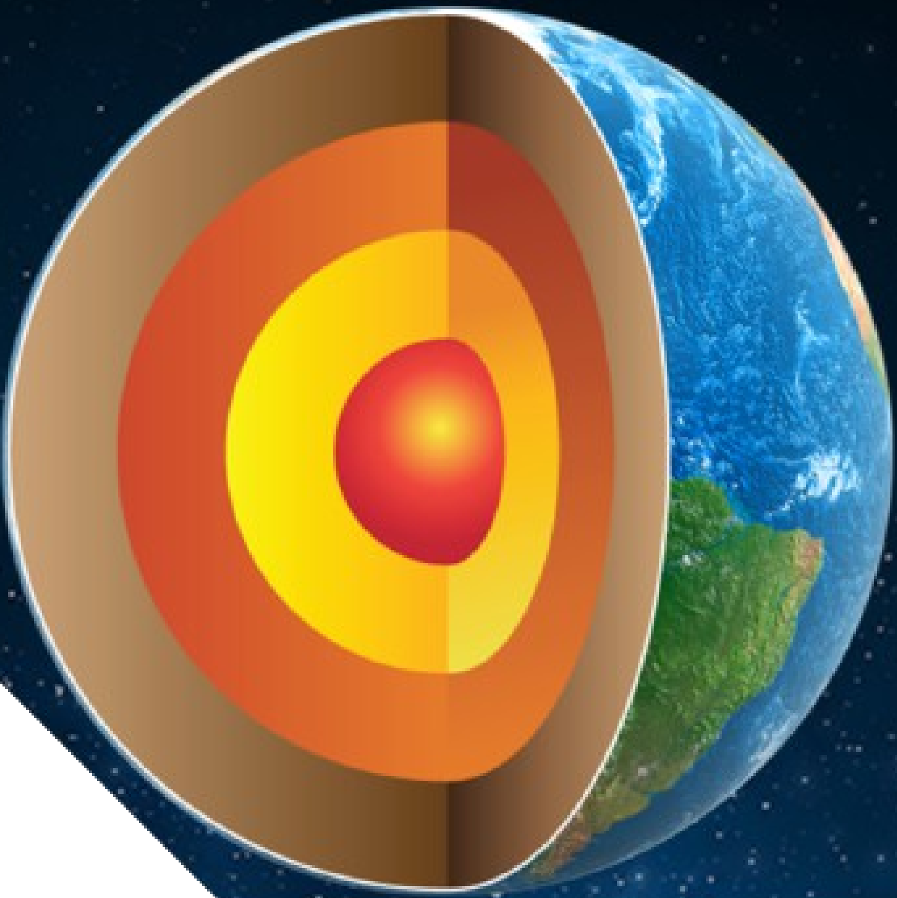


Mantle Programming Guide and API Reference

Revision 1.0
March 6, 2015

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions and typographical errors.

The information contained herein is subject to change and may be rendered inaccurate for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product releases, product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION.

AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY DIRECT, INDIRECT, SPECIAL OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

ATTRIBUTION

TABLE OF CONTENTS

Chapter I.

Introduction

Motivation

While previous generation PC graphics programming models – OpenGL and DirectX® 11 – have provided a solid 3D graphics foundation for quite some time, they are not necessarily ideal solutions in scenarios where developers want tighter control of the graphics system and require lower execution overhead.

The proposed new programming model and application programming interface (API) attempts to bridge PC and consoles in terms of flexibility and performance, address efficiency problems, and provide a forward-looking, system-level foundation for graphics programming.

Solution Overview

The proposed solution implements a lower system-level programming model designed for high-performance graphics that makes the PC graphics programming environment look a bit more like that found on gaming consoles. While allowing applications to build hardware command buffers with very small operational overhead, Mantle provides a reasonable level of abstraction in terms of the pipeline definition and programming model. As a part of improving the programming model, the Mantle API removes some legacy features found in other graphics APIs.

While the proposed programming model draws somewhat on the strengths of OpenGL and DirectX® 11, it was based on the following main design concepts:

- Performance for both CPU and GPU is the primary goal
- The solution is forward looking in terms of the abstraction and the programming model
- The solution supports multiple operating systems and platform configurations
- The application is the arbiter of correct rendering and the sole handler of persistent state. Analysis of current APIs indicates that an efficient small batch solution can only be achieved when the driver is as stateless as possible
- Where generic feature implementation have been proven to be too inefficient in other APIs and driver models, the responsibility is shifted to the application. An application generally has a better knowledge of the rendering context and can implement more intelligent optimization strategies. As an example, video memory management becomes an application responsibility in Mantle

Due to its lower level control of memory and synchronization features, the Mantle API is targeted at sophisticated developers. Effective use of the API requires in-depth knowledge of 3D graphics, familiarity with the underlying hardware architecture, and capabilities of modern GPUs, as well as an understanding of performance considerations. The proposed solution is primarily targeted at advanced graphics programmers familiar with the game console programming environment. Despite some of its lower-level implementation features, the expectation is that Mantle can still benefit a wide range of projects, as specialized higher-level, middle-ware Mantle-based solutions and engines become available.

Developer Manifesto

The Mantle API imposes upon PC graphics developers a new set of rules. Because of the abstraction level in Mantle, which is different from previous graphics API solutions in the PC space, some developer expectations need to be adjusted accordingly.

Mantle attempts to close a gap between PCs and consoles, in terms of flexibility and performance, by implementing a lower system-level programming model. In achieving this, Mantle places a lot more responsibility in the hands of developers. Due to the lower level of the API, there are many areas where the driver is no longer capable of providing safety, performance improvements, and workarounds. The driver essentially gets out of the developers' way as much as possible to allow applications to extract every little bit of performance out of modern GPUs. The driver does not create extra CPU threads behind the application's back, does not perform extensive validation on performance critical paths, nor does it recompile shaders in the background or perform other actions that application does not expect.

When using Mantle, developers need to take responsibility for their actions with extensive validation: fixing all instances of incorrect API usage, efficiently driving GPUs, and ensuring the implementation is forward looking to support future GPU architectures. The reason for this is that in order for the driver to be as efficient as possible, many problems can no longer be efficiently worked around in the driver. This extra responsibility is the cost developers have to pay to benefit from Mantle's advantages.

Mantle is designed for those graphics developers who are willing to accept this new level of responsibility.

Chapter II.

Programming Overview

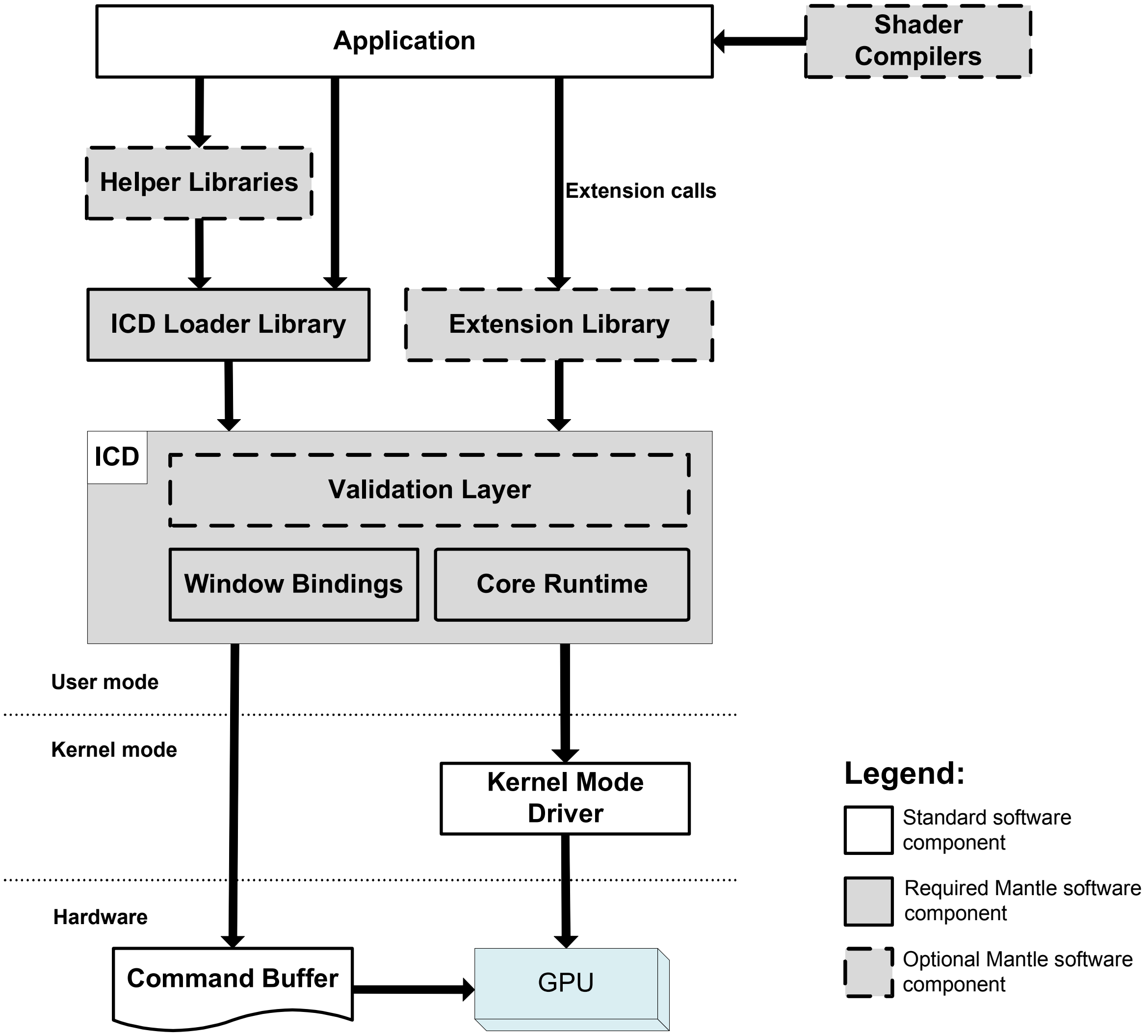
Software Infrastructure

Mantle provides a programming environment that takes advantage of the graphics and compute capabilities of PCs equipped with one or more Mantle compatible GPUs. The Mantle infrastructure includes the following components:

- a hardware platform with Mantle compatible GPUs
- an installable client driver (ICD) implementing:
    - a core Mantle API
    - platform specific window system bindings
    - Mantle API extensions
    - an API validation layer
- a generic ICD loader library with Mantle API interface
- optional extension interface libraries
- optional helper libraries to simplify Mantle development
- optional shader compilers and translators

The following diagram depicts the simplified conceptual view of Mantle software infrastructure.

Figure 1. Mantle software infrastructure



Mantle Libraries in Windows

To use Mantle in the Windows® programming environment, an application links with the core API loader library and the optional extension library, either statically through using .lib libraries or dynamically by querying entry points in .dll libraries. There are 32-bit and 64-bit versions of libraries available. An application should not redistribute Mantle .dll libraries, as they are distributed as a part of the AMD Catalyst™ driver installer.

The following static libraries are available.

Table 1. Statically linked Mantle libraries

Library file name	Description
mantle32.lib	32-bit static Mantle core API library
mantle64.lib	64-bit static Mantle core API library
mantleaxl32.lib	32-bit static AMD Mantle extension library
mantleaxl64.lib	64-bit static AMD Mantle extension library

The corresponding dynamic libraries are:

Table 2. Dynamically linked Mantle libraries

Library file name	Description
mantle32.dll	32-bit Mantle loader and core API dynamic library
mantle64.dll	64-bit Mantle loader and core API dynamic library
mantleaxl32.dll	32-bit AMD Mantle extension dynamic library
mantleaxl64.dll	64-bit static Mantle extension dynamic library

The function entry points for API and extension libraries are declared in header files:

Table 3. Mantle header files

Header file name	Description
mantle.h	Mantle core API
mantleExt.h	AMD Mantle extension interface
mantleWsiWinExt.h	Windows® specific WSI extension interface
mantlePlatform.h	Platform specific definitions
mantleDbg.h	Mantle debug API
mantleExtDbg.h	Debug features for AMD Mantle extensions
mantleWsiWinExtDbg.h	Debug features for Windows® specific WSI extension interface

Since Mantle libraries might not be available on all systems, an application could use delayed dynamic library loading. This would allow the application to avoid loading issues on the systems that do not have Mantle libraries installed. The following code snippet checks for the presence of a 64-bit Mantle library and delay loads it.

Listing 1. Check presence and delay load Mantle library in Windows® OS

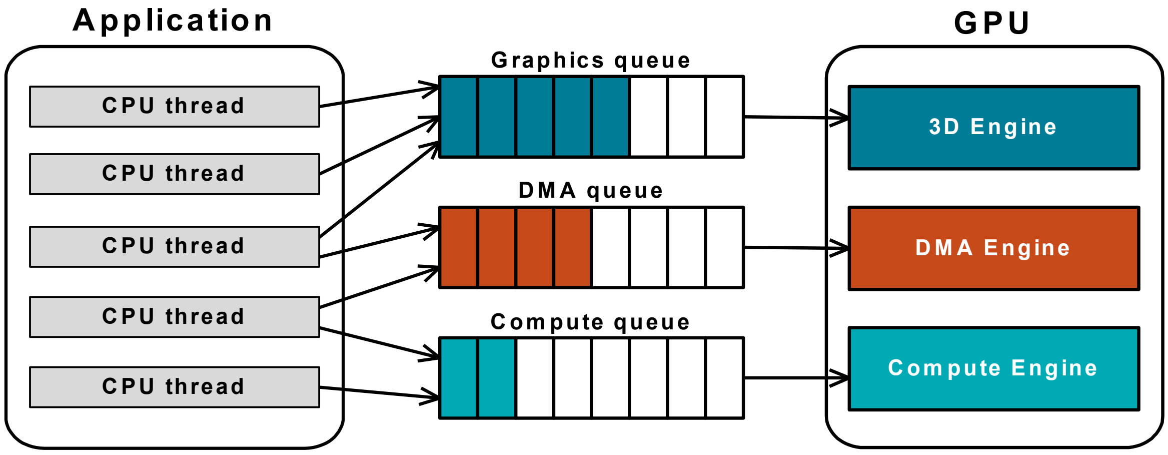
```
// application is linked with /DELAYLOAD:mantle64.dll
GR_RESULT InitMantle(
    const GR_APPLICATION_INFO* pAppInfo,
    GR_UINT*                   pGpuCount,
    GR_PHYSICAL_GPU            gpus[GR_MAX_PHYSICAL_GPUS])
{
    // Check Mantle library presence by trying to load it
    HMODULE hModule = LoadLibrary(TEXT("mantle64.dll"));
    if (hModule == NULL) {
        // Mantle library is not found
        return GR_ERROR_UNAVAILABLE;
    } else {
        // Decrement Mantle library reference count and unload
        FreeLibrary(hModule);
        // Implicitly load library and initialize Mantle
        return grInitAndEnumerateGpus(pAppInfo, NULL, pGpuCount, gpus);
    }
}
```

An application should avoid talking to Mantle drivers directly by circumventing loader and extension libraries.

Execution Model

Modern GPUs have a number of different engines capable of executing in parallel — graphics, compute, direct memory access (DMA) engine, as well as various multimedia engines. The basic building block for GPU work is a command buffer containing rendering, compute, and other commands targeting one of the GPU engines. Command buffers are generated by drivers and added to an execution queue representing one of the GPU engines, as shown in Figure 2. When the GPU is ready, it picks the next available command buffer from the queue and executes it. Mantle provides a thin abstraction of this execution model.

Figure 2. Queue submission model



An application in the Mantle programming environment controls the GPU devices by constructing command buffers containing native GPU commands through the Mantle API. The command buffer construction is extremely efficient — the API commands are directly translated to native GPU commands with minimal driver overhead, providing a high-performing solution. To achieve this performance, the driver’s core implementation performs minimal error checking while building command buffers in the release build of an application. It is the developers who are responsible for ensuring correct rendering during the development process. To facilitate input validation, profiling, and debugging, a special validation layer can be enabled on top of the core API that contains comprehensive state checking which notifies the developer of errors (invalid rendering operations) and warnings (potentially undefined rendering operations and performance concerns). Additional tools and libraries can also be used to simplify debugging and performance profiling. To improve performance on systems with multi-core CPUs, an application can build independent command buffers on multiple CPU threads in a thread-safe manner.

After command buffers are built, they are submitted to the appropriate queue for execution on the GPU device. The Mantle programming model uses a separate command queue for each of the engines so they can be controlled independently. The command buffer execution within a queue is serialized, but different queues could execute asynchronously. An application is responsible for using GPU synchronization primitives to synchronize execution between the queues as necessary.

Command buffer execution happens asynchronously from the CPU. When a command buffer is submitted to a queue, control is returned to an application before the command buffer executes. There can be a large number of submitted command buffers queued up at any time. The synchronization objects provided by the Mantle API are used to determine completion of various GPU operations and to synchronize CPU and GPU execution.

In Mantle, an application explicitly manages GPU memory allocations and resources required for rendering operations. At the time a command buffer is to be executed, the system ensures all resources and memory referenced in the command buffer are available to the GPU. If necessary, this is done by marshaling memory allocations according to the application-provided memory object reference list. In the Mantle programming environment, it is an application’s responsibility to provide a complete list of memory object references for each command buffer submission. Failure to specify an exhaustive list of memory references used in the command buffer might result in resources not being paged in and thus resulting in a fault or incorrect rendering.

A system could include multiple Mantle capable GPUs, each of them exposed as a separate *physical GPU*. The Mantle driver does not automatically distribute rendering tasks to multiple physical GPUs present in the system; it is an application’s responsibility to distribute rendering tasks between GPUs and synchronize operations as required. The API provides functionality for an efficient implementation of multi-GPU rendering techniques.

Memory in Mantle

A Mantle *device* operates on data stored in GPU *memory objects*. Internally, memory objects are referenced with a unique virtual address in a process address space. A Mantle GPU operates in a virtual address space which is separate from the CPU address space. Depending on the platform, a GPU device has a choice of different memory heaps with different properties for memory object placement. These heaps might include local video memory, remote (non-local) video memory, and other GPU accessible memory. Further, the memory objects in remote memory heaps could be CPU cacheable or write-combined, as indicated by the heap properties. An application can control memory object placement by indicating heap preferences and restricting the memory object placement to a specific set of heaps. The operating system and Mantle driver are free to move memory objects between heaps within the constraints specified by the application.

GPU memory is allocated on the block-size boundary, which in most cases is equal to the GPU page size. If an application needs smaller allocations, it sub-allocates from larger memory blocks.

