ATI Technologies Inc.

The Radeon X1000 Series Programming Guide

Revision: 1.1
Created: Mar. 2006
Author: Guennadi Riguer

ATI Technologies Inc.

Contents

1. Introduction

The Radeon™ X1900, X1800, X1600 and X1300 represent new additions to the legendary Radeon™ family of 3D graphics hardware from ATI Technologies Inc. These new products are the third generation of DirectX® 9 hardware, now supporting Shader Model 3.0 and other advanced features to cover all user demands. From the ultra-high-end through the mainstream and value market segments, this new addition to the Radeon family allows developers to easily scale performance without sacrificing any application features.

The Radeon X1800 and X1900 represents the leadership in performance with their very high-performance 16 and 48 pixel processors correspondingly. The mainstream Radeon X1600 has 12 ALU pixel shader engines, 8 Z-pipes and 4 back-end pixel pipes that process colors. The value market segment solution, the Radeon X1300, has 4 pixel pipes throughout the pixel pipeline. The following table summarizes the new hardware configurations.

Card	Market	Vertex Engines	Pixel Engines	Texture Pipes	Z Pipes	Back-end Color Pipes
X1900	Performance	8	48	16	16	16
X1800	Performance	8	16	16	16	16
X1600	Mainstream	5	12	4	8	4
X1300	Value	2	4	4	4	4

One interesting thing to note about the Radeon X1900 and X1600 is the 3:1 ratio of ALU to texture pipes in the pixel shader. This represents a current trend of complex shaders to tip the performance balance more towards ALU operations.

The latest Shader Model 3.0 products from ATI pack a lot of functionality and performance-enabling features. However, these are quite complex chips, and without a good understanding of how they work one can be easily make sub-optimal programming choices and deliver sub-optimal performance. This guide explains the most important features and nuances of the latest ATI architecture and how to make the most out of it. The majority of optimizations and recommendations described in this document apply to the complete family of Radeon X1xxx products. Cases where performance or functionality differs between chipset revisions are explicitly mentioned.

2. Vertex Processing

The following section of this document outlines several key points with respect to performance-friendly vertex processing on the Radeon X1x00 family of hardware.

2.1. Vertex Caching

Just like previous ATI graphics chips, the Radeon X1x00 family of chips has pre- and post-vertex processor caches. The pre-vertex processor cache primarily helps reducing vertex fetch bandwidth and hides memory access latency. While memory clocks have been somewhat increased, the addition of more vertex processors and a higher core clocks puts more burden on the vertex fetcher and the pre-vertex processor cache, potentially making vertex fetching a bottleneck. Thus, as never before, it is very important to have cache-friendly meshes to get the most of the vertex processors. Whenever it makes sense, try to align your vertex structure sizes to a multiple of the 32 bytes and use as few streams as possible. Also, reorganize mesh indices to maximize vertex re-use in adjacent triangles and reorder vertices in vertex buffers for locality of access. We recommend using the **`ID3DXMesh::Optimize()`** API from the D3DX library as it will perform both of these tasks based on the cache sizes of the underlying 3D device.

2.2. Vertex Textures

The Radeon X1900 and all other Shader Model 3.0 Radeon family members (including future ATI DirectX® 9 products) do not support vertex texturing. Vertex texturing is a feature that would require substantial architectural changes to be implemented at good performance on current graphics hardware. All existing hardware implementations of vertex texturing are limited in terms of performance and features, to the point of making this feature hardly usable in real-time applications. This is the reason why Radeon X1x00 cards do not support vertex texturing. It is important not to assume availability of vertex textures upon detecting VS 3.0. Always use the **`CheckDeviceFormat()`** DirectX® 9 API method with the **`D3DUSAGE_QUERY_VERTEXTEXTURE`** usage query flag to determine support of vertex texturing for a specific surface format. If none of the surfaces expose this query flag, then the hardware does not support vertex texturing.

2.3. Static Flow Control in Vertex Shaders

The static flow control is a type of flow control that does not depend on any computations performed in the shader. Just like in VS 2.0, VS 3.0 provides 16 boolean constants and 16 integers for implementing static conditionals and loops. Static flow control can sometimes help with shader management and reduce the combinatorial explosion of shaders in some applications. Keep in mind that flow control, even if it is static (which automatically guarantees the coherency of execution paths in the shader), can still be detrimental to shader performance. With flow control – and especially short conditional clauses – the compiler does not have as much freedom in scheduling instructions in the most efficient way. To improve the performance of shaders utilizing static flow control, the driver might attempt to recompile vertex shaders without flow control based on the provided constants. The driver would then cache these conditionally compiled shaders to avoid redundant shader recompilation. If you are using static flow control in a vertex shader you should pre-cache recompiled variations of that shader

in the driver to avoid on-the-fly shader optimization during application runtime. You can do this on a very first frame by rendering a dummy triangle with the shader and all the combinations of boolean constants that will be used throughout the execution of the application.

2.4. Dynamic Flow Control in Vertex Shaders

Dynamic flow control in vertex shaders is a new feature in VS 3.0, which allows conditional branching within a shader depending on previously computed results. There are several ways to implement branched calculations in VS 3.0: using predication and actual dynamic flow control instructions. Usually, the majority of interesting computations happen at a per-pixel level meaning that vertex shaders are usually only responsible for setting up tangent space and other ancillary information. In this typical, limited use of vertex shaders, there is very little need for dynamic flow control; consequently, not as much emphasis was placed on dynamic flow control efficiency in vertex shaders as was in pixel shaders.

To make the most out of vertex shaders on Radeon X1x00, it is strongly recommended to minimize vertex shader flow control and avoid it completely if possible. Using predication or a couple of simple conditional statements with fairly small clauses are the best examples of how to use dynamic flow control in vertex shaders. Always keep in mind that dynamic flow control in pixel shaders on Radeon X1x00s is much more efficient than in vertex shaders.

2.5. Instancing

The major performance bottleneck in many graphical applications is the number of submitted geometry batches. While vertex throughput has substantially increased over the years, the number of batches or draw-primitive calls that can be rendered each frame have remained roughly the same. The problem is that the number of batches that can be rendered per frame is directly tied to CPU performance and any GPU performance improvements have little impact on this bottleneck. To help alleviate this problem, the DirectX® 9 API introduced the ability to render instanced geometry with a single draw call. Instead of rendering many similar objects one at a time, the application can now specify the common instanced object data in one vertex stream and per-instance parameters (e.g. position, color, size, etc.) in another vertex stream; those streams are then combined in the vertex shader. The hardware will automatically replicate the object vertex data for each of the rendered objects while pairing it with per-instance data. Check the DirectX® 9 SDK documentation for more information about instancing.

