha compare passes if write select alpha is less than reference alpha
C3D_EACMP_LEQUAL	alpha compare passes if write select alpha is less than or equal to reference alpha
C3D_EACMP_EQUAL	alpha compare passes if write select alpha is equal to reference alpha
C3D_EACMP_GEQUAL	alpha compare passes if write select alpha is greater than or equal to reference alpha
C3D_EACMP_GREATER	alpha compare passes if write select alpha is greater than reference alpha
C3D_EACMP_NOTEQUAL	alpha compare passes if write select alpha is not equal to reference alpha
C3D_EACMP_ALWAYS	alpha compare always passes
C3D_EACMP_MAX	invalid enumeration

Description

C3D_EACMP constants specify the compare function to use during destination alpha testing. The compare function compares the alpha value from the current alpha write source with a reference alpha value maintained in the rendering context. If the compare passes, the primitive's RGB data is written to the destination RGB channels, and the alpha value from the alpha write source is written to the destination alpha channel. Otherwise, the destination reamins unchanged.

The alpha write source is represented by the *C3D_EASEL* enumeration and is selected by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_ALPHA_DST_WRITE_SELECT. The reference alpha value is set by calling ATI3DCIF_ContextSetState with the state flag

C3D_ERS_ALPHA_DST_REFERENCE and the state data pointing to a DWORD representing the reference alpha value. Note that the reference value must be in the range of the alpha channel bit depth. For example, if the channel is eight bits wide, as in the case of the ARGB8888 pixel format, the reference alpha must be in the range of 0 to 255.

The compare function is set by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_ALPHA_DST_TEST_FNC and the state data pointing to a C3D_EACMP enum. Destination alpha testing is enabled and disabled by calling ATI3DCIF_ContextSetState with the state flag C3D_ERS_ALPHA_DST_TEST_ENABLE and the state data pointing to a BOOL variable specifying the enable state.

See Also

ATI3DCIF_ContextSetState, *C3D_EASEL*

C3D_EADST

Version

1.0

Syntax

```
typedef enum {
    C3D_EADST_ZERO = 0,
    C3D_EADST_ONE = 1,
    C3D_EADST_SRCCLR = 2,
    C3D_EADST_INVSRCCLR = 3,
    C3D_EADST_SRCALPHA = 4,
    C3D_EADST_INVSRCALPHA = 5,
    C3D_EADST_DSTALPHA = 6,          // (RAGE PRO)
    C3D_EADST_INVDSTALPHA = 7,       // (RAGE PRO)
    C3D_EADST_NUM = 8,
    C3D_EADST_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EADST, * C3D_PEADST;
```

Constants

C3D_EADST_ZERO	Blend factor is (0, 0, 0)
C3D_EADST_ONE	Blend factor is (1, 1, 1)
C3D_EADST_SRCCLR	Blend factor is (Rs, Gs, Bs), where (Rs, Gs, Bs) is the source RGB color
C3D_EADST_INVSRCCLR	Blend factor is (1-Rs, 1-Gs, 1-Bs)
C3D_EADST_SRCALPHA	Blend factor is (As, As, As), where As is the source alpha value
C3D_EADST_INVSRCALPHA	Blend factor is (1-As, 1-As, 1-As)
C3D_EADST_DSTALPHA	Blend factor is (Ad, Ad, Ad) (RAGE PRO)
C3D_EADST_INVDSTALPHA	Blend factor is (1-Ad, 1-Ad, 1-Ad) (RAGE PRO)
C3D_EADST_NUM	invalid enumeration

Description

C3D_EADST constants represent the destination alpha blending factors which may be set for the rendering context. Alpha blending is performed according to the following equation:

destination color = (source color x source alpha factor) + (destination color x destination alpha factor)

The source alpha blending factors are represented by the *C3D_EASRC* enumeration.

The default mode set on context creation is C3D_EADST_ZERO. To modify the destination alpha blending factor, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_ALPHA_DST and pRStateData set to the address of a C3D_EADST object specifying the new state.

See Also

ATI3DCIF_ContextSetState, *C3D_EASRC*, *C3D_ERSID*

C3D_EASEL

Version

1.0

Syntax

```
typedef enum {
    C3D_EASEL_ZERO = 0,
    C3D_EASEL_ONE = 1,
    C3D_EASEL_SRCALPHA = 4,
    C3D_EASEL_INVSRCALPHA = 5,
    C3D_EASEL_DSTALPHA = 6,
    C3D_EASEL_INVDSTALPHA = 7,
    C3D_EASEL_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EASEL, *C3D_PEASEL;
```

Constants

C3D_EASEL_ZERO	write 0 to all alpha bits
C3D_EASEL_ONE	write 1 to all alpha bits
C3D_EASEL_SRCALPHA	write source alpha
C3D_EASEL_INVSRCALPHA	write 1 - source alpha
C3D_EASEL_DSTALPHA	write destination alpha
C3D_EASEL_INVDSTALPHA	write 1 - destination alpha

Description

C3D_EASEL constants specify the alpha data written to the destination alpha channel if destination alpha testing is enabled and the current alpha test compare function has passed the pixel write operation. For example, if C3D_EASEL_ZERO is selected, all destination alpha bits will be set to zero. The alpha write data is selected by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_ALPHA_DST_WRITE_SELECT and the state data pointing to a C3D_EASEL enum set to the desired alpha write data. The destination alpha test compare function is selected by calling ATI3DCIF_ContextSetState with the state flag C3D_ERS_ALPHA_DST_WRITE_FNC and the state data pointing to a *C3D_EACMP* enum set to the compare function. Destination alpha testing is enabled and disabled by calling ATI3DCIF_ContextSetState with the state flag C3D_ERS_ALPHA_DST_WRITE_EN and the state data pointing to a BOOL variable specifying the enable state.

See Also

ATI3DCIF_ContextSetState, *C3D_EACMP*

C3D_EASRC

Version

1.0

Syntax

```
typedef enum {
    C3D_EASRC_ZERO = 0,
    C3D_EASRC_ONE = 1,
    C3D_EASRC_DSTCLR = 2,
    C3D_EASRC_INVDSTCLR = 3,
    C3D_EASRC_SRCALPHA = 4,
    C3D_EASRC_INVSRCALPHA = 5,
    C3D_EASRC_DSTALPHA = 6,          // (RAGE PRO)
    C3D_EASRC_INVDSTALPHA = 7,       // (RAGE PRO)
    C3D_EASRC_NUM = 8,
    C3D_EASRC_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EASRC, * C3D_PEASRC;
```

Constants

C3D_EASRC_ZERO	Blend factor is (0, 0, 0)
C3D_EASRC_ONE	Blend factor is (1, 1, 1)
C3D_EASRC_DSTCLR	Blend factor is (Rd, Gd, Bd), where (Rd, Gd, Bd) is the destination RGB color
C3D_EASRC_INVDSTCLR	Blend factor is (1-Rd, 1-Gd, 1-Bd)
C3D_EASRC_SRCALPHA	Blend factor is (As, As, As), where As is the source alpha value
C3D_EASRC_INVSRCALPHA	Blend factor is (1-As, 1-As, 1-As)
C3D_EASRC_DSTALPHA	Blend factor is (Ad, Ad, Ad) (RAGE PRO)
C3D_EASRC_INVDSTALPHA	Blend factor is (1-Ad, 1-Ad, 1-Ad) (RAGE PRO)
C3D_EASRC_NUM	invalid enumeration

Description

C3D_EASRC constants represent the source alpha blending factors which may be set for the rendering context. Alpha blending is performed according to the following equation:

destination color = (source color x source alpha factor) + (destination color x destination alpha factor)

The destination alpha blending factors are represented by the *C3D_EADST* enumeration.

