Advanced DX9 Capabilities for ATI Radeon Cards

AMD Graphics Products Group
devrel@amd.com

Introduction

This white paper gives implementation details on some of the advanced capabilities exposed by Radeon graphic drivers under the DirectX 9 API. Those capabilities target a feature level not directly exposed by DirectX 9, and therefore require an additional programming interface to take advantage of them. The programming interface to access those advanced features depends on the feature being targeted, although they often share common principles like special FourCC codes or “magic” values to set in specific renderstates etc.
The detection of those advanced features usually relies on the use of the `CheckDeviceFormat()` DirectX 9 API, whereby a specific FourCC code can be checked against other rendering parameters to determine support. It is strongly advised to use this detection mechanism over graphic vendor or device ID-based detection as some of those features are shared across different GPU brands. On top of that detecting with `CheckDeviceFormat()` ensures proper operation of the game when running on future graphic hardware that was not available at development time.

Table of Contents

Depth Texture Format: INTZ

Description: An additional texture format called INTZ is exposed that allows a 24-bit depth buffer previously used for rendering to be bound as a texture. Any fetches from this texture will return the depth values stored. Shadow mapping applications relying on Percentage-Closer Filtering should prefer the use of DX9 Depth Stencil Textures instead. INTZ additionally exposes an 8-bit stencil buffer when used for depth buffering, allowing stencil operation to be carried out when an INTZ surface is bound as the active depth stencil buffer. However the contents of the stencil buffer will *not* be available as texture data when binding the surface as a texture.
Note that an INTZ depth buffer may be used as a texture concurrently to the same INTZ surface being used for depth buffering, *as long as depth writes are disabled*.
Supported hardware: ATI Radeon 4000 series and above.

Implementation details

A FourCC depth texture format is exposed:

```
#define FOURCC_INTZ ((D3DFORMAT)(MAKEFOURCC('I','N','T','Z')))
```

- To check support for this feature:

```
// Determine if INTZ is supported
HRESULT hr;
hr =pd3d->CheckDeviceFormat(AdapterOrdinal, DeviceType, AdapterFormat,
                            D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_TEXTURE,
                            FOURCC_INTZ);
BOOL bINTZDepthStencilTexturesSupported = (hr == D3D_OK);
```

- To create an INTZ depth stencil texture:

```
// Create an INTZ depth stencil texture
IDirect3DTexture9 *pINTZDST;
pd3dDevice->CreateTexture(dwWidth, dwHeight, 1,
                          D3DUSAGE_DEPTHSTENCIL, FOURCC_INTZ,
                          D3DPOOL_DEFAULT, &pINTZDST,
                          NULL);
```

- To bind the depth stencil texture as an active depth buffer:

```
// Retrieve depth buffer surface from texture interface
IDirect3DSurface9 *pINTZDSTSurface;
pINTZDST->GetSurfaceLevel(0, &pINTZDSTSurface);

// Bind depth buffer
pd3dDevice->SetDepthStencilSurface(pINTZDSTSurface);
```

Note that calling `GetSurfaceLevel()` increases the reference count of `pINTZDST` so you will need to `Release()` it when no longer needed.

- To bind an INTZ depth buffer texture as a texture:

```
// Bind depth buffer texture
pd3dDevice->SetTexture(0, pINTZDST);
```

Render Target Format: NULL

Description: A render target format called NULL is exposed. Render targets created with this format will not have any memory allocated internally, but will satisfy validation requirements imposed by the DX9 runtime. This functionality is useful when performing rendering onto a depth buffer without wishing to update an actual color render target since the DX9 runtime enforces the use of a valid color render target for all rendering operations. Therefore common depth-only rendering applications like shadow mapping or depth pre-pass can benefit from using this "dummy" format to save memory that would otherwise be required to bind a valid color render target for this operation.
Supported hardware: ATI Radeon 4000 series and above.