Older ATI DirectX® 9 hardware also supports instancing; however, it requires some special API-level tweaks to take advantage of. The ATI Radeon SDK contains examples of how to enable instancing on older hardware. On Radeon X1x00 family of video cards, geometry instancing is a first class citizen since all Shader Model 3.0 hardware natively supports this feature by definition. Also, all Radeon X1x00s have small architectural improvements that can sometimes improve rendering performance of very low polygon count instances. Please use

instancing whenever it makes sense to make your application less CPU dependent while increasing the complexity of the rendered scenes.

3. Texturing

The new generation of Shader Model 3.0 hardware from ATI includes new texturing capabilities in addition to a wide range of previously supported formats. This section explains the major texture unit changes compared to the previous generation of ATI video cards.

3.1. Large Textures

The Radeon X1x00 hardware quadruples the previous maximum texture size to 4096x4096. While it might be tempting to use these huge textures to increase the detail of graphic scenes, one should consider the additional memory footprint and its implication on performance. There are more sophisticated texture-LOD solutions that you might consider for adding extra details to your scenes. Lastly, always use mip-maps to improve performance with large textures.

3.2. New Texture Formats

The new additions to the Radeon family have several new texture formats and some improvements to existing ones. Please see section 9 (*Summary of Texture Formats*) for more information about supported formats.

3.2.1. ATI1N

The 3Dc technology and the ATI2N normal map compression format in particular was a very important milestone in the development of compressed texture formats. The new generation of graphics hardware takes texture compression technology further by providing an additional single channel format that can be used for storing luminance values, height maps, and many other different types of data. This new format is called ATI1N and there is a new Four-CC code for accessing it.

You can use the Four-CC code to check format availability as well as for texture creation as shown in a code snippet below.

```
#define FMT_ATI1N ((D3DFORMAT)MAKEFOURCC('A', 'T', 'I', '1'))

// Check support
if (SUCCEEDED(pD3D->CheckDeviceFormat(
      D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, D3DFMT_X8R8G8B8,
      0, D3DRTYPE_TEXTURE, FMT_ATI1N))
{
      bSupportsATI1N = true;
}
```

The internals of the ATI1N format are actually quite simple. It is effectively the alpha block of the DXT5 color compression format, which provides 2:1 compression ratio compared to **D3DFMT_A8** or **D3DFMT_L8** formats. The single available channel in this format is red and the pixel shader should use appropriate swizzles to replicate that data to any other channels as necessary. It is important to take this into consideration whenever you want to use ATI1N as a replacement for **D3DFMT_A8** or **D3DFMT_L8** formats.

The ATI1N format is supported for volume textures as well as regular 2D textures, and is a perfect option for compressing large luminance volumes and other volumes of monochromatic data. Each slice of the volume texture is compressed independently as a normal 2D surface and all other processing happens transparently.

3.2.2. Depth Textures

Recent drivers have added a special depth texture format that allows sampling 16-bit depth buffer information as a texture on all ATI DirectX® 9 video cards. This is especially useful for implementing shadow maps and other techniques that rely on the scene's depth information. Previously, applications had to rely on depth values output from the pixel shader to a high-precision render target, sometimes using a separate depth rendering pass. This new format allows an application to bind a depth texture surface as a depth buffer and later re-use its contents as a texture without extra rendering overhead.

While 16-bit depth textures are very useful, some algorithm implementations might find the 16-bit precision insufficient. To solve this problem the Radeon X1900, X1600 and X1300 added a new, more precise 24-bit depth texture format. Both 16-bit and 24-bit formats are implemented as Four-CC codes and application should query their support before trying to use them. An application can create a depth texture with one of the available formats and set it both as a depth buffer and a texture. It is prohibited to simultaneously render to the depth texture and fetch from it as a texture. Because rendering with depth textures is generally somewhat slower than with a normal depth buffer, they should not be used as replacement for the primary depth buffer.

When rendering shadow maps, only the depth information is relevant and scene color information can be disregarded. To save fill rate you should disable color output using a color

write mask. But even with a color write mask disabling the color output, a color buffer of matching multisample type to depth texture (**D3DMULTISAMPLE_NONE**) should still be created and bound to the D3D device. For large shadow maps this color buffer could waste a lot of space, so it should be created with the smallest renderable surface format available (such as **D3DFMT_R5G6B5**).

The following sample code shows how to properly detect and use depth textures on ATI hardware for rendering shadow maps.

```
#define FOURCC_DF16  ((D3DFORMAT) MAKEFOURCC('D','F','1','6'))
#define FOURCC_DF24  ((D3DFORMAT) MAKEFOURCC('D','F','2','4'))

D3DFORMAT fmtDepthTex = D3DFMT_UNKNOWN;

// Check DF24 and DF16 support
if (bNeedHighPrecision && SUCCEEDED(pD3D->CheckDeviceFormat(
      D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, D3DFMT_A8B8G8R8,
      D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_TEXTURE, FOURCC_DF24))
{
      fmtDepthTex = FOURCC_DF24;
}
else if (SUCCEEDED(pD3D->CheckDeviceFormat(
      D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, D3DFMT_A8B8G8R8,
      D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_TEXTURE, FOURCC_DF16))
{
      fmtDepthTex = FOURCC_DF16;
}

// Try creating a depth texture
if (fmtDepthTex != D3DFMT_UNKNOWN)
{
      pd3dDevice->CreateTexture(DST_WIDTH, DST_HEIGHT, 1,
            D3DUSAGE_DEPTHSTENCIL, fmtDepthTex, D3DPOOL_DEFAULT,
            &pDepthTex, NULL);

      // Get depth texture surface
      pDepthTex->GetSurfaceLevel(0, &pDepthSurf);
}

// Create dummy color buffer
pd3dDevice->CreateRenderTarget(DST_WIDTH, DST_HEIGHT, D3DFMT_R5G6B5,
      D3DMULTISAMPLE_NONE, 0, FALSE, &pColorBuffer, NULL);

// Set dummy color buffer and disable color writes
pd3dDevice->SetRenderTarget(0, pColorBuffer);
pd3dDevice->SetRenderState(D3DRS_COLORWRITEENABLE, 0);


// Set depth texture as a depth buffer
pd3dDevice->SetDepthStencilSurface(pDepthSurf);
```

```
// Set new viewport for DST
newViewport.X = 0;
newViewport.Y = 0;
newViewport.Width = DST_WIDTH;
newViewport.Height = DST_HEIGHT;
newViewport.MinZ = 0.0f;
newViewport.MaxZ = 1.0f;
pd3dDevice->SetViewport(&newViewport);

// Render to depth surface
// ...

// Restore color and depth buffer
pd3dDevice->SetRenderTarget(0, pOldColorBuffer);
pd3dDevice->SetRenderState(D3DRS_COLORWRITEENABLE,
      D3DCOLORWRITEENABLE_RED | D3DCOLORWRITEENABLE_GREEN |
      D3DCOLORWRITEENABLE_BLUE | D3DCOLORWRITEENABLE_ALPHA);
pd3dDevice->SetDepthStencilSurface(pOldZBuffer);

// Restore viewport
pd3dDevice->SetViewport(&oldViewport);

// Set depth texture for sampling
pd3dDevice->SetTexture(0, pDepthTex);
// ...
```