The GPU memory is not accessible by the CPU unless it is explicitly mapped into the CPU address space. In some implementations, local video memory heaps might not be CPU visible at all; therefore, not all GPU memory objects can be directly mapped by the CPU. An application should make no assumptions about direct memory visibility. Instead, it should rely on heap properties reported by Mantle. In the case when a particular memory heap cannot be directly accessed by a CPU, the data are loaded to a memory location using GPU copy operations from a CPU accessible memory object.

The memory objects do not automatically provide *renaming* functionality – employing multiple copies of memory on *discard* type memory mapping operations. An application is responsible for tracking memory object use in the queued command buffers, recycling them when possible and allocating new memory objects for implementing renaming functionality.

Objects in Mantle

The devices, queues, state objects, and other entities in Mantle are represented by the internal Mantle objects. At the API level, all objects are referenced by their appropriate handles. Conceptually, all objects in Mantle can be grouped in the following broad categories:

- Physical GPU objects
- Device management objects (devices and queues)
- Memory objects
- Shader objects
- Generic API objects

The relationship of objects in Mantle is shown in Appendix A. Mantle Class Diagram. Some of the objects might have requirements for binding GPU memory as described in API Object Memory Binding. These memory requirements are implementation dependent.

The objects are created and destroyed through the Mantle API, though some of the objects are destroyed implicitly by Mantle. It is an application's responsibility to track the lifetime of the objects and only delete them once objects are no longer used by command buffers that are queued for execution. Failure to properly track object lifetime causes undefined results due to premature object deletion.

Mantle objects are associated with a particular device and cannot be directly shared between devices in multi-GPU configurations. There are special mechanisms for sharing some memory objects and synchronization primitives between capable GPUs. See Chapter VI. Multi-device Operation for more details. It is an application's responsibility to create multiple sets of objects, per device, and use them accordingly.

Pipelines and Shaders

The GPU pipeline configuration defines the graphics or compute operations that a GPU performs on the input data to generate an image or computation result. Pipelines provide a level of abstraction that supports existing graphics and compute operations, as well as enable exposure of new pipeline configurations in the future, such as hybrid graphics/compute pipelines. Depending on its type, a pipeline is composed of one or more shaders and a portion of a fixed function GPU state.

A *compute pipeline* includes a single compute shader while a *graphics pipeline* is composed of several programmable shaders and fixed function stages (some are optional, connected in a predefined order). The capability of the graphics and compute pipelines is similar to that of DirectX® 11. In the future, more pipeline configurations might be made available.

Compute queues support workloads performed by compute pipelines, while universal queues support workloads performed by both graphics and compute pipelines. A universal queue's command buffer independently specifies graphics and compute pipelines along with any associated state.

The pipelines are constructed from shaders. The Mantle API does not include any high-level shader compilers, and shader creation takes a binary form of an *intermediate language* (IL) shader representation as an input. The Mantle drivers could support multiple IL choices and the API should generally be considered IL agnostic. At present, an IL is based on a subset of AMD IL. Other options could be adopted in the future.

Window and Presentation Systems

In the most common case, an application has a user interface and displays rendering results in a window. The integration of Mantle with a window system is performed using a platform-specific w*indow system interface* (WSI) extension inter-operating with core Mantle API. For Windows® OS, the interface is provided by Window System Interface for Windows.

It is also possible to use Mantle in a headless configuration that lacks a graphical user interface. In this scenario, an application does not need to use the window system interface API, and it could directly render to an off-screen surface.

Error Checking and Return Codes

Under normal operation, the Mantle driver detects only a small subset of potential errors that are reported back to applications using error codes. Functions used for building command buffers do not return any errors, and in case of an error, silently fail the recording of the operations in a command buffer. Submitting such command buffer results in undefined behavior.

Mantle's design philosophy is to avoid error checking as much as possible during performance-critical paths, such as command buffer and descriptor set building. Whenever possible, the driver is designed to result in an application crash, as opposed to hung hardware, as the outcome of an invalid operation.

The return codes in Mantle are grouped in three categories:

- *Successful completion code* – `GR_SUCCESS` is returned when no problems are encountered.
- *Alternative successful completion code* – returned when function successfully completes and needs to communicate an additional information to the application (e.g., `GR_NOT_READY`).
- *Error code* – returned when a function does not successfully complete due to error condition.

Because the Mantle API exposes some lower-level functionality with minimal error checking, such as the ability to introduce an infinite wait in the queue, there is a higher risk of encountering either a hang of the GPU engines or an appearance of a hang. It is expected that a possibility of such occurrences is minimized by extensive debugging and validation at development and testing time. The Mantle driver implementation relies on system recovery mechanisms, such as *timeout detection and recovery* (TDR) in Windows® OS, to detect GPU hang conditions and gracefully recover without a need to reboot the whole system.

Lost Mantle Devices

An application is notified via `GR_ERROR_DEVICE_LOST` error code that either the GPU has been physically removed from the system or it is inoperable due to a hang and recovery execution. When an application detects a lost device error, it quits submitting command buffers, releases all memory and objects, re-enumerates devices by calling `grInitAndEnumerateGpus()`, and re-initializes all necessary device objects. Failing to correctly respond to this error code results in incorrect or missing rendering and compute operations.

Debug and Validation Layer

To facilitate debugging, a special validation layer can be optionally enabled at execution time. It is capable of detecting and reporting many more errors and dangerous conditions at the expense of performance. The debug error messages can be logged to the debug output or reported to an application through the debugger callback functionality, as described in Chapter VII. Debugging and Validation Layer.

> ! Applications that are not completely error and warning free with the comprehensive error checking in the validation layer might not execute correctly on some Mantle compatible platforms. Failure to address the warnings or errors could result in intermittent rendering or other problems, even if the application might seem to perform correctly on some system configurations.

Chapter III.

Basic Mantle Operation

GPU Identification and Initialization

Each Mantle capable GPU in a system is represented by a *physical GPU* object referenced with a `GR_PHYSICAL_GPU` object handle. There could be multiple physical GPUs visible to a Mantle application, such as in a case of multi-GPU graphics boards. A *device* represents a logical view or a *context* of an individual physical GPU, and provides associations of memory allocations, pipelines, states, and other objects with that GPU context. Mantle API objects cannot be shared across different devices.

To use Mantle, an application first needs to initialize and enumerate available physical GPU devices by calling `grInitAndEnumerateGpus()`, which retrieves the number of physical GPUs and their object handles. If no Mantle capable GPUs are found in the system, `grInitAndEnumerateGpus()` returns a GPU count of zero. In multi-GPU configurations, each physical GPU is reported separately in arbitrary order. See Chapter VI. Multi-device Operation for more information about multi-device configurations in Mantle. `grInitAndEnumerateGpus()` can be called multiple times. Calling it more than once forces driver reinitialization.

Mantle requires applications to identify themselves to the driver at initialization time. This identification helps the driver to reliably implement API versioning and application-specific driver strategies. The `GR_MAKE_VERSION` macro is used to encode the API version, application, and engine versions provided on initialization in the `GR_APPLICATION_INFO` structure. The application and engine identification is optional, but the API version used by the application is mandatory. Additionally, an application can provide optional `pfnAlloc` and `pfnFree` function callbacks for system memory management of memory used internally by the Mantle driver. If system memory allocation callbacks are not provided, the driver uses its own memory allocation scheme. The ICD loader does not use these allocation callbacks.

These allocation callback functions are called whenever the driver needs to allocate or free a block of system memory. On allocation, the driver requests memory of a certain size and alignment requirement. The alignment of zero is the equivalent of 1 byte or no alignment. To fine-tune allocation strategy, the driver provides a reason for allocation, which is indicated by GR_SYSTEM_ALLOC_TYPE type. When grInitAndEnumerateGpus() is called multiple times, the same callbacks have to be provided on each invocation. Changing the callbacks on subsequent calls to grInitAndEnumerateGpus() causes it to fail with GR_ERROR_INVALID_POINTER error.

To make a selection of GPU devices suitable for an application's purpose, an application retrieves GPU properties by using the grGetGpuInfo() function. Basic physical GPU properties are retrieved with information type parameter set to GR_INFO_TYPE_PHYSICAL_GPU_PROPERTIES, which are returned in GR_PHYSICAL_GPU_PROPERTIES structure. GPU performance characteristics could be obtained with the information type parameter set to GR_INFO_TYPE_PHYSICAL_GPU_PERFORMANCE, which returns performance properties in GR_PHYSICAL_GPU_PERFORMANCE structure.

Device Creation

A device object in Mantle represents an execution context of a GPU and is referenced by the GR_DEVICE handle. Once physical GPUs are enumerated and selected, an associated device is created by using the grCreateDevice() function for a given physical GPU device. Only a single device per GPU, per process is supported. Attempts to create multiple devices for the same physical GPU fail with GR_ERROR_DEVICE_ALREADY_CREATED error code.

Listing 2. Mantle initialization and device creation example

```
GR_RESULT result;
// Setup application and API version info
GR_APPLICATION_INFO appInfo = {};
appInfo.apiVersion = GR_API_VERSION;
// Init Mantle and get the physical GPU handles
GR_PHYSICAL_GPU gpus[GR_MAX_PHYSICAL_GPUS] = {};
GR_UINT gpuCount = 0;
result = grInitAndEnumerateGpus(&appInfo, NULL, &gpuCount, gpus);
// Device will need one universal queue
GR_DEVICE_QUEUE_CREATE_INFO queueInfo = {};
queueInfo.queueType  = GR_QUEUE_UNIVERSAL;
queueInfo.queueCount = 1;
// WSI extension needs to be initialized for presentation
static const GR_CHAR* const ppExtensions[] =
{
    "GR_WSI_WINDOWS",
};
// Setup device info
GR_DEVICE_CREATE_INFO deviceInfo = {};
deviceInfo.queueRecordCount        = 1;
deviceInfo.pRequestedQueues        = &queueInfo;
deviceInfo.extensionCount          = 1;
deviceInfo.ppEnabledExtensionNames = ppExtensions;
#if defined(DEBUG) || defined(_DEBUG)
// Validation layer is very useful for debugging
deviceInfo.maxValidationLevel      = GR_VALIDATION_LEVEL_4;
deviceInfo.flags                   |= GR_DEVICE_CREATE_VALIDATION;
#else
// Release build should never use validation layer
devInfo.maxValidationLevel         = GR_VALIDATION_LEVEL_0;
#endif
// Created device for the first available physical GPU
GR_DEVICE device = GR_NULL_HANDLE;
result = grCreateDevice(gpus[0], &deviceInfo, &device);
```

At device creation time, an application requests what queues should be available on the device. An application should only request queues that are available for the given physical GPU. A list of available queue types and the number of queues supported can be queried by using the `grGetGpuInfo()` function, with information type parameter set to `GR_INFO_TYPE_PHYSICAL_GPU_QUEUE_PROPERTIES`.

To access advanced or platform-specific Mantle features, an application can use the extension mechanism. Before creating a device, an application should determine if a desired extension is supported. If so, it can be requested at device creation time by adding the extension name to the table of enabled extensions in the device creation parameters. Extensions that are not explicitly requested at device creation time are not available for use. For more information about Mantle extensions, see Chapter VIII. Mantle Extension Mechanism.

An application might optionally request creation of a device that implements debug infrastructure for validation of various aspects of GPU operation and consistency of command buffer data. Refer to Chapter VII. Debugging and Validation Layer for more information.

Once an application finishes rendering and no longer needs a device, it is destroyed by calling grDestroyDevice(). To avoid memory leaks, an application must completely drain all command queues and destroy all objects associated with a device before its destruction.

GPU Memory Heaps

The GPU operates on data stored in GPU accessible memory. The GPU memory is represented by a variety of video memory *heaps* available in a system. The choice of heaps and their properties are platform dependent and the application queries memory heap properties to derive the best allocation strategy. On a typical PC with a discrete GPU, there would generally be one or more local video memory heaps and one or more non-local, or remote heaps. Other platforms might have different heap configurations. While heap identities are provided, the GPU proximity and strategy for managing heap priorities should be inferred from heap performance characteristics and other properties. The reported heap sizes are approximate and do not account for the amount of memory already allocated. An application might not be able to allocate as much memory as there is in a heap due to other running processes and system constraints.

> It is a good idea to avoid oversubscribing memory. The reported heap size gives a reasonable upper bound estimate on how much memory could be used.

To get the number of available memory heaps a device supports, an application calls grGetMemoryHeapCount(). The returned number of heaps is guaranteed to be at least one or greater.

Heaps are identified by a heap ID ranging from 0 up to the reported count minus 1. An application queries each heap's properties by calling grGetMemoryHeapInfo() with infoType set to GR_INFO_TYPE_MEMORY_HEAP_PROPERTIES value. The properties are returned in GR_MEMORY_HEAP_PROPERTIES structure.

Listing 3. Example of enumerating memory heaps

```
GR_RESULT result;
// Retrieve number of memory heaps
GR_UINT heapCount = 0;
result = grGetMemoryHeapCount(device, &heapCount);
// Allocate memory for heap info
GR_MEMORY_HEAP_PROPERTIES* pHeapInfo = new GR_MEMORY_HEAP_PROPERTIES[heapCount];
// Retrieve info for each heap
for (GR_UINT i = 0; i < heapCount; i++)
{
    GR_SIZE dataSize = sizeof(GR_MEMORY_HEAP_PROPERTIES);
    result = grGetMemoryHeapInfo(device, i, GR_INFO_TYPE_MEMORY_HEAP_PROPERTIES,
                                 &dataSize, &pHeapInfo[i]);
}
```

The heap properties contain information about heap memory type, heap size, page size, access flags, and performance ratings. The heap size and page size are reported in bytes. The heap size is a multiple of the page size.

Performance ratings for each memory heap are provided to help applications determine the best memory allocation strategy for any given access scenario. The performance rating represents an approximate relative memory throughput for a particular access scenario, either for CPU or GPU access for read and write operations; it should not be taken as an absolute performance metric. For example, if two heaps in a system have performance ratings of 1.0 and 2.0, it can safely be assumed that the second heap has approximately twice the throughput of the first. For heaps inaccessible by the CPU, the read and write performance rating of the CPU is reported as zero. While the performance ratings are consistent within the system, they should not be used to compare different systems as the performance rating implementation could vary.

GPU Memory Objects

A Mantle GPU operates on data contained in *memory objects* that are referenced in the API by a `GR_GPU_MEMORY` handle. There are several types of memory objects in Mantle which serve different purposes. The most common memory objects are *real memory objects* which are created by calling `grAllocMemory()`. An application specifies required size for the memory object along with its preferred placement in memory heaps and other options in `GR_MEMORY_ALLOC_INFO` structure. The other types of memory objects are discussed in the following sections of this guide.

Listing 4. Example of creating a memory object

```
GR_RESULT result;
// Setup memory allocation info
GR_MEMORY_ALLOC_INFO allocInfo = {};
allocInfo.size        = gpuPageSize * numPages;
allocInfo.alignment   = 0;
allocInfo.memPriority = GR_MEMORY_PRIORITY_NORMAL;
allocInfo.heapCount   = 1;
allocInfo.heaps[0]    = firstHeapChoiceId;
// Allocate memory
GR_GPU_MEMORY mem = GR_NULL_HANDLE;
result = grAllocMemory(device, &allocInfo, &mem);
```

> Whenever possible, an application should provide multiple heap choices to increase flexibility of memory object placement and memory management in general.

The Mantle driver allocates video memory in blocks aligned to the page size of the heap. The page size is system and GPU dependent and is specified in the heap properties. Different memory heaps might use different page sizes. When specifying multiple heap choices for a memory object, the largest of the allowed heap page sizes should be used for the granularity of the allocation. For example, if one heap has a page size of 4KB and another of 64KB, allocating a memory block that could reside in either of those heaps should be 64KB aligned.

If the application needs to allocate blocks smaller than a memory page size, the application is required to implement its own memory manager for sub-allocating smaller memory requests. An attempt to allocate video memory that is not page size aligned fails with `GR_ERROR_INVALID_ALIGNMENT` error code. When memory is allocated, its contents are considered undefined and must be initialized by an application.

By default, a memory object is assigned a GPU virtual address that is aligned to the largest page size of the requested heaps. Optionally an application can request memory object GPU address alignment to be greater than a page size. If the specified memory alignment is greater than zero, it must be a multiple of the largest page size of the requested heaps. The optional memory object alignment is used when memory needs to be used for objects that have alignment requirements that exceed a page size. For example, if page size is reported to be 64KB in heap properties, but an alignment requirement for a texture is 128KB, then the memory object that is used for storing that texture's contents has to be 128KB aligned. The object memory requirements are described in API Object Memory Binding.

> ! Avoid unnecessary memory object alignments as it might exhaust GPU virtual address space more quickly.

A memory object is freed by calling `grFreeMemory()` when it is no longer needed. Before freeing a memory object, an application must ensure the memory object is unbound from all API objects referencing it and that it is not referenced by any queued command buffers. Failing to ensure that a memory allocation is not referenced results in corruption or a fault.

GPU Memory Priority

A *memory object priority* is used to indicate to the memory management system how hard it should try to keep an allocation in the memory heap of the highest preference when under significant memory pressure. The memory priority behavior is platform specific and might have no effect when only one memory heap is available or when the GPU memory manager does not support memory object migration.

> ! The priority is just a hint to the memory management system and does not guarantee a particular memory object placement.

Memory objects containing render targets, depth-stencil targets and write-access shader resources should typically use either high memory priority `GR_MEMORY_PRIORITY_HIGH` or very high priority `GR_MEMORY_PRIORITY_VERY_HIGH`. Most other objects should use normal priority `GR_MEMORY_PRIORITY_NORMAL`. When it is known that a memory object will not be used by the

GPU for an extended period of time, it could be assigned `GR_MEMORY_PRIORITY_UNUSED` priority value. This indicates to the memory manager that a memory object could be paged out without any impact on performance. If an application decides to start using that memory allocation again, it should bump up its priority according to usage scenario.

> The memory priority provides coarse grained control of memory placement and an application should avoid frequent priority changes.

The initial memory object priority is specified at creation time; however, in systems that support memory object migration, it can be adjusted later on to reflect a change in priority requirements. An application is able to adjust memory object priority by calling `grSetMemoryPriority()` with one of the values defined in `GR_MEMORY_PRIORITY`.

