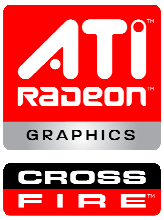


ATI OpenGL Programming and Optimization Guide

Introduction

This guide focuses on how to get the most out of ATI graphics hardware under OpenGL. This guide focuses on the R300 and its derivatives as indicated in Appendix A. Most of the performance advice contained in this document is generally relevant to all graphics accelerators. When something is extremely specific to the R300, R400 or R500 family it is called out as such. In addition to performance, this guide also looks closely at how to access the latest features. This guide does not attempt to discuss extensions for older HW in detail, only how they interact with the R300+ series. Please see the ATI OpenGL extensions guide for details on which extensions are found on which products.

What's New

This update adds new hardware (R400, R500 asic families and newer R300 derivatives), a section on GLSL features and support, sample codes for programming user shaders, and a table matching product names to chip families. An appendix referencing various tools and libraries to facilitate OpenGL development has also been added (Appendix F).

Basic Architecture

To understand how one's application is going to perform on a particular platform, it is best to understand the basic architecture. From the programmer's standpoint, the R300+ series is very similar to previous programmable graphics accelerators. Architecturally, it has richer functionality and higher performance than its predecessors – the primary advancement being the support for floating point color in the texture engine, the shader engine, and the frame buffer. Furthermore, the R500+ series most importantly adds support for dynamic branching in its fragment shader unit, floating point blending, and floating point multisampling.

The number of vertex engines within the vertex processing unit varies across the multiple flavors of the R300+ asics family. Refer to Appendix A for a table enumerating the engine count for the specific asics. Generally, peak transform rate is approximately vertex engine count divided by 4

(number of dot products for a model to clip space transform) vertices per clock. The ideal peak rate may not be attainable in real-world situations, but it should provide a good basis for gauging geometry throughput.

Each fragment shader unit on R300+ series executes a texture instruction and a set of arithmetic instructions every clock cycle. The instructions are executed on number of fragments in parallel depending on the number of fragment engines (sometimes called the pixel engine but will be referred to as fragment engine throughout this document). Refer to Appendix A for number of fragment engines of a particular asic flavor. Generally, one ALU instruction is executed per clock but the fragment shader engine on the RV530 asic has the unique ability to execute up to three ALU instructions in parallel per clock. As with geometry throughput, peak fragment throughput is only theoretical. The real-world performance is almost certainly affected by such things as memory bandwidth or starvation.

Transform, Clip, and Lighting

Data specification

OpenGL allows an application to render a number of ways – immediate mode, display list mode, and vertex array mode. Among the three methods, immediate mode is likely to be the most inefficient and should be avoided as the predominate rendering method. The increase in CPU and graphics processor speeds has far exceeded that of system memory bandwidth. The large amount of data moving around per frame during immediate mode rendering exposes this gap between processing speed versus memory throughput and effectively starves the graphics HW of data to process.

Vertex arrays can be the fastest way to provide geometry data to the R300+ asics. First, they greatly reduce software overhead by reducing the number of necessary function calls to specify vertex data and can eliminate the need to send duplicate data. Additionally, the use of Vertex Buffer Objects can allow an application to write the vertex data directly to either AGP or video memory. The graphics HW then directly addresses the Vertex Buffer Object within local or AGP memory eliminating the need for the vertex data to travel over the system front side bus. The R300+ series supports both vertex and index data storage in the Vertex Buffer Objects. The gl(Multi)DrawArrays or gl(Multi)Draw(Range)Elements entry point should be used to render with these buffer objects. Avoid using glArrayElement since this will lead to significant per call overhead as opposed to the other Vertex Array render calls. Refer to Appendix B for sample code on how to render using a vertex arrays stored in a Vertex Buffer Object and glDrawElements.

To ensure maximum performance using Vertex Buffer Objects, please see the table below outlining the native formats of the R300+ series. Specifying data in a Vertex Buffer Object that is in a format different than the listed ones will have a significant performance penalty, and will likely be slower than using display lists or even immediate mode.

Type	Native	Alignment	Components	Range
GLdouble	No			
GLfloat	Yes	32-bit	1,2,3,4	+/- MAX_FLOAT
GLuint	No			
GLint	No			
GLushort	Yes	32-bit	2,4	[0,65536]
GLshort	Yes	32-bit	2,4	[-32768,32767]
GLushort (normalized)	Yes	32-bit	2,4	[0,1]
GLshort (normalized)	Yes	32-bit	2,4	[-1,1]
GLubyte	Yes	32-bit	4	[0,255]
GLbyte	Yes	32-bit	4	[-128,127]
GLubyte (normalized)	Yes	32-bit	4	[0,1]
GLbyte (normalized	Yes	32-bit	4	[-1,1]

The transform engine has pre-transform and post-transform vertex caches. When using indexed primitives via gl(Multi)Draw(Range)Elements, both of these caches can reduce the load on the vertex engine. The pre-transform cache starts by ensuring that multiple vertices that share the same cache line will only need to be read once, reducing the memory bandwidth required. The post-transform cache will allow a vertex whose index matches a recently processed vertex to not be reprocessed, effectively freeing a vertex engine to process another uncached vertex. Both of these optimizations require that the index list contain a locality of reference to enable optimal performance, the former requiring that neighboring indices reference contiguous vertices, and the latter requiring that reuse of any given index be as local as possible. Finally, these and other efficiency concerns mean that vertex array draw commands that consist of primitives with 64 or more vertices will make better use of the vertex caches as well as allow the vertex engine to stream process the data more efficiently. Avoid drawing with less than 10 vertices per primitive.

Geometry data can also be compiled into and rendered out of a display list. The driver will compile, optimize, and store the display list into video or AGP memory under most circumstances. Always attempt to compile a handful of larger display lists rather than numerous amounts of small display lists since each glCallList(s) invokes a certain overhead which should be avoided if possible. The R300+ series is optimized for processing large primitives so it is advantageous to arrange the geometry data into primitive strips of 10 vertices or longer. To ensure that the display list is stored in the optimal manner, avoid compiling evaluators, edge flags, generic vertex program attributes, and texture coordinates with four components into the display list.