3.3. Border Color Texture Addressing Mode

DirectX® 9 supports a variety of texture addressing modes that describe how texture coordinates outside of the [0, 1] range are processed. Previous generations of ATI DirectX® 9 hardware did not fully support the border color texture addressing mode and this functionality was not exposed through the caps. The new Radeon X1x00 family of cards fully supports border texture color and applications can now fully rely on this feature. Always use the caps to check the availability of this feature prior to using it. To verify support of the border color addressing mode use the **D3DPTADDRESSCAPS_BORDER** cap bit in the TextureAddressCaps field of the D3DCAPS9 structure.

3.4. Floating Point Textures

Floating point textures appeared in the first generation of DirectX® 9 hardware and revolutionized 3D rendering by providing means to implement high dynamic range (HDR) imaging and other rendering techniques that require higher precision and range than previously available. The Radeon X1x00 family has improved floating point surface support by adding blending and multisampling, which will be discussed later; however, floating point texture filtering is not supported. This is not a huge concern, and in rare cases where filtering

precision is very important, it can be simulated in pixel shaders. In a wide variety of situations like HDR rendering, other less expensive high-quality solutions can be used.

When trying to emulate floating point filtering, check if any of the 16-bit per channel filterable integer formats would provide a good alternative. In many cases where data does not span very large numeric range, integer formats with fixed point representation is the best choice and in some cases could provide better precision than floating point formats.

If using floating point textures is your only choice, always step back and check if floating point filtering emulation can be optimized. For example, when downsampling textures in half by averaging 2x2 texel regions you do not need full bilinear filtering emulation and it is sufficient to fetch and average 4 texels. Sometimes, bilinear filtering, whenever it is supported, can be used as an optimization primitive for implementing large filter kernels. For instance, a 3x3 filter can be implemented using 4 bilinear fetches. If these fetches are emulated in the shader, you will end up with 16 point sampled fetches where you really need only 9.

The following code snippet shows an example of optimized bilinear filtering emulation using floating point textures without mip-maps, which takes advantage of texel rounding of point-sampled filtering. The texel offsets used in the code are slightly smaller than 0.5 to work around the incorrect behavior caused by snapping to texels in point sampled filtering. This delta from 0.5 value (`fudge` constant) can be tweaked based on hardware and texture dimensions to produce the best results.

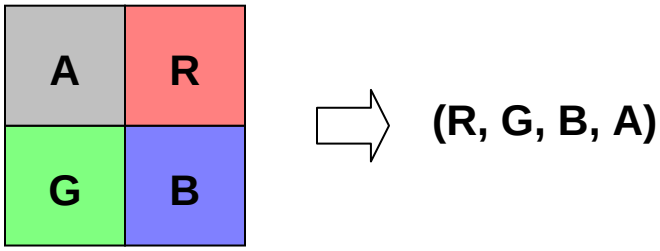
```
float2 texWidthHeight = {TEX_WIDTH, TEX_HEIGHT};
float4 texOffsets = {-0.5/TEX_WIDTH+fudge, -0.5/TEX_HEIGHT+fudge,
      0.5/TEX_WIDTH-fudge, 0.5/TEX_HEIGHT-fudge};

float4 tex2D_bilerp(sampler s, float2 texCoord)
{
   float4 offsetCoord = texCoord.xyxy + texOffsets;

   float2 fracCoord = frac(offsetCoord.xy * texWidthHeight);
   float4 s00 = tex2D(s, offsetCoord.xy);
   float4 s10 = tex2D(s, offsetCoord.zy);
   float4 s01 = tex2D(s, offsetCoord.xw);
   float4 s11 = tex2D(s, offsetCoord.zw);

   s00 = lerp(s00, s10, fracCoord.x);
   s01 = lerp(s01, s11, fracCoord.x);
   s00 = lerp(s00, s01, fracCoord.y);
   return s00;
}
```

For more ideas and inspiration on how to emulate floating point filtering, please, refer to our **HDR Texturing** whitepaper available as a part of ATI Radeon SDK as well as on the ATI developer web site (www.ati.com/developer).

3.5. Fetch-4

Lately, shadow mapping has become one of the most popular shadow rendering methods. Shadow mapping requires special filtering methods to antialias shadows and the most frequently used approach is PCF, or *percentage-closer-filtering*. PCF works as follows. Multiple samples containing depth values from the shadow map are fetched and compared to the distance to the rendered surface. This comparison produces a binary result for each of the tested samples indicating whether they are in shadow or not. The results of comparison are combined to produce the shadow intensity at a given point. Using large filter kernel sizes can produce really nice soft shadows.

The Radeon X1900, X1600 and X1300 have a new feature that is conveniently suited to accelerate PCF implementations. This feature is called **Fetch-4** and with one texture fetch it can retrieve 4 neighboring texels (2x2 texel block) from a single-channel texture map. Four individual samples of a single-channel texture are swizzled into RGBA channels when they are fetched from the texture. The swizzling of 2x2 texel block into 4 channels is illustrated by the following diagram.