The default mode set on context creation is C3D_EASRC_ONE. To modify the source alpha blending factor, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_ALPHA_SRC and pRStateData set to the address of a C3D_EASRC object specifying the new state.

See Also

ATI3DCIF_ContextSetState, *C3D_EADST*, *C3D_ERSID*

C3D_EC

Version

1.0

Syntax

```
typedef enum {
    C3D_EC_OK = 0,
    C3D_EC_GENFAIL = 1,
    C3D_EC_MEMALLOCFAIL = 2,
    C3D_EC_BADPARM = 3,
    C3D_EC_UNUSED0 = 4,
    C3D_EC_BADSTATE = 5,
    C3D_EC_NOTIMPYET = 6,
    C3D_EC_UNUSED1 = 7,
    C3D_EC_CHIPCAPABILITY = 8,
    C3D_EC_NUM = 9,
    C3D_EC_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EC, * C3D_PEC;
```

Constants

C3D_EC_OK	success
C3D_EC_GENFAIL	generic failure
C3D_EC_MEMALLOCFAIL	memory allocation failure
C3D_EC_BADPARM	invalid parameter passed to function
C3D_EC_UNUSED0	not used
C3D_EC_BADSTATE	object entered invalid state
C3D_EC_NOTIMPYET	functionality not implemented yet
C3D_EC_UNUSED1	not used
C3D_EC_CHIPCAPABILITY	feature not available on this version of 3D RAGE
C3D_EC_NUM	invalid enumeration

Description

C3D_EC constants represent the error codes which may be returned by ATI3DCIF functions.

See Also

None.

C3D_ECI_TMAP_TYPE

Version

1.0

Syntax

```
typedef enum {
    C3D_ECI_TMAP_TRUE_COLOR = 0,
    C3D_ECI_TMAP_4BIT_HI = 1,
    C3D_ECI_TMAP_4BIT_LOW = 2,
    C3D_ECI_TMAP_8BIT = 3,
    C3D_ECI_TMAP_VQ = 4,            // (RAGE PRO)
    C3D_ECI_TMAP_NUM = 5,
    C3D_ECI_TMAP_FORCE_U32  = C3D_FORCE_SIZE
} C3D_ECI_TMAP_TYPE;
```

Constants

C3D_ECI_TMAP_TRUE_COLOR	texture format is true color: no palette
C3D_ECI_TMAP_4BIT_HI	texture format is CI4 packed in high nibble: 16 entry palette
C3D_ECI_TMAP_4BIT_LOW	texture format is CI4 packed in low nibble: 16 entry palette
C3D_ECI_TMAP_8BIT	texture format is CI8: 256 entry palette
C3D_ECI_TMAP_VQ	texture format is 256 entry codebook (RAGE PRO)
C3D_ECI_TMAP_NUM	invalid enumeration

Description

C3D_ECI_TMAP_TYPE constants are used in the function *ATI3DCIF_TexturePaletteCreate* to specify what kind of texture palette to create.

NOTE: Texture palettes are only available with the 3D RAGE II graphics accelerator or later.

See Also

ATI3DCIF_TexturePaletteCreate

C3D_EPIXFMT

Version

1.0

Syntax

```
typedef enum {
    C3D_EPF_RGB1555 = 3,
    C3D_EPF_RGB565 = 4,
    C3D_EPF_RGB8888 = 5,
    C3D_EPF_RGB332 = 6,
    C3D_EPF_Y8 = 7,
    C3D_EPF_YUV422 = 8,
    C3D_EPF_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EPIXFMT, * C3D_PEPIXFMT;
```

Constants

C3D_EPF_RGB1555	1 bit alpha, 5 bits red, 5 bits green, 5 bits blue
C3D_EPF_RGB565	0 bits alpha, 5 bits red, 6 bits green, 5 bits blue
C3D_EPF_RGB8888	8 bit alpha, 8 bits red, 8 bits green, 8 bits blue
C3D_EPF_RGB332	0 bit alpha, 3 bits red, 3 bits green, 2 bits blue
C3D_EPF_Y8	8 bits Y
C3D_EPF_YUV422	the pixel format is YUV422 packed YUYV

Description

C3D_EPIXFMT constants are used to specify the pixel format of the drawing surface.

On context creation, the drawing surface pixel format is set to that of the desktop. To modify the drawing surface pixel format, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_SURF_DRAW_PF and pRStateData set to the address of a C3D_EPIXFMT object specifying the new state.

See Also

ATI3DCIF_ContextSetState

C3D_EPRIM

Version

1.0

Syntax

```
typedef enum {
    C3D_EPRIM_LINE = 0,
    C3D_EPRIM_TRI = 1,
    C3D_EPRIM_QUAD = 2,
    C3D_EPRIM_RECT = 3,
    C3D_EPRIM_POINT = 4,
    C3D_EPRIM_NUM = 5,
    C3D_EPRIM_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EPRIM, * C3D_PEPRIM;
```

Constants

C3D_EPRIM_LINE	line primitive
C3D_EPRIM_TRI	triangle list or strip primitive
C3D_EPRIM_QUAD	quadrilateral list or strip primitive
C3D_EPRIM_RECT	screen aligned rectangle strip or list primitive
C3D_EPRIM_POINT	point list or strip primitive
C3D_EPRIM_NUM	invalid enumeration

Description

C3D_EPRIM constants represent the primitive types which may be rendered within the rendering context when drawing a primitive list or strip. The default mode set on context creation is C3D_EPRIM_TRI. To modify the primitive type, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_PRIM_TYPE and pRStateData set to the address of a C3D_EPRIM object specifying the new state.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*

NOTE: The C3D_EPRIM_RECT and C3D_EPRIM_POINT types are not available on some earlier versions of ATI3DCIF. Applications should call *ATI3DCIF_GetInfo* and query the u32CIFCaps1 member of the *C3D_3DCIFINFO* structure to verify the availability of these primitive types.

C3D_ERSID

Version

1.0

Syntax

```
typedef enum {
    C3D_ERS_FG_CLR = 0,
    C3D_ERS_VERTEX_TYPE = 1,
    C3D_ERS_PRIM_TYPE = 2,
    C3D_ERS_SOLID_CLR = 3,
    C3D_ERS_SHADE_MODE = 4,
    C3D_ERS_TMAP_EN = 5,
    C3D_ERS_TMAP_SELECT = 6,
    C3D_ERS_TMAP_LIGHT = 7,
    C3D_ERS_TMAP_PERSP_COR = 8,
    C3D_ERS_TMAP_FILTER = 9,
    C3D_ERS_TMAP_TEXOP = 10,
    C3D_ERS_ALPHA_SRC = 11,
    C3D_ERS_ALPHA_DST = 12,
    C3D_ERS_SURF_DRAW_PTR = 13,
    C3D_ERS_SURF_DRAW_PITCH = 14,
    C3D_ERS_SURF_DRAW_PF = 15,
    C3D_ERS_SURF_VPORT = 16,
    C3D_ERS_FOG_EN = 17,
    C3D_ERS_DITHER_EN = 18,
    C3D_ERS_Z_CMP_FCN = 19,
    C3D_ERS_Z_MODE = 20,
    C3D_ERS_SURF_Z_PTR = 21,
    C3D_ERS_SURF_Z_PITCH = 22,
    C3D_ERS_SURF_SCISSOR = 23,
    C3D_ERS_COMPOSITE_EN = 24,
    C3D_ERS_COMPOSITE_SELECT= 25,
    C3D_ERS_COMPOSITE_FNC = 26,
    C3D_ERS_COMPOSITE_FACTOR = 27,
    C3D_ERS_COMPOSITE_FILTER = 28,
    C3D_ERS_COMPOSITE_FACTOR_ALPHA = 29,
    C3D_ERS_LOD_BIAS_LEVEL = 30,
    C3D_ERS_ALPHA_DST_TEST_ENABLE = 31,
    C3D_ERS_ALPHA_DST_TEST_FNC = 32,
    C3D_ERS_ALPHA_DST_WRITE_SELECT = 33,
    C3D_ERS_ALPHA_DST_REFERENCE = 34,
    C3D_ERS_SPECULAR_EN = 35,
    C3D_ERS_NUM = 36,
    C3D_ERS_FORCE_U32 = C3D_FORCE_SIZE
} C3D_ERSID, * C3D_PERSID;
```