Vertex array calls can be compiled into a display list. Compiling the regular OpenGL 1.2 vertex array usage may potentially allow the driver to perform various optimizations on the input vertex data. But it should be noted that if vertex arrays are intended to be compiled into a Display List, the vertex array should not be stored in a Vertex Buffer Object. Compiling vertex arrays stored in Vertex Buffer Objects will cause a significant performance hit when the vertex data is read back from video memory during the display list compilation.

Regardless of the rendering mode used, it is always good practice to render primitives of similar state in batches. State changes (i.e. stipple pattern changes, light state changes) will invoke a validation overhead in the driver and within the graphics HW. Haphazard and redundant state changes within a rendering frame will lead to poor performance.

Transform and Vertex Shading Engine

All geometry processing is performed by a number of parallel vertex engines in the R300+ series. The peak geometry throughput is roughly the floor of the number of operations –or– the number of operations divided by the number of vertex engines. All fixed function and user vertex shaders use the same resources, so the approximate penalty of a feature in fixed function is equivalent to the cost if it were hand-coded in a vertex program.

The ARB_vertex_program is one way to program user shaders for the vertex engine. Refer to Appendix C for sample code creating and binding an ARB_vertex_program. When writing an ARB_vertex_program, several items must be considered to achieve maximum performance. Most importantly, the smallest number of instructions should be used. The driver will collapse and optimize code, but it is always best to start with the best code possible. It is also important to minimize the number of constants and temporaries used by the program. The fewer temporaries used by the program the closer the hardware comes to reaching the theoretical performance limit. As with the instructions, the driver will attempt to reduce the use of temps where appropriate. Refer to Appendix D for a table mapping the instructions in the ARB_vertex_program to the various HW resources required to execute that instruction.

The resources available to the vertex shading unit are as follows:

ALU Instructions: 256 4-tuple vector + scalar (R300, R400), 1024 4-tuple vector + scalar (R500)
Temporary registers per vertex: 32 4-tuple (R300, R400, R500)
Constant registers: 256 4-tuple (R300, R400, R500)

The OpenGL Shading Language (GLSL) can also be used to specify a custom vertex shading program. GLSL allows users to write their shader in higher level C-like syntax and also abstracts HW shader

feature support. Most importantly, GLSL enables the user to program flow control into their shader logic. The native HW program used to implement the flow control is emitted by the GLSL compiler within the driver. All native HW support is abstracted by the compiler. The section below on the OpenGL Shading Language divulges more information about the language and subtleties associated with its use.

Static flow control is supported for the R300+ series through emulation. When a vertex shader program using static flow control is detected, the driver will recompile the program without flow control based on the provided static constants. The recompiled program will then be cached away to avoid redundant shader recompilation during the runtime. If you are using static flow control in a vertex program you should pre-cache recompiled variations of that program in the driver by rendering a dummy triangle on the very first frame with all combinations of constant values that will be relevant throughout the execution of the application.

Dynamic flow control is natively supported in the R500 asics. Empirical analysis indicates that most shader programs use dynamic flow control at the fragment shading level. Therefore, the dynamic flow control performance within the transform engine is not optimized as it would be at the fragment level. To maximize vertex program performance for scenes rendering a significant number of primitives, it is advised to avoid dynamic flow control altogether in the vertex program.

Clipping

The R300+ series has support for six user specified clip-planes in addition to the frustum clip planes. The cost of clipping is determined by the number of clip-planes enabled and the amount of geometry being clipped but not trivially accepted or rejected. To ensure that the hardware clip-plane support is being utilized, the user must use a projection matrix that is non-singular as all clipping occurs in clip-space.

Rasterization

Component Interpolation

The R300+ series can interpolate ten sets of 4-tuple vectors. Two sets are reserved for the primary and secondary colors, while the other eight are used for texture coordinates. The color interpolators have two inputs each, one each for front and back colors. The decision as to whether to use the front or back colors is done at setup and the appropriate colors are then interpolated. The interpolated colors have a range of [0-1] and are limited to 12 bits of precision. When multisampling is enabled, the colors are sampled at the centroid of the covered portion of the fragment as is specified in the

SGIS_multisample specification. The texture coordinate interpolators differ from the color interpolators in that they always sample at the fragment center and that they are interpolated using at least 24 bit floating point. All interpolations are performed with perspective correction. Historically, calling glHint(GL_PERSPECTIVE_CORRECTION_HINT, GL_FASTEST) led to screen-space interpolation. But this is not guaranteed to be the case any more due to advances in the HW. If screen-space effects are desired, the user must undo the perspective correction in a fragment shader.

Stipple and Anti-Aliasing

While the R300+ series accelerates polygon stippling, line stippling, and line anti-aliasing, the resources used to support it overlap the texture resources. As a result, enabling polygon stippling, line stippling, or line anti-aliasing reduces the number of texture units accelerated in hardware to seven. Using more than seven textures in the fixed function case or more than seven texture coordinate sets in the fragment shader/program case will result in a fallback to software rendering.

Depth and Stencil Testing

The R300+ series supports multiple methods to accelerate rendering by culling pixels that are not visible. First, the R300+ series supports an accelerated depth buffer clear that effectively makes clears free. Not only is the clear free, but also the clear optimizes the first depth buffer reads for each frame rendered. Applications should always clear the buffer rather than attempt to use Z tricks to avoid the clear. This is even more important when bandwidth intensive operations like multi-sampling are enabled. To ensure that the fast Z clear optimization is used, the app should have scissor disabled or enable a full screen viewport size. If the graphics context has uses a stencil buffer, it should also be cleared at the same time as the depth buffer.

The R300+ series also supports hierarchical depth testing. This allows the graphics HW to depth check blocks of fragments efficiently. To maximize the benefit of the Hi-Z optimization, the app must keep the depth compare function constant over the course of a frame. The app must not switch from a test using *less than* to using *greater than* or vice versa. Avoid using *always*. This naturally is not a problem if the depth test is switched and the depth mask is set to prevent updating the depth buffer. Additionally, other operations can preclude the use of Hi-Z for a set of primitives. These operations include: outputting a depth in the fragment shading unit and updating the stencil buffer on depth fail.