Fetch-4 is controlled on a per sampler basis and can be enabled by sending special “magic” tokens to the driver using the **`D3DTSS_MIPMAPLODBIAS`** texture sampler state. The application should submit these “magic” tokens to the API only on hardware that supports Fetch-4 functionality. Note that point sampling filtering must also be enabled for Fetch-4 to be triggered. Fetch-4 is supported on all ATI hardware that supports the DF24 format, so you should check for DF24 format support before using Fetch-4. The following code shows how to detect, enable and disable Fetch-4 operation.

```
#define FETCH4_ENABLE ((DWORD)MAKEFOURCC('G', 'E', 'T', '4'))
#define FETCH4_DISABLE ((DWORD)MAKEFOURCC('G', 'E', 'T', '1'))

#define FOURCC_DF24  ((D3DFORMAT) MAKEFOURCC('D','F','2','4'))

BOOL bFetch4Supported = FALSE;

// Check for DF24 support
if (SUCCEEDED(pD3D->CheckDeviceFormat(
      D3DADAPTER_DEFAULT, D3DDEVTYPE_HAL, D3DFMT_X8R8G8B8,
      D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_TEXTURE, FOURCC_DF24))
{
      bFetch4Supported = TRUE;
}

if (bFetch4Supported)
{
      // Enable Fetch-4 on sampler 0
      pd3dDevice->SetSamplerState(0,
            D3DSAMP_MIPMAPLODBIAS, FETCH4_ENABLE);

      // Set point sampling filtering (required for Fetch-4 to work)
      pd3dDevice->SetSamplerState(0, D3DSAMP_MAGFILTER, D3DTEXF_POINT);
      pd3dDevice->SetSamplerState(0, D3DSAMP_MINFILTER, D3DTEXF_POINT);

      // ...

      // Disable Fetch-4 on sampler 0
      pd3dDevice->SetSamplerState(0,
            D3DSAMP_MIPMAPLODBIAS, FETCH4_DISABLE);
}
```

This approach works extremely well for PCF implementations with regular grid positioning of the filter taps and, for example, allows implementing a 4x4 PCF kernel with only 4 texture fetches. To get better results, PCF implementations sometimes use jittered sample locations or otherwise more sophisticated sample distribution in the filter kernel. Fetch-4 can also help in those cases as well. It is possible to achieve similar visual results by using fewer Fetch-4 jittered samples than point-sampled taps. For example, instead of 16 point-sampled jittered samples you might get away with 8 to 12 jittered Fetch-4 fetches, while maintaining the same or better visual quality.

Fetch-4 works not only on DST (depth-stencil textures) formats (like DF24), but with all other single-channel formats as well; therefore, this feature can be used for implementing various filter kernels that operate on single-channel textures.

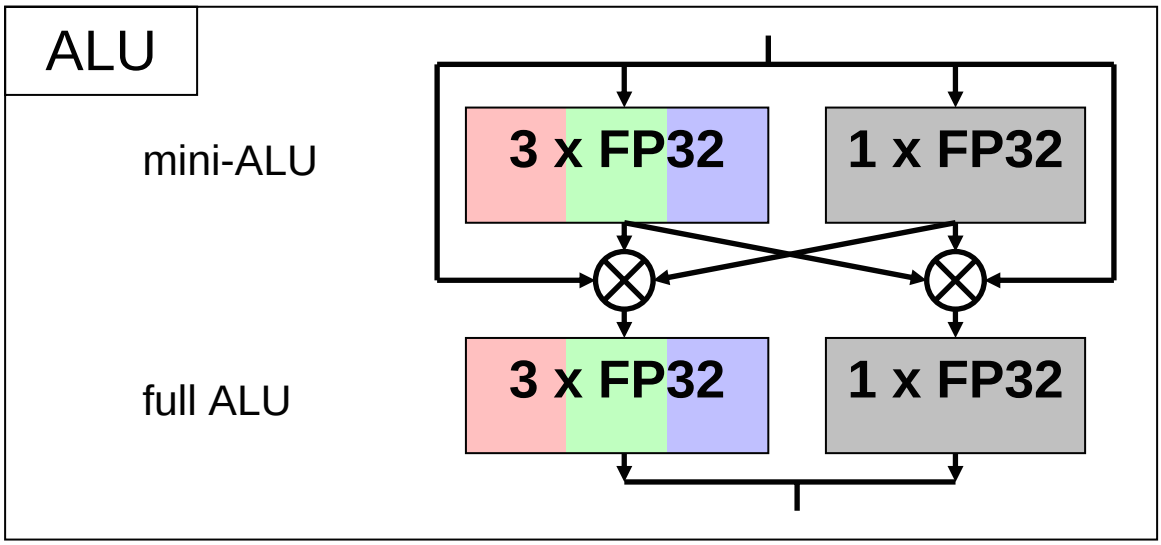
4. Pixel Shader

The Radeon X1x00 series of chips incorporate extremely sophisticated high-performance pixel shader engines completely redesigned to support all Shader Model 3.0 features without sacrificing performance. This section covers the architecture of the pixel processor as well as tips and techniques to unleash the raw pixel processing power of the Radeon X1x00 family of graphics hardware.

4.1. Radeon X1xxx Pixel Shader Architecture

The Radeon X1x00 family of chips has a brand new pixel shader engine designed to support all the new pixel shader features while maintaining the high, predictable performance characteristics of previous generation hardware. As was the case in previous generation pixel shader engines, the Radeon X1x00 cards support only a single precision mode – high precision. To fully match the requirements of Shader Model 3.0 the hardware was upgraded to provide an IEEE 32-bit equivalent floating point implementation. The section of this document on *floating point rules* explains the intricacies of the floating point implementation in pixel shaders.

As with previous ATI hardware generations, the Radeon X1x00 pixel engine can simultaneously execute a texture and an ALU instruction each clock. Each ALU instruction can be a full 4D vector or a combination of a 3D vector and a scalar. Also, in the addition to the main ALU, there is a mini-ALU that can execute a subset of ALU instructions. The following diagram illustrates the ALU block architecture.



The major redesign of the pixel shader engine comes from the fact that the shader pipe now fully supports flow control with predication, branching and looping (both static and dynamic).

The main emphasis for this generation of shader architecture is on the efficiency of the flow control as it is the most important and useful feature of PS 3.0. The shader engines of the Radeon X1x00 family are capable of executing many pixel threads in parallel at very high performance, with a smaller thread size than the competition supports. These features bring a new level of unsurpassed pixel performance never seen before in any graphics hardware.

4.2. Instruction Counts

The Radeon X1x00 cards expose 512 pixel shader instruction slots that along with flow control provide a lot of flexibility for implementing very sophisticated algorithms. With 4 levels of nested loops you can be potentially executing up to trillions of instructions per pixel! This might cause the hardware to appear to have stopped responding, so you have to exercise caution when coding shaders with flow control. On the other hand, some shaders might fail to compile if they exceed the number of allotted instruction slots. Note that some instructions take more than one slot, as specified in DirectX® 9 specification. Standalone HLSL and ASM compilers, as well as the ones implemented in the D3DX library, included in the DirectX® 9 SDK may only report approximate instruction counts and they might sometimes be smaller than the real instruction slots count that the runtime verifies against.

4.3. Instruction Vectorization, Swizzles and Write Masks

The shader processors are vector units; thus, to achieve the highest shader processor utilization one should make sure that as many computations as possible are vectorized and are executed simultaneously. The compiler built into the driver can pair 3D and scalar instructions and you should use write masks to give the compiler a chance to co-issue instructions whenever possible. For example, if you do not care about the alpha output, you could use the `.rgb` write mask in all your color calculations and let the alpha come from a texture fetch or some other instruction.