Constants

C3D_ERS_FG_CLR	set fog color
C3D_ERS_VERTEX_TYPE	set vertex structure type
C3D_ERS_PRIM_TYPE	set primitive type
C3D_ERS_SOLID_CLR	set solid color
C3D_ERS_SHADE_MODE	set primitive shading mode
C3D_ERS_TMAP_EN	enable or disable texture mapping
C3D_ERS_TMAP_SELECT	select texture map
C3D_ERS_TMAP_LIGHT	set texture lighting method
C3D_ERS_TMAP_PERSP_COR	set texture perspective correction level
C3D_ERS_TMAP_FILTER	set texture filtering method
C3D_ERS_TMAP_TEXOP	set texture rendering operation
C3D_ERS_ALPHA_SRC	set source alpha blending factor
C3D_ERS_ALPHA_DST	set destination alpha blending factor
C3D_ERS_SURF_DRAW_PTR	set draw surface address
C3D_ERS_SURF_DRAW_PITCH	set draw surface pitch
C3D_ERS_SURF_DRAW_PF	set draw surface pixel format
C3D_ERS_SURF_VPORT	set draw surface viewport coordinates
C3D_ERS_FOG_EN	enable or disable fog
C3D_ERS_DITHER_EN	enable or disable dither
C3D_ERS_Z_CMP_FNC	set Z compare function
C3D_ERS_Z_MODE	set Z testing mode
C3D_ERS_SURF_Z_PTR	set Z buffer address
C3D_ERS_SURF_Z_PITCH	set Z buffer pitch
C3D_ERS_SURF_SCISSOR	set draw surface clipping coordinates
C3D_ERS_COMPOSITE_EN	enable texture compositing
C3D_ERS_COMPOSITE_SELECT	select secondary composite texture. Primary composite texture is selected by C3D_ERS_TMAP_SELECT
C3D_ERS_COMPOSITE_FNC	select texture composite function
C3D_ERS_COMPOSITE_FACTOR	select blending factor for texture composite function
C3D_ERS_COMPOSITE_FILTER	set texture filtering method for secondary composite texture
C3D_ERS_COMPOSITE_FACTOR_ALPHA	force blend factor for blend texture compositing function to be taken from composite texture's alpha channel
C3D_ERS_LOD_BIAS_LEVEL	set LOD bias for mipmap level switching
C3D_ERS_ALPHA_DST_TEST_ENABLE	enable destination alpha testing
C3D_ERS_ALPHA_DST_TEST_FNC	select destination alpha test compare function
C3D_ERS_ALPHA_DST_WRITE_SELECT	select destination alpha test alpha write source
C3D_ERS_ALPHA_DST_REFERENCE	set destination alpha testing reference alpha
C3D_ERS_SPECULAR_EN	enable specular lighting
C3D_ERS_NUM	invalid enumeration

Description

This enumeration is used to specify the state to change when calling *ATI3DCIF_ContextSetState*. A valid C3D_ERSID constant must be passed as the eRStateID argument to change the associated state for the context referenced by hRC.

NOTE: z-buffers are only supported in the 3D RAGE II graphics accelerator or later.

See Also

ATI3DCIF_ContextSetState

C3D_ESHADE

Version

1.0

Syntax

```
typedef enum {
    C3D_ESH_NONE = 0,
    C3D_ESH_SOLID = 1,
    C3D_ESH_FLAT = 2,
    C3D_ESH_SMOOTH = 3,
    C3D_ESH_NUM = 4,
    C3D_ESH_FORCE_U32 = C3D_FORCE_SIZE
} C3D_ESHADE, * C3D_PESHADE;
```

Constants

C3D_ESH_NONE	shading mode is undefined
C3D_ESH_SOLID	primitives are shaded according to the rendering context solid color. On context creation, this color is set to black. It may be modified by calling *ATI3DCIF_ContextSetState*.
C3D_ESH_FLAT	primitives are flat shaded according to the color of the last vertex in each triangle or quadrilateral in the primitive list or strip. The rendering context vertex structure type must include r, g, b, and a color members.
C3D_ESH_SMOOTH	primitives are Gouraud shaded according to the color of each vertex in the triangle or quadrilateral in the primitive list or strip. The primitive color is interpolated from one vertex to the other, resulting in a smooth gradation over the entire primitive. The rendering context vertex structure type must include r, g, b, and a color members.
C3D_ESH_NUM	invalid enumeration

Description

C3D_ESHADE constants represent the primitive shading modes which may be set for the rendering context. The default mode set on context creation is C3D_ESH_SMOOTH. To modify the shading mode, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_SHADE_MODE and pRStateData set to the address of a C3D_ESHADE object specifying the new state.

If texture mapping is enabled and the texture lighting state is set to C3D_ETL_MODULATE, the color of each texel will be modulated by the color of the primitive as defined by the shading mode. Therefore, texels may be modulated by the solid, flat, or Gouraud shaded color of the primitive.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*, *C3D_ETLIGHT*

C3D_ETEXCOMPFCN

Version

1.0

Syntax

```
typedef enum {
    C3D_ETEXCOMPFCN_BLEND = 0,
    C3D_ETEXCOMPFCN_MOD = 1,
    C3D_ETEXCOMPFCN_MAX = 3,
    C3D_ETEXCOMPFCN_FORCE_U32 = C3D_FORCE_SIZE
} C3D_ETEXCOMPFCN, * C3D_PETEXCOMPFCN;
```

Constants

C3D_ETEXCOMPFCN_BLEND	composite texel = (primary texel x (1-(blend factor/16))) + (secondary texel x blend factor/16)
C3D_ETEXCOMPFCN_MOD	composite texel = primary texel x secondary texel
C3D_ETEXCOMPFCN_MAX	invalid enumeration

Description

C3D_ETEXCOMPFCN constants specify the texture compositing function to use if texture compositing is enabled. The default texture compositing function is set to C3D_ETEXCOMPFCN_BLEND on context creation. For the blend compositing function represented by C3D_ETEXCOMPFCN_BLEND, the blend factor must be an integer value between 0 and 15, which is set by calling *ATI3DCIF_ContextSetState* with the state flag C3D_ERS_COMPOSITE_FACTOR and the state data set to the desired blend factor.