Implementation details

A FourCC color render target format is exposed:

```
#define FOURCC_NULL ((D3DFORMAT)(MAKEFOURCC('N','U','L','L')))
```

- To check support for this feature:

```
// Determine if NULL is supported
HRESULT hr;
hr = pd3d->CheckDeviceFormat(AdapterOrdinal, DeviceType, AdapterFormat,
                             D3DUSAGE_RENDERTARGET, D3DRTYPE_SURFACE,
                             FOURCC_NULL);
BOOL bNULLRenderTargetSupported = (hr == D3D_OK);
```

- To create a NULL render target surface:

```
// Create a NULL render target with 4x multisampling
IDirect3DSurface9* pDummyRenderTarget;
pd3dDevice->CreateRenderTarget(dwWidth, dwHeight, FOURCC_NULL,
                               D3DMULTISAMPLE_4_SAMPLES, 0,
                               FALSE, &pDummyRenderTarget, NULL);
```

- To bind a NULL render target surface:

```
// Bind a NULL render target at slot 0 so that we don't have to bind
// a real color buffer; this allows memory savings
pd3dDevice->SetRenderTarget(0, pDummyRenderTarget);
```

Depth Stencil Textures using DirectX 9 depth buffer formats

Description: Allows the use of DirectX 9 depth buffers as depth stencil textures on top of normal depth buffering. Percentage-Closer Filtering will be applied on all texture fetches from a depth stencil texture. This is the preferred method for DX9 shadow mapping applications relying on Percentage-Closer Filtering.
Supported Hardware: ATI Radeon 2000, 3000, 4000 series and above.

Implementation details

A DirectX 9 depth buffer format can be specified in the `CreateTexture()` API, allowing a depth buffer previously rendered on to be bound as a texture for shadow mapping applications. Supported depth buffer formats are:

```
D3DFMT_D16
D3DFMT_D24X8
D3DFMT_D24S8
```

For performance it is recommended to use the `D3DFMT_D16` format. Otherwise the preferred format is `D3DFMT_D24X8`. Note that `D3DFMT_D24S8` only returns depth values when bound as a texture; no stencil values are available for reading when binding this format as a texture input.

- To check support for this feature (example below using the `D3DFMT_D16` format):

```
// Determine if DX9 depth stencil textures are supported
HRESULT hr;
hr =pd3d->CheckDeviceFormat(AdapterOrdinal, DeviceType, AdapterFormat,
                            D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_TEXTURE,
                            D3DFMT_D16);
BOOL bDX9DepthStencilTexturesSupported = (hr == D3D_OK);
```

- To create a depth stencil texture:

```
// Create a 16-bit depth stencil texture
IDirect3DTexture9 * pDepthStencilTexture;
pd3dDevice->CreateTexture(dwWidth, dwHeight, 1,
                          D3DUSAGE_DEPTHSTENCIL, D3DFMT_D16,
                          D3DPOOL_DEFAULT, &pDepthStencilTexture,
                          NULL);
```

- To bind the depth stencil texture as an active depth buffer:

```
// Retrieve depth buffer surface from texture interface
IDirect3DSurface9 *pDepthStencilTextureSurface;
pDepthStencilTexture->GetSurfaceLevel(0, & pDepthStencilTextureSurface);

// Bind depth buffer
pd3dDevice->SetDepthStencilSurface(pDepthStencilTextureSurface);
```

Note that calling `GetSurfaceLevel()` increases the reference count of `pDepthStencilTexture` so you will need to `Release()` it when no longer needed.

- To bind a depth buffer texture as a texture for PCF filtering

```
// Bind depth buffer texture for Percentage-Closer Filtering shadows
pd3dDevice->SetTexture(0, pDepthStencilTexture);
```

Multisampled Depth Buffer Resolve: RESZ

Description: Allows the resolve of a multisampled depth buffer into a depth stencil texture. This capability is often useful for post-process effects and other rendering operations requiring access to depth. The resolve operation copies the depth value *from the first sample only* into the resolved depth stencil texture.
The RESZ interface also works with non-multisampled depth buffers to preserve orthogonality between multisampled and non-multisampled rendering. In this case the resolve is effectively a simply copy from the source depth buffer to the specified destination depth texture format will be performed. The RESZ interface is a good alternative to use for cards that don't support INTZ.
Supported hardware: ATI Radeon 2000, 3000, 4000 series and above.

Implementation details

A FourCC value is exposed to check support for this feature:

```
#define FOURCC_RESZ ((D3DFORMAT)(MAKEFOURCC('R','E','S','Z')))
```

A magic value is exposed to trigger the resolve:

```
#define RESZ_CODE 0x7fa05000
```

The resolve operation will need a valid recipient to receive the contents of the resolved multisampled depth buffer. The destination non-multisampled surface can be any of the following Depth Stencil Texture formats:

```
D3DFMT_D24S8
D3DFMT_D24X8,
ATI_FOURCC_DF16
ATI_FOURCC_DF24
FOURCC_INTZ
```

A multisampled depth buffer is created and used for rendering as normal. Once a resolve is needed the destination Depth Stencil Texture must be bound to texture sampler 0. A dummy draw call is required after setting the destination Depth Stencil Texture to sampler 0 to ensure the texture is set before the next renderstate instruction (as the DirectX9 runtime may send commands out of order). After the dummy draw call the `D3DRS_POINTSIZE` renderstate must be set to the magic value of `RESZ_CODE` – this will trigger the resolve of the current multisampled depth buffer into texture sampler 0 (i.e. the destination Depth Stencil Texture). From there on the Depth Stencil Texture will contain the resolved contents of the multisampled depth buffer and can be used for further rendering operations.