While the compiler does a good job of pairing instructions, its task can often be made simpler by explicitly vectorizing computations in the source code. This is especially important for 2D + 2D cases where compiler cannot easily perform this task. A lot of post-processing shaders operate on 2D texture coordinates and these calculations are the ideal target for manual vectorization. Shader Model 3.0 supports arbitrary swizzles in the pixel shader and it makes programmers' task of vectorizing calculations much simpler. The following HLSL shader code snippets illustrate an example of such an optimization.

```
// Sub-optimal code
float2 v0 = t + offset0;
float2 v1 = t + offset1;
float4 c = tex2D(s, v0) + tex2D(s, v1);
```

```
// Optimized code
float4 v01 = t.xyxy + offset01;
float4 c = tex2D(s, v01.xy) + tex2D(s, v01.zw);
```

The few instructions saved here and there will quickly accumulate and in a typical post-processing shader, where texture coordinate calculations dominate, it could result in a significant performance boost.

4.4. SINCOS instruction

The Radeon X1x00 family of graphics chips natively supports the SINCOS instruction in the pixel shader and the hardware is capable of computing a SIN or COS function in a single clock cycle. If both function results are needed, the computation will take two clock cycles. If you need only one of the functions computed, make sure to use the appropriate write mask.

4.5. Instruction Balancing

The Radeon X1x00 pixel shader pipe is built on a lot of concepts that made the older DirectX® 9 Radeons so successful. One of those is the ability to execute texture and arithmetic instructions simultaneously. For each processed pixel, the Radeon X1800 and the Radeon X1300 can fetch a texture and execute one ALU instruction per clock (assuming memory bandwidth is not a limiting factor). Fetching 64-bit textures, volume textures, or using trilinear or more expensive filtering would take more clocks to execute, allowing a greater number of ALU instructions to be executed during the texture fetch. Ideally you should target anywhere around 1:4 texture to ALU instruction ratios; however, the optimal ratios vary based on filtering, texture formats and other factors.

In the past, from the early DirectX® 8 days, we have seen the texture to ALU instruction ratios slowly increase from 1:1 or 1:2 to much higher numbers as shader complexity grew. As we move towards more and more complex shaders that implement complex lighting models, procedural materials and other advanced effects, the computational complexity will grow disproportionably to texture fetch requirements. To sustain the performance growth we all have enjoyed for years, shader developers will have to rely more and more on shader computations rather than texture fetches, as the former continues to grow with each new GPU generation at a much quicker pace than the available memory bandwidth.

The Radeon X1900 and X1600 marks a new and exciting trend of targeting higher than ever before texture to ALU instruction ratios by executing up to 3 ALU instructions per clock per pixel, while performing at most one texture fetch. Volume textures, 64-bit or more textures, and expensive filtering will require more than one clock cycle for a texture fetch, allowing 6 or even

more ALU instructions to be executed at the same time. To maintain the highest pixel pipe efficiency the texture to ALU instruction ratio has to be 1:8 or even higher. Future hardware is expected to continue this trend of targeting higher texture to ALU ratios; however exact numbers might vary from product to product.

4.6. Pixel Shaders and Flow Control

The biggest and most important feature of Shader Model 3.0 is the addition of flow control to pixel shaders. The supported flow control constructs include subroutine calls, predication, conditional statements and loops. Also, based on the condition that invokes the flow control, there are two types of flow control available in shaders: static flow control and dynamic flow control. The former is controlled by parameters that are known before shader execution, while the latter is based on the parameters derived during shader execution.

This section provides in-depth explanation of flow control, the implications of its use, as well as general recommendations and optimizations.

4.6.1. Subroutine Calls

Subroutines calls in PS 3.0 assembly language allow shaders to exceed the 512 instruction limit by moving common code into a subroutine. Because of the extra overhead of calling routines, the HLSL compiler as well as the shader compiler in the driver will try to eliminate the subroutine calls by in-lining the functions into the body of the caller. The only real use for subroutine calls in assembly is to work around the 512 instruction limit in cases when subroutine calls reduce number of instruction slots used.

4.6.2. Static Flow Control

As advanced graphics techniques mature and applications rely more and more on shaders, we are faced with a combinatorial explosion of shaders due to the number of lights, materials and many effects that might need to be combined in shaders. PS 3.0 includes static flow control functionality, previously available only in vertex shaders, to help ease the pain of shader management. Shader developers can now produce a shader that contains all possible elements that could be used at one time – an über-shader. At runtime conditionals are used to pick the combinations of code parts that achieve the desired result.

As with vertex shaders, there are pros and cons to using static flow control in pixel shaders. On one hand, static flow control allows the simplification of shader management and thus a reduction in CPU overhead, which can lead to some performance improvements in CPU bound cases. On the other hand, the use of static flow control can impact shader performance, which would have a negative effect on performance in fill-rate bound cases. By themselves, flow control instructions are not very expensive. However, their placement in the shader code restricts the compiler's ability to reorganize other shader instructions for co-issue and achieve

the most optimal scheduling. In some extreme cases pixel shader performance can suffer by 50% or more if a lot of small branches are used throughout the shader. The driver is capable of recognizing situations like this and compiling out flow control in the shader based on the currently set conditionals. The driver will cache those shaders to make sure they are not recompiled every time boolean shader constants change. To make sure the driver does not have to recompile shaders during application execution, it is recommended to “warm” the shader cache by rendering on a very first frame a dummy triangle with static flow control shaders and the most common boolean permutations that will be used with these shaders. For compatibility with earlier shader models the application might adopt a similar approach of pre-caching the most common shaders that are compiled ahead of time based on some boolean conditionals that are used for shader fragment linking.

4.6.3. Dynamic Flow Control

Dynamic flow control gives shader developers the ability to implement conditionals and loops, where execution might vary from pixel to pixel. This allows creation of very complex materials that might dynamically combine various lighting or material components per pixel. Another big use for dynamic flow control is to skip parts of the computations and texture fetches that would not contribute to the final result.