Finally, the R300+ series has a more fine-grained mechanism to cull occluded fragments. It can perform the depth and stencil tests prior to shading the fragment. This is only a win if the shader has multiple instructions such that the rasterizer is able to produce fragments faster than the shader can consume. So a user should not be concerned with this optimization unless they are performing a relatively expensive shading operation. To enable the optimization, the app only needs to ensure that the shader is not writing a depth value and that pixels are not being killed by the shader or by the alpha test when the depth values are being updated.

In addition to all these pixel culling optimizations, the R300+ series has the ability to control its stencil functions and stencil ops based on the facing direction of the primitive. This allows algorithms such as shadow volume computations to be accelerated, as they need to alter the stencil buffer differently based on the facing orientation of the primitive. Please see the extension spec ATI_separate_stencil for more details.

Fragment Shader Unit

For mostly all R300+ asics, the fragment shading unit can execute one texture instruction and one ALU instruction in each clock cycle. The one pleasant exception is the RV530 which can actually execute up to three ALU instructions plus one texture instruction per clock. Each ALU instruction can actually be a 4-tuple operation or a combination of a 3-tuple plus a scalar operation. All instructions are operated on in 24-bit floating point numbers for R300/R400 and 32-bit floating point for R500.

The ARB_fragment_program is primarily used to program a user shader for the fragment shading unit. Like programming the vertex engine with the ARB_vertex_program, one should use the least amount of instructions possible to allow for maximum performance. While resources have increased for the R500, it is still good practice to be sensitive to the number of resources used in the fragment program. Refer to Appendix D for a table mapping the ARB_fragment_program instructions to the various HW resources required to execute the instruction.

The resources available in the shader unit are as follows:

ALU Instructions: 64 3-tuple vector + scalar (R300), 512 3-tuple vector + scalar (R400 + R500)
Texture Instructions: 32 (R300), 512 (R400 and R500)
Temporary registers: 32 4-tuple (R300), 64 4-tuple (R400), 128 4-tuple (R500)
Constant registers: 32 4-tuple (R300 and R400), 256 4-tuple (R500)

Each fragment engine on the R300+ series operates as two independent units - one operating on scalar data and one operating on 3-tuple data. This means that the implementation can optimize certain operations if the user can isolate 3-tuple operations to occur only on the rgb components by using the write masks. Additionally, all of the native scalar ops (EX2, LG2, RSQ, and RCP) are always executed on the scalar unit, so it is best to have the sources and destinations come from the alpha channel. In general, these tips will only reduce shader size. Ignoring them will not cause the driver to expand the operation into more instructions.

When writing an ARB_fragment_program, one should also consider the source modifiers used. While all the different source modifier variations are natively supported on the R500, not all of the source modifiers are natively supported on the R300 and R400 series. The native source modifiers for the R300 and R400 are negate (-src), r replicate (src.r), g replicate, b replicate, a replicate, gbr*, brg*,

and abg*. Additionally, these modifiers are not native on the texture or kill operations. When using non-native source modifiers, the driver will insert up to four extra instructions to generate the swizzle.

As with the vertex shading unit, one can also program the fragment shading unit using the OpenGL Shading Language (GLSL). The benefits of using GLSL for the fragment shading unit is nearly the same as the benefits for using GLSL for the vertex shading unit. As a matter of fact, the GLSL compiler can be of more significance for a GLSL fragment program than a vertex program due to the increase in complexity of the fragment shading unit's architecture over the vertex shading unit's. Like the vertex shading case, the user can easily program flow control into their fragment shader logic as well as enjoy the abstractions and optimizations the compiler implements for the underlying HW. For more information on the usage of GLSL, refer to the OpenGL Shading Language section below.

While the R500 natively supports flow control in the fragment shading unit, the R300 and R400 asics does not. Static flow control for the R300 and R400 is emulated by the driver compiling out unused conditionals and unrolling loops based on the set constants. Even though the R500 asics family natively support flow control, the driver will still attempt to compile out static flow conditions enabling it to reorganize shader instructions for better instruction scheduling. The driver will also try to cache away the compiled shader for a specific static flow condition set in anticipation for its reuse. So when writing a fragment program that uses static flow control, it is recommended to "warm" the shader cache by rendering a dummy triangle on the very first frame that uses the common static conditional permutations relevant for the life of the shader.

The R500 asics family fully supports dynamic flow control in the fragment shading unit. Dynamic flow control allows a shader program to execute varying shader code as well as selectively skip over portions of execution code and texture fetches for a group of fragments also known as the execution thread. When implementing dynamic flow control into a shader program, one should be sensitive to the parallel nature of the fragment engine and how it processes the execution thread in lockstep. If flow control causes fragments that are executed within a thread to take a different code path, all the fragments in the thread, regardless of whether they should or should not execute the path, will be "dragged along". For the fragments that should not execute the path, the instructions will be ignored while the relevant fragments will be processed as usual. The thread size for the R500 family consists of 16 fragments so a fragment program should not be written so that dynamic flow control varies execution paths for fragments smaller than this granularity. Like static flow control, it is best to avoid small conditional statements since it will limit compiler optimization efficiency. Generally, fragment shaders should avoid loops as well as more than 6 levels of dynamic branching. If possible, a low iteration count dynamic loop should be unrolled into nested conditional statements.

Regardless if GLSL or ARB_fragment_program is used, one should consider the memory latency of texture fetches when writing a fragment program. Even though the fragment shader unit can execute one texture instruction along with one ALU instruction, due to memory latency and time to filter the texture sample, it will very likely take more than one cycle for the sample result to be available. For

the R300+ asics other than the RV530, the fragment program should have a 1:4 texture to ALU instruction ratio. For the RV530, the ratio should be around 1:8 since it can execute more ALU instructions per clock. The most optimal ratio will also vary depending on the texture filter mode, format and general memory load on the HW.