See Also

ATI3DCIF_ContextSetState

C3D_ETEXFILTER

Version

1.0

Syntax

```
typedef enum {
    C3D_ETFILT_MINPNT_MAGPNT = 0,
    C3D_ETFILT_ MINPNT_MAG2BY2 = 1,
    C3D_ETFILT_MIN2BY2_MAG2BY2 = 2,
    C3D_ETFILT_MIPLIN_MAGPNT = 3,
    C3D_ETFILT_MIPLIN_MAG2BY2 = 4,
    C3D_ETFILT_MIPTRI_MAG2BY2 = 5,        // (RAGE PRO)
    C3D_ETFILT_MIN2BY2_MAGPNT = 6,        // (RAGE PRO
    C3D_ETFILT_NUM = 7,
    C3D_ETFILT_FORCE_U32 = C3D_FORCE_SIZE
} C3D_ETEXFILTER, * C3D_PETEXFILTER;
```

Constants

C3D_ETFILT_MINPNT_MAGPNT	pick-nearest on minification, pick-nearest on magnification
C3D_ETFILT_MINPNT_MAG2BY2	pick-nearest on minification, bi-linear on magnification
C3D_ETFILT_MIN2BY2_MAG2BY2	bi-linear on minification, bi-linear on magnification
C3D_ETFILT_MIPLIN_MAGPNT	mip-linear on minification, pick-nearest on magnification
C3D_ETFILT_MIPLIN_MAG2BY2	mip-linear on minification, bi-linear on magnification
C3D_ETFILT_MIPTRI_MAG2BY2	tri-linear on minification, bi-linear on magnification (RAGE PRO)
C3D_ETFILT_MIN2BY2_MAGPNT	bilinear on minification, pick nearest on magnification (RAGE PRO)
C3D_ETFILT_NUM	invalid enumeration

Description

C3D_ETEXFILTER constants represent the texel filtering modes which may be set for the rendering context. The default mode set on context creation is C3D_ETFILT_MINPNT_MAG2BY2. This mode causes texels to be blended bi-linearly on magnification, and selected by the pick-nearest criterion on minification.

To modify the texel filtering mode, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_TMAP_FILTER and pRStateData set to the address of a C3D_ETEXFILTER object specifying the new state.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*

C3D_ETEXFMT

Version

1.0

Syntax

```
typedef enum {
    C3D_ETF_CI4 = 1,
    C3D_ETF_CI8 = 2,
    C3D_ETF_RGB1555 = 3,
    C3D_ETF_RGB565 = 4,
    C3D_ETF_RGB8888 = 6,
    C3D_ETF_RGB332 = 7,
    C3D_ETF_Y8 = 8,
    C3D_ETF_YUV422 = 11,
    C3D_ETF_RGB4444 = 15,
    C3D_ETF_VQ = 20,                // (RAGE PRO)
    C3D_ETF_FORCE_U32 = C3D_FORCE_SIZE
} C3D_ETEXFMT, * C3D_PETEXFMT;
```

Constants

C3D_ETF_CI4	4 bpp index into palette (pseudo color)
C3D_ETF_CI8	8 bpp index into palette (pseudo color)
C3D_ETF_RGB1555	1 bit alpha, 5 bits red, 5 bits green, 5 bits blue
C3D_ETF_RGB565	0 bits alpha, 5 bits red, 6 bits green, 5 bits blue
C3D_ETF_RGB8888	8 bits alpha, 8 bits red, 8 bits green, 8 bits blue
C3D_ETF_RGB332	0 bits alpha, 3 bits red, 3 bits green, 2 bits blue
C3D_ETF_Y8	8 bits Y
C3D_ETF_YUV422	the pixel format is YUV 422 packed YUYV
C3D_ETF_RGB4444	4 bits alpha, 4 bits red, 4 bits green, 4 bits blue
C3D_ETF_VQ	VQ compressed texture

Description

C3D_ETEXFMT constants are used to specify a texture's texel format in the eTexFormat member of the *C3D_TMAP* structure. This structure is used to provide texture information to the ATI3DCIF module when registering a texture map.

NOTE: The C3D_ETF_CI4 and C3D_ETF_CI8 formats are only available with the 3D RAGE II graphics accelerator or later. All other formats are available with the entire 3D RAGE accelerator family.

See Also

ATI3DCIF_ContextSetState

C3D_ETEXOP

Version

1.0

Syntax

```
typedef enum {
    C3D_ETEXOP_NONE = 0,
    C3D_ETEXOP_CHROMAKEY = 1,
    C3D_ETEXOP_ALPHA  = 2,
    C3D_ETEXOP_ALPHA_MASK = 3,
    C3D_ETEXOP_NUM =  4,
    C3D_ETEXOP_FORCE_U32 = C3D_FORCE_SIZE
} C3D_ETEXOP, * C3D_PETEXOP;
```

Constants

C3D_ETEXOP_NONE	all texels are rendered
C3D_ETEXOP_CHROMAKEY	texels are not rendered if equal to the chroma key color
C3D_ETEXOP_ALPHA	texels are alpha blended by passing the texel alpha to the alpha blender
C3D_ETEXOP_ALPHA_MASK	texels are not rendered if the least significant bit in the alpha channel is set to 0
C3D_ETEXOP_NUM	invalid enumeration

Description

C3D_ETEXOP constants represent the texel rendering operations which may be performed during texture mapping. The default mode set on context creation is C3D_ETEXOP_NONE. Texel operations include texel transparency based on chroma keying or alpha masking, and alpha blending using the texel alpha channel.

To modify the texel render operation, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_TMAP_TEXOP and pRStateData set to the address of a C3D_ETEXOP object specifying the new state.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*

C3D_ETLIGHT

Version

1.0

Syntax

```
typedef enum {
    C3D_ETL_NONE = 0,
    C3D_ETL_MODULATE = 1,
    C3D_ETL_ALPHA_DECAL = 2,
    C3D_ETL_NUM = 3,
    C3D_ETL_FORCE_U32 = C3D_FORCE_SIZE
} C3D_ETLIGHT, * C3D_PETLIGHT;
```

Constants

C3D_ETL_NONE	texel colors are not modulated
C3D_ETL_MODULATE	texel colors are modulated according to the shading mode. If the shading mode is C3D_ERS_FLAT or C3D_ERS_SMOOTH, texel colors are modulated by the color of the primitives in flat or Gouraud shading, respectively. If the shading mode is C3D_ERS_SOLID, texel colors are modulated according to the rendering context solid color.
C3D_ETL_ALPHA_DECAL	texel colors are modulated by a combination of the primitive shading color and an alpha value supplied in the texture map. The texel value becomes (texel color x texel alpha) + (primitive color x (1 - texel alpha)), where the primitive color is determined by the shading mode.
C3D_ETL_NUM	invalid enumeration

Description

C3D_ETLIGHT constants represent the texture lighting modes which may be set for the rendering context. The default mode set on context creation is C3D_ETL_NONE. To modify the texture lighting mode, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_TMAP_LIGHT and pRStateData set to the address of a C3D_ETLIGHT object specifying the new state.