As it is the case with static flow control, dynamic flow control can be good or bad, depending on how it is used. Besides the potential small performance impact from limiting instruction reshuffling, dynamic flow control potentially has a bigger problem. The reason modern GPUs are so fast is because they are massively parallel in architecture, with many pixels in flight at the same time. The smallest processing element is a 2x2 pixel quad; and a number of these quads run in lockstep, executing the same instructions. This collection of pixel quads constitutes an execution thread, and on modern graphics hardware many threads are sharing the same shader processor. If flow control makes pixels that are executed within a thread take different paths, all the pixels in the thread, regardless of whether they should or should not execute the path, will be dragged along. The pixels that should not execute a given path will just ignore the instructions, while pixels that should execute this path will be processed as usual. This sounds very bad and inefficient, but in reality it is not all that gloomy as long as there is a fair amount of coherency in branch selection within reasonably sized pixel blocks. The Radeon X1x00 family has a thread size of 16 pixels, which is the smallest in the industry, and it provides absolutely the best dynamic flow control efficiency on the market.

Besides reasonable coherency of flow control execution there are several other tips that you should follow. Avoid many small conditional statements scattered throughout a shader. Just like in the static flow control case, this will impact optimization efficiency. Dynamic flow control should rather be used to skip fairly large portions of the code.

While the dynamic flow control on Radeon X1x00 graphics hardware is extremely fast and efficient, there is a way to improve it even further. The fastest pixel shaders with dynamic flow control are the ones that do not contain loops and have no more than 6 levels of dynamic branching. If the shader is to execute a dynamic loop with a fairly low iteration count, you might

be better off substituting loops with a number of conditional statements as illustrated in the example below. However, if all loops in the shader cannot be substituted with branching there will be no performance improvement from this substitution.

```
// Sub-optimal code
int i = 0;
float4 val;
float2 dx = ddx(texCoord);
float2 dy = ddy(texCoord);
do
{
      val = tex2D(s, texCoord, dx, dy);
      texCoord += 0.1;
} while (i++ < 4 && val.a > 0);
```

```
// Better code
float4 val;
float2 dx = ddx(texCoord);
float2 dy = ddy(texCoord);
val = tex2D(s, texCoord, dx, dy);
if (val.a > 0)
{
      texCoord += 0.1;
      val = tex2D(s, texCoord, dx, dy);
      if (val.a > 0)
      {
            texCoord += 0.1;
            val = tex2D(s, texCoord, dx, dy);
            if (val.a > 0)
            {
                  texCoord += 0.1;
                  val = tex2D(s, texCoord, dx, dy);
            }
      }
}
```

4.6.4. Using Dynamic Flow Control for Early-Out

While dynamic flow control can be seen as a menace to performance, it should also be looked at as a great optimization opportunity. As it was mentioned before, one of the significant uses for dynamic flow control is skipping unnecessary calculations or exiting early out of the shader. There are too many examples to be listed here that can be optimized by dynamic flow control: skipping lighting computations in the shadow, optimizing shadow map filters and many others. Whenever you multiply results of a fairly long instruction chain by a zero value or whenever the contribution of your texture fetches is zero, you should see it as an opportunity to optimize a

shader with dynamic flow control. Always try experimenting with dynamic flow control optimizations whenever you find an opportunity.

It should be noted though that while the X1x00 has high performance with dynamic branching, and shader based branching is very convenient, it can in some cases still be faster to use alternative approaches such as utilizing early stencil rejection, depending on the load on the vertex shader, number of draw calls, the percentage of pixels that can be rejected, and other factors.

4.6.5. Screen Gradients and Dynamic Flow Control

The partial derivatives or screen gradients describe a change in values from pixel to pixel on the screen. There are many uses for the gradients in the shaders such as implementing shader antialiasing, constructing custom anisotropic filters, and computing texture LOD. PS 3.0 includes special instructions to compute gradients; however, some caution has to be exercised with respect to dynamic flow control since gradients might be undefined inside flow control statements. The gradients are computed on a per quad basis by computing change in values across the quad. If pixels within the quad take on different code paths, the values used for gradient computation might not be available for all quad pixels, which might result in incorrect results. In assembly pixel shaders only the gradients of the texture coordinates and other interpolated values can be computed inside flow control statements because only these values are guaranteed to be available and correct for all pixels anywhere in the shader. If gradients have to be computed for shader-derived values, it has to be done outside of the flow control statement. Shader developers also have to make sure that inputs to gradient instructions have been initialized for all execution paths of the shader that could lead to gradient calculation.

The HLSL compiler also performs checks on gradient computations and does not permit these instructions to be placed in flow control statements if the gradients are computed from shader-derived values. Whenever possible, the HLSL compiler will try to move gradient instructions outside of the flow control. Failure to do so will result in shaders compiled without actual flow control instructions.

4.6.6. Texture Fetches Inside of Dynamic Flow Control

Texture fetches rely on the texture coordinate gradients to compute the appropriate mip-level and degree of anisotropy. As we have noted above, the gradients for shader-derived values cannot be computed inside of dynamic flow control. This means mip-level calculation of texture fetches inside of flow control statements cannot be based on shader-computed texture coordinates. To solve this problem Shader Model 3.0 includes special texture sampling instructions that accept user-supplied texture LOD or gradients. Since these instructions do not rely on automatically computed gradients, their use inside of flow control statements is permitted. The pixel shader compiler performs shader validation to enforce proper use of

texture fetches within flow control. In assembly only the fetches with interpolated texture coordinates are permitted inside of flow control.

The HLSL compiler enforces rules for texture fetches inside of flow control, just like it does for gradient calculations. Failure to explicitly specify gradients or LOD with texture fetches based on computed texture coordinates inside of flow control will either disable the generation of flow control instructions or move significant chunks of code, including potentially expensive texture fetches, outside the flow control, resulting in much smaller performance gains than what otherwise would be possible.

The following code fragment shows examples of invalid and correct use of texturing and gradient functions in the shader.

```
// BAD: no flow control instructions generated
float diffuse = dot(normal, lightVec);
if (diffuse > 0)
{
      // PROBLEM: Use of texture fetch inside of flow control
      // with shader derived texture coordinates
      float4 base = tex2D(Base, texCoord + 0.5);
      finalColor = lightColor * diffuse * base + ambient;
}
```

```
// GOOD: flow control instructions generated
float diffuse = dot(normal, lightVec);
if (diffuse > 0)
{
      // GOOD: Texture fetch with interpolated coordinates
      // (assuming texCoord comes from interpolator)
      float4 base = tex2D(Base, texCoord);
      finalColor = lightColor * diffuse * base + ambient;
}
```

```
// GOOD: flow control instructions generated
float diffuse = dot(normal, lightVec);
// GOOD: Derivatives do not depend on values computed inside of flow control
float2 dx = ddx(texCoord);
float2 dy = ddy(texCoord);
if (diffuse > 0)
{
      // GOOD: Texture fetch with user gradients is permitted
      float4 base = tex2D(Base, texCoord + 0.5, dx, dy);
      finalColor = lightColor * diffuse * base + ambient;
}
```

4.6.7. Predication

By definition, predication is a flow control technique used in vector processors where all possible code branches are executed in parallel before the branch condition is proved. Shader Model 3.0 includes a form of predication that, without invoking special flow control constructs, can conditionally control instruction execution per channel. A special predication register containing 4 boolean values (one per channel) is used for execution control. The specification is quite flexible in how flow control can be implemented and it is allowed to substitute predication with flow control instructions and vice versa. The shader compiler will pick the most appropriate method for the internal flow control implementation based on the shader heuristics and intimate knowledge of the hardware, so there is no need to explicitly use predication in the shaders.