CPU Access to GPU Memory Objects

A memory object created with `grAllocMemory()` represents a block of GPU virtual address space, and by default is not directly CPU accessible. Memory objects that can be made CPU accessible are considered to be *mappable*. An application retrieves a CPU virtual address pointer to the beginning of a mappable memory object by calling `grMapMemory()`. All of the memory heap choices for the mappable memory object must be CPU visible, which is indicated by `GR_MEMORY_HEAP_CPU_VISIBLE` heap property flag. If any heap used for the memory object is not CPU visible, the memory cannot be mapped. Attempts to map memory objects located in memory heaps invisible to the CPU fail with a `GR_ERROR_NOT_MAPPABLE` error code.

The memory is mapped without any checks for memory being used by the GPU. It is an application's responsibility to both synchronize memory accesses and to guarantee that data needed for rendering queued to the GPU are not overwritten by the CPU. An application is expected to implement its own internal memory *renaming* schemes or take other corrective actions, if necessary.

Once the CPU access to a memory object is no longer needed by the application, it can be removed by calling `grUnmapMemory()`.

The `grMapMemory()` and `grUnmapMemory()` functions are thread safe, provided the different threads are accessing different memory objects.

> On the Windows® 7-8.1 platforms, it is disallowed to keep memory objects that could be placed in local video memory heaps mapped while they are referenced by executing command buffers. Due to implementation of video memory manager in Windows®, such operation might result in intermittent data loss.

Pinned Memory

On some platforms, system memory allocations can be pinned (pages are guaranteed to never be swapped out), allowing direct GPU access to that memory. This provides an alternative to CPU mappable memory objects. An application determines support of memory pinning by examining flags in `GR_PHYSICAL_GPU_MEMORY_PROPERTIES` structure, which is retrieved by calling the `grGetGpuInfo()` function with the information type parameter set to `GR_INFO_TYPE_PHYSICAL_GPU_MEMORY_PROPERTIES`. If `GR_MEMORY_PINNING_SUPPORT` is set, the memory pinning is supported.

A *pinned memory object* representing a pinned memory region is created using `grPinSystemMemory()`. The pinned memory object is associated with the heap capable of holding pinned memory objects identified by the `GR_MEMORY_HEAP_HOLDS_PINNED` flag, as if it were allocated from that heap. Mantle guarantees that only one heap will be capable of holding pinned memory objects.

The pinned memory region pointer and size have to be aligned to a page boundary for the pinning to work. The page size can be obtained from the properties of the heap marked with the `GR_MEMORY_HEAP_HOLDS_PINNED` flag.

The memory is unpinned by destroying pinned memory object using the `grFreeMemory()` function. Pinned memory objects can be used as regular memory objects; however, they have a notable difference. Their priority cannot be specified. Pinned memory objects can be mapped, which would just return a cached CPU address of the system allocation provided at creation time.

Multiple system memory regions can be pinned; however. the total size of pinned memory in a system is limited and an application must avoid excessive use of pinning. Memory pinning fails if the total size of pinned memory exceeds a limit imposed by the operating system.

Pinning too much memory negatively impacts overall system performance.

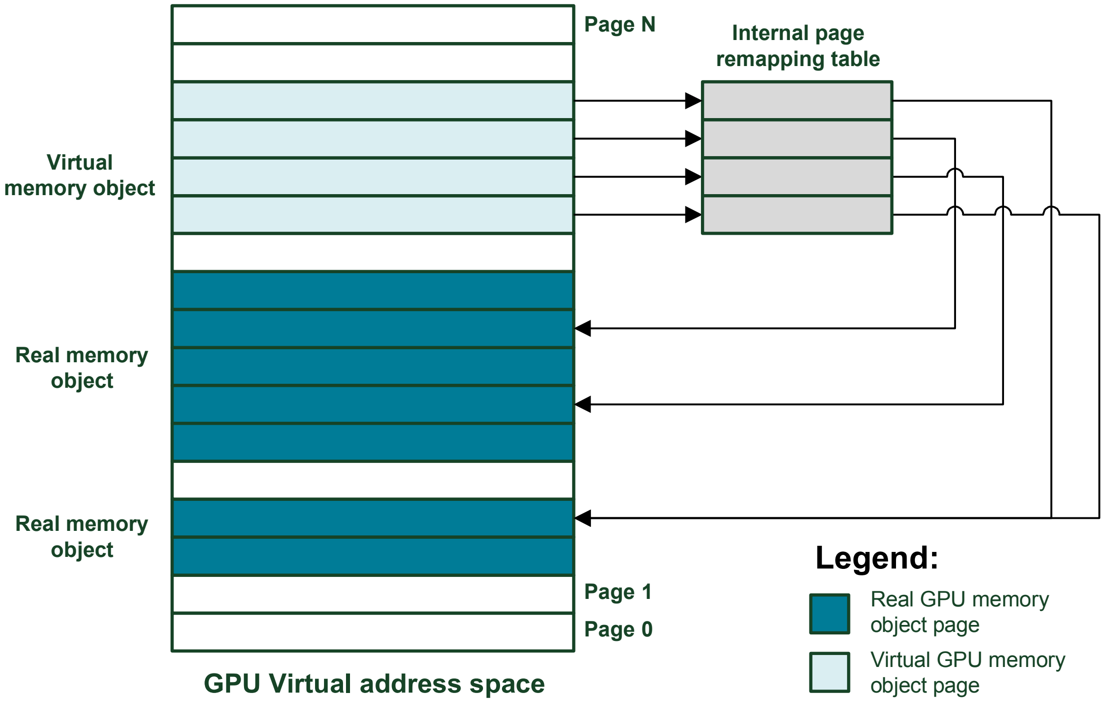
Virtual Memory Remapping

On some platforms, the Mantle API allows reservation of GPU address space by exposing *virtual memory objects* that can be remapped later to *real memory objects*. Since Mantle GPUs operate in a virtual machine (VM) environment, all memory objects are part of the GPU virtual address space. To avoid confusion, the following terminology is used: *real memory objects* are those backed by physical memory, while *virtual memory objects* refer to GPU virtual address space reservations without physical memory backing. The granularity of virtual memory mapping is the page size for virtual allocations, which can be queried in the device properties.

An application determines support of virtual memory remapping by examining flags in

`GR_PHYSICAL_GPU_MEMORY_PROPERTIES` structure, which is retrieved by calling the `grGetGpuInfo()` function with the information type parameter set to `GR_INFO_TYPE_PHYSICAL_GPU_MEMORY_PROPERTIES`. The virtual memory remapping is supported when `GR_MEMORY_VIRTUAL_REMAPPING_SUPPORT` is set.

Figure 3. Conceptual view of virtual memory remapping


Page N
Internal page remapping table
Virtual memory object
Real memory object
Real memory object
Page 1
Page 0
GPU Virtual address space
Legend:
Real GPU memory object page
Virtual GPU memory object page

Virtual memory objects are created by calling `grAllocMemory()` with the `GR_MEMORY_ALLOC_VIRTUAL` flag in the creation parameters. When a virtual allocation is created, none of its pages are backed by actual physical memory and they need to be remapped prior to use as described further. A virtual memory object is destroyed by using the `grFreeMemory()` function.

Virtual memory objects cannot be mapped for CPU access and their priority cannot be changed. If an application wants to update memory in virtual memory objects, it should do so by updating the real memory objects backing the virtual allocations.

Multiple virtual memory objects can exist simultaneously to provide very flexible memory management schemes. A page from real memory objects can be mapped to one or more pages in one or more virtual memory objects. The remapped memory access is transparent to the user and is internally implemented by adjusting the VM page table. There is no direct application access to the page tables; the driver provides `grRemapVirtualMemoryPages()` function for managing virtual memory page remapping. The remapping functionality is only valid for virtual allocations and calls to `grRemapVirtualMemoryPages()` with a real allocation or pinned memory object fail.

`grRemapVirtualMemoryPages()` specifies how multiple ranges of virtual memory pages are remapped to real memory objects. The remapping specified with each function invocation is additive and represents a delta state for page mapping. Previously mapped virtual pages can be unmapped by specifying the GR_NULL_HANDLE value for the target memory object.

The remapping happens asynchronously with operations queued to the GPU. Changing page mapping for objects at the time they are accessed by the GPU results in undefined behavior. To guarantee proper ordering of remapping with other GPU operations, two sets of queue semaphores can be provided by an application. The use of semaphores is optional if the application can guarantee proper execution order of operations using other methods. Before remapping, grRemapVirtualMemoryPages() function waits on semaphores to be signaled. After remapping, it signals another set of semaphores, indicating completion of remapping. Multiple invocations of grRemapVirtualMemoryPages() are executed sequentially with each other, and with back-to-back remapping operations, it is sufficient to provide semaphores on the first and the last remapping operations.

Memory pages are only remapped for virtual memory objects and the remapping only points to pages in real memory. Only one level of remapping is allowed, and it is invalid to remap pages to other virtual memory objects.

> ! When remapping memory pages containing texture data for tiled images, an application should be careful to avoid using the same page for different regions of images. Due to some tiling implementations, the tiling pattern of different image regions might not match.

Memory Allocation and Management Strategy

The optimal memory management strategy is dependent on the type of platform, the type and version of the operating system, and other factors. Mantle provides very flexible memory management facilities to enable a wide range of performance and ease-of-use tradeoffs. For example, an application could trade the cost of managing multiple smaller allocations versus the larger memory footprint. The following are some of the guidelines that applications might want to adopt.

Memory allocation and management strategy employed by an application depends on the capabilities of the GPU memory manager available on a platform. Some platforms might support memory object migration between the heaps, while others might not. An application determines the GPU memory manager's ability to migrate memory objects by examining flags in GR_PHYSICAL_GPU_MEMORY_PROPERTIES structure, which is retrieved by calling grGetGpuInfo() function with the information type parameter set to GR_INFO_TYPE_PHYSICAL_GPU_MEMORY_PROPERTIES. If GR_MEMORY_MIGRATION_SUPPORT is set, the migration is supported.

In general, an application should avoid over-subscription of GPU memory to provide ideal memory object placement, which ensures high performance. In the Windows® OS, where memory management is not completely under the application's control, a multi-tiered approach to memory objects can be applied. In this approach, parts of the memory management are handled by the Windows® video memory manager and parts of it rest on the application's shoulders. First, an application should use reasonably sized memory pools of different priorities. The "reasonable" size depends on how much video memory a graphics board has, how much memory is needed, and other factors. Using memory pools of 16-32MB is a good starting point for experimentation. Resources should be grouped in memory pools by their type, read or write access, and priority. Objects with larger memory requirements, such as multisampled targets, might use their own dedicated memory objects. The key to extracting maximum performance from a number of configurations and platforms is making memory management configurable.

When deciding on memory placement, an application should evaluate performance characteristics of different memory heaps to sort and filter heaps according to its requirements. An application should be prepared to deal with a wide range of memory heap configurations – from supporting a single heap to supporting heaps of new types, such as `GR_HEAP_MEMORY_EMBEDDED`. The exposed memory heaps are likely to change in the future due to ongoing platform, OS, and hardware developments.

An application should generally specify multiple heaps for memory objects, if memory usage allows for it. This gives the driver and video memory manager the best chance of placing the memory object in the best location under high memory pressure. The controlling of memory placement is done by adjusting the heap order.

Further, the memory should be grouped in pools of different priorities and object assignment to memory should be performed according to the memory priority. It is recommended to define 3-5 memory pool priority types. See GPU Memory Priority for discussion of memory priorities.

An application should avoid marking all memory objects with the same memory priority. Under heavy memory pressure, the video memory manager in Windows® might get confused trying to keep all memory objects in video memory, resulting in unnecessary movement of data between local and non-local memory heaps.

All resources that are written by the GPU (i.e., target images, and read/write images) should be in high-priority memory pools, others can be placed in medium or low priority pools. The application should ensure that, whenever possible, high and medium priority pools do not oversubscribe available local video memory, including all visible and non-visible local heaps on the graphics card. The threshold for determining oversubscribed video memory conditions depends on the platform and the execution conditions, but setting it to about 60-80% of local video memory for high and medium priority allocations would be a safe choice for full screen applications. To avoid crossing the memory threshold for high and medium pools, the application should manage resource placement based on the memory working set. If parts of the memory in high and medium priority polls do not fit under that 60-80% threshold, the application can use an asynchronous DMA, as described in Chapter X. AMD Extension: DMA Queue, to move resource between local and non-local memory when necessary, providing more intelligent memory management of video memory under pressure.

Buffer-like resources, as well as small, infrequently used and compressed textures, could be lower priority than more frequently GPU-accessed images of larger texel size. On the systems which support memory object migration, it is reasonable to allow lower priority memory objects to be spilled by the OS to non-local video memory without the application worrying too much about their migration.

On the systems with relatively small visible local memory heap, the application should be careful with the placement of memory objects inside of it. Only high priority memory pools should be in both local non-visible and local visible, specified in that order. Medium priority pools probably should not be in a local visible heap if it is a scarce resource, but it depends on what else needs to go into the local visible heap.

> With integrated graphics, which are part of an accelerated processing unit (APU), the application should generally use non-local memory heaps instead of local visible heap for memory objects that require CPU access.

Pipeline objects and descriptor sets should generally be in local visible heaps, provided that they do not take up too much memory. For pipelines, an application can reduce memory requirements by just keeping a working set of pipelines bound to memory and binding/unbinding them on the fly as necessary. An application might want to maintain multiple pools of memory for pipelines and descriptor sets for efficient binding/unbinding. This could help ensure the memory objects containing pipelines and descriptor sets are not paged out to non-local memory by the Windows® video memory manager.

Generic Mantle API objects

The Mantle API objects, other than physical GPUs, devices, queues, and memory objects, are grouped into a broad *generic API object* category. These objects have common API functions for querying object properties, managing memory binding, and destruction.

API Object Destruction

Once a generic API object is no longer needed, it is destroyed by calling the `grDestroyObject()` function. If an object has previous memory binding, it is required to unbind memory from an API object before it is destroyed.

The object should not be destroyed while it is referenced by any other object or while there are references to an object in any command buffer queued for execution.

Querying API Object Properties

Mantle API objects have a variety of properties that an application queries to enable proper object operation. There are several functions for querying properties depending on the object type. For generic API objects, most of the properties can by queried by calling `grGetObjectInfo()`.

Querying Parent Device

In cases when objects are created outside of the application's control, it might be necessary for an application to know to what device an API object belongs. An example of such use could be Mantle API interception tools and utilities for frame capture and debugging. These applications can query parent device information for any API object by calling `grGetObjectInfo()` with the information type parameter set to `GR_INFO_TYPE_PARENT_DEVICE`. The parent device handle is returned in the `GR_PARENT_DEVICE` structure.

Additionally, an application can query a physical GPU handle that was used for device creation by calling `grGetObjectInfo()` for the device object with information type parameter set to `GR_INFO_TYPE_PARENT_PHYSICAL_GPU`. The parent device handle is returned in `GR_PARENT_PHYSICAL_GPU` structure.

API Object Memory Binding

In Mantle, some API objects require GPU memory storage for their data. Developers are responsible for explicitly managing GPU memory allocations for these objects based on memory requirements reported at run-time. These API objects must be bound to memory before they can be used.

The most obvious objects requiring video memory are images, but other objects, such as state and pipeline objects, might also require GPU memory storage depending on the implementation. The only objects that are guaranteed to have no external memory requirements are devices, queues, command buffers, shaders, and memory objects. Device, queue, and command buffer objects manage their own internal memory allocations. Shader objects are also special because they are not directly referenced by Mantle GPUs.

For the object types which can be bound to memory, an application should not make assumptions about memory requirements, as requirements might change between GPUs and even between versions of the Mantle driver. An application queries object memory requirements by calling `grGetObjectInfo()` with a handle of the object of interest and the `GR_INFO_TYPE_MEMORY_REQUIREMENTS` information type. The returned data are in the `GR_MEMORY_REQUIREMENTS` structure and includes memory size, alignment, and a list of compatible memory heaps.

Not all objects have memory requirements, in which case it is valid for the requirements structure to return zero size and alignment, and no heaps. For objects with valid memory requirements, at least one valid heap is returned. If the returned memory size is greater than zero, then memory needs to be allocated and associated to the API object. To bind an object to memory, an application should call `grBindObjectMemory()` with the desired target memory object handle and an offset within the memory object.

Listing 5. Example of binding object memory

```
GR_RESULT result;
// Get memory requirements
GR_MEMORY_REQUIREMENTS memReqs = {};
GR_SIZE reqSize = sizeof(GR_MEMORY_REQUIREMENTS);
result = grGetObjectInfo(object, GR_INFO_TYPE_MEMORY_REQUIREMENTS,
                         &reqSize, &memReqs);
// Don't need to bind memory if not required
if (memReqs.size > 0)
{
    // Find appropriate memory location
    GR_GPU_SIZE bindOffset = 0;
    GR_GPU_MEMORY mem = GetMatchingMemObjectAndOffset(&memReqs, &bindOffset);
    // Bind memory at appropriate offset
    grBindObjectMemory(object, mem, bindOffset);
}
```

The memory alignment for some objects might be larger than the video memory page size. If that is the case, an application must create memory objects with an alignment multiple of API object alignment requirements. A single memory object can have multiple API objects bound to it, as long as the bound memory regions do not overlap.

Memory heap allowed for binding different API objects could vary with implementation and an application should make no assumptions about heap requirements. That information is provided as a part of the object memory requirements using an allowed heap list. Compatible heaps are represented by heap ordinals (i.e., the same ones used with `grGetMemoryHeapInfo()`). Only the heaps on that list can be used for the object memory placement. An application could filter the heaps according to its additional requirements. For example, it could remove CPU invisible heaps to ensure guaranteed CPU access to the memory. The heaps in the list are presorted according to the driver's performance preferences, but the order of heaps for a memory allocation does not need to match the order returned in object requirements and can optionally be changed by the application.

> Driver provided heap preferences are just a suggestion, and a sophisticated application could adjust preferred heap order according to its requirements.

The driver ensures that the required heap capabilities for any given object match at least one of the heaps present in the system.

The driver fails memory binding for any one of the following reasons:

- If the memory heaps used for memory object creation do not match memory heap requirements of the particular API object
- If the memory placement requirements make the object data extend past the memory object
- If the required memory alignment does not match the provided offset

When objects other than images are bound to a memory object, the necessary data might be committed to memory automatically by the Mantle driver without an API involvement. The handling of memory binding is different for image objects and is described in Image Memory Binding.

> When a pipeline object has memory requirements, binding its memory automatically initializes the GPU memory by locking it and updating it with the CPU. If the memory object used for pipeline binding resides in local video memory at the time of binding while being referenced in queued command buffers, the memory object cannot be safely mapped on Windows® 7-8.1 platforms, and such pipeline binding leads to undefined results.