If the texture lighting method is set to C3D_ETL_MODULATE, the color of each texel will be modulated by the color of the primitive as defined by the shading mode. Therefore, texels will be modulated by the solid, flat, or Gouraud shaded color of the primitive. Shading modes are defined by the *C3D_ESHADE* enumeration constants.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*, *C3D_ESHADE*

C3D_ETPERSPCOR

Version

1.0

Syntax

```
typedef enum {
    C3D_ETPC_NONE = 0,
    C3D_ETPC_ONE = 1,
    C3D_ETPC_TWO = 2,
    C3D_ETPC_THREE = 3,
    C3D_ETPC_FOUR = 4,
    C3D_ETPC_FIVE = 5,
    C3D_ETPC_SIX = 6,
    C3D_ETPC_SEVEN = 7,
    C3D_ETPC_EIGHT = 8,
    C3D_ETPC_NINE = 9,
    C3D_ETPC_NUM = 10,
    C3D_ETPC_FORCE_U32 = C3D_FORCE_SIZE
} C3D_ETPERSPCOR, * C3D_PETPERSPCOR;
```

Constants

C3D_ETPC_NONE	no perspective correction
C3D_ETPC_ONE	level one perspective correction
C3D_ETPC_TWO	level two perspective correction
C3D_ETPC_THREE	level three perspective correction
C3D_ETPC_FOUR	level four perspective correction
C3D_ETPC_FIVE	level five perspective correction
C3D_ETPC_SIX	level six perspective correction
C3D_ETPC_SEVEN	level seven perspective correction
C3D_ETPC_EIGHT	level eight perspective correction
C3D_ETPC_NINE	level nine perspective correction
C3D_ETPC_NUM	invalid enumeration

Description

C3D_ETPERSPCOR constants represent the texture perspective correction levels which may be set for the rendering context. The default mode set on context creation is C3D_ETL_NONE. To modify the perspective correction level, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_TMAP_ PERSP_COR and pRStateData set to the address of a C3D_ETPERSPCOR object specifying the new state.

Texture perspective correction produces better image quality. However, this comes at the expense of frame rate. The frame rate will vary inversely to the level of perspective correction set. Level C3D_ETPC_NONE will offer no correction but the fastest frame rate, whereas level C3D_ETPC_NINE will offer full correction but the poorest frame rate. The recommended level which offers the best compromise is level C3D_ETPC_THREE. For a full discussion of performance and image quality issues, see *Chapter 6*, *3D RAGE / ATI3DCIF Porting and Performance Notes*.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*, *C3D_ESHADE*

C3D_EVERTEX

Version

1.0

Syntax

```
typedef enum {
    C3D_EV_VF = 0,
    C3D_EV_VCF = 1,
    C3D_EV_VTF = 2,
    C3D_EV_VTCF = 3,
    C3D_EV_TLVERTEX = 4,        // (RAGE PRO)
    C3D_EV_NUM = 5,
    C3D_EV_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EVERTEX, * C3D_PEVERTEX;
```

Constants

C3D_EV_VF	vertex defined by C3D_VF structure
C3D_EV_VCF	vertex defined by C3D_VCF structure
C3D_EV_VTF	vertex defined by C3D_VTF structure
C3D_EV_VTCF	vertex defined by C3D_VTCF structure
C3D_EV_TLVERTEX	vertex defined by C3D_TLVERTEX structure (RAGE PRO)
C3D_EV_NUM	invalid enumeration

Description

C3D_EVERTEX constants represent the vertex structures which may represent vertex data within the rendering context. The default mode set on context creation is C3D_EV_VTCF. To modify the vertex structure type, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_VERTEX_TYPE and pRStateData set to the address of a C3D_EVERTEX object specifying the new state.

All vertex structures contain x, y, and z location coordinate members. Vertex structures with a C in the structure name also contain r, g, b, and a color component members. Vertex structures with a T in the structure name contain s, t, and w texture coordinate members. All vertex structures represent data in floating point format.

To perform flat or Gouraud shading and source or destination alpha blending, the vertex structure must contain r, g, b, and a members. To perform texture mapping, the vertex structure must contain s, t and w members (or tu and tv if using the C3D_TLVERTEX structure). To modulate texel colors, the vertex structure must contain both color r, g, b, and a members and texture s, t, and w members (or tu and tv if using the C3D_TLVERTEX structure).

See Also

ATI3DCIF_ContextSetState

C3D_EZCMP

Version

1.0

Syntax

```
typedef enum {
    C3D_EZCMP_NEVER = 0,
    C3D_EZCMP_LESS = 1,
    C3D_EZCMP_LEQUAL = 2,
    C3D_EZCMP_EQUAL = 3,
    C3D_EZCMP_GEQUAL = 4,
    C3D_EZCMP_GREATER = 5,
    C3D_EZCMP_NOTEQUAL = 6,
    C3D_EZCMP_ALWAYS = 7,
    C3D_EZCMP_MAX = 8,
    C3D_EZCMP_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EZCMP, * C3D_PEZCMP;
```

Constants

C3D_EZCMP_NEVER	Z compare never passes
C3D_EZCMP_LESS	Z compare passes if test z is less than buffered z
C3D_EZCMP_LEQUAL	Z compare passes if test z is less than or equal to buffered z
C3D_EZCMP_EQUAL	Z compare passes if test z is equal to buffered z
C3D_EZCMP_GEQUAL	Z compare passes if test z is greater than or equal to buffered z
C3D_EZCMP_GREATER	Z compare passes if test z is greater than buffered z
C3D_EZCMP_NOTEQUAL	Z compare passes if test z is not equal to buffered z
C3D_EZCMP_ALWAYS	Z compare always passes
C3D_EZCMP_MAX	invalid enumeration

Description

C3D_EZCMP constants specify what kind of Z compare function to use during z-buffer testing. The default Z compare function is set to C3D_EZCMP_ALWAYS on context creation. The compare function performs a logical operation to select or reject a pixel for rendering.

NOTE: z-buffers are only supported in the 3D RAGE II graphics accelerator or later.

See Also

None.

C3D_EZMODE

Version

1.0

Syntax

```
typedef enum {
    C3D_EZMODE_OFF  = 0,
    C3D_EZMODE_TESTON = 1,
    C3D_EZMODE_TESTON_WRITEZ = 2,
    C3D_EZMODE_MAX = 3,
    C3D_EZMODE_FORCE_U32 = C3D_FORCE_SIZE
} C3D_EZMODE, * C3D_PEZMODE;
```

Constants

C3D_EZMODE_OFF	Disable Z testing
C3D_EZMODE_TESTON	Test Z, do not update the z-buffer
C3D_EZMODE_TESTON_WRITEZ	Test Z, update the z-buffer
C3D_EZMODE_MAX	invalid enumeration

Description

C3D_EZMODE constants specify the state of z-buffer testing. The default z-buffer testing mode is set to C3D_EZMODE_OFF on context creation. Z-buffer testing can be disabled, set to test Z and update the z-buffer, or set to test Z and not modify the z-buffer.

NOTE: z-buffers are only supported in the 3D RAGE II graphics accelerator or later.

See Also

None.

C3D_HRC

Version

1.0

Syntax

typedef void* C3D_HRC;

Description

This handle identifies an ATI3DCIF rendering context created by calling *ATI3DCIF_ContextCreate*. This handle must be used to reference the rendering context when calling the following functions:

ATI3DCIF_ContextDestroy

ATI3DCIF_ContextSetState

ATI3DCIF_RenderBegin

ATI3DCIF_RenderSwitch

See Also

ATI3DCIF_ContextDestroy, *ATI3DCIF_ContextSetState*, *ATI3DCIF_RenderBegin*, *ATI3DCIF_RenderSwitch*

C3D_HTX

Version

1.0

Syntax

```
typedef void * C3D_HTX;
typedef C3D_HTX * C3D_PHTX;
```

Description

This handle identifies a texture map registered with the ATI3DCIF module. The handle is obtained by calling *ATI3DCIF_TextureReg* with a pointer to a *C3D_TMAP* structure describing the texture's attributes. To use the texture map, an application must select it by calling *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_TMAP_SELECT and pRStateData set to the address of its C3D_HTX handle.

See Also

ATI3DCIF_TextureReg, *ATI3DCIF_ContextSetState*, *C3D_TMAP*

C3D_HTXPAL

Version

1.0

Syntax

```
typedef void * C3D_HTXPAL;
typedef C3D_HTXPAL *C3D_PHTXPAL;
```

Description

This handle identifies a logical texture palette created internally within ATI3DCIF. Palettes associated with CI8 or CI4 textures must be created and then assigned to the textures in the htxpalTexPalette member of the *C3D_TMAP* structure before the textures are registered. The handle is obtained by calling *ATI3DCIF_TexturePaletteCreate*.