4.7. Pixel Shader Constants

Quite often shader developers use some common constants like 2.0, 0.5, -1.0 and so on throughout the shader. The most widespread uses of such constants are scale, scale and bias operations, and many others. Some developers prefer to set these constants in the application instead of using constant literals in the shader code, which is generally a bad practice. By embedding constants in the shader, not only will you produce more readable code, but the HLSL compiler as well as the driver shader compiler will have a better chance of optimizing shader code with respect to the constants.

The following example highlights the proper use of literal constants.

```
// Sub-optimal code
float4 userConsts; // i.e. set by application to (2.0, 1.0, 0.5, 4.0)
Normal = Normal * userConsts.x - userConsts.y;
Result /= userConsts.w;
```

```
// Better code
Normal = 2.0 * Normal – 1.0;
Result /= 4.0;
```

This optimization applies not only to floating point constants used for computations, but also to the constants used for loop counts. If literal shader constants are used for loops, the compiler might be able to unroll the loops, resulting in more optimal code.

5. Optimal HLSL Use

The HLSL compiler, whether we are talking about the standalone tool or shader compilation functions embedded in the D3DX library, exposes a number of compiler options that allow developers to tweak shader compilation. When compiling HLSL shaders for the Radeon X1x00 family of hardware you should adhere to the following guidelines to get the best possible performance.

- Vertex shaders compiled with the VS 3.0 model should be preferably compiled with the option to *avoid* flow control statements. Use the **`D3DXSHADER_AVOID_FLOW_CONTROL`** compiler flag with D3DX compiler functions or the **`/Gfa`** command line option with the fxc.exe compiler.
- Pixel shaders compiled with the PS 3.0 model should be compiled with the option to *prefer* flow control statements, since flow control in pixel shaders is one of the strong points of the Radeon X1x00 architecture. Use the **`D3DXSHADER_PREFER_FLOW_CONTROL`** compiler flag with D3DX compiler functions or the **`/Gfp`** command line option with the fxc.exe compiler.
- Another option that is worth investigating is skipping shader optimization altogether. Generally you would want to keep this option enabled to make sure the HLSL compiler produces the smallest code possible. This is especially important for cases where unoptimized shader code would exceed the number of available instructions. However, there are cases when disabling shader optimizations would lead to slightly higher performance as the driver shader compiler would be able to pick up more optimization opportunities.

Experimenting with various compiler options is the best strategy to produce the best performing shaders.

6. FP16 Render Targets

High-precision floating point render targets made their first appearance a while ago in the first generation of ATI DirectX® 9 hardware. One major functionality missing from the previous implementations was alpha blending. The Radeon X1x00 hardware adds this missing piece of the puzzle, which greatly increases utility of FP16 surfaces. Alpha blending is supported on the most frequently used 4-channel FP16 format, while 1- and 2-channel formats are supported without blending. This is generally not a problem since such 1- and 2-channel formats are usually employed for storing values other than color, for which blending is not applicable anyway.

6.1. Fog and FP16 Render Targets

Fixed function fog is one of the antiquated rendering methods that somehow have stayed with us for a very long time. Shader Model 3.0 marks a new “fogging era”, as it is not compatible with fixed function fog and developers are now expected to implement fog as a part of the shader whenever they use PS 3.0. It is possible, however, to use fixed function fog with lower shader models. Because Radeon X1x00 hardware primarily targets FP16 rendering and Shader Model 3.0, fixed-function fog with FP16 surfaces is not a first class citizen and developers are encouraged to implement their own fog in pixel shaders to obtain the best performance when rendering to FP16 surfaces (even with earlier shader models). Pixel shaders 2.0 and 2.x are more than capable of handling fog computations in the shader.

The 8-bit and 10-bit per channel surfaces still treat fixed function fog as a first class citizen to ensure the highest rendering performance in legacy applications.

6.2. FP16 Render Targets and MSAA

FP16 HDR rendering is quite often tied to very high visual fidelity and photorealism. Up until now the major obstacle in achieving the highest possible quality with HDR rendering and other post-processing methods that rely on FP16 surfaces was the lack of multisampling. Even the most beautiful HDR scene can look wrong and the perception of reality could be completely destroyed by something seemingly as small as jagged lines. The Radeon X1x00 features a solution that brings HDR rendering to the next level – multisampling support with FP16 surfaces. When creating FP16 surfaces to be used for scene rendering, always check if multisampling is available and make the best use of it whenever possible. The new graphics cards sporting 512MB of video memory are the best candidates for multisampled FP16 HDR implementation, as conventional applications rarely require this much video memory for normal rendering.

Using multisampling with FP16 render targets not only requires more memory, but also significantly increases memory bandwidth. This is especially noticeable when alpha blending is enabled. If alpha blended geometry is rendered with a fairly simple pixel shader it makes sense, whenever possible, to use alpha testing or TEXKILL to reduce the number of blended pixels. For example, when using additive blending, “killing” black pixels could provide a substantial performance boost. This optimization works well only with fairly small pixel shaders because alpha test and TEXKILL disables top-of-the-pipe Z rejection. Large and expensive pixel shaders might become a greater bottleneck than the saved memory bandwidth if top-of-the-pipe Z rejection is disabled. In most cases this is not a problem since nearly all alpha blended geometry is rendered with a very simple shader (e.g. smoke particles).

7. RGBA1010102 Render Targets

10-bit formats are not anything new, and they are widely used to provide higher quality than standard 8-bit textures and render targets. The Radeon X1x00 includes some improvements

for 10-bit render targets – now, on top of the filtering, these renderable surfaces also support alpha blending and multisampling. With the addition of these features the 10-bit surfaces have become first-class citizens, and they may be used in many scenarios including a high-fidelity replacement for standard 8-bit surfaces. These surfaces could also be useful for HDR whenever the dynamic range is not too large. This is representative of many games with fairly dark indoor environments and strategy games where the sun – the highest intensity source – is not visible. Now HDR effects are more of a reality on mainstream and value market segment hardware since 10-bit surfaces are much less expensive than FP16.