The object is unbound from memory by specifying the `GR_NULL_HANDLE` value for the memory object when calling the `grBindObjectMemory()` function.

An application is able to rebind objects to different memory locations as necessary. This ability to rebind object memory is particularly useful for some cases of application controlled image *renaming,* as image objects would not need to be recreated. It is not necessary to explicitly unbind previously bound memory before binding a new one. The rules for rebinding memory are different for images and all other object types. Rebinding of a given non-image object should not occur from the time of building a command buffer or a descriptor set which references that object to the time at which the GPU has finished execution of that command buffer or descriptor set. If a new memory location is bound to a non-image object while that object is referenced in a command buffer scheduled for execution on the GPU, the execution results are not guaranteed after memory rebinding.

Image Memory Binding

Image objects have slightly specialized memory binding rules. The image's object data are not initialized on memory binding and previous memory contents are preserved. The non-target images are assumed to be in the `GR_IMAGE_STATE_DATA_TRANSFER` state upon memory binding. Images used as color targets or depth-stencil implicitly start in the `GR_IMAGE_STATE_UNINITIALIZED` state, and must be transitioned to a proper state and cleared before first use.

Target images should never rely on the previous memory contents after memory binding. Failing to initialize state and clear target images before the first use results in undefined results.

Image memory can be rebound at any time, even during command buffer construction or descriptor set building. A current memory binding of an image (its GPU memory address) is recorded in the command buffer on direct image reference. Likewise, memory binding of an image is stored in the descriptor set at image attachment time. Rebinding image memory does not affect previously recorded image references in command buffers and descriptor sets. To ensure integrity of the data, any images that might have been written to by the GPU must be transitioned to a particular state before unbinding or re-binding memory. Non-target images must be transitioned to the `GR_IMAGE_STATE_DATA_TRANSFER` state before memory unbinding, while images used as color targets or depth-stencil must be transitioned to the `GR_IMAGE_STATE_UNINITIALIZED` state. Alternatively, images of any type can be transitioned to `GR_IMAGE_STATE_DISCARD` state. See Memory and Image States for more information about image states.

Queues and Command Buffers

In Mantle, all commands are sent to the GPU by recording them in command buffers and submitting command buffers to the GPU queues, along with a complete list of used memory object references.

Queues

Mantle GPU devices can have multiple execution *engines* represented at the API level by *queues* of different types. The type and maximal number of queues supported by a GPU, along with their properties, is retrieved from physical GPU properties by calling the `grGetGpuInfo()` function with the information type parameter set to `GR_INFO_TYPE_PHYSICAL_GPU_QUEUE_PROPERTIES`, which returns an array of `GR_PHYSICAL_GPU_QUEUE_PROPERTIES` structures, one structure per queue type. Since the number of available queue types and the amount of returned data could vary, to determine the data size an application calls `grGetGpuInfo()` with a `NULL` data pointer, the expected data size for all queue property structures is returned in `pDataSize`.

Mantle API defines two queue types: a universal queue (`GR_QUEUE_UNIVERSAL`) and an asynchronous compute queue (`GR_QUEUE_COMPUTE`). Other queue types, such as DMA and so on are exposed through extensions. There is at least one universal queue available for the Mantle device; other queues are optional.

The universal queues support both graphic rendering and compute operations, which are dispatched synchronously, even though their execution in some cases might overlap. The additional compute-only queues operate asynchronously with the universal and other queues, and it is an application’s responsibility to synchronize all queue execution. While the execution across multiple queues could be asynchronous, the execution order of command buffers within any queue is well defined and matches the submission order.

The queues in Mantle are referenced using GR_QUEUE object handles. The queue objects cannot be explicitly created. Instead, when a device is created, an application requests a number of universal, compute, and other queues up to the maximum number of queues supported by the device. There must be at least one queue requested on device creation. Requesting more queues than are available on a device fails the device creation. It is invalid to request the same queue type multiple times on device creation.

Once a device is created, the queue handles are retrieved from the device by calling grGetDeviceQueue() with a queue type and a requested logical queue ID. The logical queue ID is a sequential number starting from zero and referencing up to the number of queues requested at device creation. Each queue type has its own sequence of IDs starting at zero.

Listing 6. Example of creation of a device with universal and compute queues

```
GR_RESULT result;
// Device with 1 universal and 1 compute queue
GR_DEVICE_QUEUE_CREATE_INFO queueInfo[2] = {};
queueInfo[0].queueType  = GR_QUEUE_UNIVERSAL;
queueInfo[0].queueCount = 1;
queueInfo[1].queueType  = GR_QUEUE_UNIVERSAL;
queueInfo[1].queueCount = 1;
// Setup device info
GR_DEVICE_CREATE_INFO deviceInfo = {};
deviceInfo.queueRecordCount        = 2;
deviceInfo.pRequestedQueues        = queueInfo;
deviceInfo.extensionCount          = 0;
deviceInfo.ppEnabledExtensionNames = NULL;
devInfo.maxValidationLevel         = GR_VALIDATION_LEVEL_0;
// Created device with requested queues
GR_DEVICE device = GR_NULL_HANDLE;
result = grCreateDevice(gpus, &deviceInfo, &device);
// Retrieve universal queue
GR_QUEUE universalQueue = GR_NULL_HANDLE;
result = grGetDeviceQueue(device, GR_QUEUE_UNIVERSAL, 0, &universalQueue);
// Retrieve compute queue
GR_QUEUE computeQueue = GR_NULL_HANDLE;
result = grGetDeviceQueue(device, GR_QUEUE_COMPUTE, 0, &computeQueue);
```

The queue objects cannot be destroyed explicitly by an application and are automatically destroyed when the associated device is destroyed. Once the device is destroyed, attempting to use a queue results in undefined behavior.

Command Buffers

Command buffers are objects that contain GPU rendering and other commands recorded by the driver on the application's behalf. The command buffers in Mantle are referenced using

GR_CMD_BUFFER object handles. A command buffer can be executed by the GPU multiple times and recycled, provided that the command buffer is not pending execution by the GPU at the time of recycling.

The command buffers are fully independent and there is no persistence of *GPU state* between the command buffers. When a new command buffer is recorded, the state is undefined. All relevant state must be explicitly set by the application before state-dependent operations such as draws and dispatches can be recorded in a command buffer.

An application can create a command buffer by calling grCreateCommandBuffer(). At creation time, a command buffer is designated for use on a particular queue type. A command buffer created for execution on universal queues is called a *universal command buffer*, the one created for a compute queue is called a *compute command buffer*.

An application must ensure that the command buffer is not submitted and pending execution before destroying it by calling grDestroyObject().

COMMAND BUFFER BUILDING

The Mantle driver supports multithreaded command buffer construction using independent command builder contexts. There is no hard limit on how many command buffers could be constructed in parallel at any given time.

An application calls grBeginCommandBuffer() to start recording a command buffer. An application must ensure the command buffer object is not previously scheduled for execution when it begins recording. Once recording starts, an application records a sequence of state binds, draws, dispatches, and other commands, then terminates construction by calling grEndCommandBuffer(). After a command buffer is fully constructed, it can be submitted for execution as many times as necessary.

Command buffer commands may only be recorded between the grBeginCommandBuffer() and grEndCommandBuffer() command buffer functions that put command buffer in a *building state*. Attempts to record command buffer while it is not in the building state results in a silent fail of commands unless it is running with the validation layer enabled.

Listing 7. Example of creating and building a command buffer

```
GR_RESULT result;
// Create compute command buffer
GR_CMD_BUFFER_CREATE_INFO cmdBufInfo = {};
cmdBufInfo.queueType = GR_QUEUE_COMPUTE;
GR_CMD_BUFFER cmdBuffer = GR_NULL_HANDLE;
result = grCreateCommandBuffer(device, &cmdBufInfo, &cmdBuffer);
// Start building command buffer, optimize fo single time submittion
result = grBeginCommandBuffer(cmdBuffer,
                              GR_CMD_BUFFER_OPTIMIZE_ONE_TIME_SUBMIT);
// Record command buffer commands
grCmdSetEvent(cmdBuffer, event);
// Finish recording
grEndCommandBuffer(cmdBuffer);
```

While a command buffer could contain a large number of GPU operations, there might be a practical limit to the GPU command buffer length or total amount of recorded command buffer data. If an application runs out of memory reserved for command buffers, no more new command buffers are built until previously recorded command buffers are recycled and command buffer memory is freed.

> ! In general, it is not recommended to record huge command buffers. If a command buffer is taking too long to execute, a system might interpret the condition as a hardware hang and could attempt to reset the GPU device.

An application may avoid the overhead of creating new command buffer objects by recycling a command buffer not referenced by the GPU. Calling `grBeginCommandBuffer()` implicitly recycles the command buffer before starting a new recording session. An application could explicitly recycle the command buffer by calling `grResetCommandBuffer()`. An explicit command buffer reset by an application allows the driver to release the memory and any other internal command buffer resources as soon as possible without re-recording the command buffer. A command buffer can be recycled or reset by an application as soon as the buffer finishes its last queued execution and an application no longer needs it. It is the application's responsibility to ensure that the command buffer is not referenced by the GPU and is not scheduled for execution.

Command buffer construction could fail for a number of different reasons: running out of memory or other resources, hitting an error condition, and so on. The error is only guaranteed to be returned upon the command buffer termination with `grEndCommandBuffer()`. The error is not returned during the command buffer construction, and the command buffer building function silently fails unless running with the validation layer enabled. An application must be able to gracefully handle a case when termination of a command buffer fails.

Once any Mantle API object, such as image view or a GPU memory, is referenced in commands used during command buffer recording, it should not be destroyed until the command buffer recording is finished by calling `grEndCommandBuffer()`.

Command Buffer Optimizations

At command buffer building time, an application specifies optional optimization hints that could help the Mantle driver to tailor command buffer contents for different performance scenarios. Specifying the `GR_CMD_BUFFER_OPTIMIZE_ONE_TIME_SUBMIT` hint indicates to the driver that command buffer will be submitted only once. This allows the driver to apply submission time optimizations if multiple command buffers are submitted in a single batch.

A number of other hints target GPU optimizations in command buffers. Specifying the `GR_CMD_BUFFER_OPTIMIZE_GPU_SMALL_BATCH` hint optimizes command buffer for GPU command stream processing that could become a bottleneck in cases of small or lightweight draw and dispatch operations. The `GR_CMD_BUFFER_OPTIMIZE_PIPELINE_SWITCH` hint optimizes command buffer for cases when the application frequently changes pipelines between draw and dispatch operations. Similarly, `GR_CMD_BUFFER_OPTIMIZE_DESCRIPTOR_SET_SWITCH` optimizes command buffer for the case when descriptor sets are changed very frequently.

Multiple optimization flags can be specified at the same time. The command buffer optimization hints could increase CPU overhead during command buffer building and provide a mechanism for trading CPU performance versus the GPU performance.

> An application could detect at run time if it is CPU or GPU bound and dynamically adjust command buffer optimization hints to better balance CPU and GPU performance.

Command Buffer Submission

Once a command buffer is built, it is submitted for execution on a queue of a matching type. For example, a command buffer created for universal queues cannot be executed on compute queues and vice versa. An attempt to submit a command buffer to the queue of a wrong type fails submission.

Command buffers are submitted to a queue by calling `grQueueSubmit()`. Multiple command buffers can be submitted as a batch in a single submit operation. Submission places the provided command buffers in a queue and does not guarantee their immediate execution upon immediate return from the `grQueueSubmit()` function. When submitting multiple command buffers in a single batch, they are executed in the order in which they are provided in the list.

> Submitting multiple command buffers in one operation might help reduce the CPU and GPU overhead. Also, avoid submitting a lot of small command buffers, as there might be a fixed GPU overhead per command buffer, and GPU execution time needs to be sufficient to cover the scheduler latency.

If an application needs to track command buffer execution status, it can supply an optional fence object in the function parameters; otherwise, `GR_NULL_HANDLE` could be used instead. The fence is reached when the last provided command buffer in a submission batch has finished execution.

Listing 8. Example of submitting a command buffer and waiting for completion

```
GR_RESULT result;
// Setup memory references used by command buffer
const GR_MEMORY_REF memRefs[] =
{
    { targetMem, 0 },
    { miscMem, 0 },
    { constMem, GR_MEMORY_REF_READ_ONLY },
};
static const GR_UINT MemRefCount = sizeof(memRefs) / sizeof(memRefs[0]);
// Submit command buffer and associate a fence
result = grQueueSubmit(queue, 1, &cmdBuffer, MemRefCount, memRefs, fence);
// Wait for completion using a fence with 2s timeout
result = grWaitForFences(device, 1, &fence, GR_TRUE, 2.0f);
```

It is allowed to record and submit empty command buffers with no actual commands between `grBeginCommandBuffer()` and `grEndCommandBuffer()` calls.

> An application should avoid submitting an excessive number of empty command buffers, as each submitted command buffer adds CPU and GPU overhead.

Once command buffer is submitted, an API object, directly or indirectly referenced by the command buffer, must not be destroyed until the command buffer execution completes.

GPU Memory References

On submission, an application provides a complete list of memory objects used by the submitted command buffers, including virtual and pinned memory objects. The memory reference for a memory object is specified using the `GR_MEMORY_REF` structure. The supplied memory object handle cannot be `GR_NULL_HANDLE`. It is an application's responsibility to guarantee completeness of the memory references list. This includes all memory used by all Mantle objects directly or indirectly referenced in command buffers.

When using virtual memory allocations, an application must include all real allocations that the remapped virtual memory objects are referencing. Failing to include all memory references results in incorrect rendering since memory objects might not be resident on the GPU at the command buffer execution time.

There are two complimentary methods for supplying memory references. First, a list of memory references is specified at the command buffer submission time. Second, a set of *global memory references* is made available on a per-queue basis using the `grQueueSetGlobalMemReferences()` function. The references are global for a queue in the sense that they are used by all command buffers submitted to the queue. For example, these might be used with memory objects storing device object data referenced in all of the submitted command buffers.

If an application needs to make memory references global to the device, it should separately set them on all used queues.

Specifying a global memory references list completely overwrites the previously specified list. The previous memory reference list can be removed by specifying a zero number of global memory references along with NULL reference list pointer. Use of the global memory reference list is optional and is present only as an optimization. A snapshot of the global memory references is taken at submission time and applied to the submitted command buffers. Changing the global memory references does not apply to already submitted command buffers.

The grQueueSetGlobalMemReferences() function is not thread safe and the application needs to ensure it cannot be called simultaneously with other functions accessing a queue.

There is a limit on how many total memory references can be specified per command buffer at execution time. This limit applies to the global memory references, as well as the references from the list supplied on submission, and the sum of both should not exceed the specified limit. Exceeding the limit results in failed a command buffer submission. The maximal number of memory references can be queried from the physical GPU properties.

While building command buffers, an application has to keep an eye on the number of referenced memory objects per command buffer. If it grows too large, the command buffer cannot be safely submitted.

Read-only GPU Memory References

As an optimization, an application could specify the GR_MEMORY_REF_READ_ONLY flag to indicate that the memory object is used for read-only GPU access and its contents will not change during the command buffer execution. Table 4 lists memory access type for various operations. A memory object is considered to be read-only if all of its uses are for read-only access.

Table 4. Memory access type for command buffer operations

Operation	Access Type
Memory bound to pipeline and state objects	Read
Memory bound to descriptor sets	Read
Memory for index data	Read
Memory for dynamic memory view	Read/Write
Memory for memory views attached to descriptor sets	Read/Write
Memory bound to images used as image views attached to descriptor sets	Read/Write

Operation	Access Type
Memory bound to images used as color targets	Write
Memory bound to images used as depth-stencil	Write
Memory used in state transitions	Write
Memory bound to images used in state transitions	Write
Memory for draw or dispatch argument data	Read
Source for memory copy	Read
Destination for memory copy	Write
Memory bound to images used as source for copy	Read
Memory bound to images used as destination for copy	Write
Memory bound to images used as source for cloning	Write
Memory bound to images used as destination for cloning	Write
Memory bound to images used as source for resolve	Read
Memory bound to images used as destination for resolve	Write
Memory for immediate update from command buffer	Write
Memory for fill operation	Write
Memory bound to cleared color images	Write
Memory bound to cleared depth-stencil images	Write
Memory bound to set or reset event objects	Write
Memory for queue atomic operations	Write
Memory bound to query pool objects cleared or counter	Write
Memory for timestamps	Write
Memory for loading atomic counters	Read
Memory for saving atomic counters	Write
Memory used in any state transitions	Read/Write
Memory bound to images used in any state transitions	Read/Write

Specifying the read-only memory flag, while actually writing memory contents from within a command buffer, results in undefined memory contents.

Avoid mixing read-only and read-write memory uses within the same memory object.

Compute Dispatch Operations

The Mantle API supports dispatching compute operations using a compute pipeline and a currently bound command buffer compute state. The compute is dispatched with explicit work dimensions by calling `grCmdDispatch()`, which is available on both universal and compute queues.

The work dimensions for compute dispatch cannot be zero.

Indirect Dispatch

The compute job dimensions could be specified to come from memory by using the `grCmdDispatchIndirect()` function. The dispatch argument data must be 4-byte aligned and the memory range containing the indirect data must be in the `GR_MEMORY_STATE_INDIRECT_ARG` state. The layout of the indirect dispatch argument data is shown in Table 5.

Table 5. Argument data layout for indirect dispatch

Offset	Data type	Description
0x00	GR_UINT32	Number of thread groups in X direction
0x04	GR_UINT32	Number of thread groups in Y direction
0x08	GR_UINT32	Number of thread groups in Y direction

The indirect version of compute dispatch is available on both universal and compute queues.

Rendering Operations

An application renders graphics primitives using graphics pipelines and a currently bound command buffer graphics state. All parts of the state must be properly set for the rendering operation to produce the desired result. There are separate functions for rendering indexed and non-indexed geometry.

Non-indexed geometry can be rendered by calling the `grCmdDraw()` function for rendering both instanced and non-instanced objects. Indexed geometry can be rendered with `grCmdDrawIndexed()`. Indexed geometry can only be rendered when the valid index data memory is bound to the command buffer state with `grCmdBindIndexData()`. If objects are not instanced, `firstInstance` should be set to zero and the `instanceCount` parameters should be set to one.

> The vertex, index, and instance count cannot be zero.

The rendering operations are only valid for command buffers built for execution on universal queues.