Texture Operations

As mentioned above, the fragment shader can execute a maximum of 32 texture fetches for the R300/R400 and 512 texture fetches for the R500. Each of these fetches may come from any of the available 16 texture contexts for all R300+ asics. This flexibility is only available by using the fragment program extension. In fixed function, the available number of textures is limited to the number of texture coordinate supported by OpenGL which is eight for version 1.2.

The speed of texture operations is controlled by the precision of the input texture and the number of bilinear blends the filter requires. LINEAR, NEAREST, NEREST_MIPMAP_NEAREST, and LINEAR_MIPMAP_NEAREST filter modes on 1D and 2D textures all qualify as requiring a single bilinear blend. LINEAR_MIPMAP_LINEAR on 1D and 2D textures and the previous operations on 3D textures require two bilinear blends. Operations such as anisotropic filter take a variable number of blends based on the level of anisotropy. Each bilinear blend requires one clock cycle if the amount of data being blended is 32 bits or less. This includes texture of type RGBA8 and LUMINANCE8_ALPHA8. For texture formats larger than 32 bits per component, blending can take 2 or more cycles

The R300+ series supports two floating point texture formats. The first is a standard 32 bit IEEE float with 8 bits exponent, 23 bits mantissa, and a sign bit. The second is a 16 bit floating point number a 5 bit exponent, a 10 bit mantissa, and a sign bit. Floating point textures have some restrictions to its usage. They may only be sampled as NEAREST or NEAREST_MIPMAP_NEAREST since floating point texture filtering is not supported. In the case where floating point texture filtering is required, it can be simulated in a fragment program. The use of the 16 bit integer per channel format can also be entertained as an alternative to using the floating point format since filtering is supported for the 16 bit integer format.

Also, the R300/R400 asics do not support the texture border color when using floating point texture formats. So only parameters that do not use the border color such as GL_REPEAT and GL_CLAMP_TO_EDGE are supported as wrap modes. The R500 fully supports texture border colors regardless of texture format. In all other ways, floating point textures are identical to other textures. They can be used with the 1D, 2D, 3D, CUBE_MAP, and RECTANGLE targets.

The maximum texture size supported by the R300 and R400 series is 2048 in all dimensions and 4096 in all dimensions for the R500. Allocating the maximum texture size is impossible since doing so

exceeds the 32-bit address space. For optimum performance, individual textures should be kept to sizes 32MB or smaller since it allows the driver more flexibility in where to place the texture in memory.

The 16 bit integer formats do not support a border color. As a result, the wrap modes CLAMP and CLAMP _TO_BORDER are not supported for these formats.

As a final note on performance, an application should attempt to use smaller texture formats where applicable. Modern 3D graphics is often extremely bandwidth intensive, any reduction in bandwidth usage often equates to an increase in performance. Judiciously using the compressed texture formats is one way to reduce bandwidth usage as well as memory footprint. Often textures like base maps can be compressed with few or no visual artifacts, while textures used as normal maps may suffer terrible quality degradation. Being careful to selectively reduce texture format sizes will improve bandwidth with minimal quality loss.

Backend Operations

Blending

The R300+ series supports most modes of frame buffer blending in use today. It supports the blend_func_separate extension and the blending operations included in the imaging subset. The important thing to keep in mind from a performance point of view with the R300+ series is that it can optimize identity blending operations and save bandwidth. For instance, if the source factor is DEST_COLOR and the destination factor is ZERO, the blend can be optimized if the pixel is (1,1,1,1). To enable this optimization, the programmer should either ensure that unneeded quantities are either masked or set to the identity value for the blend. An example of this is rendering without attention to nor caring about what is being placed into the destination alpha. If the app either sets the alpha mask to false or takes care to ensure that the value produced on alpha is the one that is the identity for that blend equation, then the optimization will be applied. Otherwise, the HW will have to blend those pixels just to keep the destination alpha correct even though they may never be used.

The R500 asics family also supports alpha blending for the 4-channel 16-bit floating point and RGBA1010102 format render targets. One and 2-channel 16-bit floating point format blending is not supported but 1 and 2-channel formats are generally used for storing non-color values for which blending is not applicable.

Multisampling

The R300+ series supports ARB_multisample and is a true multisampling implementation. Each fragment may be written to two, four, or six samples based on coverage and depth. The pixel formats for

multisample buffers are only available when queries are made with WGL_ARB_pixel_format. To get the most out of multisampling on the R300+ series, the programmer should be aware that the resolve is done in a gamma corrected space. The result is that blends between extreme values will provide the perceptually correct value. To allow this to work optimally, the programmer should not attempt to provide a gamma correction curve. The resolve operation is tuned for an sRGB color space which expects that the gamma table be linear. This value works on essentially all displays presently in use.

The R500 asics family also supports multisampling for the 16-bit floating point and RGBA1010102 format render target to allow for High Dynamic Range rendering. It should be noted that 16-bit floating point multisampling will significantly increase memory footprint as well as memory bandwidth usage. The 1010102 format may be a better performance alternative to the 16-bit floating point format.

Miscellaneous Categories

OpenGL Shading Language (GLSL)

A programmable vertex and fragment shader program can also be specified using the OpenGL Shading Language or GLSL. All API functions needed to create, compile and use a GLSL program has been promoted into OpenGL 2.0 and is supported by all variants of the R300 family of products. The language specification for GLSL can be found at http://www.opengl.org/documentation/oglsl.html.

The GLSL syntax is similar to the C programming language. The programmer need not worry about GLSL support since all OpenGL 2.0 compliant HW must support GLSL. The low-level translation of the program to machine language is hidden from the GLSL programmer. This abstraction allows the compiler in the graphics driver to seamlessly enable future HW improvements compiling the same GLSL program.

