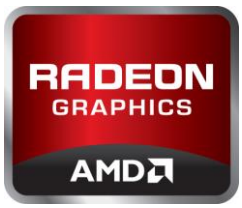

AGS3 Library Documentation

AMD Graphics Developer Relations Team

Revision 11 – 28th January 2016

AGS3 Library Overview

This document provides an overview of the AGS (AMD GPU Services) library, including a presentation of available functionality and related entry points. The AGS library provides software developers with the ability to query AMD GPU software and hardware state information that is not normally available through standard operating systems or graphic APIs. Version 3.1 of the library includes support for querying graphics driver version info, GPU performance, Crossfire (AMD's multi-GPU rendering technology) configuration info, as well as Eyefinity (AMD's multi-display rendering technology) configuration info. AGS also exposes some additional functionality supported in the DirectX11 AMD driver.

This paper only presents AGS library APIs and associated functionality. Additional information on Catalyst drivers, as well as on Crossfire and Eyefinity technologies is available at www.amd.com. Graphics programming recommendations are detailed in the *Harnessing the Performance of CrossfireX* and in the *Gaming under Eyefinity* whitepapers, both available at developer.amd.com.

What's new in AGS3.1 since 3.0

AGS3.1 now returns a lot more information from the GPU in addition to exposing the explicit Crossfire API for DX11. The following changes are new to AGS3.1:

- The initialization function can now return information about the GPU:
    - Whether the GPU is GCN or not.
    - The adapter string and device id.
    - The driver version is now rolled into this structure instead of a separate function call.
    - Performance metrics such as the number of compute units and clock speeds.
- A whole new API to transfer resources on GPUs in Crossfire configuration in DirectX11.
- A method to register your app and engine with the DirectX11 driver.
- The screen rect primitive is now available in DirectX11 if supported.

Using the AGS library

It is recommended to take a look at both the AGS Sample and the EyefinitySample source code. The AGS Sample application demonstrates the code required to initialize AGS and use it to query the GPU and Eyefinity state.

To add AGS support to an existing project, follow these steps:

- Link your project against the correct import library. Choose from either the 32 bit or 64 bit version.
- Copy the AGS dll into the same directory as your game executable.
- Include the amd_ags.h header file from your source code.
- Declare a pointer to an AGSContext and make this available for all subsequent calls to AGS.
- On game initialization, call agsInit() passing in the address of the context. On success, this function will return a valid context pointer.

Initializing the API

The AGS library must be initialised before making any subesquent calls to the API. This can be performed before the device is created. The API is cleaned up using `agsDeInit()`.

AGSReturnCode **agsInit**(AGSContext** context , AGSGPUInfo* info)		
In/Out Param	context	Address of a pointer to a context. This function allocates a context on the heap which is then required for all subsequent API calls.
In/Out Param	info	Optional pointer to a AGSGPUInfo struct which will get filled in for the primary adapter.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	
Notes	This function will fail with AGS_ERROR_LEGACY_DRIVER function will fail in Catalyst versions before 12.20	

AGSReturnCode **agsDeInit** (AGSContext* context)		
In Param	context	A valid pointer to an AGSContext structure.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

AGSGPUInfo struct	
version	The GPU architecture version.
adapterString	The adapter name, eg “AMD Radeon R9 Fury Series”.
deviceId	The device id.
revisionId	The revision id which can be used to differentiate some products that share the same device id.
driverVersion	A string containing the current packaged driver version. eg. “14.502.1014.1001-150526a-184425E”
iNumCUs	The number of compute units. This value is zero for non GCN hardware.
iCoreClock	The core clock speed in MHz when the GPU is running at 100% performance. This value is zero for non GCN hardware.
iMemoryClock	The memory clock speed in MHz when the GPU is running at 100% performance. This value is zero for non GCN hardware.
fTflops	GPU compute power in Teraflops. This value is zero for non GCN hardware.

Querying Crossfire State

The `agsGetCrossfireGPUCount()` function returns the number of AMD GPUs that operate in AMD Crossfire mode.