Indirect Rendering

In addition to rendering geometry with application-supplied arguments, Mantle supports indirect draw functions whose execution is driven by data stored in GPU memory objects. Indirect rendering is performed by either calling the `grCmdDrawIndirect()` or `grCmdDrawIndexedIndirect()` function, depending on the presence of index data.

The draw argument data must be 4-byte aligned and the memory range containing the indirect data must be in the `GR_MEMORY_STATE_INDIRECT_ARG` state. The layout of the indirect draw argument data is shown in Table 6 and Table 7.

Table 6. Argument data layout for indirect draw

Offset	Data type	Description
0x00	GR_UINT32	Number of vertices per instance
0x04	GR_UINT32	Number of instances
0x08	GR_INT32	Vertex offset
0x0C	GR_UINT32	Instance offset

Table 7. Argument data layout for indexed indirect draw

Offset	Data type	Description
0x00	GR_UINT32	Number of indices per instance
0x04	GR_UINT32	Number of instances
0x08	GR_UINT32	Index offset
0x0C	GR_INT32	Vertex offset
0x10	GR_UINT32	Instance offset

Primitive Topology

Mantle supports a wide range of standard primitive topologies, along with tessellated patches, and special rectangle list primitives. Primitive topology is specified as a part of the graphics pipeline static state. See Graphics Pipeline State.

The rectangle list is a special geometry primitive type that can be used for implementing post-processing techniques or efficient copy operations. There are some special limitations for rectangle primitives. They cannot be clipped, must be axis aligned, and cannot have depth gradient. Failure to comply with these restrictions results in undefined rendering results.

Queries

Mantle supports occlusion and pipeline statistics queries. Occlusion queries are only available on universal queues, while pipeline statistics queries are available on universal and compute queues.

Queries in the Mantle API are managed using query pools – homogeneous collections of queries of a certain type. Query pools are represented by `GR_QUERY_POOL` object handles. The query type and number of query slots in a pool is specified at creation time. The query pools are created with `grCreateQueryPool()`.

Occlusion queries are used for counting the number of samples that pass the depth and stencil tests. They could be helpful when an application needs to determine visibility of a certain object. The result of an occlusion query can be accessed by the CPU to let the application make rendering decisions based on visibility.

Pipeline statistics queries can be used to retrieve shader execution statistics, as well as the number of invocations of some other fixed function parts of the geometry pipeline. Naturally, the compute queue statistics have only a compute related subset of statistics information available.

A query needs to be reset after creation and binding to memory, or if a query has already been used before. Failing to reset a query prior to use produces undefined results. To reset queries in a pool, an application uses `grCmdResetQueryPool()`. Multiple queries in a pool could be reset in just one reset call by specifying a contiguous range of query slots to reset.

> Resetting a range of queries in one operation is a lot more optimal than resetting individual query slots.

The query counts query-specific events between the `grCmdBeginQuery()` and `grCmdEndQuery()` commands embedded in the command buffer. The query commands can only be issued in command buffers that support queries of the given type.

The same query cannot be used in a command buffer more than once; otherwise, the results of the query are undefined. Also, the query cannot span more than a single command buffer and should be explicitly terminated before the end of a command buffer. Failing to properly terminate a query, by matching every `grCmdBeginQuery()` function call with a `grCmdEndQuery()`, results in an undetermined query result value, invalid query completion status, and could produce an undetermined rendering result. For example, calling `grCmdBeginQuery()` twice in a row matched by a single `grCmdEndQuery()` call, or matching a single `grCmdBeginQuery()` call with multiple `grCmdEndQuery()` is not allowed.

Occlusion queries support an optional `GR_QUERY_IMPRECISE_DATA` flag that could be used as an optimization hint by the GPU. If a flag is set, the query value is only guaranteed to be zero when no samples pass depth or stencil tests. In all other cases, the query returns some non-zero value.

An application retrieves results of any query in a pool by calling `grGetQueryPoolResults()`. One or multiple consecutive query results can be retrieved in a single function call. If any of the requested results are not yet available, which is indicated by the `GR_NOT_READY` return code, the returned data are undefined for all requested query slots. An application must ensure there is enough space provided to store results for all requested query slots. Calling `grGetQueryPoolResults()` with a `NULL` data pointer could be used to determine expected data size.

> To retrieve query results or to check for completion, the driver performs a memory map operation, which could be relatively expensive. If the application needs to perform a lot of frequent query checks, and memory assignment for query pool objects allows it, the query pool objects can be bound to pinned memory. This ensures expensive memory map operations are not performed.
>
> If the memory object bound to the query pool resides in local video memory while being referenced in queued command buffers, the memory object cannot be safely mapped on Windows® 7-8.1 platforms, and calls to grGetQueryPoolResults() lead to undefined results.

The results for an occlusion query are returned as a 64-bit integer value and pipeline statistics are returned in a `GR_PIPELINE_STATISTICS_DATA` structure.

TIMESTAMPS

For timing the execution of operations in command buffers, Mantle provides the ability to write GPU *timestamps* to memory from command buffers using `grCmdWriteTimestamp()` functions. The timestamps are 64-bit time values counted with a stable GPU clock, independent of the GPU engine or memory clock. To time a GPU operation, an application uses the difference between two timestamp values. The frequency of the timestamp clock is queried from the physical GPU information as described in GPU Identification and Initialization.

There are two types of locations in a pipeline from where the timestamp could be generated: *top of pipeline* and *bottom of pipeline*. The top of pipeline timestamp is generated immediately when the timestamp write command is executed, while the bottom of pipeline timestamp is written out when the previously launched GPU work has finished execution.

The timestamp destination memory offset for universal and compute queues has to be aligned to an 8-byte boundary. Other queue types might have different alignment requirements. Before a timestamp can be written out, the destination memory range has to be transitioned into the `GR_MEMORY_STATE_WRITE_TIMESTAMP` state using an appropriate preparation operation.

The bottom of pipeline timestamps are supported on universal and compute queues, while the top of pipeline timestamps are supported on universal queues only.

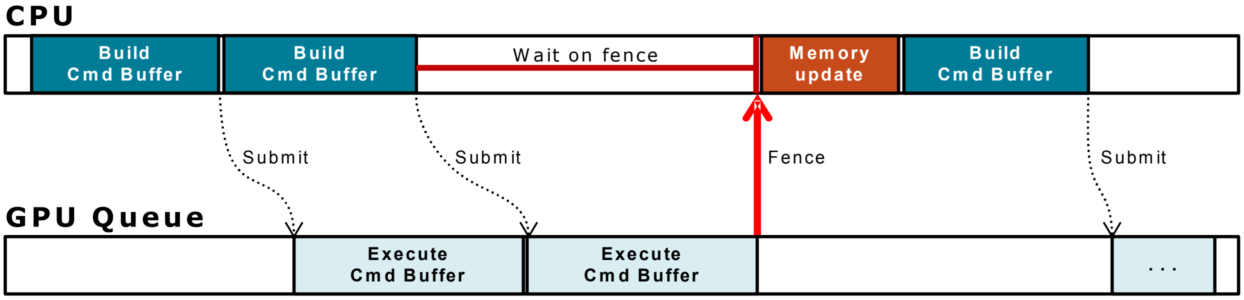
SYNCHRONIZATION

The Mantle API provides a comprehensive set of synchronization primitives to synchronize between a CPU and a GPU, as well as between multiple GPU queues.

COMMAND BUFFER FENCES

Command buffer fences provide a coarse level synchronization between a GPU and a CPU on command buffer boundaries by indicating completion of command buffer execution. Figure 4 demonstrates an example of a CPU waiting on a GPU fence before it performs a resource load operation.

Figure 4. Synchronization with fences



A fence object, represented by the GR_FENCE object handle, can be created by calling the grCreateFence() function and can optionally be attached to command buffer submissions as described in Command Buffer Submission.

Once a command buffer with a fence is submitted, the fence status can be checked with the grGetFenceStatus() function. If the fence has not been reached, the GR_NOT_READY code is returned to the application. An attempt to check the fence status before it is submitted returns a GR_ERROR_UNAVAILABLE error code. If a fence object has been used for the command buffer submission, it must not be reused or destroyed until the fence has been reached.

An application can also sleep one of its threads while waiting for a fence or a group of fences to come back by calling grWaitForFences(). If multiple fences are specified and the grWaitForFences() is instructed to wait for all fences, the function waits for all the fences to complete, otherwise any returned fence wakes an application thread. A timeout in seconds can be specified on the fence wait to prevent a thread from sleeping for excessive periods of time.

If an application receives a GR_ERROR_DEVICE_LOST error while waiting for a fence with grWaitForFences() or by periodically checking fence status, it should immediately stop waiting and proceed with appropriate error handling.

Events

Events in Mantle can be used for more fine-grain synchronization between a GPU and a CPU than fences, as an application could use events to monitor progress of the GPU execution inside of the command buffers. An event object can be set or reset by both the CPU and GPU, and its status can be queried by the CPU. The events in Mantle are represented by the GR_EVENT object handle.

Event objects are created by calling the grCreateEvent() function, and are set and reset by the CPU by using the grSetEvent() and grResetEvent() functions. From command buffers, the events are similarly manipulated using the grCmdSetEvent() and grCmdResetEvent() functions. Event operations are supported by both universal and compute queues.

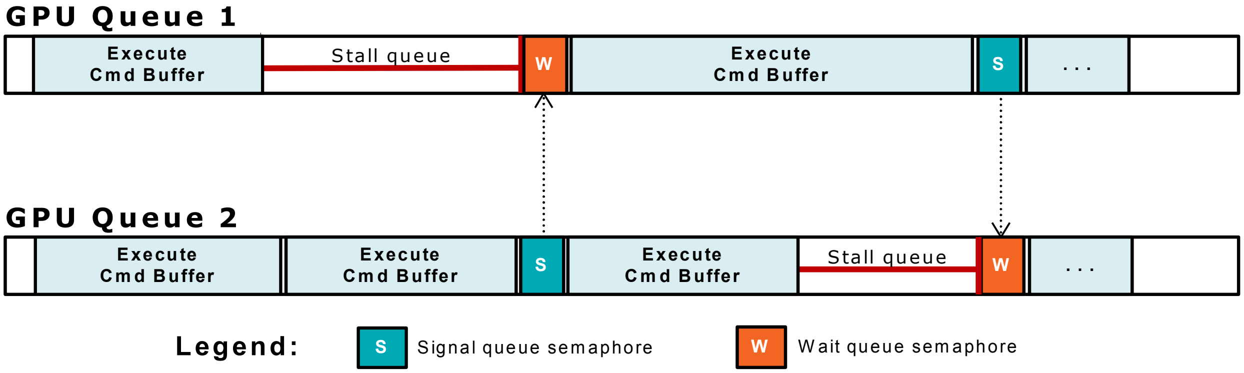
An application checks the event's state using the CPU by calling grGetEventStatus(). When created, the event starts in an undefined state, and it should be explicitly set or reset before it can be queried. If an application receives a GR_ERROR_DEVICE_LOST error while waiting for an event, it should immediately stop waiting and proceed with appropriate error handling.

To retrieve event status with the CPU, the driver performs a memory map operation, which could be relatively expensive. If the application needs to perform a lot of frequent event status checks, and memory assignment for event objects allows it, the event objects can be bound to pinned memory. This ensures expensive memory map operations are not performed.

Queue Semaphores

Queue semaphores are used to synchronize command buffer execution between multiple queues and between capable GPUs in multi-GPU configurations. See Queue Semaphore Sharing for a discussion on synchronization in multi-GPU configurations. The semaphores are also used for synchronizing virtual allocation remapping with other GPU operations. The following figure shows an example of synchronization between queues to guarantee a required order of execution.

Figure 5. Queue synchronization with semaphores



Queue semaphore objects are represented by GR_QUEUE_SEMAPHORE object handles and are created by calling grCreateQueueSemaphore(). At creation time, an application can specify an initial semaphore count that is equivalent to signaling the semaphore that many times.

An application issues *signal* and *wait* semaphore operations on the queues by calling grSignalQueueSemaphore() and grWaitQueueSemaphore() functions. It is an application's responsibility to ensure proper matching of signals and waits. In the case where a queue is stalled for excessive periods of time, the debug infrastructure is able to detect a timeout condition and reports an error to the application.

For performance reasons, it is recommended to ensure *signal* is issued before *wait* on the Windows® platform.

Draining Queues

For some operations, it might be required to ensure a particular queue, or even all of the device queues, are completely drained before proceeding. The Mantle API provides the functions `grQueueWaitIdle()` and `grDeviceWaitIdle()` to stall and wait for the queues to drain. These functions are not thread safe, and all submissions and other API operations must be suspended while waiting for idle. The `grDeviceWaitIdle()` function waits for all queues to fully drain and virtual memory remapping operations to complete.

> For performance reasons, it is recommended to avoid draining queues unless absolutely necessary.

Queue Memory Atomics

The Mantle GPU is capable of executing memory atomics operating on 32-bit and 64-bit integers from the command buffer, similar to how memory atomic operations are performed in shaders. Besides synchronization, atomics can be used to perform some arithmetic operations on memory values directly from GPU queues. The memory location operated on by an atomic operation is provided by the memory object, and the application is responsible for issuing appropriate memory preparation operations. The memory range for the queue atomic operation must be in the `GR_MEMORY_STATE_QUEUE_ATOMIC` state.

An atomic operation can be recorded in a command buffer using `grCmdMemoryAtomic()`. The memory offset for an atomic location has to be aligned to 4-bytes for 32-bit integer atomics and 8-bytes for 64-bit atomics. The 32-bit atomic operations use the lower 32-bits of the literal value provided in the source data argument. Atomic operations performed on unaligned addresses cause undefined results.

Shader Atomic Counters

The Mantle shader model exposes atomic counters that could be used for implementing unordered data queues using atomic increment and decrement operations. The atomicity of operations guarantees that no two shader threads see the same counter value returned. The underlying counter is 32-bits, representing a [0, 2^{32-1}] range of values. Going outside of this value range causes the counter to wrap. The atomic counters in Mantle are independent from images and other API objects.

Each universal and compute queue has some number of independent atomic counter resources per pipeline type. There are guaranteed to be at least 64 atomic counters per pipeline type for universal queues, but for other queue types, the atomic counters are optional and may be zero.

The number of available atomic counters is queried in the queue properties as described in Queues.

> Before using atomic counters, an application should query a queue's properties to confirm the number of available counter slots.

Atomic counters are referenced by a slot number varying from 0 to the number of available atomic counters for that queue minus one. If a number of counters reported for a particular queue is zero, atomic counters cannot be used in any of the shaders used by compute or graphics workloads executing on that queue. Attempting to use atomic counters outside of the available counter slot range results in undefined behavior.

Atomic counter values are not preserved across command buffer boundaries, and it is an application's responsibility to initialize the counters to a known value before the first use, and later save them off to memory if necessary.

Before accessing it from a shader, an atomic counter should be initialized to a specific value by loading data with `grCmdInitAtomicCounters()` or by copying the data from a memory object using `grCmdLoadAtomicCounters()`. An atomic counter value could also be saved into a memory location using `grCmdSaveAtomicCounters()`.

The GPU memory offsets for loading and storing counters have to be aligned to a 4-byte boundary. The source and destination memory for the counter values have to be in the `GR_MEMORY_STATE_DATA_TRANSFER` state before issuing the load or save operation.

Chapter IV.

Resource Objects and Views

The Mantle GPU operates on data stored in memory objects. There are several ways the data can be accessed depending on its intended use. Texture and render target data are represented by *image* objects and are accessed from shader and pipeline back-end using appropriate *views*. Many other operations work directly on raw data stored in memory objects, and shader access to raw memory is performed through *memory views*.

Memory Views

A buffer-like access to raw memory from shaders is performed using *memory views*. There are no objects in the Mantle API representing them due to the often dynamic nature of such data. Shader memory views describe how raw memory is interpreted by the shader and are specified during *descriptor set* construction (see Resource Shader Binding) or bound dynamically using *dynamic memory views* (see Dynamic Memory View).

A memory view describes a region of memory inside of the memory object that is made accessible to a shader. Additionally, memory view specifies how the shader sees and interprets the raw data in memory: a format and element stride could be specified. The memory view is defined by the `GR_MEMORY_VIEW_ATTACH_INFO` structure.

Interpretation of memory view data depends on a combination of view parameters and shader instructions used for data access. Here are the rules for setting up memory views for different shader instruction types:

- For *typed buffer* shader instructions, the format has to be valid and stride has to be equal to the format element size.
- For *raw buffer* shader instructions, the format is irrelevant and the stride has to be equal to one.
- For *structured buffer* shader instructions, the format is irrelevant and the stride has to be equal to the structure stride. Specifying zero stride makes shader access the first structure stored in memory, regardless of the specified index. The actual structure or type of the data is expressed inside of the shader.

Memory view offset, as well as the data accessed in the shader, must be aligned to the smaller of the fetched element size or the 4-byte boundary. Memory accesses outside of the memory view boundaries or unaligned accesses produce undefined results. It is an application's responsibility to avoid out of bounds memory access.

IMAGES

Images in Mantle are containers used to store texture data. They are also used for color render targets and depth-stencil buffers.

Unlike many other graphic APIs, where image objects refer to the actual data residing in video memory along with meta-data describing how that data are to be interpreted by the GPU, Mantle decouples the storage of the image data and the description of how the GPU is supposed to interpret it. Data storage is provided by memory objects, while Mantle images are just CPU side objects that reference the data in memory objects and store information about data layout and their other properties. With this approach, developers are able to manage video memory more efficiently.

An image is composed of 1D, 2D or 3D *subresources* containing texels organized in a layout that depends on the type of image tiling selected, as well as other image properties. At image creation time, a texel format is specified for the purpose of determining the storage requirements; however, it can be overwritten later with a compatible format at view creation time. The image dimensions are specified in texels for the topmost mipmap level for all image formats. This applies to compressed images as well. The size of compressed images must be a multiple of the compression block size.

An image of any supported type is created by calling `grCreateImage()`. All appropriate usage flags are set at creation time and must match the expected image usage. For images that are not intended for view creation and used for data storage only (e.g., data transfer), it is allowed to omit all usage flags.

The application should specify a minimal set of image usage flags. Specifying extra flags might result in suboptimal performance.

Once an image object is created, an application queries its memory requirements at run-time. The video memory requirements include the memory needed to store all *subresources,* as well as *internal image meta-data.* An application either creates a new memory object for the image data, or sub-allocates a memory block from an existing memory object if the memory size allows. Before an image is used, it should be bound to an appropriate memory object and, if necessary, cleared and prepared according to the intended use.