Another new feature of the 10-bit surface format support is the fact that this format is now also *displayable* on the Radeon X1x00 family of hardware. An RGBA1010102 surface can now be used for both back and front buffers (fullscreen only), allowing a better rendition of colors onto the screen. A good quality CRT monitor, or an LCD supporting 10-bits precision are required to accurately represent the gain in image quality compared to a standard RGBA8888 format, but even older and cheaper 8-bit LCD monitors can benefit from 10-bit displayable surfaces since display engine in Radeon X1x00 video cards automatically supports advanced dithering when working with less than 10-bit LCDs.

8. Floating Point Rules

The Radeon X1x00 family marks a real breakthrough with full 32-bit IEEE floating point support throughout the shader pipeline and 16-bit floating point in the raster backend. The use of floating point calculations opens new possibilities for very flexible techniques, but it creates an opportunity for making mistakes at the same time. Floating point formats have a number of special rules to deal with numbers that are outside of the supported range and results that are not representable. This section explains some important facts about the floating point implementation on Radeon X1x00 cards. Failure to recognize the importance of and to honor these rules might lead to visual artifacts that could be hard to debug. As an example, during a post-processing pass an invalid value of even a single pixel might propagate to large regions of the screen and cause white or black areas.

8.1. Floating Point in Vertex Shaders

The floating point computations in vertex shaders on the Radeon X1x00 cards are performed on standard 32-bit IEEE numbers with most of the IEEE computation rules honored. However, there are some minor deviations from the IEEE standard. Computations are performed without intermediate rounding, and fused operations might have slightly different results and precision from simple calculations. As with all shader-based vertex processors, when computing vertex position in multi-pass rendering, always use the same sequence of instructions to guarantee exactly the same results. Failure to do so might result in Z-fighting.

8.2. Floating Point in Pixel Shaders

The floating point implementation in pixel shader engine is very close to the IEEE 32-bit floating point standard. In fact, it is much closer than required by Shader Model 3.0 specification. However, there are several very small differences that shader developers might need to be aware of. The biggest divergence from the IEEE rules is that denorms are flushed to appropriately signed zero. For instance, all comparisons with denorms will be equivalent to comparisons with zero. Another small deviation from the IEEE standard is the inconsistency of rounding modes for various operations. Regardless, operations are generally accurate within 1 ULP (Unit of Least Precision). A lot of operations maintain a higher internal precision than IEEE requires ensuring that the results of the fused operations are acceptable. With dot products, nevertheless, there is a possibility of losing some precision if added values differ greatly in magnitude.

8.3. Floating Point in Texture Unit

The texture unit accepts input texture coordinates in floating point format and the hardware implements several special cases of handling special values. Just like in pixel shaders, the denorms used for texture fetching will be flushed to appropriately signed zero. NaN values do not make sense as texture coordinates and are automatically converted to +Inf. Avoid generating very large values to be used for texture lookups as these values might generate infinite values during texture projection and cubemap lookup processing.

8.4. Floating Point in Alpha Blender

The 16-bit floating point implementation in the alpha blender generally follows the FP16 rules set out in the DirectX® 9 specification. The FP16 format contains 1 sign bit, 5 bits of biased exponent (bias of 15.0) and 10 bits of fraction with the additional hidden bit. The implementation does not support special cases such as +NaN/-NaN and +Inf/-Inf, however it does support denorms. Because the mentioned specials are not supported, the extra exponent value can be used to represent values up to 131,008.0, which is twice the required range. The alpha blender will also not create negative zeros and will convert them to positive zeros. Because alpha blending is a fused operation, the internal precision of the calculation is slightly higher than required for an FP16 implementation to insure the correct blending results.

9. Summary of Texture Formats

The following tables summarize support of various surface formats and texturing capabilities on Radeon X1x00 hardware.

Integer texture formats

Format	Renderable	Blend	Filter	Auto Mip	sRGB Read	sRGB Write
A8R8G8B8	Yes	Yes	Yes	Yes	Yes	Yes
X8R8G8B8	Yes	Yes	Yes	Yes	Yes	Yes
R5G6B5	Yes	Yes	Yes	Yes	No	No
X1R5G5B5	Yes	Yes	Yes	Yes	No	No
A1R5G5B5	Yes	Yes	Yes	Yes	No	No
A4R4G4B4	Yes	Yes	Yes	Yes	Yes	Yes
A8	No	No	Yes	No	No	No
A2B10G10R10	Yes	No	Yes	Yes	No	No
G16R16	Yes	No	Yes	Yes	No	No
A2R10G10B10	Yes	Yes	Yes	Yes	No	No
A16B16G16R16	Yes	No	Yes	Yes	No	No
L8	No	No	Yes	No	Yes	No
A8L8	No	No	Yes	No	Yes	No
V8U8	No	No	Yes	No	No	No
L6V5U5	No	No	Yes	No	No	No
X8L8V8U8	No	No	Yes	No	No	No
Q8W8V8U8	No	No	Yes	No	No	No
V16U16	No	No	Yes	No	No	No
A2W10V10U10	No	No	Yes	No	No	No
L16	No	No	Yes	No	No	No
Q16W16V16U16	No	No	Yes	No	No	No

Floating texture point formats

Format	Renderable	Blend	Filter	Auto Mip	sRGB Read	sRGB Write
R16F	Yes	No	No	Yes	No	No
G16R16F	Yes	No	No	Yes	No	No
A16B16G16R16F	Yes	Yes	No	Yes	No	No
R32F	Yes	No	No	Yes	No	No
G32R32F	Yes	No	No	Yes	No	No
A32B32G32R32F	Yes	No	No	Yes	No	No

Compressed texture formats

Format	Renderable	Blend	Filter	Auto Mip	sRGB Read	sRGB Write
DXT1	No	No	Yes	No	Yes	No
DXT2	No	No	Yes	No	Yes	No
DXT3	No	No	Yes	No	Yes	No
DXT4	No	No	Yes	No	Yes	No
DXT5	No	No	Yes	No	Yes	No
ATI1	No	No	Yes	No	Yes	No
ATI2	No	No	Yes	No	Yes	No

Depth texture formats

Format	Renderable	Blend	Filter	Auto Mip	sRGB Read	sRGB Write
DF16	No	No	Yes	No	No	No
DF24 *	No	No	Yes	No	No	No

Note: * - available only on Radeon X1900, X1600 and X1300

Video texture formats

Format	Renderable	Blend	Filter	Auto Mip	sRGB Read	sRGB Write
UYVY	No	No	Yes	No	No	No
YUY2	No	No	Yes	No	No	No