A GLSL program consists of various shader objects. Currently, it can be a vertex shader or a fragment shader targeting, respectively, the transform/vertex engine or the fragment shading unit within the graphics HW. After the shaders are compiled, it is then attached to a program object. The program object is then linked into the GL pipeline and replaces the targeted fixed pipeline functionality. Refer to Appendix E for sample vertex, fragment shader code and the code needed to create, compile, and use a shader and program object.

While GLSL abstracts the resource limitations of the hardware, a programmer must take some care in developing programs that are well suited to the target hardware. With the rapid expansion of hardware capabilities and improvements in graphics hardware compiler technology, this is quickly becoming a smaller problem, but care is still needed to support older graphics HWs. To help identify limitations, the ATI OpenGL driver will attempt to add a helpful message to the info log at link time if the capabilities

of the underlying HW platform have been exceeded. The driver will always report the word “software” in the info log if the shader forces the driver to not render fully hardware accelerated.

All ATI graphics HW have a few items that deserve special consideration when using GLSL. The first major item of note is the absence of vertex texture units. This means that vertex texturing is never available, and all shaders attempting to use texture functions in the vertex shader will fail to link. Additionally, dynamic index on samplers is presently not supported in either hardware or software. This means that while it is possible to declare an array of samplers and use it within a shader, if the array indices are recognized as not being compile time constants, then the shader will fail to compile. Finally, the full generality required to handle all cases of user defined clip planes is also not supported. As a result, if the vertex shader writes gl_ClipVertex, the program will use software vertex processing. If a user wishes to use user clip planes with GLSL on ATI hardware, they can use *ftransform* and specify the clip planes in eye-space as they would with fixed function. ATI defines the unspecified result in this case to be equivalent to fixed function clipping.

In addition to the general caveats, the R300 and R400 asics have some additional GLSL limitations. The first important item to note is that they lack support for dynamic flow control. As a result, all branching code paths must execute and extra instructions simulating predication are added to the compiled shader program to resolve the final solution. Loop branches are also unrolled. The practical result of the lack of branching support is that flow control attempting to avoid extra work is ineffective on this hardware, and that deeply nested if statements will place heavy demands on resources. The practical result of the lack of dynamic loop support is that the driver can only support fairly simple *for* loops with reasonably small loop counts in hardware. The R300 and R400 asics also lack support for derivative operations. This means that dFdx, dFdy, and fwidth will cause the shader to run in software. Finally, the R300 and R400 can support up to three indirections when accessing textures (the R500 asics do not have this restriction). The compiler will do a good job of rearranging code to avoid running into this limit, but occasionally it is impossible to avoid. Additionally, when combined with large numbers of texture fetches this limitation can artificially increase the use of register resources potentially exceeding HW limits.

Finally, the GLSL shaders compete for hardware resources with a few other states. As a result, the user may see a shader that would otherwise run in hardware fall back to software rendering. The states that compete for shader resources are anti-aliased points and lines, and stippled lines and polygons. Additionally, wide points and lines are incompatible with the gl_FragCoord variable.

To ensure the best performance when writing GLSL code, it is best to keep a few simple rules in mind. First, the hardware described here is essentially a SIMD vector machine. This means that a user should avoid small optimizations that might be done on a scalar machine. Adding a pair of three component vectors can actually be cheaper than recognizing that the y component of one of them is always zero and performing two scalar additions instead. Further, it is typically best to try to take full advantage of all the built-in functions, as these are coded to make maximal use of the hardware.

Pixel Transfer Operations

The flexibility of the R300+ series' shader pipe allows it to support acceleration on most forms of draw and copy pixels. The only formats that DrawPixels does not natively support are GLint and GLuint. The only packed formats that are not natively supported are the BITMAP, 2_3_3_REV, 10_10_10_2, and 5_5_5_1 formats. Additionally, the scale, bias, and zoom operations are all supported directly by hardware. For non-packed formats, all pixel transfer operations are most efficient when operating on four component data. This same flexibility exists in the texture specification path, but for maximum performance textures should be specified in BGRA order. For the internal bit representations of the various GL pixel formats, refer to Appendix F.

ReadPixels is presently accelerated only for color components. To get the best performance an application should be programmed to read colors back in BGRA format as GLubyte's, GLushort's, or GLfloat's with four components on a 32 bit desktop. To prepare for future acceleration an app should read back depth values as 32-bit floats.

While glBitmap is hardware accelerated on the R300+ series, it is relatively performance intensive. Rendering bitmaps on a 3D scene may cause a noticeable slowdown. To get the best performance out of bitmaps, they should typically be compiled into a display list. For even better performance, a user should use textures for rendering widgets and text on top of their 3D screen.

Floating Point Rendering

All R300+ hardware has the capability to render to floating point frame buffers through various extensions to OpenGL. The preferred method of rendering to a floating point buffer is with the use of the EXT_framebuffer_object (FBO) extension. This extension supersedes the ARB_pbuffer extensions in many ways. Most importantly, an FBO allows much greater software efficiency by eliminating the need to use separate contexts to render to floating point buffers. Regardless of the mechanism used, the buffers may be created to use either IEEE 32-bit floats or the 16-bit s10e5 floats described previously.

Certain limitations apply when using floating point buffers on the R300 and R400 hardware and the user must be aware of them. We decided not to put these limitations into the extensions because we did not want to create an extension that would need revising in the future when hardware without these limitations, such as the R500, is available. First, floating point buffers are not displayable on R300 or R400 series hardware, but 16 bit floating point buffers are displayable on the R500 series. This later capability is presently unavailable under Microsoft Windows due to limitations with the desktop color buffer bit depth. As a result, all floating point rendering must be targeted at an offscreen buffer using either the ARB_pbuffer extension or the FBO extension. Refer to the EXT_framebuffer_object SDK sample for more information and sample code on the topic (http://www.ati.com/developer/radeonSDK.html). Floating point buffers also lack hardware support for

some of the operations after specular add or fragment program execution. Enabling any of these operations will force the implementation to fall back to software rendering. The unsupported operations are:

- alpha test (Supported for 16-bit floats on the R500 series)
- blending (Supported for 16-bit floats on the R500 series)
- fog

The supported operations are:

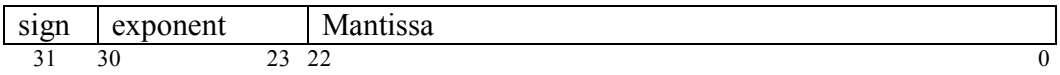
- depth test
- stencil test
- color mask.