AGSReturnCode **agsGetCrossfireGPUCount**(AGSContext* context, int* numGPUs)		
In Param	context	A valid pointer to an AGSContext structure.
Out Param	numGPUs	A valid pointer to an int that will store the number of Crossfired GPUs.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

Querying GPU Memory

The `agsGetTotalGPUCount()` function returns the total number of GPUs enumerated by the system. This can include GPUs from other hardware vendors other than AMD. To query the memory local to each GPU, call agsGetGPUMemorySize() which returns the amount in bytes.

AGSReturnCode **agsGetTotalGPUCount**(AGSContext* context, int* numGPUs)		
In Param	context	A valid pointer to an AGSContext structure.
Out Param	numGPUs	A valid pointer to an int that will store the total number of GPUs found on the machine.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

AGSReturnCode **agsGetGPUMemorySize**(AGSContext* context, int gpuIndex, long long* sizeInBytes)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	gpuIndex	The zero-based index of this GPU. Use agsGetTotalGPUCount() to get the number of GPUs in the system.
Out Param	sizeInBytes	A valid pointer to a long long that will store the size in bytes of the memory of this GPU.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

Querying Eyefinity State

Querying Eyefinity configuration state information can be accomplished with the `agsGetEyefinityConfigInfo()` function which returns the following information:

- Whether Eyefinity is enabled or not;
- The SLS (single large surface) grid configuration of displays used (3x1 layout, 3x2 layout, etc);
- The SLS size of the surface that spans the displays;
- Whether bezel compensation is enabled or not;
- The SLS grid coordinate for each display;
- The total rendering area for each display;
- The visible rendering area for each display;
- The preferred display (to properly position UI elements in games for example).

In order to use this function correctly, you must call the function *twice*. Once to query the number of displays from numDisplayInfo, then again to fill in the array of displaysInfo. The usage would be thus:

- Call once with NULL eyefinityInfo and NULL displaysInfo entries. This will return the number of monitors used in the Eyefinity configuration.
- Allocate an array of AGSDisplayInfo structures based on the value of numDisplaysInfo.
- Call a second time, this time supplying all parameters. The function will now fill out the displaysInfo field.

Please refer to the AGSSample program source for correct usage of this function.

AGSReturnCode **agsGetEyefinityConfigInfo**(AGSContext* context, int displayIndex, AGSEyefinityInfo* eyefinityInfo, int* numDisplaysInfo, AGSDisplayInfo* displaysInfo)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	displayIndex	This is an operating system specific display index identifier. The value used should be the index of the display used for rendering operations.
In Param	eyefinityInfo	A pointer to an AGSEyefinityInfo structure that contains system Eyefinity configuration information. This can be NULL if you are only interested in querying the number of displays.
In/Out Param	numDisplaysInfo	A pointer to an int storing the number of displays.
Out Param	displaysInfo	A pointer to the user allocated array of AGSDisplayInfo structures. It is the user's responsibility to free this up after use.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

AGSEyefinityInfo struct	
iSLSActive	Indicates if Eyefinity is active for the operating system display index passed into agsGetEyefinityConfigInfo(). 1 if enabled and 0 if disabled.
iSLSGridWidth	Contains width of the multi-monitor grid that makes up the Eyefinity Single Large Surface. For example, a 3 display wide by 2 high Eyefinity setup will return 3 for this entry.
iSLSGridHeight	Contains height of the multi-monitor grid that makes up the Eyefinity Single Large Surface. For example, a 3 display wide by 2 high Eyefinity setup will return 2 for this entry.
iSLSWidth	Contains width in pixels of the multi-monitor SLS. The value returned is a function of the width of the SLS grid, of the horizontal resolution of each display, and of whether or not bezel compensation is enabled.
iSLSHeight	Contains height in pixels of the multi-monitor SLS. The value returned is a function of the height of the SLS grid, of the vertical resolution of each display, and of whether or not bezel compensation is enabled.
iBezelCompensatedDisplay	Indicates if bezel compensation is used for the current SLS display area. 1 if enabled and 0 if disabled.

