Radeon 9500/9600/9700/9800 OpenGL Programming and Optimization Guide

Version: 1.0
April 5, 2010

Introduction

This guide focuses on how to get the most out of the Radeon 9500/9600/9700/9800 series under OpenGL. These cards will be referred to as the 9500+ series for the purposes of this guide. Most of the performance advice contained in this document is not specific to the 9500+ series, and can be applied to other ATI graphics accelerators and even those from other companies. When something is extremely specific to the 9500+ 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 9500+ series. Please see the ATI OpenGL extensions guide for details on which extensions are found on which products.

Basic Architecture

To understand how one's application is going to perform on a particular platform, it is best to understand the basic architecture. The Radeon 9500+ series is very similar to programmable graphics accelerators before it from a programmer's standpoint. It just elevates the levels of functionality and performance. Its primary advancement is the inclusion of support for floating point color in the texture engine, the shader engine, and the frame buffer.

The transform engine on the 9500, 9500 Pro, 9700, 9700 Pro, 9800, and 9800 Pro has four vertex engines all able to execute a vector operation per clock, while the transform engine on the 9600 and 9600 Pro has two vertex engines able to execute a vector operation per clock. This puts the peak transform rate at approximately one vertex every clock or one vertex every other clock respectively. Naturally, this may not be attainable in real-world situations, but it should provide a good basis for understanding geometry throughput.

The shader engine on the 9500+ series executes a texture instruction and a set of arithmetic instructions every clock cycle. On the 9500, 9600, and 9600 Pro, the instructions are executed across four pixels in parallel. On other chips in the family, the instructions are executed across eight pixels in parallel. As with the vertex engines, the real-world performance is almost certainly more limited by such things as memory bandwidth or starvation.

Transform, Clip, and Lighting

Data specification

The fastest way to provide geometry data to the Radeon 9500+ series is to place the data into vertex array objects or vertex buffer objects, so that the chip can access the data directly in either AGP or video memory. The 9500+ series supports both vertex and index data in these buffers. The drawing with these buffers should be done using the vertex array entry points and not the array element path. To ensure maximum performance from vertex array objects, please see the table below outlining the native formats of the 9500+ series. Data that in a VAO or VBO that is in a format different than the listed ones will have a significant performance penalty, and will likely be slower than other methods of specifying data.

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]

Transform Engine

All geometry processing is performed by the four vertex engines in the 9500+ series. The peak geometry rate is roughly the number of operations per vertices divided by four. 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 table below provides guideline for the number of ops required for each of the instructions in ARB_vertex_program.

ARB_vertex_program is the primary mode of programming the TCL engine for user shaders. The following tables provide information on the resources available and the resource usage by certain instructions.

Op-Code	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	0	0

When using a user specified vertex program, several items must be considered to achieve maximal performance. Most important is using the smallest number of instructions necessary. The driver will collapse and optimize code, but it is always best to start with the best code possible. Next most important is to minimize the number of constants and temporaries used by the program. The fewer temporaries in use by the program, the closer the hardware comes to reaching the theoretical performance limit. As with instructions, the driver will attempt to reduce the use of temps where appropriate.

Display Lists

The Radeon 9500+ series can store geometry from a display list in video memory in most circumstance. To ensure that the display list is stored in the optimal manner, avoid including evaluators, edge flags, generic vertex program attributes, and texture coordinates with four components. For a typical game application, it is best to use vertex arrays with GL_ATI_vertex_array_object or GL_ARB_vertex_buffer_object as they are more flexible and work best with vertex programs.

Clipping

The Radeon 9500+ 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 enabled and the amount of geometry being clipped and 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 Radeon 9500+ 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 at full precision. All interpolation is performed with perspective correction. If screen-space effects are desired, the user must undo the perspective in the fragment shader.

Stipple and Anti-Aliasing

While the Radeon 9500+ series accelerates polygon stippling, line stippling, and line anti-aliasing, the resources used to support it overlap the texture resources. As a result, enabling any of 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 Radeon 9500+ series supports multiple methods to accelerate rendering by culling pixels that are not visible. First, the 9500+ 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 in the frame, because of this it is better for the app to clear the buffer than to attempt to use Z tricks to avoid clearing it. 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 set to the full viewport size and to clear the stencil buffer at the same time if the context has a stencil buffer.

The next depth testing optimization the 9500+ series supports is hierarchical depth testing. This allows the VPU to eliminate occluded blocks of pixels efficiently. To get the benefit of this Hi-Z optimization, the app must maintain a constant sense to its depth compare function over the course of a frame. In other words, the app must not switch from a test using less to one using greater or vice versa or ever 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 9500+ series has a more fine-grained mechanism to cull occluded pixels. 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 pixels faster than the shader can consume them, so a user should not be concerned with enabling 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 9500+ 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 direction of the primitive. Please see the extension spec ATI_separate_stencil for more details.

Shader Unit

The fragment shading unit can execute one texture instruction, one rgb instruction, and one alpha instruction in each clock cycle. This seems very similar to most accelerators before it, but the performance characteristics are somewhat different as it is the first graphics accelerator to perform all its computations on floating point numbers. This means that certain operations that were once free may now have an actual cost. The actual resources available in the shader unit are as follows:

ALU Instructions:	64 of both rgb and alpha
Texture Instructions:	32
Temporary registers:	32 4-tuple
Constant registers:	32 4-tuple

To get the most out of this new fragment unit, the user should use ARB_fragment_program. The following table provides information on the amount of HW resources required to implement the ops in fragment program on the 9500+ series.