Image Organization and Subresources

The following image types are natively supported in Mantle:

- 1D images
- 2D images
- 3D images

Along with the *image views*, these types are used to represent all supported images, including cubemaps and image arrays.

Image objects are composed of one or more *subresources* – image array slices, mipmap levels, etc. – that vary based on the resource type and dimensions. A subresource within an image is referenced by a descriptor defined as a `GR_IMAGE_SUBRESOURCE` structure. Some operations can be performed on a contiguous range of image subresources. Such a subresource range is represented by a `GR_IMAGE_SUBRESOURCE_RANGE` structure.

Image Aspects

Some images could have multiple components: depth, stencil, or color. Each of these components is represented by an *image aspect*. Each such image component or image aspect is logically represented by its own set of subresources. The image aspects are described by values in a `GR_IMAGE_ASPECT` enumeration.

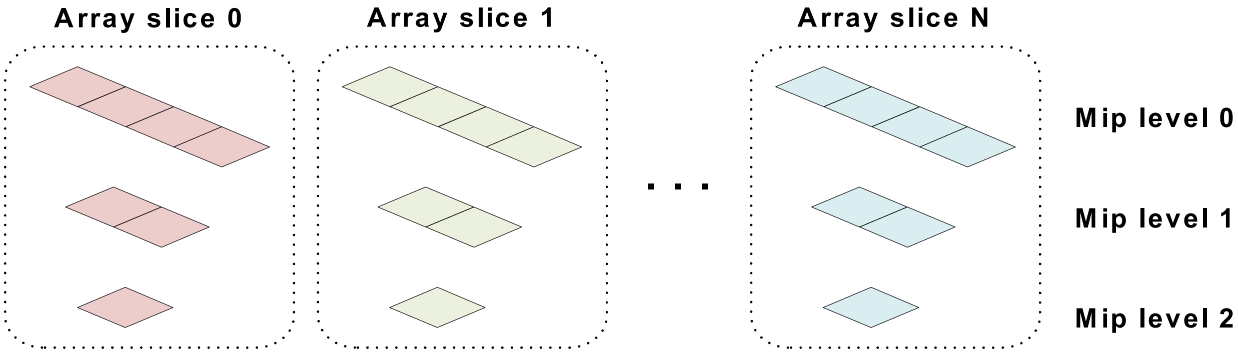
While some operations might refer to images in their entirety, some operations require specification of a particular image aspect. For example, rendering to a depth-stencil image uses the entire set of aspects (in this case, depth, and stencil), while a specific aspect is specified to access depth or stencil image data from a shader.

1D IMAGES

1D image type objects can store 1D images or 1D image arrays, with or without mipmaps. 1D images cannot be multisampled and cannot use block compression formats.

An example of 1D image array organization is shown in Figure 6.

Figure 6. 1D image organization

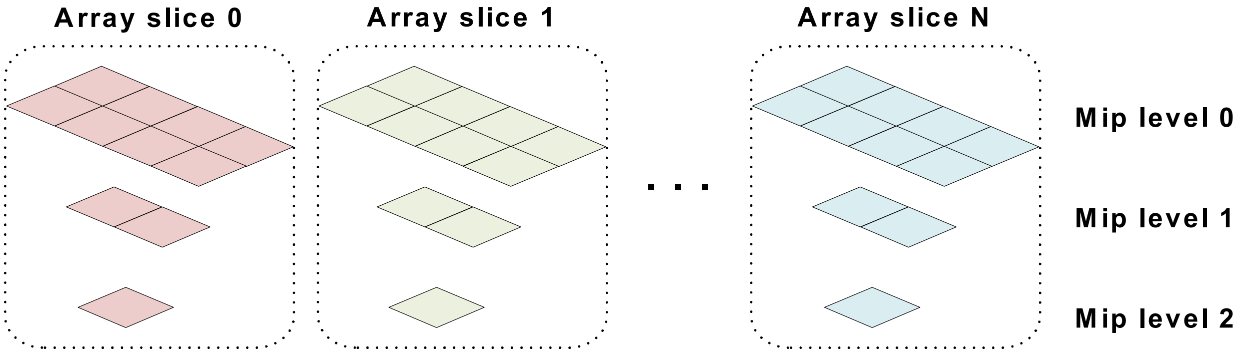


2D IMAGES

2D image type objects can store 2D images, 2D image arrays, cubemaps, color targets, and depth-stencil targets, including multisampled targets. Multisampled 2D images cannot have mipmap chains.

An example of 2D image array organization is shown in Figure 7.

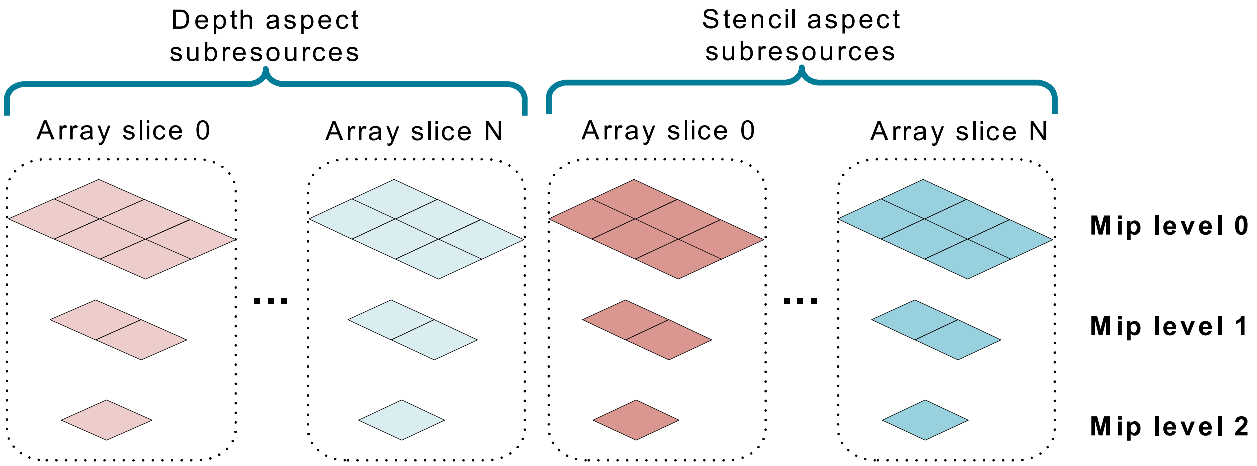
Figure 7. 2D image organization



2D images used as depth-stencil targets have separate subresources for its depth and stencil aspects. For GPUs that do not support separate depth and stencil image aspect storage, the same memory offsets might be reported for depth and stencil subresources.

An example of depth-stencil image organization is shown in Figure 8.

Figure 8. Depth-stencil image organization



Cubemaps

Cubemap images are a special case of 2D image arrays. From the storage perspective, cubemaps are essentially 2D image arrays with 6 slices. Arrays of cubemaps are also 2D image arrays, with a number of slices equal to 6 times the number of cubemaps. The cubemap slices have to be square in terms of their dimensions. Cubemap images cannot be multisampled.

The slice number within a cubemap or a cubemap array can be computed as follows:

```
slice = 6 * cube_array_slice + faceID
```

The cubemap face IDs and their orientation are listed in the following table.

Table 8. Cubemap face ID decoding from face orientation

Direction	Face ID
Positive X	0
Negative X	1
Positive Y	2
Negative Y	3
Positive Z	4
Negative Z	5

3D Images

3D image type objects can only store volume textures, and like other types of images, can contain mipmaps. 3D images cannot be multisampled or created as arrays.

In 3D images, each subresource represents a mipmapped volume starting with the topmost mipmap level. An example 3D image organization is show in the Figure 9.

Figure 9. 3D image organization

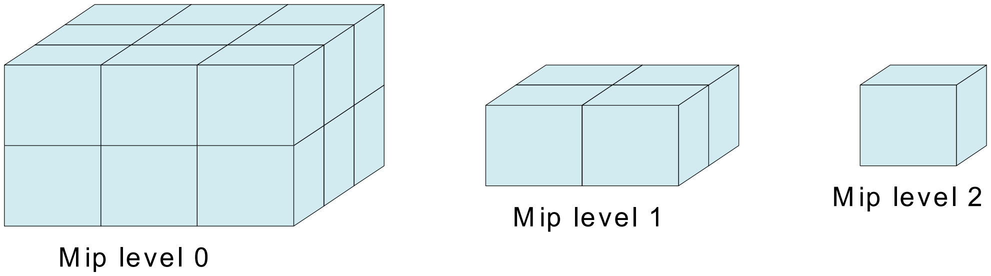


Image Tiling and Image Data Organization

There are several options available for internal image texel organization. In *linear tiling*, the texels are stored linearly within an image row and the image width is padded to a required pitch. While simple and efficient for CPU access, the linear tiling does not play well with a GPU memory system. For the highest GPU performance, an *optimal tiling* should be used. The internal implementation of the optimal tiling could vary depending on the image type and usage. The only reliable way to upload to or download data from optimally tiled images is to copy their data to and from linearly tiled images that could be directly accessed by the CPU. Image tiling types are defined in `GR_IMAGE_TILING`.

Some image operations can only be performed on images of certain tiling. An application should check format capabilities for the tiling of interest to verify the tiling type is supported for the operations with the intended image usage.

In Mantle, depending on the resource type and usage, images are broadly classified as *transparent* or *opaque,* in terms of their data layout. Transparent images are non-target images with linear tiling. Memory contents of these images can be directly accessed by the CPU as the data layout is well defined. Opaque images, while technically accessible by the CPU in a raw form, do not make any guarantees about the data layout. Opaque images are the optimally tiled images, as well any target images (e.g., color targets, depth-stencil targets, and multisampled images). The primary use for the transparent images is data transfer to and from the GPU.

Retrieving Image Properties

When it comes to support of images in Mantle, different GPUs might expose slightly different capabilities. For example, different maximum number of array slices or maximum dimensions could be supported. Additionally, GPUs might have different maximum alignment requirements for images. In some cases, knowledge of maximum alignment requirements could be used to simplify memory management for images. An application retrieves image properties for a GPU by using the `grGetGpuInfo()` function with the information type parameter set to

GR_INFO_TYPE_PHYSICAL_GPU_IMAGE_PROPERTIES. Image properties are returned in the GR_PHYSICAL_GPU_IMAGE_PROPERTIES structure.

Resource Formats and Capabilities

The *resource format* is used for specifying an image element type and memory view element type for shader access. It is specified using a GR_FORMAT format descriptor that contains information about the *numeric format* and the *channel format*. The numeric format describes how the data are to be interpreted while the channel specification describes the number of channels and their bit depth. The GR_NUM_FMT_DS numeric format is a special case format used specifically for creating depth and stencil images.

The channel layout in memory is specified in this particular order: R, G, B, A, with the leftmost channel stored at the lowest address. The exceptions are the compressed formats that have a different encoding scheme per block, and formats with an alternative channel ordering which are used to handle certain OS-specific interoperability issues, such as GR_CH_FMT_B5G6R5 and GR_CH_FMT_B8G8R8A8.

Not all channel and numeric format combinations are valid, and only a subset of them can be used for color and depth-stencil targets. An application can query format capabilities using grGetFormatInfo(). A separate set of capabilities is reported for linear and optimal tiling modes in the GR_FORMAT_PROPERTIES structure. Refer to Appendix D. Format Capabilities for a summary of basic format capabilities guaranteed for all Mantle devices.

Listing 9. Example of querying format properties

```
GR_RESULT result;
// Setup RGBA8 normalized format
GR_FORMAT fmt = {};
fmt.channelFormat = GR_CH_FMT_R8G8B8A8;
fmt.numericFormat = GR_NUM_FMT_UNORM;
// Retrieve format properties
GR_FORMAT_PROPERTIES fmtInfo = {};
GR_SIZE infoSize = sizeof(GR_FORMAT_PROPERTIES);
result = grGetFormatInfo(device, fmt, GR_INFO_TYPE_FORMAT_PROPERTIES,
                         &infoSize, &fmtInfo);
// Check support flags
bool supportsLinearConversion =
    (fmtInfo.linearTilingFeatures & GR_FORMAT_CONVERSION) != 0;
bool supportsTiledMsaa =
    (fmtInfo.optimalTilingFeatures & GR_FORMAT_MSAA_TARGET) != 0;
```

If no capabilities are reported for a given combination of channel format and numeric format, that format is unsupported. For formats with multisampling capabilities, more detailed support of multisampling can be validated as described in Multisampled Images.

Compressed Images

Compressed images are the images that use block compression channel formats (GR_CH_FMT_BC1 through GR_CH_FMT_BC7). Compressed images have several notable differences that an application should properly handle:

- Image creation size is specified in texels, but size for copy operations is specified in compression blocks.
- Compressed images can only use optimal tiling. Since linear tiling cannot be used for compressed images, their uploads should use non-compressed formats of the texel size equivalent to the block compression size.

Multisampled Images

Depth-stencil and color targets can be created as multisampled 2D images. A more fine control of image multisampling options on AMD platforms can be performed through the Advanced Multisampling extension.

An application can check multisampled image support for various combinations of samples and other image creation parameters by attempting to create a multisampled image. The image creation is lightweight enough to not cause any performance concerns for performing these checks.

Image Views

Image objects cannot be directly accessed by pipeline shaders for reading or writing image data. Instead, *image views* representing contiguous ranges of the image subresources and containing additional meta-data are used for that purpose. Views can only be created for images of compatible types and should represent a valid subset of image subresources. The resource usage flags should have GR_IMAGE_USAGE_SHADER_ACCESS_READ and/or GR_IMAGE_USAGE_SHADER_ACCESS_WRITE set for successful creation of image views of all types. If image view overwrites image format, the image should be created with the GR_IMAGE_CREATE_VIEW_FORMAT_CHANGE flag.

The types of the image views for shader access that can be created are listed below:

- 1D image view
- 1D image array view
- 2D image view
- 2D image array view

- 2D MSAA image view
- 2D MSAA image array view
- Cubemap view
- Cubemap array view
- 3D image view

An image view is created by calling `grCreateImageView()`. The exact image view type is partially implicit, based on the resource characteristics — resource type, multisampling settings, and the number of array slices — as well as the view creation parameters. Some of the image creation parameters are inherited by the view.

Listing 10. Example of creating read-only image and image view

```
GR_RESULT result;
// Setup 256x256 image with 3 mip levels
GR_IMAGE_CREATE_INFO imgInfo = {};
imgInfo.extent.width  = 256;
imgInfo.extent.height = 256;
imgInfo.extent.depth  = 1;
imgInfo.format.channelFormat = GR_CH_FMT_R8G8B8A8;
imgInfo.format.numericFormat = GR_NUM_FMT_UNORM;
imgInfo.imageType     = GR_IMAGE_2D;
imgInfo.arraySize     = 1;
imgInfo.mipLevels     = 3;
imgInfo.samples       = 1;
imgInfo.tiling        = GR_OPTIMAL_TILING;
imgInfo.usage         = GR_IMAGE_USAGE_SHADER_ACCESS_READ;
// Create image
GR_IMAGE image = GR_NULL_HANDLE;
result = grCreateImage(device, &imgInfo, &image);
// Image should have memory bound before creating a view
CreateAndBindImageMemory(device, image);
// Setup shader access image view
GR_IMAGE_VIEW_CREATE_INFO viewInfo = {};
viewInfo.image      = image;
viewInfo.viewType   = GR_IMAGE_VIEW_2D;
viewInfo.format.channelFormat = GR_CH_FMT_R8G8B8A8;
viewInfo.format.numericFormat = GR_NUM_FMT_UNORM;
viewInfo.channels.r = GR_CHANNEL_SWIZZLE_R;
viewInfo.channels.g = GR_CHANNEL_SWIZZLE_G;
viewInfo.channels.b = GR_CHANNEL_SWIZZLE_B;
viewInfo.channels.a = GR_CHANNEL_SWIZZLE_A;
viewInfo.subresourceRange.aspect         = GR_IMAGE_ASPECT_COLOR;
viewInfo.subresourceRange.baseMipLevel   = 0;
viewInfo.subresourceRange.mipLevels      = GR_LAST_MIP_OR_SLICE;
viewInfo.subresourceRange.baseArraySlice = 0;
viewInfo.subresourceRange.arraySize      = GR_LAST_MIP_OR_SLICE;
viewInfo.minLod     = 0.0f;
// Create image view
GR_IMAGE_VIEW view;
result = grCreateImageView(device, &viewInfo, &view);
```

The Table 9 describes required image and view creation parameters compatible with shader resources of different types. Attempting to create a view with image formats or image types incompatible with the parent image resource fails view creation.

Table 9. Image and image view parameters for shader resource types

Shader resource type	Image creation parameters	Image view creation parameters

1D image	imageType = 1D
	width >= 1
	height = 1
	depth = 1
	arraySize = 1
	samples = 1
		viewType = 1D
		baseArraySlice = 0
		arraySize = 1

1D image array	imageType = 1D
	width >= 1
	height = 1
	depth = 1
	arraySize > 1
	samples = 1
		viewType = 1D
		baseArraySlice >= 0
		arraySize > 1

2D image	imageType = 2D
	width >= 1
	height >= 1
	depth = 1
	arraySize >= 1
	samples = 1
		viewType = 2D
		baseArraySlice >= 0
		arraySize = 1

2D image array	imageType = 2D
	width >= 1
	height >= 1
	depth = 1
	arraySize > 1
	samples = 1
		viewType = 2D
		baseArraySlice >= 0
		arraySize > 1

2D MSAA image	imageType = 2D
	width >= 1
	height >= 1
	depth = 1
	arraySize = 1
	samples > 1
		viewType = 2D
		baseArraySlice = 0
		arraySize = 1

Shader resource type	Image creation parameters	Image view creation parameters
2D MSAA image array	imageType = 2D width >= 1 height >= 1 depth = 1 arraySize > 1 samples > 1	viewType = 2D baseArraySlice >= 0 arraySize > 1

Cubemap image	imageType = 2D
	width >= 1
	height = width
	depth = 1
	arraySize = 6
	samples = 1
		viewType = CUBE
		baseArraySlice = 0
		arraySize = 1

Cubemap image array	imageType = 2D
	width >= 1
	height = width
	depth = 1
	arraySize = 6*N
	samples = 1
		viewType = CUBE
		baseArraySlice >= 0
		arraySize = N

3D image	imageType = 3D
	width >= 1
	height >= 1
	depth >= 1
	arraySize = 1
	samples = 1
		viewType = 3D
		baseArraySlice = 0
		arraySize = 1

The number of mipmap levels and array slices has to be a subset of the subresources in the parent image. If the application wants to use all mipmap levels or slices in an image, the number of mipmap levels or slices can be set to a special value of GR_LAST_MIP_OR_SLICE without knowing the exact number of mipmap levels or slices.