- To check support for this feature:

```
// Determine if RESZ is supported
HRESULT hr;
hr = pd3d->CheckDeviceFormat(AdapterOrdinal, DeviceType, AdapterFormat,
                             D3DUSAGE_RENDERTARGET, D3DRTYPE_SURFACE,
                             FOURCC_RESZ);
BOOL bRESZSupported = (hr == D3D_OK);
```

- To create the destination Depth Stencil Texture that will receive the contents of the resolve operation from the current multisampled depth buffer (using the `FOURCC_INTZ` format in this example):

```
// Create an INTZ depth stencil texture that will receive the resolve
IDirect3DTexture9 *pINTZDST;
pd3dDevice->CreateTexture(dwWidth, dwHeight, 1,
                          D3DUSAGE_DEPTHSTENCIL, FOURCC_INTZ,
```

```
                                D3DPOOL_DEFAULT, &pINTZDST,
                                NULL);
```

- To bind the depth stencil texture as texture sampler 0:

```
// Bind depth stencil texture to texture sampler 0
pd3dDevice->SetTexture(0, pINTZDST);
```

- To render a dummy draw call. Note that this can even be a real draw call; all that matters is that texture sampler 0 is bound to the Depth Stencil Texture that will receive the contents of the resolve operation when this call is made:

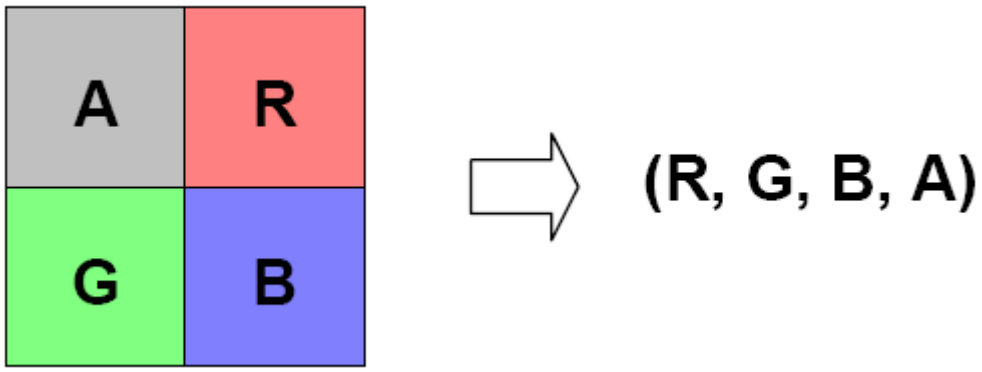
```
// Perform a dummy draw call to ensure texture sampler 0 is set before the
// resolve is triggered
// Vertex declaration and shaders may need to me adjusted to ensure no
debug
// error message is produced
D3DXVECTOR3 vDummyPoint(0.0f, 0.0f, 0.0f);
pd3dDevice->SetRenderState(D3DRS_ZENABLE, FALSE);
pd3dDevice->SetRenderState(D3DRS_ZWRITEENABLE, FALSE);
pd3dDevice->SetRenderState(D3DRS_COLORWRITEENABLE, 0);
pd3dDevice->DrawPrimitiveUP(D3DPT_POINTLIST, 1, vDummyPoint,
                            sizeof(D3DXVECTOR3));
pd3dDevice->SetRenderState(D3DRS_ZWRITEENABLE, TRUE);
pd3dDevice->SetRenderState(D3DRS_ZENABLE, TRUE);
pd3dDevice->SetRenderState(D3DRS_COLORWRITEENABLE, 0x0F);
```

- To trigger the resolve of the current multisampled depth buffer into the non-multisampled Depth Stencil Texture bound at texture sampler 0:

```
// Trigger the depth buffer resolve; after this call texture sampler 0
// will contain the contents of the resolve operation
pd3dDevice->SetRenderState(D3DRS_POINTSIZE, RESZ_CODE);
```