AGSDisplayInfo struct	
iGridCoordX	Contains horizontal SLS grid coordinate of the display. The value is zero based with increasing values from left to right of the overall SLS grid. For example, the left-most display of a 3x2 Eyefinity setup will have the value 0, and the right-most will have the value 2.
iGridCoordY	Contains vertical SLS grid coordinate of the display. The value is zero based with increasing values from top to bottom of the overall SLS grid. For example, the top display of a 3x2 Eyefinity setup will have the value 0, and the bottom will have the value 1.
displayRect	Contains the base offset and dimensions in pixels of the SLS rendering area associated with this display. If bezel compensation is enabled, this area will be larger than what the display can natively present to account for bezel area. If bezel compensation is disabled, this area will be equal to what the display can support natively.
displayRectVisible	Contains the base offset and dimensions in pixels of the SLS rendering area associated with this display that is visible to the end user. If bezel compensation is enabled, this area will be equal to what the display can natively, but smaller than the area described in the displayRect entry. If bezel compensation is disabled, this area will be equal to what the display can support natively and equal to the area described in the displayRect entry. ***Developers wishing to place UI, HUD, or other assets on a given display so that it is visible and accessible to end users need to locate them inside of the region defined by this rect.***
iPreferredDisplay	Indicates whether or not this display is the preferred one for rendering of game HUD and UI elements. Only one display out of the whole SLS grid will have this be true if it is the preferred display and 0 otherwise. Developers wishing to place specific UI, HUD, or other game assets on a given display so that it is visible and accessible to end users need to locate them inside of the region defined by this rect. If no display is marked as preferred, then it may be either down to the game to determine where to position the HUD or assume the HUD should cover the entire SLS such as in the case of 2x1 4k resolutions.

Using the DirectX11 Driver Extensions

The AMD DirectX11 driver supports a number of useful extensions that can be accessed via AGS. In order to use these, AGS must already be initialized and agsDriverExtensions_Init must be called. This function returns which extensions are supported by the current hardware and driver configuration.

AGSReturnCode **agsDriverExtensions_Init** (AGSContext* context, ID3D11Device* device, unsigned int* extensionsSupported)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	device	A valid pointer to the DirectX11 device.
Out Param	extensionsSupported	A valid pointer to a bit mask of the extensions supported by the driver.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	
Notes	The AGS needs to be initialized first in order to obtain an AGS context. Direct3D11 will then need to be initialized in order to obtain the ID3D11Device. Only then can this function be called. To check which extensions are supported, AND the AGSDriverExtension enumerated bits against the extensionsSupported value returned from this function call.	

AGSReturnCode **agsDriverExtensions_DeInit** (AGSContext* context)		
In Param	context	A valid pointer to an AGSContext structure.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	
Notes	This needs to be called before the D3D11 device is released.	

Quad List and Screen Rect Driver Extension

The Quad List extension is a convenient way to submit quads without using an index buffer. Note that this still submits two triangles at the driver level. In order to use this function, AGS must already be initialized and agsDriverExtensions_Init must have been called successfully.

The Screen Rect extension, which is only available on GCN hardware, allows the user to pass in three of the four corners of a rectangle. The hardware then uses the bounding box of the vertices to rasterize the rectangle primitive (ie as a rectangle rather than two triangles). Note that this will not return valid interpolated values, only valid SV_Position values.

If either the Quad List or Screen Rect extension are used, then agsDriverExtensions_IASetPrimitiveTopology should be called in place of the native DirectX11 equivalent all the time.

AGSReturnCode **agsDriverExtensions_IASetPrimitiveTopology**(AGSContext* context, D3D_PRIMITIVE_TOPOLOGY topology)		
In Param	context	A valid pointer to an AGSContext structure.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

UAV Overlap Driver Extension

When calling back-to-back draw calls or dispatch calls that write to the same UAV, the AMD DX11 driver will automatically insert a barrier to ensure there are no write after write (WAW) hazards. If the app can guarantee there is no overlap between the writes between these calls, then this extension will remove those barriers allowing the work to run in parallel on the GPU.

Usage would be as follows:

```
// Disable automatic WAW syncs
agsDriverExtensions_BeginUAVOverlap( m_agsContext );

// Submit back-to-back dispatches that write to the same UAV
m_device->Dispatch( ... );  // First half of UAV
m_device->Dispatch( ... );  // Second half of UAV

// Reenable automatic WAW syncs
agsDriverExtensions_EndUAVOverlap( m_agsContext );
```

In order to use this function, AGS must already be initialized and agsDriverExtensions_Init must have been called successfully.