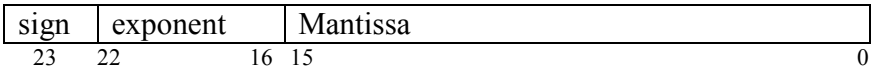
There are also operations that are do not make sense when applied to floats. These operations are:

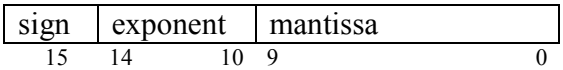
- dither
- logical operation

Data Formats

The R300+ series supports three different floating point numeric formats that are in some way exposed to the programmer. When performing calculations that require an extensive amount of precision, the programmer should be familiar with these formats, where they occur, and what their limitations are. Textures and frame buffers may use either a 16 bit or 32 bit floating point format. The 32 bit format is identical to a single precision IEEE 754 float, except that denormals will be interpreted as zero. The 16 bit floating point format behaves in the same manner, but it naturally has different range and precision statistics. Additionally, the R300 and R400 asics use a 24 bit format for internal shader computations while the R500 uses a 32 bit floating point format. The layout of the formats and their limitations are in the following tables.







Format	Mantissa	Exponent	Max Value	Min Positive Value
16 bit	10 bits	5 bits	6.550400E+04	5.960464E-08
24 bit	16 bits	7 bits	1.844660E+19	2.168404E-19
32 bit	23bits	8 bits	3.402823466e+38	1.175494E-38

Multiple Output Colors and Destination Color Buffers

The R300+ series supports simultaneous rendering to multiple color buffers. This allows it to fully accelerate render modes such as FRONT_AND_BACK. Additionally, the shader may output separate colors for each of the buffers being written to. These independent color outputs and the control of their associated destination buffers are defined in the GL_ATI_draw_buffers extension. The R300+ series supports up to four simultaneous color outputs through this extension.

Under fixed function OpenGL, the same color value will be written to all of the draw buffers. When fragment programs are enabled, they may optionally write separate outputs for each of the draw buffers. In either case, all the buffers share the same alpha-blending and color masking state.

Appendix A: R300+ series Extension and New Feature Usage

Accumulations buffers

The R300+ series is the first consumer level card to accelerate accumulation buffers in hardware. It represents the data using a signed 16-bit format, and all accumulation operations are directly supported. The user should be careful to only select a pixel format with an accumulation buffer if desired. Otherwise, using accumulation buffers can cause a memory overhead. But the driver tries to avoid the overhead by detecting when the buffer goes unused.

All accumulations buffer rendering is accomplished using the entry point glAccum and no extensions to the standard accumulation capabilities are provided. If enhanced accumulation operations are desired, they can be accomplished via fragment programs and rendering to FBOs.

ARB_vertex_program

The R300+ series presently implements ARB_vertex_program as its primary method for specifying a user vertex program. Please see the information above for implementation details.

ARB_fragment_program

The R300+ series presently implements ARB_fragment_program as its primary method for exposing per-fragment programmability to the user. Implementation details on how it operates on the R300+ are included above.

ARB_depth_texture, ARB_shadow, ARB_shadow_ambient

The R300+ series supports shadow buffers via ARB_depth_texture, ARB_shadow and ARB_shadow_ambient. The R300+ series only supports NEAREST and NEAREST_MIPMAP_NEAREST filtering on depth textures. If better filtering is desired, the user should write a fragment program and perform whatever version of filtering is desired.

ATI_separate_stencil

The R300+ series supports an extension to accelerate the rendering of stencil shadow volumes in the form of ATI_separate_stencil. This extension allows an application to specify different stencil update functions for primitives based on whether they are front or back facing. This reduces the transform overhead in when using common shadow volume techniques, since the geometry must only be sent once rather than once for front faces and once for back faces.

ATI_draw_buffers

The multiple output colors are defined using the ATI_draw_buffers fragment program option which extends the fragment program grammar with result.color[n]. The draw buffers for each of the output colors is defined with glDrawBuffersATI(GLsizei n, const GLenum *bufs).

ATI_pixel_format_float

This extension provides a new pixel format type for floating point data. This pixel format is only available for off-screen rendering using the ARB_pbuffer or the EXT_framebuffer_object extension. On the R300+ series, certain rendering capabilities are either restricted or emulated in software when rendering to targets with a floating point pixel format. The exact limitations are contained in the section “Floating Point Rendering”.

ATI_texture_float

Floating point texture formats are exposed through the ATI_texture_float extension. As noted in the texturing section, the floating point formats have certain limitations that developer should be aware of. The additional formats are exposed as new internal formats for all the base texture formats. (RGB, RGBA, etc) The new internal formats have versions for both 16-bit floating point numbers and 32-bit floating point numbers.