Fetch-4

Description: this feature allows the fetching of four unfiltered neighboring texels (2x2 texel block) in a single texture instruction. Four individual samples of a single-channel texture are swizzled into RGBA channels when they are fetched from the texture. The swizzling of the 2x2 texel block into four channels is illustrated by the following diagram:



Note that this swizzle is different from the swizzle pattern used with the DirectX10.1 and DirectX11 `Gather()` HLSL shader instruction. Implementations wishing to use Fetch-4 for Percentage-Closer Filtering should prefer the use of DX9 Depth Stencil Textures instead.
Supported hardware: ATI Radeon X13x0, X16x0, X19x0, ATI Radeon 2000, 3000, 4000 series and above. Note that all Radeon cards supporting the DF24 depth texture FourCC format support Fetch4.

Implementation details

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.

- To check support for this feature:

```
// Determine Fetch-4 supported: all ATI Radeon cards supporting Fetch-4
also
// support the DF24 depth texture FourCC format so use this for detection.
#define ATI_FOURCC_DF24 ((D3DFORMAT)(MAKEFOURCC('D','F','2','4')))
HRESULT hr;
hr = pd3d->CheckDeviceFormat(AdapterOrdinal, DeviceType, AdapterFormat,
                             D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_TEXTURE,
                             ATI_FOURCC_DF24);
BOOL bFetch4Supported = (hr == D3D_OK);
```

- To enable Fetch-4 on a texture sampler (sampler 0 in this example):

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

// Enable Fetch-4 on sampler 0 by overloading the MIPMAPLODBIAS render
state
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);
```

- To disable Fetch4 on a texture sampler (sampler 0 in this example):

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

Depth Texture Formats: DF16 and DF24

Description: Additional texture formats DF16 and DF24 are exposed that allow a depth buffer previously used for rendering to be bound as a texture. Any fetches from those textures will return the depth values stored. Shadow mapping applications relying on Percentage-Closer Filtering should prefer the use of DX9 Depth Stencil Textures instead.

Supported hardware:

DF16: ATI Radeon 9000, X1000, 2000, 3000, 4000 series and above.

DF24: ATI Radeon X13x0, X16x0, X19x0, ATI Radeon 2000, 3000, 4000 series and above.

Implementation details

Two ATI FourCC depth texture formats are exposed:

```
#define ATI_FOURCC_DF16 ((D3DFORMAT)(MAKEFOURCC('D','F','1','6')))
#define ATI_FOURCC_DF24 ((D3DFORMAT)(MAKEFOURCC('D','F','2','4')))
```

`ATI_FOURCC_DF16` is a 16-bit per pixel depth stencil texture format.
`ATI_FOURCC_DF24` is a 24-bit per pixel depth stencil texture format.
For performance it is advised to prefer the `ATI_FOURCC_DF16` format over `ATI_FOURCC_DF24`.

- To check support for this feature (example below using the `ATI_FOURCC_DF16` format):

```
// Determine if ATI DF16 is supported
HRESULT hr;
hr = pd3d->CheckDeviceFormat(AdapterOrdinal, DeviceType, AdapterFormat,
                             D3DUSAGE_DEPTHSTENCIL, D3DRTYPE_TEXTURE,
                             ATI_FOURCC_DF16);
BOOL bATIDF16DepthStencilTexturesSupported = (hr == D3D_OK);
```

- To create a DF16 depth stencil texture:

```
// Create a DF16 depth stencil texture
IDirect3DTexture9 * pDepthStencilTexture;
pd3dDevice->CreateTexture(dwWidth, dwHeight, 1,
                          D3DUSAGE_DEPTHSTENCIL, ATI_FOURCC_DF16,
                          D3DPOOL_DEFAULT, &pDF16DST,
                          NULL);
```

- To bind the depth stencil texture as an active depth buffer:

```
// Retrieve depth buffer surface from texture interface
IDirect3DSurface9 *pDF16DSTSurface;
pDF16DST->GetSurfaceLevel(0, &pDF16DSTSurface);

// Bind depth buffer
pd3dDevice->SetDepthStencilSurface(pDF16DSTSurface);
```

Note that calling `GetSurfaceLevel()` increases the reference count of `pDF16DST` so you will need to `Release()` it when no longer needed.

- To bind a DF16 depth buffer texture as a texture:

```
// Bind depth buffer texture
pd3dDevice->SetTexture(0, pDF16DST);
```

Alpha to Coverage

Description: Alpha to coverage is a graphic feature that converts an incoming alpha value into a multisample coverage mask. Typical applications of alpha to coverage include the smoothing of edges in transparent textures such as foliage and fences in multisampling anti-aliasing scenarios. For an example of alpha to coverage in action please see the AlphaToCoverage DirectX 9 sample in the ATI SDK.
Supported hardware: ATI Radeon X1000, 2000, 3000, 4000 series and above.

Implementation details

Two special renderstate values are exposed via the FourCC interface:

```
#define ALPHA_TO_COVERAGE_ENABLE  (MAKEFOURCC('A','2','M','1'))
#define ALPHA_TO_COVERAGE_DISABLE (MAKEFOURCC('A','2','M','0'))
```

Alpha to coverage is globally enabled or disabled by overloading the `D3DRS_POINTSIZE` renderstate with those special values.

- To check support for this feature:

All ATI cards supporting the DirectX9 feature set expose alpha to coverage.

- To enable alpha to coverage:

```
// Enable alpha to coverage
pd3dDevice->SetRenderstate(D3DRS_POINTSIZE, ALPHA_TO_COVERAGE_ENABLE);
```

- To disable alpha to coverage:

```
// Disable alpha to coverage
pd3dDevice->SetRenderstate(D3DRS_POINTSIZE, ALPHA_TO_COVERAGE_DISABLE);
```

Texture Formats: ATI2N and ATI1N

Description: Two additional texture formats ATI2N and ATI1N are exposed, allowing compression of 2-channels (ATI2) or single-channel (ATI1) textures. The ATI2N format is also known as 3DC. ATI2N is functionally equivalent to the BC5 format while ATI1N is functionally equivalent to the BC4 format. Both BC4 and BC5 format were introduced in the DirectX10 API.
Graphic tools such as AMD Compressonator and the ATI Compression library support those formats.
For more info on ATI2N (3DC) please read the *ATI3Dc_Developer_Brief.pdf* document available in the ATI SDK.
Supported hardware:
ATI2N: ATI Radeon 9000, X1000, 2000, 3000, 4000 series and above.
ATI1N: X1000, 2000, 3000, 4000 series and above.

Implementation details

Two FourCC formats are exposed:

```
#define FOURCC_ATI1N ((D3DFORMAT)MAKEFOURCC('A', 'T', 'I', '1'))
#define FOURCC_ATI2N ((D3DFORMAT)MAKEFOURCC('A', 'T', 'I', '2'))
```

- To check support for ATI1N:

```
// Check if ATI1N is supported
HRESULT hr;
hr = pd3d->CheckDeviceFormat(AdapterOrdinal, DeviceType, AdapterFormat,
                             0, D3DRTYPE_TEXTURE, FOURCC_ATI1N);
BOOL bATI1NSupported = (hr == D3D_OK);
```

- To check support for ATI2N:

```
// Check if ATI2N is supported
HRESULT hr;
hr = pd3d->CheckDeviceFormat(AdapterOrdinal, DeviceType, AdapterFormat,
                             0, D3DRTYPE_TEXTURE, FOURCC_ATI2N);
BOOL bATI2NSupported = (hr == D3D_OK);
```

- To create an ATI1N texture:

```
// Create an ATI1N texture
IDirect3DTexture9 * pATI1NTexture;
pd3dDevice->CreateTexture(dwWidth, dwHeight, 0, 0, FOURCC_ATI1N,
                          D3DPOOL_DEFAULT, &pATI1NTexture, NULL);
```

- To create an ATI2N texture:

```
// Create an ATI2N texture
IDirect3DTexture9 * pATI2NTexture;
pd3dDevice->CreateTexture(dwWidth, dwHeight, 0, 0, FOURCC_ATI2N,
                          D3DPOOL_DEFAULT, &pATI2NTexture, NULL);
```

Other Advanced Capabilities

Render to Vertex Buffer (R2VB)

For a description and implementation details of the Render to Vertex Buffer interface (R2VB) please read the *R2VB programming.pdf* white paper in the ATI SDK.

Geometry Tessellation

For a description and implementation details of the Geometry Tessellation interface please read the ATI tessellation SDK available from the following link:
http://developer.amd.com/gpu/radeon/Tessellation

Document updates:

09 September 2009: Added details on RESZ interface also being supported for non-multisampled depth buffers.

Advanced Micro Devices
One AMD Place
P.O. Box 3453
Sunnyvale, CA 94088-3453

www.amd.com
http://ati.amd.com/developer