AGSReturnCode **agsDriverExtensions_BeginUAVOverlap** (AGSContext* context)		
In Param	context	A valid pointer to an AGSContext structure.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

AGSReturnCode **agsDriverExtensions_EndUAVOverlap** (AGSContext* context)		
In Param	context	A valid pointer to an AGSContext structure.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

Depth Bounds Driver Extension

The depth bounds test is enables geometry to be clipped outside a specific depth range. In order to use this function, AGS must already be initialized and agsDriverExtensions_Init must have been called successfully.

AGSReturnCode **agsDriverExtensions_SetDepthBounds**(AGSContext* context, bool enabled, float minDepth, float maxDepth)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	enabled	Whether to enable the depth bounds test or not. If disabled, the min and max depth values are ignored.
In Param	minDepth	The min depth value of the clipping region.
In Param	maxDepth	The max depth value of the clipping region.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

Multi Draw Indirect Driver Extension

The multi draw indirect extensions allow multiple sets of DrawInstancedIndirect to be submitted in one API call. The draw calls are issued on the GPU's command processor (CP), potentially saving the significant CPU overheads incurred by submitting the equivalent draw calls on the CPU.

The extension allows the following code:

```
// Submit n batches of DrawIndirect calls
for ( int i = 0; i < n; i++ )
    DrawIndexedInstancedIndirect( buffer, i * sizeof( cmd ) );
```

To be replaced by the following call:

```
// Submit all n batches in one call
agsDriverExtensions_MultiDrawIndexedInstancedIndirect( m_agsContext,
                                                       n, buffer, 0, sizeof( cmd ) );
```

The buffer used for the indirect args must be of the following formats:

```
// Buffer layout for agsDriverExtensions_MultiDrawInstancedIndirect
struct DrawInstancedIndirectArgs
{
    UINT VertexCountPerInstance;
    UINT InstanceCount;
    UINT StartVertexLocation;
    UINT StartInstanceLocation;
}

// Buffer layout for agsDriverExtensions_MultiDrawIndexedInstancedIndirect
struct DrawIndexedInstancedIndirectArgs
{
    UINT IndexCountPerInstance;
    UINT InstanceCount;
    UINT StartIndexLocation;
    UINT BaseVertexLocation;
    UINT StartInstanceLocation;
}
```

In order to use this function, AGS must already be initialized and agsDriverExtensions_Init must have been called successfully.

AGSReturnCode **agsDriverExtensions_MultiDrawInstancedIndirect**(AGSContext* context, unsigned int drawCount, ID3D11Buffer* pBufferForArgs, unsigned int alignedByteOffsetForArgs, unsigned int byteStrideForArgs)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	drawCount	The number of draws to execute from the args buffer.
In Param	pBufferForArgs	A pointer to the buffer containing the list of args.
In Param	alignedByteOffsetForArgs	The DWORD aligned offset into the args buffer.
In Param	byteStrideForArgs	The DWORD aligned stride of the args buffer, byteStrideForArgs >= sizeof(DrawInstancedIndirectArgs).
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

AGSReturnCode **agsDriverExtensions_MultiDrawIndexedInstancedIndirect**(AGSContext* context, unsigned int drawCount, ID3D11Buffer* pBufferForArgs, unsigned int alignedByteOffsetForArgs, unsigned int byteStrideForArgs)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	drawCount	The number of draws to execute from the args buffer.
In Param	pBufferForArgs	A pointer to the buffer containing the list of args.
In Param	alignedByteOffsetForArgs	The DWORD aligned offset into the args buffer.
In Param	byteStrideForArgs	The DWORD aligned stride of the args buffer, byteStrideForArgs >= sizeof(DrawIndexedInstancedIndirectArgs).
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

Crossfire API Driver Extension

The Crossfire API allows explicit control of resource transfers when running in Crossfire AFR mode in DirectX11.

agsDriverExtensions_SetCrossfireMode allows more control over how the app interacts with the driver in Crossfire AFR mode.