NOTE: Texture palettes, C3D_ETF_CI4, and C3D_ETF_CI8 texel formats are only available with the 3D RAGE II graphics accelerator or later. All other formats are available with the entire 3D RAGE accelerator family.

See Also

ATI3DCIF_TexturePaletteCreate, *C3D_TMAP*

C3D_PALETTENTRY

Version

1.0

Syntax

```
typedef union {
    struct {
            unsigned r: 8;
            unsigned g: 8;
            unsigned b: 8;
            unsigned flags: 8;
    };
    C3D_UINT32 u32All;
} C3D_PALETTENTRY , * C3D_PPALETTENTRY;
```

Members

r	8 bit red color component
g	8 bit green color component
b	8 bit blue color component
flags	flag controlling palette entry loading

Description

This structure is used to specify the color values of palette entries when creating a texture palette with *ATI3DCIF_TexturePaletteCreate*. When creating a palette for a CI8 texture, a 256 element C3D_PALETTENTRY array is used to specify the palette colors. When creating a palette for a CI4 texture, a 16 element C3D_PALETTENTRY array is used to specify the palette colors. r, g, and b specify the 8 bit RGB color components of each palette entry. flags control the loading of individual entries in the palette. If flags is set to C3D_LOAD_PALETTE_ENTRY, the physical palette entry corresponding to the C3D_PALETTENTRY element in the array will be replaced with the specified color. If flags is set to C3D_NO_LOAD_PALETTE_ENTRY, the physical palette entry will not be modified.

NOTE: Texture palettes, C3D_ETF_CI4, and C3D_ETF_CI8 texel formats are only available with the 3D RAGE II graphics accelerator or later. All other formats are available with the entire 3D RAGE accelerator family.

See Also

ATI3DCIF_TexturePaletteCreate

C3D_PRSDATA

Version

1.0

Syntax

```
typedef void* C3D_PRSDATA;
```

Description

The C3D_PRSDATA type is used in the function *ATI3DCIF_ContextSetState* to specify the address of the data object containing the new state data to set.

See Also

ATI3DCIF_ContextSetState

C3D_RECT

Version

1.0

Syntax

```
typedef struct {
    C3D_INT32 top;
    C3D_INT32 left;
    C3D_INT32 bottom;
    C3D_INT32 right;
} C3D_RECT, * C3D_PRECT;
```

Members

top	y coordinate of the upper left corner of the rectangle
left	x coordinate of the upper left corner of the rectangle
bottom	y coordinate of the bottom right corner of the rectangle
right	x coordinate of the bottom right corner of the rectangle

Description

This structure defines the upper left and bottom right corners of a rectangular region. It is used to pass the rectangular viewport coordinates to the *ATI3DCIF_ContextSetState* function when setting the viewport region of the drawing surface. Primitives are clipped to the bounds of the viewport region when rendered.

See Also

ATI3DCIF_ContextSetState

C3D_TLVERTEX

Version

1.0

Syntax

```
typedef struct {
        union {
                C3D_FLOAT32 sx;
                C3D_FLOAT32 x;
        };
        union {
                C3D_FLOAT32 sy;
                C3D_FLOAT32 y;
        };
        union {
                C3D_FLOAT32 sz;
                C3D_FLOAT32 z;
        };
        union {
                C3D_FLOAT32 rhw;
                C3D_FLOAT32 w;
        };
        union {
                C3D_UINT32  color;
                struct {
                        C3D_UINT8  b;
                        C3D_UINT8  g;
                        C3D_UINT8  r;
                        C3D_UINT8  a;
                };
        };
        union {
                C3D_UINT32  specular;
                struct {
                        C3D_UINT8  spec_b;
                        C3D_UINT8  spec_g;
                        C3D_UINT8  spec_r;
                        C3D_UINT8  spec_a;
                };
        };
        union {
                C3D_FLOAT32  tu;
                C3D_FLOAT32s;
        };
        union {
                C3D_FLOAT32  tv;
                C3D_FLOAT32  t;
        };
        struct {
            C3D_FLOAT32  reserved1;
            C3D_FLOAT32  reserved2;
            C3D_FLOAT32  reserved3;
        } composite;
} C3D_TLVERTEX;
```

Members

sx	vertex x coordinate
sy	vertex y coordinate
sz	vertex z coordinate
rhw	reciprocal of the homogeneous vertex w coordinate
color	vertex diffuse color
specular	vertex specular color
tu	non-homogeneous texture u coordinate
tv	non-homogeneous texture v coordinate
reserved	reserved for future use

Description

This structure may be used to describe a vertex in terms of its screen location coordinates, its u and v texture coordinates, *tu* and *tv*, and its diffuse and specular color components. If the rendering context uses this structure to represent vertices, flat or Gouraud shading, source or destination alpha blending, texture mapping and texel modulation can be performed. This structure represents color data members in unsigned format, all other data members are floating point. This structure is highly portable and is intended to replace older vertex types, although the C3D_VTCF vertex type will continue to be supported.

The tu and tv members of this structure represent non-homogenous u,v texture coordinates. That is s and t are derived from u and v as: s = u/w and t = v/w.

The default vertex type used to represent vertex data on context creation is C3D_VTCF. To modify the vertex structure type to use C3D_TLVERTEX, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_VERTEX_TYPE and pRStateData set to the address of an *C3D_EVERTEX* object containing C3D_EV_TLVERTEX.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*, *C3D_EVERTEX*, *C3D_VTCF*

C3D_TMAP

Version

1.0

Syntax

```
typedef struct {
      C3D_UINT32 u32Size;
      BOOL bMipMap;
      C3D_PVOID apvLevels [cu32MAX_TMAP_LEV];
      C3D_UINT32 u32MaxMapXSizeLg2;
      C3D_UINT32 u32MaxMapYSizeLg2;
      C3D_ETEXFMT eTexFormat;
      C3D_COLOR clrTexChromaKey;
      C3D_HTXPAL htxpalTexPalette;
      C3D_BOOL_bClampS;          // (RAGE PRO)
      C3D_BOOL_bClampT;          // (RAGE PRO)
      C3D_BOOL_bAlphaBlend;      // (RAGE PRO)
      C3D_ETEXTILEeTexTiling;    // (RAGE PRO)
} C3D_TMAP, * C3D_PTMAP;
```

Members

u32Size	size of C3D_TMAP structure
bMipMap	mip map enable flag
apvLevels	array of pointers to individual maps which compose a mip map texture
u32MaxMapXSizeLg2	the log 2 x-axis size of largest map
u32MaxMapYSizeLg2	the log 2 y-axis size of largest map
eTexFormat	texel format
clrTexChromaKey	texel transparency chroma key color
htxpalTexPalette	handle to texture palette
bClampS	smear/repeat s=1 if s>1 (RAGE PRO)
bClampT	smear/repeat t=1 if t>1 (RAGE PRO)
bAlphaBlend	use this texture's alpha as blend factor for the blend compositing factor (RAGE PRO)
eTexTiling	texture minimized for local reference (RAGE PRO)

Description

The C3D_TMAP structure is used to provide information describing a texture map to the ATI3DCIF module when registering the texture with the function *ATI3DCIF_TextureReg*. This information specifies how the hardware should interpret the texture cached in the frame buffer.

u32Size specifies the size of the C3D_TMAP structure. The client application must set this member to the size of the C3D_TMAP structure prior to calling *ATI3DCIF_TextureReg*. bMipMap set to TRUE signals that the texture is a mip map. If this is the case, the first element in the apvLevels array contains the address of the base map in the frame buffer, and subsequent elements contain the address of sequentially smaller maps. If bMipMap is FALSE, the first element contains the address of the texture map and the rest are ignored. The height and width of the texture maps must be in powers of 2 (e.g. 2, 4, 8, etc.) to a maximum of 1024 lines or pixels. u32MaxMapXSizeLg2 specifies the base 2 log of the width of the texture or the largest map in a mip map. u32MaxMapYSizeLg2 specifies the base 2 log of the height of the largest map. clrTexChromaKey specifies the transparency chroma key color. htxpalTexPalette is the handle to the texture's palette if the texture format is C3D_ETF_CI4 or C3D_ETF_CI8. The texture palette must have been created beforehand by calling *ATI3DCIF_TexturePaletteCreate*.