Appendix B: Product family chart

Product	Chip Family	Bus	Memory	Fragment Engines	Vertex Engines
Radeon 9700 (Pro)	R300	AGP	128 MB	8	4
Radeon 9500	R300	AGP	128 MB	4	4
Radeon 9500 Pro	R300	AGP	128 MB	8	4
Radeon 9800 (Pro)	R350	AGP	128/256 MB	8	4
Radeon 9800XT	R360	AGP	256 MB	8	4
FireGL X1	R300	AGP	128 MB	8	4
FireGL X1-256	R300	AGP Pro	256 MB	8	4
FireGL Z1	R300	AGP	128 MB	4	4
Radeon 9600 (Pro)	RV350	AGP	128 MB	4	2
Radeon 9600XT	RV360	AGP	128 MB	4	2
FireGL T2	RV350	AGP	128 MB	4	2
FireGL T2e	RV360	AGP	128 MB	4	2
Radeon X800	R420/R423	AGP/PCI-E	256 MB	12	6
Radeon X800PE	R420/R423	AGP/PCI-E	256 MB	12	6
FireGL V3100	RV370	PCI-E	128 MB	4	2
FireGL V3200	RV380	PCI-E	128 MB	4	2
FireGL V5100	R423	PCI-E	128 MB	12	6
FireGL V7100	R423	PCI-E	256 MB	12	6
Radeon X700	RV410	PCI-E	128/256 MB	8	6
FireGL V5000	RV410	PCI-E	128 MB	8	6
Radeon X600	RV380	PCI-E	128/256 MB	4	2
Radeon X300	RV370	PCI-E	128 MB	4	2
FireGL V3300	RV515Pro-GL	PCI-E	128 MB	4	2
FireGL V3400	RV530Pro-GL	PCI-E	128 MB	4 (3x ALU)	5
FireGL V5200	RV530XT-GL	PCI-E	256 MB	4 (3x ALU)	5
FireGL V7200	R520XT-GL	PCI-E	256 MB	16	8
FireGL V7300	R520XT-GL	PCI-E	512 MB	16	8
FireGL V7350	R520XT-GL	PCI-E	1024MB	16	8
Radeon X1300	RV515	PCI-E	128/256 MB	4	2
Radeon X1600	RV530	PCI-E	256/512 MB	4 (3x ALU)	5
Radeon X1800	R520	PCI-E	256/512 MB	16	8

Radeon X1900	R580	PCI-E	512 MB	16 (3x ALU)	8

Appendix C: Rendering using Vertex Buffer Objects

Below is a coding example on how to create, bind and draw using a Vertex Buffer Object and the glDrawElements call. More information can be found at http://www.ati.com/developer/sdk/RadeonSDK/Html/Info/Extensions/GL_ARB_vertex_buffer.html.

```
#include <glATI.h>

// Initialize a VBO array buffer
glGenBuffersARB( 1, &unVtxBufferObj );
glBindBufferARB( GL_ARRAY_BUFFER_ARB, unVtxBufferObj );

// Create data store of the buffer object and copy vertex data
// into the buffer object
glBufferDataARB( GL_ARRAY_BUFFER_ARB,
                 vCount * 3 * sizeof(GLfloat), vtxArray,
                 GL_STATIC_DRAW_ARB );

// Do the same for indexed arrays
glGenBuffersARB( 1, &unIndexBufferObj );
glBindBufferARB( GL_ELEMENT_ARRAY_BUFFER_ARB, unIndexBufferObj );
glBufferDataARB( GL_ELEMENT_ARRAY_BUFFER_ARB,
                 N_PRIMS_TO_DRAW * sizeof(GLuint) * 3,
                 elemIdxArray,
                 GL_STATIC_DRAW_ARB );

// Enable vertex arrays
glEnableClientState( GL_VERTEX_ARRAY );

// Set the vertex array to use the buffer object by setting
// the vertex pointer to NULL
glVertexPointer( 4, GL_FLOAT, 0, NULL );

// Draw elements using the indexed buffer object by also
// passing in a NULL index pointer
glDrawElements( GL_TRIANGLES, (N_PRIMS_TO_DRAW*3),
                GL_UNSIGNED_INT, 0 );

// Delete the buffer objects
glDeleteBuffersARB( 1, &unVtxBufferObj );
glDeleteBuffersARB( 1, &unIndexBufferObj );
```

Appendix D: Creating and binding an ARB_vertex and ARB_fragment program

```
#include <glATI.h>

// Vertex Program
const char vertexProg[] =
"!!ARBvp1.0                                    \n"
"PARAM black = { 1.0, 0.0, 0.0, 1.0};         \n"
"PARAM c0 = { 1.0, 0.0, 0.0, 0.0};            \n"
"PARAM c1 = { 0.0, 1.0, 0.0, 0.0};            \n"
"PARAM c2 = { 0.0, 0.0, 1.0, 0.0};            \n"
"PARAM c3 = { 0.0, 0.0, 0.0, 1.0};            \n"
"ATTRIB vtx = vertex.position;                \n"
"OUTPUT oV = result.position;                 \n"
"OUTPUT oC = result.color;                    \n"
"MOV oV, vtx;                                 \n"
"MOV oC, black;                               \n"
"                                             \n"
"END";

// Fragment Program
const char fragmentProg[] =
"!!ARBfp1.0                                   \n"
"MOV result.color, fragment.color;            \n"
"END";


// Initialize, compile, and bind a vertex and fragment program
glGenProgramsARB( 2, VFprogArr );
glBindProgramARB( GL_VERTEX_PROGRAM_ARB, VFprogArr[0] );
glProgramStringARB( GL_VERTEX_PROGRAM_ARB,
                    GL_PROGRAM_FORMAT_ASCII_ARB,
                    (GLsizei)(strlen(vertexProg)), vertexProg);
if (glGetError() != GL_NO_ERROR) {
    printf("Vertex Shader failed compile: %s\n",
            glGetString(GL_PROGRAM_ERROR_STRING_ARB));
}
glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB, VFprogArr[1] );
glProgramStringARB( GL_FRAGMENT _PROGRAM_ARB,
                    GL_PROGRAM_FORMAT_ASCII_ARB,
                    (GLsizei)(strlen(fragmentProg)), fragmentProg);
if (glGetError() != GL_NO_ERROR) {
    printf("Fragment Shader failed compile: %s\n",
            glGetString(GL_PROGRAM_ERROR_STRING_ARB));
}


// Enable vertex program
```

```
glEnable( GL_VERTEX_PROGRAM_ARB );
glEnable( GL_FRAGMENT_PROGRAM_ARB );
glBindProgramARB( GL_VERTEX_PROGRAM_ARB, VFprogArr[0] );
glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB, VFprogArr[1] );

// Render
// ...

// Delete the program
glBindProgramARB( GL_VERTEX_PROGRAM_ARB, 0 );
glBindProgramARB( GL_FRAGMENT_PROGRAM_ARB, 0 );
glDeleteProgramsARB( 2, VFprogArr );
```

Appendix E: Tables mapping ARB_vertex_program and ARB_fragment_program to HW resources

The following tables provide guideline for the number of ops required for each of the instructions in ARB_vertex_program and ARB_fragment_program and information on the resources available and the resource usage by certain instructions.