Instruction	HW Instructions	HW Temps	HW Constants
ABS	0-1	0	0
FLR	2	1	0
FRC	1	0	0
LIT	7	1	1
MOV	0-1	0	0
COS	11	2	3
EX2	1	0	0
LG2	1	0	0
RCP	1	0	0
RSQ	1	0	0
SIN	10	2	3
SCS	6-8	1	2
POW	3	1	0
ADD	1	0	0
DP3	1	0	0
DP4	1	0	0
DPH	2	1	0
DST	4	1	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	1	0
TEX	1 Texture	0	0
TXP	1 Texture	0	0
TXB	1 Texture	0	0
KIL	1 Texture	0	0

Additionally, not all of the source modifiers are natively supported on the 9500+ series. The native source modifiers 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.

When attempting to write an optimal fragment program, it is best to keep in mind the Radeon 9500+ series' architecture. The shader on the 9500+ 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 segregate 3-tuple operations to occurring 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 increase force the driver to expand the operation into more instructions.

Texture Operations

The fragment shader on the Radeon 9500+ series can execute a maximum of 32 texture fetches as mentioned above. Each of these 32 fetches may come from any of the available 16 texture contexts. This full 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 sets which is eight.

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 textures requiring 32 to 64 bits it uses two cycles and so on.

The Radeon 9500+ 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 a few limitations that other types do not have. First, they may only be sampled as NEAREST or NEAREST_MIPMAP_NEAREST. Secondly, the texture border color is not available, so only formats that do not use the border color such as GL_REPEAT and GL_CLAMP_TO_EDGE are supported as wrap modes. 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 Radeon 9500+ series is 2048 in all dimensions. Allocating the maximum texture size is impossible as it exceeds a 32-bit address space. For optimum performance, individual textures should be kept to sizes 32MB or smaller as it allows more flexibility in placing the texture in memory.

The sixteen bit integer formats also have one notable limitation. These 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 often equates to an increase in performance. Judiciously using the compressed texture formats is one way to do this. 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 Radeon 9500+ 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 9500+ series is that is 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 take care to either ensure that unneeded quantities are either masked or set to the identity value for the blend. A typical example of this is rendering and not paying 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 driver will have to blend those pixels just to keep the destination alpha correct even though they may never be used.

Multisampling

The Radeon 9500+ 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 multi-sample buffers are only available when queries are made with wgl_ARB_pixel_format. To get the most out of multisampling on the 9500+ 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 is linear. This value works on essentially all displays presently in use.

Miscellaneous Categories

Pixel Transfer Operations

The flexibility of the Radeon 9500+ series's 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.

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.

Representation of Texture Formats

Format	Red bits	Green bits	Blue bits	Alpha bits	Lum bits	Int bits
RGBA8	8	8	8	8		
RGB8	8	8	8			
RGB10	10	10	10			
RGB12	16	16	16			
RGB16	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

While glBitmap is hardware accelerated on the Radeon 9500+ 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

The Radeon 9500+ series supports rendering to floating point color buffers via the extension WGL_ATI_pixel_format_float. The color buffers may be allocated to contain either IEEE 32-bit floats or the 16-bit s10e5 floats described previously. All allocations for these formats must be made through the WGL_ARB_pixel_format extension.

With Radeon 9500+ series floating point color buffers, certain limitations apply and the user must be aware of them. We decided not to put these limitation into the WGL_ATI_pixel_format_float extension because we did not want to create a crippled extension that would need replacing in the future when hardware without these limitations is available. First, floating point buffers are not displayable, so they may only be allocated as Pbuffers. Second, they 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
- blending
- fog

The supported operations are:

- depth test
- stencil test
- color mask.

There are also operations that are defined to not happen in the WGL_ATI_pixel_format_float extension because they do not make sense when applied to floats. These operations are:

- dither
- logical operation

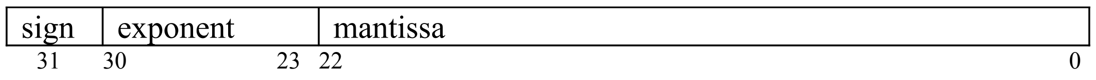
Multiple Output Colors and Destination Color Buffers

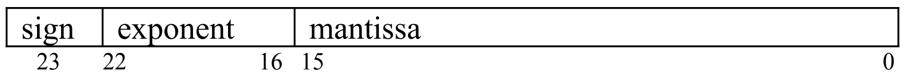
The Radeon 9500+ series supports simultaneous rendering to multiple color buffers. This allows it to fully accelerate render modes such as FRONT_AND_BACK. Additionally, the shader may output separate colors for each of the buffers being written to. These independent color outputs and the control of their associated destination buffers are defined in the GL_ATI_draw_buffers extension. The 9500+ series supports up to four simultaneous color outputs through this extension.

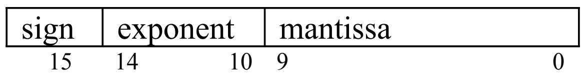
Under fixed function OpenGL, the same color value will be written to all of the draw buffers. When fragment programs are enabled, they may optionally write separate outputs for each of the draw buffers. In either case, all the buffers share the same alpha-blending and color masking state.

Data Formats

The Radeon 9500+ 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 9500+ series uses a 24 bit format for internal shader computations. 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

Appendix A: Radeon 9500+ series Extension and New Feature Usage

Accumulations buffers

The Radeon 9500+ 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 when they desire it as it could cause a memory overhead, but the driver is smart enough to avoid the overhead 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 pbuffers.

ARB_vertex_program

The Radeon 9500+ 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 Radeon 9500+ 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 9500+ are included above.

ARB_depth_texture, ARB_shadow, ARB_shadow_ambient

The Radeon 9500+ series supports shadow buffers via ARB_depth_texture, ARB_shadow, and ARB_shadow_ambient. The 9500+ series only supports NEAREST and NEAREST_MIPMAP_NEAREST 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 Radeon 9500+ 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 (pbuffers). On the 9500+ 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.