NOTE: Texture palettes, C3D_ETF_CI4, and C3D_ETF_CI8 texel formats are only available with the 3D RAGE II graphics accelerator or later. All other formats are available with the entire 3D RAGE accelerator family.

See Also

ATI3DCIF_TextureReg

C3D_VCF

Version

1.0

Syntax

```
typedef struct {
    C3D_FLOAT32 x;
    C3D_FLOAT32 y;
    C3D_FLOAT32 z;
    C3D_FLOAT32 r;
    C3D_FLOAT32 g;
    C3D_FLOAT32 b;
    C3D_FLOAT32 a;
} C3D_VCF, * C3D_PVCF;
```

Members

x	vertex x coordinate
y	vertex y coordinate
z	vertex z coordinate
r	vertex red color component
g	vertex green color component
b	vertex blue color component
a	vertex alpha value

Description

This structure may be used to describe a vertex in terms of its x, y, and z location coordinates and its r, g, b, and a color components. Because this structure does not contain texture coordinate members (s, t, and w), texture mapping cannot be performed if the rendering context uses this structure to represent vertices. This structure represents data members in floating point format.

The default vertex structure type set on context creation is C3D_VTCF. To modify the vertex structure type, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_VERTEX_TYPE and pRStateData set to the address of an *C3D_EVERTEX* object with the value C3D_EV_VCF.

See Also

ATI3DCIF_ContextSetState, C3D_ERSID, *C3D_EVERTEX*

C3D_VF

Version

1.0

Syntax

```
typedef struct {
    C3D_FLOAT32 x;
    C3D_FLOAT32 y;
    C3D_FLOAT32 z;
} C3D_VF, * C3D_PVF;
```

Members

x	vertex x coordinate
y	vertex y coordinate
z	vertex z coordinate

Description

This structure may be used to describe a vertex in terms of its x, y, and z location coordinates. Because this structure does not contain color component members (r, g, b, and a) and texture coordinate members (s, t, and w), flat or Gouraud shading, source or destination alpha blending, and texture mapping cannot be performed if the rendering context uses this structure to represent vertices. This structure represents data members in floating point format.

The default vertex structure type set on context creation is *C3D_VTCF*. To modify the vertex structure type, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_VERTEX_TYPE and pRStateData set to the address of an *C3D_EVERTEX* object with the value C3D_EV_VF.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*, *C3D_EVERTEX*

C3D_VLIST

Version

1.0

Syntax

typedef void ** C3D_VLIST;

Description

The C3D_VLIST type is used in the function *ATI3DCIF_RenderPrimList* to specify the array of pointers to the vertex structures representing the vertices in the primitive list.

See Also

ATI3DCIF_RenderPrimList

C3D_VSTRIP

Version

1.0

Syntax

typedef void * C3D_VSTRIP;

Description

The C3D_VSTRIP type is used in the function *ATI3DCIF_RenderPrimStrip* to specify the array of vertex structures representing the vertices in the primitive strip.

See Also

ATI3DCIF_RenderPrimStrip

C3D_VTCF

Version

1.0

Syntax

```
typedef struct {
    C3D_FLOAT32 x;
    C3D_FLOAT32 y;
    C3D_FLOAT32 z;
    C3D_FLOAT32 s;
    C3D_FLOAT32 t;
    C3D_FLOAT32 w;
    C3D_FLOAT32 r;
    C3D_FLOAT32 g;
    C3D_FLOAT32 b;
    C3D_FLOAT32 a;
} C3D_VTCF, * C3D_PVTCF;
```

Members

x	vertex x coordinate
y	vertex y coordinate
z	vertex z coordinate
s	vertex s texture coordinate
t	vertex t texture coordinate
w	vertex w texture coordinate
r	vertex red color component
g	vertex green color component
b	vertex blue color component
a	vertex alpha value

Description

This structure may be used to describe a vertex in terms of its x, y, and z location coordinates, its s, t, and w texture coordinates and its r, g, b, and a color components. If the rendering context uses this structure to represent vertices, flat or Gouraud shading, source or destination alpha blending, texture mapping and texel modulation can be performed. This structure represents data members in floating point format.

This structure is the default used to represent vertex data on context creation. To modify the vertex structure type, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_VERTEX_TYPE and pRStateData set to the address of an *C3D_EVERTEX* object containing the new vertex type's enumeration constant.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*, *C3D_EVERTEX*

C3D_VTF

Version

1.0

Syntax

```
typedef struct {
    C3D_FLOAT32 x;
    C3D_FLOAT32 y;
    C3D_FLOAT32 z;
    C3D_FLOAT32 s;
    C3D_FLOAT32 t;
    C3D_FLOAT32 w;
} C3D_VTF, * C3D_PVTF;
```

Members

x	vertex x coordinate
y	vertex y coordinate
z	vertex z coordinate
s	vertex s texture coordinate
t	vertex t texture coordinate
w	vertex w texture coordinate

Description

This structure may be used to describe a vertex in terms of its x, y, and z location coordinates and its s, t, and w texture coordinates. Because this structure does not contain color component members (r, g, b, and a), flat or Gouraud shading, source or destination alpha blending, and texel modulation cannot be performed if the rendering context uses this structure to represent vertices. This structure represents data members in floating point format.

The default vertex structure type set on context creation is *C3D_VTCF*. To modify the vertex structure type, call *ATI3DCIF_ContextSetState* with eRStateID set to C3D_ERS_VERTEX_TYPE and pRStateData set to the address of an *C3D_EVERTEX* object with the value C3D_EV_VTF.

See Also

ATI3DCIF_ContextSetState, *C3D_ERSID*, *C3D_EVERTEX*

Chapter 6
3D RAGE / ATI3DCIF Porting and Performance Notes

Introduction

The ATI 3D RAGE is a very powerful 3D rendering chip with advanced capabilities including:

- Gouraud Shading
- Perspective Correct Texture Mapping
- Texture Lighting
- Texture Filtering
- Interpolated Alpha Blending
- Interpolated Fog

The ATI3DCIF interface is provided for low-level access to this functionality under Windows 95.

While all of these features are available on the 3D RAGE chip, some features run faster than others for a variety of reasons, including the number of reads and writes of pixel and texel data per pixel rendered and the number of registers that must be set up per polygon operation. For this reason, a step by step approach to porting a game or other application to the ATI3DCIF will result in the best combination of high frame rate performance and superior image quality.