ARB_vertex_program Instruction	HW Instructions	HW Temps	HW Constants
ABS	1	0	0
FLR	2	1	0
FRC	1	0	0
LIT	1	0	0
MOV	1	0	0
EX2	1	0	0
EXP	1	0	0
LG2	1	0	0
LOG	1	0	0
RCP	1	0	0
RSQ	1	0	0
POW	1	0	0
ADD	1	0	0
DP3	1	0	0
DP4	1	0	0
DPH	1	0	0
DST	1	0	0
MAX	1	0	0
MIN	1	0	0
MUL	1	0	0

SGE	1	0	0
SLT	1	0	0
SUB	1	0	0
XPD	2	1	0
MAD	1	0	0
SWZ	0/1 (R300, R400), 0 (R500)	0	0

ARB_fragment_program Instruction	HW Instructions	HW Temps	HW Constants
ABS	0-1	0	0
FLR	2	1	0
FRC	1	0	0
LIT	8	1	1
MOV	0-1	0	0
COS	11 (R300, R400), 1 (R500)	2 (R300, R400), 0 (R500)	3
EX2	1	0	0
LG2	1	0	0
RCP	1	0	0
RSQ	1	0	0
SIN	10 (R300, R400) 1 (R500)	2 (R300, R400), 0 (R500)	3
SCS	6-8 (R300, R400), 2 (R500)	1 (R300, R400), 0 (R500)	2
POW	3	1	0
ADD	1	0	0
DP3	1	0	0
DP4	1	0	0
DPH	2 (R300, R400), 1 (R500)	1 (R300, R400), 0 (R500)	0
DST	3	0	0
MAX	1	0	0
MIN	1	0	0
MUL	1	0	0
SGE	2	1	0
SLT	2	1	0
SUB	1	0	0
XPD	2	1	0
CMP	1	0	0
LRP	1-2	0-1	0
MAD	1	0	0
SWZ	1-4 (R300, R400), 1 (R500)	1 (R300, R400), 0 (R500)	0
TEX	1 Texture	0	0
TXP	1 Texture	0	0

TXB	1 Texture	0	0
KIL	1 Texture	0	0

Appendix F: Using the OpenGL Shading Language

Below is a coding example on how to create, compile and link a GLSL program object. More info about GLSL and the language specification can be found at http://www.opengl.org/documentation/oglsl.html.

```
// Vertex Shader
const char vertexProg[] =
"void main()                                                    \n"
"{                                                              \n"
"    // transform vertices into projection space                \n"
"    gl_Position = gl_ModelViewProjectionMatrix*gl_Vertex;\n"
"                                                               \n"
"    // output color                                            \n"
"    gl_FrontColor = gl_Color;                                  \n"
"}";

// Fragment Shader
const char fragmentProg[] =
"void main()                                                  \n"
"{                                                            \n"
"    // Fragment color                                        \n"
"    gl_FragColor = gl_Color;                                 \n"
"}";

// Create and compile a vertex and fragment shader object
vshader = glCreateShaderObject(GL_VERTEX_SHADER);
glShaderSource(vshader, 1,(const GLcharARB**) &vertexProg,NULL);
glCompileShader(vshader);
fShader = glCreateShaderObject(GL_FRAGMENT_SHADER);
glShaderSource(fShader, 1,(const GLcharARB**) &fragmentProg,NULL);
glCompileShader(fShader);

// Attach and use the new shader programs
prog = glCreateProgramObject();
glAttachObject(prog, vshader);
glAttachObject(prog, fShader);
glLinkProgramObject(prog);
glUseProgramObject(prog);
```

Appendix G: Table of internal bit representation of the GL Pixel and Texture formats

Internal Representation of Pixel and Texture Formats

Format	Red bits	Green bits	Blue bits	Alpha bits	Lum bits	Int bits	Depth bits	Unused Bits
RGBA8	8	8	8	8				
RGB8	8	8	8					8
RGB10	10	10	10					2
RGB12	16	16	16					16
RGB16	16	16	16					16
RGB10_A2	10	10	10	2				
RGBA12	16	16	16	16				
RGBA16	16	16	16	16				
RGB5	5	6	5					
RGB4	5	6	5					
RGBA4	4	4	4	4				
ALPHA4				8				
ALPHA8				8				
ALPHA12				16				
ALPHA16				16				
LUM4					8			
LUM8					8			
LUM12					16			
LUM16					16			
LUM4_A4				8	8			
LUM8_A8				8	8			
LUM12_A4				16	16			
LUM12_A12				16	16			
LUM16_A16				16	16			
INT4						8		
INT8						8		
INT12						16		
INT16						16		
RGBA32F	32	32	32	32				
RGBA16F	16	16	16	16				
RGB32F	32	32	32					32
RGB16F	16	16	16					16
ALPHA32F				32				
ALPHA16F				32				
INT32F						32		

INT16F						16		
LUM32F					32			
LUM16F					16			
LUM_A32F				32	32			
LUM_A16F				16	16			
DEPTH32							16	
DEPTH24							16	
DEPTH16							16	

Appendix H: Tools and Libraries to facilitate OpenGL development

GLUT and GLU are the most commonly used utility libraries in OpenGL development. The GLUT library allows one to develop OpenGL applications that are agnostic of the underlying graphics/windowing subsystem. By using the GLUT library, the OpenGL application becomes portable across many different platforms without the need to write platform specific code.
The GLU library exposes various utility functions that help in developing an OpenGL application such as texturing utilities (ie mipmap creation) and drawing functionality (i.e. NURBS).

The gDEBugger application, developed by Graphic Remedy (http://www.gremedy.com/), can be used to debug and profile one's OpenGL application. With gDEBugger, a developer can analyze the OpenGL API call stream to debug or optimize the application. A trial version of the application can be found at their website.