AGS_CROSSFIRE_MODE_DRIVER_AFR is the default path apps will go down on mGPU systems. This will run AFR mode and the driver will auto detect which resources to copy from one GPU to the next.

AGS_CROSSFIRE_MODE_EXPLICIT_AFR allows the app to determine which resources get transferred and when using the AGS Crossfire API. AGS_EXTENSION_NOT_SUPPORTED will be returned if this option is not available and the default Crossfire driver behavior will be enabled.

AGSReturnCode **agsDriverExtensions_SetCrossfireMode**(AGSContext* context, AGSCrossfireMode mode)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	mode	The AFR mode to use.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

Resources must be created with the appropriate Create function in order to specify the AFR transfer type per resource.

AGSReturnCode **agsDriverExtensions_CreateBuffer/Texture1D/2D/3D**(AGSContext* context, D3D11_BUFFER/TEXTURE_DESC desc, D3D11_SUBRESOURCE_DATA* initialData, ID3D11Resource** resource, AGSAFRTransferType transferType)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	desc	Pointer to the D3D11 resource description.
In Param	initialData	Optional pointer to the initializing data for the resource.

In/Out Param	resource	Returned pointer to the resource.
In Param	transferType	The transfer behavior. See AGSAfrTransferType for more details.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

AGSAFRTransferType enum	
AGS_AFR_TRANSFER_DEFAULT	The default Crossfire driver resource tracking, i.e. if the driver detects that this resource is required by the next frame, a p2p copy will be performed, potentially at a suboptimal time.
AGS_AFR_TRANSFER_DISABLE	Turns off driver resource tracking completely for this resource, i.e. no p2p copies are ever done.
AGS_AFR_TRANSFER_1STEP_P2P	App controlled GPU to next GPU transfer.
AGS_AFR_TRANSFER_2STEP_NO_BROADCAST	App controlled GPU to next GPU transfer using intermediate system memory.
AGS_AFR_TRANSFER_2STEP_WITH_BROADCAST	App controlled GPU to all render GPUs transfer using intermediate system memory.

agsDriverExtensions_NotifyResourceEndWrites is required to notify the driver that we have finished writing to the resource this frame. This will initiate a transfer for AGS_AFR_TRANSFER_1STEP_P2P, AGS_AFR_TRANSFER_2STEP_NO_BROADCAST and AGS_AFR_TRANSFER_2STEP_WITH_BROADCAST.

AGSReturnCode **agsDriverExtensions_NotifyResourceEndWrites**(AGSContext* context, ID3D11Resource* resource, const D3D11_RECT* transferRegions, const unsigned int* subresourceArray, unsigned int numSubresources)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	resource	The D3D11 resource.
In Param	transferRegions	An array of transfer regions (can be null to specify the whole area).
In Param	subresourceArray	An array of subresource indices (can be null to specify all subresources). These should be sorted in ascending value (eg 0, 1, 4, 8, rather than 4, 0, 1, 8).
In Param	numSubresources	The number of subresources in subresourceArray OR number of transferRegions. Use 0 to specify ALL subresources and one transferRegion (which may be null if specifying the whole area).
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

agsDriverExtensions_NotifyResourceBeginAllAccess is required to notify the driver that the app will begin read/write access to the resource.

AGSReturnCode **agsDriverExtensions_ NotifyResourceBeginAllAccess**(AGSContext* context, ID3D11Resource* resource)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	resource	The D3D11 resource.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	

agsDriverExtensions_NotifyResourceEndAllAccess is used for AGS_AFR_TRANSFER_1STEP_P2P to notify when it is safe to initiate a transfer. This call in frame N-(NumGpus-1) allows a 1 step P2P in frame N to start. This should be called after agsDriverExtensions_NotifyResourceEndWrites.

AGSReturnCode **agsDriverExtensions_ NotifyResourceEndAllAccess**(AGSContext* context, ID3D11Resource* resource)		
In Param	context	A valid pointer to an AGSContext structure.
In Param	resource	The D3D11 resource.
Return Code	On success AGS_SUCCESS, otherwise one of the failure codes.	