Triangle Size, Performance and Image Quality

In a nutshell, the larger the triangle being rendered, the more benefit will be realized from the use of 3D RAGE's hardware acceleration, since the driver time to setup the hardware is amortized across more pixels drawn by the hardware, allowing better parallelism of driver software and hardware drawing. For Gouraud shading, the benefit threshold is at about 10 pixels per triangle and for textured rendering the benefit threshold is at about 30 pixels per triangle. The term "benefit threshold" refers to the first point at which the time taken to send the triangle to the hardware is less than the time to render the triangle directly in software.

Having set a benefit threshold strictly by performance, it has to be remembered that there is a second very important benefit to using the 3D RAGE hardware: image quality. Use of texture filtering and mipmapping, modulation, alpha blending and fog allow new levels of image quality in games that cannot be achieved in software rendering at acceptable frame rates due to the large numbers of arithmetic and logical operations required per pixel. Another way of thinking about the higher image quality is that it effectively reduces the benefit threshold, since it would take software considerably longer to achieve these quality levels.

Porting Backgrounds and Scenery

The first place to expect a huge improvement in both performance and image quality is in drawing backgrounds and scenery. These items are often composed of large or very large polygons that leverage the hardware very efficiently. If your game is partitioned for rendering and control into backgrounds and objects, a good porting strategy is to begin by accelerating the backgrounds first.

- Turn on Gouraud shading or a texture map for the sky
- Turn on texture maps for mountains, trees, walls, and other sceneries as the game requires
- Use simple pick nearest texturing to begin with and measure the frame rate

Now experiment with more advanced features and check the effect on frame rate at each step:

- Turn on texture lighting (Modulation recommended over Alpha Decal)
- Turn on mipmapping if you have mipmapped textures available
- Turn on bilinear texture filtering
- Turn on interpolated Alpha Blending or Fog for transparency and mist effects
- Try other features of interest, such as texture compositing or specular highlighting

In conducting these experiments you will find:

- Texture lighting radically improves realism
- Mipmapping has a very small frame rate impact (<10%) and eliminates texture aliasing
- Bilinear Filtering has a frame rate impact of around 20% and eliminates the nasty “Blocky Pixel” effect that occurs when you approach close to a textured surface. Bilinear filtering also removes texture aliasing for distant texels and can be used alone or in combination with mipmapping. Experiment for best effects.
- Use of alpha blending and fog allow scenery items to approach gradually and realistically through the Yon (far) clipping plane, rather than just popping up.
- Use of Fog runs faster than Alpha Blending, but uses a constant color for the fog rather than a pre-rendered backdrop.

Game Objects

Once you are satisfied with the background effects and frame rate, it is time to experiment with game objects. In general, these will be smaller models with more polygons and fewer pixels per polygon. The more highly tessellated the game figures are, the less benefit to be had from hardware acceleration. However, since high quality textured rendering modes are now available in hardware, it may be worth experimenting with fewer polygons per model and using texture mapping effects to compensate for the use of fewer polygons. Additional performance improvements will be realized by optimizing object and polygon culling prior to setting up polygon/primitive drawing lists. This will result in fewer writes to the frame buffer and z-buffer. Another way to reduce writes is to render front-to-back to avoid overdraw and is especially important when z-buffering (see note below in “*Additional Tips for Improving Performance*”).

In general the recommendations for accelerating game objects are:

- avoid use of highly tessellated objects
- avoid excessive overdraw
- turn on minimal acceleration (gouraud and pick nearest) and measure frame rate
- turn on higher quality rendering features and measure again

If as a result of these experiments you decide to employ a mix of hardware and software rendering, remember to never mix hardware and software rendering within a meshed object. A meshed object must always be rendered in its entirety through a consistent process (hardware or software) to ensure continuity of color at boundaries and to ensure that all pixels at boundaries are rendered correctly.

Concurrency and Software Overhead

The amount of concurrency between the 3D RAGE accelerator and the CPU depends on the size of the primitives being rendered. For small primitives, the 3D RAGE can render faster than the CPU can send instructions over the bus. In this case, it is more advantageous to dispatch a series of primitives to the hardware in a single list or strip, that is, in a single *ATI3DCIF_RenderPrimList* or *ATI3DCIF_RenderPrimStrip* call. The software overhead associated with the call is amortized over several primitives.

For large primitives, more concurrency is achieved by sending primitives to the hardware in smaller lists or strips. The ATI3DCIF_RenderPrimList and ATI3DCIF_RenderPrimStrip functions queue the primitives and do not return until the last one has been set up for rendering. Given a large primitive, these functions can return before the 3D RAGE completes rendering. The CPU can take advantage of this time to perform additional calculations. Thus by rendering smaller lists or strips, more time will be available in between rendering calls for concurrent CPU operations.

Using the RAGE's 2D Engine

Some primitives can be rendered faster using 2D rather than 3D operations. For example, a solid rectangle to clear the back buffer may be drawn faster by performing a 2D rectangle fill rather than rendering two triangles or one quadrilateral in 3D using solid or flat shading. Blitting a rectangular background bitmap is faster than mapping it as a texture onto a rectangle which covers the same area if no scaling is required. If scaling is required, the bitmap may first be texture mapped onto a rectangle in off-screen memory, and then blitted to the destination in 2D. There are two advantages to using this method: (1) the application can enhance or add special effects to the bitmap by applying bi-linear filtering or alpha blending while mapping it as a texture, and (2) if the bitmap will be repeatedly blitted without being altered over several frames, these additional enhancements will only have been done once.

Under certain circumstances, blitting the back buffer to the primary buffer may be faster than page flipping due to the vertical blank synchronization delay in the flipping method. This is dependent on several factors, such as the amount of time required by the game to perform game logic between frames, and the resolution of the screen. Blitting will more likely be faster for smaller resolutions such as 320x240, 400x300, or possibly 512x384. With increasing resolution, the number of pixels to transfer increases, and blitting becomes less effective relative to page flipping. It is recommended that the developers experiment with both the page flipping and blitting methods to determine which offers the best results. In general, blitting will work without introducing noticeable flicker if the frame rate of the application is less then half the vertical refresh rate of the monitor. This method is particularly useful for games that only update a portion of the screen (for example, games which have static cockpit or dashboard consoles which frame the 3D scene).

Under Windows 95, 2D operations may be performed using DirectDraw. Note, however, that the ATI3DCIF driver interface does not require DirectDraw as its surface management layer; any valid surface pointer may be used. If DirectDraw is chosen as the surface management layer, page flipping may be used only in full-screen applications, while blitting must be used for windowed application (although it is rumored that DirectDraw will soon support windowed page flipping).

Additional Tips for Improving Performance

- Whenever possible, arrange primitives into strips. Although more complex to setup, strips use less vertices to compose objects than lists. Consequently, there is less vertex setup overhead.

- When Z buffering, render primitives from front to back and use a Z compare function that maximizes pixel rejection according to the direction Z increases. For example, if Z increases from front to back, use a less-than or less-than-equal compare while rendering primitives from front to back. If Z increases towards the front plane, use a greater-than or greater-than-equal Z compare function. Rendering from back to front may cause pixels to trivially pass the Z compare test, resulting in unnecessary overdraw.
- Eliminate triangles that do not cover a pixel center, that is, triangles whose area is zero or close to zero. Such triangles will not be drawn, but will still incur overhead due to the setup of their vertices.
- Try keeping texture coordinates less than or equal to 10.0. Larger numbers require additional processing by the ATI3DCIF driver.

Summary

A step by step approach to porting a game to the 3D RAGE as suggested above will allow the discovery of the best mix of image quality and fast frame rate. This may be more painstaking than a simple “recompile and run” approach, but will allow the engineer doing the porting to get a real feel for the cost/benefit of each of the 3D RAGE features and find an optimal mix.