It is an application's responsibility to correctly use image views based on the supported image format capabilities and usage flags requested at image creation time. For example, attempting to write to a resource of GR_CH_FMT_R4G4 or compressed format from a shader results in undefined behavior. Similarly, attempting to write to an image that did not have the GR_IMAGE_USAGE_SHADER_ACCESS_WRITE flag specified on image creation results in undefined behavior.

An image view specifies image channel remapping in `channels` member of the `GR_IMAGE_VIEW_CREATE_INFO` structure that can be used to swizzle the channel data on shader access. This swizzling applies to both image read and write operations.

Render Targets

In Mantle there are two different types of render targets:

- Color targets (render targets)
- Depth-stencil render targets

Color Targets

Color targets are 2D or 3D image objects created with the `GR_IMAGE_USAGE_COLOR_TARGET` object usage flag that designates them as color targets. An image cannot be designated as both a color target and a depth-stencil target.

Images cannot be directly bound as color targets, but rather their *color target views* are used for that purpose. A color target view is created by calling `grCreateColorTargetView()`. A color target view can represent a contiguous range of image array slices at any particular mipmap level. A color target view cannot reference multiple mipmap levels.

A variety of different formats is supported for color render targets. A valid image format must be specified for the color target view. It can be different from the image format, provided the view format is compatible with the format of the parent image and the image is created with the `GR_IMAGE_CREATE_VIEW_FORMAT_CHANGE` flag.

Listing 11. Example of creating color target image and view

```
GR_RESULT result;
// Setup 1000x800 image for color target use
GR_IMAGE_CREATE_INFO imgInfo = {};
imgInfo.extent.width  = 1000;
imgInfo.extent.height = 800;
imgInfo.extent.depth  = 1;
imgInfo.format.channelFormat = GR_CH_FMT_R8G8B8A8;
imgInfo.format.numericFormat = GR_NUM_FMT_UNORM;
imgInfo.imageType     = GR_IMAGE_2D;
imgInfo.arraySize     = 1;
imgInfo.mipLevels     = 1;
imgInfo.samples       = 1;
imgInfo.tiling        = GR_OPTIMAL_TILING;
imgInfo.usage         = GR_IMAGE_USAGE_COLOR_TARGET;
// Create image
GR_IMAGE image = GR_NULL_HANDLE;
result = grCreateImage(device, &imgInfo, &image);
// Image should have memory bound before creating a view
CreateAndBindImageMemory(device, image);
// Setup color target view
GR_COLOR_TARGET_VIEW_CREATE_INFO viewInfo = {};
viewInfo.image          = image;
viewInfo.arraySize      = 1;
viewInfo.baseArraySlice = 0;
viewInfo.mipLevel       = 0;
viewInfo.format.channelFormat = GR_CH_FMT_R8G8B8A8;
viewInfo.format.numericFormat = GR_NUM_FMT_UNORM;
// Create color target view
GR_COLOR_TARGET_VIEW view = GR_NULL_HANDLE;
result = grCreateColorTargetView(device, &viewInfo, &view);
```

A color target image can be accessed from shaders by creating appropriate image views, provided the image has necessary shader access flags and the formats are compatible.

Depth-Stencil Targets

The *depth-stencil targets* are represented by depth-stencil views created from a 2D image marked with the `GR_IMAGE_USAGE_DEPTH_STENCIL` usage flag, and could be created as depth-only, stencil-only, and depth-stencil. The depth formats supported are 16-bit integer and 32-bit floating point formats, while stencil only supports the 8-bit integer format. It is allowed to mix stencil with any of the supported depth formats. An image cannot be designated as both a color target and a depth-stencil target.

Images cannot be directly bound as depth-stencil targets, but rather their *depth-stencil views* need to be created for that purpose. A depth-stencil view is created by calling `grCreateDepthStencilView()`.

Listing 12. Example of creating depth-stencil image and view

```
GR_RESULT result;
// Setup 1000x800 image for depth buffer use
GR_IMAGE_CREATE_INFO imgInfo = {};
imgInfo.extent.width  = 1000;
imgInfo.extent.height = 800;
imgInfo.extent.depth  = 1;
imgInfo.format.channelFormat = GR_CH_FMT_R32;
imgInfo.format.numericFormat = GR_NUM_FMT_DS;
imgInfo.imageType     = GR_IMAGE_2D;
imgInfo.arraySize     = 1;
imgInfo.mipLevels     = 1;
imgInfo.samples       = 1;
imgInfo.tiling        = GR_OPTIMAL_TILING;
imgInfo.usage         = GR_IMAGE_USAGE_DEPTH_STENCIL;
// Create image
GR_IMAGE image = GR_NULL_HANDLE;
result = grCreateImage(device, &imgInfo, &image);
// Image should have memory bound before creating a view
CreateAndBindImageMemory(device, image);
// Setup depth-stencil view
GR_DEPTH_STENCIL_VIEW_CREATE_INFO viewInfo = {};
viewInfo.image          = image;
viewInfo.arraySize      = 1;
viewInfo.baseArraySlice = 0;
viewInfo.mipLevel       = 0;
viewInfo.flags          = 0;
// Create color target view
GR_DEPTH_STENCIL_VIEW view = GR_NULL_HANDLE;
result = grCreateDepthStencilView(device, &viewInfo, &view);
```

A depth-stencil target image can be accessed from shaders by creating appropriate image views, provided the image has necessary shader access flags and formats are compatible. Table 10 lists all supported depth-stencil formats and underlying storage formats for depth and stencil aspects.

Table 10. Depth-stencil image formats

Image format (channel/numeric format)	Depth aspect format (channel/numeric format)	Stencil aspect format (channel/numeric format)
GR_CH_FMT_R8 / GR_NUM_FMT_DS	N/A	GR_CH_FMT_R8 / GR_NUM_FMT_UINT
GR_CH_FMT_R16 / GR_NUM_FMT_DS	GR_CH_FMT_R16 / GR_NUM_FMT_UINT	N/A
GR_CH_FMT_R32 / GR_NUM_FMT_DS	GR_CH_FMT_R32 / GR_NUM_FMT_FLOAT	N/A
GR_CH_FMT_R16G8 / GR_NUM_FMT_DS	GR_CH_FMT_R16 / GR_NUM_FMT_UINT	GR_CH_FMT_R8 / GR_NUM_FMT_UINT

Image format (channel/numeric format)	Depth aspect format (channel/numeric format)	Stencil aspect format (channel/numeric format)
GR_CH_FMT_R32G8 / GR_NUM_FMT_DS	GR_CH_FMT_R32 / GR_NUM_FMT_FLOAT	GR_CH_FMT_R8 / GR_NUM_FMT_UINT

Separate from depth-stencil target views are image views that allow shaders to read depth-stencil target data. Only a single aspect (depth or stencil) can be accessed by the shader through image view at a time.

Target Binding

All provided color targets and depth-stencil targets are simultaneously bound to the command buffer state with `grCmdBindTargets()`. It is not required for all target information to be present for binding. Specifying the `NULL` target information unbinds previously bound targets, leaving them unbound until the next call to `grCmdBindTargets()`. All targets have to match graphics pipeline expectations at the time of the draw call execution following the state binding.

Along with target views, an application specifies the per target image state that represents the expected state for all subresources in the view at the draw time. For the depth-stencil view, a separate state is specified for depth and stencil aspects. The depth and stencil states could be different (e.g in the case of read-only depth or stencil. For unused color targets, as well as for unused depth-stencil aspects, an application should specify the `GR_IMAGE_STATE_UNINITIALIZED` state.

Targets of different sizes can be simultaneously bound; however, it is required that a scissor is enabled and restricts render target access to the smallest of the bound targets. Specifying scissor larger than the smallest target or disabling scissor while binding multiple targets of different sizes results in undefined behavior.

Read-only Depth-stencil Views

Read-only depth-stencil view allows rendering with read-only access to the depth or stencil aspect of an image while it is also used for read access from the graphics pipeline shaders. Only one of the depth or stencil aspects can be designated as read-only, but not both at the same time. The read-only depth in a view is indicated by the `GR_DEPTH_STENCIL_VIEW_CREATE_READ_ONLY_DEPTH` flag and the read-only stencil is indicated by the `GR_DEPTH_STENCIL_VIEW_CREATE_READ_ONLY_STENCIL` flag at depth-stencil creation time.

If the depth or stencil aspect is used for simultaneous read access as a depth-stencil target and as an image view from the graphic shaders, it has to be in the `GR_IMAGE_STATE_TARGET_AND_SHADER_READ_ONLY` image state. The image subresources in a read-only depth-stencil view that are read from shaders should be transitioned to that state, as well as this state should be used for binding image view and appropriate aspect for the depth-stencil target.

View Format Compatibility

An image view or color target view can be created with a format different from the original image format if the image is created with the `GR_IMAGE_CREATE_VIEW_FORMAT_CHANGE` flag. The formats are compatible when they have the same texel *bit-depth*. Compressed formats for image views are compatible with uncompressed formats of the texel bit-depth equal to the compressed image block size.

To verify a particular view format is compatible with a given image resource, an application attempts to create a view with the desired format. The view creation is lightweight enough not to cause any performance concerns for the compatibility checks.

Data Feedback Loop

There is the possibility that the same memory range, an image, or its views could be bound to multiple parts of the pipeline for both read and output operations. An example would be an image simultaneously bound for render target output and texture fetch, or a memory range bound for index fetch while it is output from one of the pipeline shaders to a writable memory view. This causes data feedback loops in the pipeline that can compromise integrity of the data. The validation layer is capable of catching a number of feedback conditions; however, under normal operation, the driver performs no checks and it is the developer's responsibility to avoid creating any data feedback loops. Results are undefined in such cases.

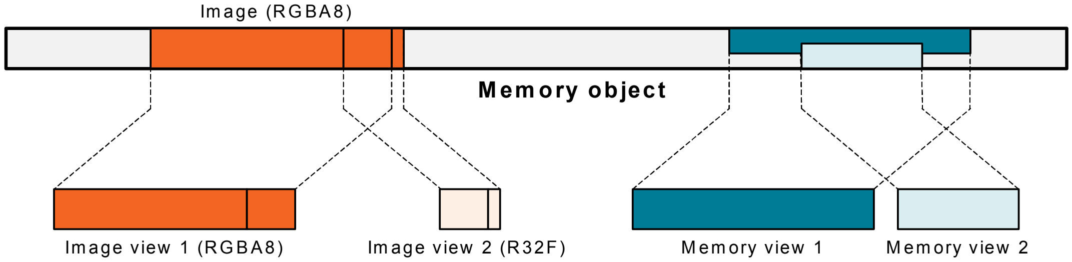
Resource Aliasing

With the flexible memory management in Mantle, it might be tempting to alias memory regions or images by associating them with the same memory location. Aliasing of raw memory or memory views is allowed and is encouraged as a means of sharing data, saving memory, and reducing memory copy operations. The subresources of transparent images (i.e., non-target images with linear tiling) can also be aliased in memory. From this perspective, transparent images behave similarly to memory views due to well defined data layout.

Different rules apply to opaque images. Because of hidden resource meta-data, tiling restrictions, and a possibility for introducing hard-to-track errors, it is illegal to directly alias opaque images. An application should use views to perform compatible format conversions for those images. The validation layer in the driver detects cases of aliased opaque images and reports an error. To avoid triggering this error when reusing memory for multiple image resources accessed at different times, the application must unbind memory from one image before rebinding it to the other.

Figure 10 demonstrates examples of allowed memory view aliasing and image reinterpretation through views.

Figure 10. Examples of data aliasing in Mantle



No assumption about preserving memory contents should be made when reusing memory between multiple target images (e.g., depth-stencil targets, color render targets, including multisampled images), and the application should perform proper preparation to initialize newly memory-bound target image resources.

One has to be careful about tracking memory and image state dependencies and properly handling their preparation (see Resource States and Preparation) when aliasing memory or using overlapping memory ranges for different purposes.

Memory view aliasing could be the source of a data feedback loop when multiple aliased views or memory ranges are simultaneously bound to the graphics pipeline for both output and read operations (also see Data Feedback Loop). The consistency of data in that case cannot be guaranteed and results are undefined.

Invariant Image Data

For non-target images, the memory contents are preserved after unbinding memory if the image is in the `GR_IMAGE_STATE_DATA_TRANSFER` state. Rebinding the same non-target image object to the previously used memory location preserves image contents. This generally is not true for binding image objects to image data left in memory from other image objects. Reusing image memory contents can be accomplished by using the `GR_IMAGE_FLAG_INVARIANT_DATA` flag. Creating a new image with exactly the same parameters and memory binding as an old image provides initial memory contents equivalent to the old image if the `GR_IMAGE_FLAG_INVARIANT_DATA` flag is specified at image creation time for both the old and new image object.

Resource States and Preparation

When the GPU accesses a resource, the accessed memory range or image is assumed to be in a particular *state* that must match the GPU expectations for its behavior, with respect to the type of access, cache residency, state of the resource meta-data, and so on. Lack of consistency between the actual memory or image state and the GPU expectations produces incorrect results. For performance reasons, the Mantle driver does not keep track of the persistent memory or image state, nor does it track hazard conditions. With Mantle, it becomes an application's responsibility to track memory and image states, and ensure their consistency with operations performed by the GPU. The application indicates when there is a change of memory or image state. For some operations, an application must also communicate to the driver the current state at the time of performing the operation.

In Mantle, the memory and image state is expressed in terms of the resource usage. The resource state represents where an image or memory can be bound, what operations can be performed on it, and provides abstracted hints for the internal resource representation. The application transitions memory and images from one state to another to indicate the change in the GPU usage of applicable resources.

Memory and Image States

There are separate states for memory and images, as they are representative of different usage and resource bind points. The memory states, represented by GR_MEMORY_STATE values, are used for memory regions directly accessed by the GPU and for memory views accessed from shaders. The image states represented by GR_IMAGE_STATE values are specially used for tracking not only memory state, but also internal image meta-data states for images. The image state can be thought of as a superset of memory state, and no separate memory state needs to be tracked for the memory associated with an image object.

When binding memory, memory views, or images to different parts of the pipeline, some of the attachment points are more restrictive in terms of the acceptable resource states than others. For example, shader resources could be in a variety of states depending on the pipeline and resource access type, while the memory range containing draw index data has to be only in the GR_MEMORY_STATE_INDEX_DATA state. The color targets or depth-stencil images could be in either the GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL or GR_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL state, which is communicated to Mantle at the target bind time. Naturally, the GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL state for color targets and depth-stencil buffers provides the best performance for rendering, but might incur an overhead when converting to any other access state or when accessing from shaders. In cases when the application expects to have light rendering followed by image shader access, it has an option of using the `GR_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL` state for rendering. A list of allowed states modes for various operations in Mantle is presented in table below.

Table 11. Allowed resource states for various operations

Operation or usage	Allowed resource states
CPU resource access	GR_MEMORY_STATE_DATA_TRANSFER GR_IMAGE_STATE_DATA_TRANSFER
GPU resource copy	GR_MEMORY_STATE_DATA_TRANSFER GR_MEMORY_STATE_DATA_TRANSFER_SOURCE GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION GR_IMAGE_STATE_DATA_TRANSFER GR_IMAGE_STATE_DATA_TRANSFER_SOURCE GR_IMAGE_STATE_DATA_TRANSFER_DESTINATION
Immediate memory update	GR_MEMORY_STATE_DATA_TRANSFER
Load/save atomic counter	GR_MEMORY_STATE_DATA_TRANSFER
Copy occlusion data	GR_MEMORY_STATE_DATA_TRANSFER
Queue atomics	GR_MEMORY_STATE_QUEUE_ATOMIC
Write timestamp	GR_MEMORY_STATE_WRITE_TIMESTAMP
Resource cloning	Any image state except GR_IMAGE_STATE_UNINITIALIZED
Indirect draw/dispatch argument data	GR_MEMORY_STATE_INDIRECT_ARG GR_MEMORY_STATE_MULTI_USE_READ_ONLY
Index data	GR_MEMORY_STATE_INDEX_DATA GR_MEMORY_STATE_MULTI_USE_READ_ONLY
Graphics shader access	GR_MEMORY_STATE_GRAPHICS_SHADER_READ_ONLY GR_MEMORY_STATE_GRAPHICS_SHADER_WRITE_ONLY GR_MEMORY_STATE_GRAPHICS_SHADER_READ_WRITE GR_MEMORY_STATE_MULTI_USE_READ_ONLY GR_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY GR_IMAGE_STATE_GRAPHICS_SHADER_WRITE_ONLY GR_IMAGE_STATE_GRAPHICS_SHADER_READ_WRITE GR_IMAGE_STATE_MULTI_SHADER_READ_ONLY GR_IMAGE_STATE_TARGET_AND_SHADER_READ_ONLY

Operation or usage	Allowed resource states
Compute shader access	GR_MEMORY_STATE_COMPUTE_SHADER_READ_ONLY GR_MEMORY_STATE_COMPUTE_SHADER_WRITE_ONLY GR_MEMORY_STATE_COMPUTE_SHADER_READ_WRITE GR_MEMORY_STATE_MULTI_USE_READ_ONLY GR_IMAGE_STATE_COMPUTE_SHADER_READ_ONLY GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY GR_IMAGE_STATE_COMPUTE_SHADER_READ_WRITE GR_IMAGE_STATE_MULTI_SHADER_READ_ONLY
Color targets	GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL GR_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL
Depth-stencil targets	GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL GR_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL GR_IMAGE_STATE_TARGET_AND_SHADER_READ_ONLY
Image clear	GR_IMAGE_STATE_CLEAR
Resolve source	GR_IMAGE_STATE_RESOLVE_SOURCE
Resolve destination	GR_IMAGE_STATE_RESOLVE_DESTINATION

Initial Resource State

When memory objects are created or non-target images are bound to memory, they are assumed to be in the GR_MEMORY_STATE_DATA_TRANSFER or GR_IMAGE_STATE_DATA_TRANSFER state. Images that could be used as color or depth-stencil targets are assumed to be in the GR_IMAGE_STATE_UNINITIALIZED state when bound to memory, and have to be initialized by transitioning them to an appropriate state on a graphics capable queue. Only the whole images can be transitioned from the GR_IMAGE_STATE_UNINITIALIZED state. Failure to initialize the whole target image produces undefined behavior.

Before unbinding GPU updated images from memory, an application transitions target images to the GR_IMAGE_STATE_UNINITIALIZED state and non-target images to the GR_IMAGE_STATE_DATA_TRANSFER state. Alternatively, images to be unbound can be transitioned to the GR_IMAGE_STATE_DISCARD state regardless of the image type. This ensures the GPU caches are properly flushed and avoids a possibility of data corruption.

Data Transfer States

The GR_MEMORY_STATE_DATA_TRANSFER or GR_IMAGE_STATE_DATA_TRANSFER states are used for a broad range of copy operations that can be performed in Mantle either with a GPU or CPU.

There are also *specialized data transfer states* for designating memory and image as a source for GPU copy operations: GR_MEMORY_STATE_DATA_TRANSFER_SOURCE and GR_IMAGE_STATE_DATA_TRANSFER_SOURCE. Likewise, there are states that designate memory and image as a destination for GPU copy operations: GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION and GR_IMAGE_STATE_DATA_TRANSFER_DESTINATION. These states can be used interchangeably with the GR_IMAGE_STATE_DATA_TRANSFER state for source or destination of the GPU copy operation; however, they cannot be used for specifying CPU access. Peer images must always use the GR_IMAGE_STATE_DATA_TRANSFER state.

If application requires CPU access to memory or image in one of the specialized data transfer states, the state needs to be transitioned to the GR_MEMORY_STATE_DATA_TRANSFER or GR_IMAGE_STATE_DATA_TRANSFER state accordingly.

For performance reasons, it is advised to use specialized data transfer states specifying only source or destination for the GPU copy.

State Preparations

An application indicates a memory range or an image state transition by adding special *resource preparation* commands into the GPU command buffer on the expected change of the memory or image usage model. A preparation command specifies how a memory range or an image was used previously (since the last preparation command) and its new usage. The non-rendering and non-compute operations that affect memory contents, such as copies, clears, and so on, also participate in the change of resource usage and require preparation commands before and after the operation. The preparation of a list of memory ranges is added to a command buffer by calling grCmdPrepareMemoryRegions().

Listing 13. Example of memory preparation

```
// Prepare first 100 bytes of memory object for shader read after data upload
GR_MEMORY_STATE_TRANSITION transition = {};
transition.mem        = mem;
transition.oldState   = GR_MEMORY_STATE_DATA_TRANSFER;
transition.newState   = GR_MEMORY_STATE_GRAPHICS_SHADER_READ_ONLY;
transition.offset     = 0;
transition.regionSize = 100;
// Record state transition in command buffer
grCmdPrepareMemoryRegions(cmdBuffer, 1, &transition);
```

Images are similarly prepared by using `grCmdPrepareImages()`.

Listing 14. Example of image preparation

```
// Prepare image for shader read after it was rendered to
GR_IMAGE_STATE_TRANSITION transition = {};
transition.image    = image;
transition.oldState = GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL;
transition.newState = GR_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY;
transition.subresourceRange.aspect         = GR_IMAGE_ASPECT_COLOR;
transition.subresourceRange.baseMipLevel   = 0;
transition.subresourceRange.mipLevels      = GR_LAST_MIP_OR_SLICE;
transition.subresourceRange.baseArraySlice = 0;
transition.subresourceRange.arraySize      = GR_LAST_MIP_OR_SLICE;
// Record state transition in command buffer
grCmdPrepareImages(cmdBuffer, 1, &transition);
```

On memory and image preparation, the driver internally generates appropriate GPU stalls, cache flushes, surface decompressions, and other required operations according to the resource state transition and the expected usage model. Some of the transitions might be "no-op" from the hardware perspective; however, all preparations have to be performed for compatibility with a wide range of GPUs, including future generations.

> It is more optimal to prepare memory or images in batches, rather than executing preparations on individual resources.

Preparation ranges for memory objects are specified at byte granularity. When zero offset and range size are used, the whole memory object range is prepared. Any part of the prepared memory range can only be specified once in a preparation call. Referencing the same location multiple times within a preparation operation produces undefined results.

Image preparation is performed at a subresource granularity, according to the specified range of subresources. Any given subresource must only be referenced once in a preparation call. Referencing a subresource multiple times within a preparation operation produces undefined results.

When an image preparation operation is recorded in a command buffer, the render target and depth-stencil view of that image cannot be bound to the current state, as it causes undefined rendering behavior following the preparation. The application must rebind target views that are based on images that have been prepared before the draw.

All memory and image states are available for transitions executed on the graphics and universal queues, but only a subset is available for transitions executed on compute queues. The queues defined in extensions might have a different set of rules regarding the preparations.

Multisampled Image Preparation

Preparation of multisampled images requires a correct *multisample anti-aliasing* (MSAA) state object (see Multisampling State) to be bound to the current command buffer state. The MSAA state object used for preparation should be with exactly the same configuration as the one used for rendering to the multisampled image. If multiple multisampled images with different MSAA configurations have to be processed, they cannot be prepared on the same invocation of the `grCmdPrepareImages()` function.

Memory and Image State Reset

If memory object contains a lot of memory regions in different states, the application is required to track all ranges and perform a lot of state transition to ensure consistency of the state. When application does not care about memory contents anymore (e.g., when it is about to load new data or generate new data with GPU), it could bulk reset memory state in a single operation by performing a state transition from the `GR_MEMORY_STATE_DISCARD` state to an appropriate state such as the `GR_MEMORY_STATE_DATA_TRANSFER` state.

Similarly, when new data are about to be loaded to an image with many subresources in different states, an application could bulk reset image state in a single operation by performing a state transition from `GR_IMAGE_STATE_DISCARD` to a new appropriate state.

After transition from the `GR_MEMORY_STATE_DISCARD` or `GR_IMAGE_STATE_DISCARD` state, the memory or image data integrity is not guaranteed, and it is assumed that resource data will be completely overwritten by an application on the next operation. For that reason, the application should not transition from the `GR_MEMORY_STATE_DISCARD` or `GR_IMAGE_STATE_DISCARD` state to any readable state.

Multi-Queue Considerations

When preparing memory ranges or images for transitioning use between queues, the preparation has to be performed on the queue that was last to use the resource. For example, if the universal queue was used to render to a color target that is used next for shader read on a compute queue, the universal queue has to execute a `GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL` to a `GR_IMAGE_STATE_COMPUTE_SHADER_READ_ONLY` transition. The only exceptions to this are transitions from any of the `GR_MEMORY_STATE_DATA_TRANSFER`, `GR_IMAGE_STATE_DATA_TRANSFER` (or any specialized data transfer states), and `GR_IMAGE_STATE_UNINITIALIZED` states, which should be performed on the queue that will use the memory or image next.

> Failing to prepare memory range or image on the queue that was last to update or otherwise use the resource might result in corruption due to residual data in caches. Additionally, the queue intended for the next operation might not have hardware capability to properly perform the state transition.

It is allowed to access the memory or image from multiple queues for read-only access using the `GR_MEMORY_STATE_MULTI_USE_READ_ONLY` and `GR_IMAGE_STATE_MULTI_SHADER_READ_ONLY` states. Before a resources is accessed by any of the multiple queues, it should be transitioned to one of those states on the queue that was the last to use the resource. After the application no longer desires to read the memory or image from multiple queues, it should perform an appropriate transition on the queue that is next to use the resource. All other queues should transition from the current read-only state to the `GR_MEMORY_STATE_DISCARD` or `GR_IMAGE_STATE_DISCARD` state to ensure caches are flushed, if necessary. Only transitions from the `GR_MEMORY_STATE_MULTI_USE_READ_ONLY` to the `GR_MEMORY_STATE_DISCARD` state and the `GR_IMAGE_STATE_MULTI_SHADER_READ_ONLY` to the `GR_IMAGE_STATE_DISCARD` state are allowed. Transition from any other state to `GR_MEMORY_STATE_DISCARD` or `GR_IMAGE_STATE_DISCARD` results in undefined behavior. Application should use queue semaphores to ensure preparations between queues are properly synchronized.

Hazards

The Mantle driver does not track any potential resource access hazards, such as *read-after-write* (RAW), *write-after-write* (WAW) or *write-after-read* (WAR), that could result from resources being written and read by different parts of the pipeline and by the overlapping nature of the shader execution in draws and compute dispatches. The resource hazard conditions are expressed in Mantle using the preparation operations.

In most cases, the graphics pipeline does not guarantee ordering of element processing in the pipeline. The ordering of execution between the draw calls is only guaranteed for color target and depth-stencil target writes – the pixels of the second draw are not written until all of the pixels from the first draw are written to the targets. Mantle also guarantees ordering of copy operations for memory ranges in the `GR_MEMORY_STATE_DATA_TRANSFER`, `GR_MEMORY_STATE_DATA_TRANSFER_SOURCE` or `GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION` states, and images in the `GR_IMAGE_STATE_DATA_TRANSFER`, `GR_IMAGE_STATE_DATA_TRANSFER_SOURCE` or `GR_IMAGE_STATE_DATA_TRANSFER_DESTINATION` states. In all other cases, hazards must be addressed by the application. For example, image writes from shaders could cause write-after-write hazards.

The read-after-write hazards must be addressed whenever there is a possibility of the GPU reading resource data produced by the GPU. Likewise, write-after-write and write-after-read hazards must be resolved when there is a possibility of concurrent or out-of-order writes. In case of back-to-back image clears, without transition to any other state, there is also a possibility of a write-after-write hazard that must be resolved by an application.

Some of the write-after-write hazards, such as executing back-to-back compute dispatches that write to the same resource or memory range, do not represent actual change in image or memory state. These can be resolved by performing a transition from the current image or memory state to the same state. For example, the write-after-write hazard for image writes from the compute pipeline in the case above can be resolved by a preparation call with a state transition from `GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY` to `GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY`. While there is not an actual transition of state, such preparation would be an indication to the Mantle driver of a write-after-write hazard condition. Inserting hazard processing ensures the non-overlapping nature of the copy operations.

> There is never a write-after-write hazard when performing operations on memory in the GR_MEMORY_STATE_DATA_TRANSFER state and on images in the GR_IMAGE_STATE_DATA_TRANSFER state (or their appropriate specialized data transfer states). When performing back-to-back copies of data, the Mantle driver ensures there are no hazards by ensuring each copy function call has finished before continuing with the next operation.

Some typical examples of hazard conditions and state transitions are listed in Table 12. Note that preparations are not only used for handling hazard conditions, but to indicate actual resource usage transition (e.g., change from shader readable state to render target use).

Table 12. Hazard and state transition examples

Usage scenario	Hazard	Transition
Read the render target previously in render-optimal state	RAW	GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL to GR_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY
Write to image from compute shader after it was read by graphics pipeline	WAR	GR_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY to GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY
Write to image from compute shader on consecutive dispatches	WAW	GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY to GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY
Write to image from pixel shader (non-target write) on consecutive draws	WAW	GR_IMAGE_STATE_GRAPHICS_SHADER_WRITE_ONLY to GR_IMAGE_STATE_GRAPHICS_SHADER_WRITE_ONLY
Draw indirect with data generated by the compute shader	RAW	GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY to GR_MEMORY_STATE_INDIRECT_ARG

Usage scenario	Hazard	Transition
Draw indirect with data loaded by the CPU	N/A	GR_MEMORY_STATE_DATA_TRANSFER to GR_MEMORY_STATE_INDIRECT_ARG
Draw with indices output by the compute shader	RAW	GR_MEMORY_STATE_COMPUTE_SHADER_WRITE_ONLY to GR_MEMORY_STATE_INDEX_DATA
Back-to-back image clears	WAW	GR_IMAGE_STATE_CLEAR to GR_IMAGE_STATE_CLEAR
Reading the GPU timestamp data by the CPU	N/A	GR_MEMORY_STATE_WRITE_TIMESTAMP to GR_MEMORY_STATE_DATA_TRANSFER

The list of the hazard conditions in the table above is non-exhaustive, and all hazards must be addressed whenever there is a possibility of reading or writing resource data in different parts of the pipeline or by different GPU engines, or in case of race conditions.

Resource Operations

In the Mantle API, images and memory content are operated on by *resource operation* commands recorded in command buffers. Using command buffers submitted on multiple queues allows some resource operations to be asynchronous with respect to rendering and dispatch commands. It is an application's responsibility to ensure proper synchronization and preparation of images and memory on accesses from compute and graphic pipelines and asynchronous resource operations executed on other queues. An application must make no assumptions about the order in which command buffers containing resource operations are executed between queues (ordering of command buffers is guaranteed only within a queue), and should rely on synchronization objects to ensure command buffer completion before proceeding with dependent operations.

The following operations can be performed on memory and images:

- Clearing images and memory
- Copying data in memory and images
- Updating memory
- Resolving multisampled images
- Cloning images

Resource Copies

An application can copy memory and image data using several methods depending on the type of data transfer. The memory data can be copied between memory objects with `grCmdCopyMemory()`, and a portion of an image could be copied to another image with `grCmdCopyImage()`. The image data can also be copied to and from memory using `grCmdCopyImageToMemory()` and `grCmdCopyMemoryToImage()`. Multiple memory or image regions can be specified in the same function call. None of the source and destination regions can overlap – overlapping any of the source or destination regions within a single copy operation produces undefined results. It is also invalid to specify an empty memory region or zero image extents.

Not all image types can be used for copy operations. While images designated as depth targets can be used as copy source, they cannot be used as copy destination. An attempt to copy to a depth image produces undefined behavior.

If the application needs to copy data into a depth image, it can do so by rendering a rectangle that covers the copy region and exporting depth information with a pixel shader.

When copying memory to and from images, the memory offsets have to be aligned to the image texel size (or compression block size for compressed images).

When copying data between images, the source and destination image type must match. That is, a part of a 2D image can be copied to another 2D image, but it is not allowed to copy a part of a 1D image to a 2D image. The multisampled images can only be copied when source and destination images they have the same number of samples. Source and destination formats do not have to match, and appropriate format conversion is performed automatically, if both the source and destination image formats support conversion, which is indicated by the `GR_FORMAT_CONVERSION` format capability flag. In that case, the pixel size (or compression block size for compressed images) has to match, and the raw image data are copied.

For compressed image formats, the conversion cannot be performed and the image extents are specified in compression blocks.

Before any of the copy operations can be used, the memory ranges involved in copy operations must be transitioned to the `GR_MEMORY_STATE_DATA_TRANSFER` state and images must be transitioned to the `GR_IMAGE_STATE_DATA_TRANSFER` state using an appropriate preparation command. After the memory or image copy is done, a preparation command indicating transition of usage from the `GR_MEMORY_STATE_DATA_TRANSFER` or `GR_IMAGE_STATE_DATA_TRANSFER` state must be performed before a source or a destination memory or image can be used for rendering or other operations. Alternatively, an appropriate specialized data transfer state can be used. With back-to-back copies to the same resource, there is no need to deal with write-after-write hazards, as each copy is guaranteed to finish before starting the next one.

> Whenever possible, an application should combine copy operations using the same image or memory objects, provided the copy regions do not overlap. Batching reduces the overhead of copy operations.

Image Cloning

The image copy operations described in Resource Copies, while flexible, require images to be put into the `GR_IMAGE_STATE_DATA_TRANSFER` or `GR_IMAGE_STATE_DATA_TRANSFER_SOURCE` state for the duration of the copy operation. That state transition might incur some overhead and in many cases for target images, might be suboptimal. If a whole resource needs to be copied without a change of its state, a special optimized *clone operation* can be used. Images are cloned by calling `grCmdCloneImageData()`.

The clone operation can only be performed on images with the same creation parameters, and memory objects must be bound to the source and destination image before executing a clone operation. Both source and destination image must be created with the `GR_IMAGE_CREATE_CLONEABLE` flag.

If, before cloning, a destination image was used on a different queue, it needs to be transitioned to the `GR_IMAGE_STATE_DISCARD` state similarly to the rules for queues that no longer require resource access described in Multi-Queue Considerations. After cloning, the application should assume the destination image object is in the same state as the source image before the clone operation. The source resource state is left intact after the cloning.

> Even though an application has direct access to the memory store of all resources, it should not rely on direct memory copy for cloning opaque objects, but should instead use the specially provided function to properly clone all image meta-data.

If the destination image for cloning operation was bound to a device state as a target during the clone operation, it needs to be re-bound before the next draw, otherwise rendering produces undefined results.

Immediate Memory Updates

Sometimes it is necessary to inline small memory updates in command buffers (e.g., to quickly feed new parameter values into shaders). In the Mantle API, this can be accomplished by using `grCmdUpdateMemory()`. The update is performed synchronously with other operations in a command buffer.

> While immediate memory update is a convenient mechanism for small data updates, it can be relatively slow and inefficient. Use immediate memory update sparingly.

The data size and destination offset for immediate memory updates have to be 4-byte aligned. The memory range must be in the GR_MEMORY_STATE_DATA_TRANSFER or GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION state for the immediate updates to work correctly. These updates can be executed on queues of all types. There is a limit on the maximum size of the uploaded data that is guaranteed to be at least 1KB. This maximum inline update size can be queried from the physical GPU properties (see GPU Identification and Initialization) by inspecting the maxInlineMemoryUpdateSize value in GR_PHYSICAL_GPU_PROPERTIES.

Resource Upload Strategies

Mantle provides a number of different data upload options that can be selected to satisfy a particular set of requirements or tradeoffs. For small infrequent updates Immediate Memory Updates might be an acceptable choice. For larger uploads there are generally two methods: direct memory update or indirect update.

To implement the direct update method, an application maps the CPU accessible memory and directly loads memory and image data using a CPU memory copy. This method generally works well for non-image dynamic data, provided the destination memory resides in a CPU visible heap.

The indirect update method uses two steps for loading data. First, the non-local (remote) memory object is mapped (or alternatively a pinned memory is used) and data are loaded into that *staging area* using the CPU memory copy. Second, the memory or image data are copied to the final destination using the GPU. If a DMA queue (see DMA Queue Extension) is available, it can be used to upload the memory or image data in parallel with rendering and compute operations. The indirect update method is particularly useful for loading the data to CPU invisible heaps and to load data for optimally tiled images. If necessary, the tiling conversion is performed during the GPU copy.

> Since compressed images can only use optimal tiling, the indirect update is the only suitable method for loading compressed images.

Memory Fill

A range of memory could be cleared by the GPU by filling it with the provided 4-byte value using grCmdFillMemory(). The destination and fill size have to be 4-byte aligned. The memory range needs to be in the GR_MEMORY_STATE_DATA_TRANSFER or GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION state for the fill operation to work correctly. The memory fill can be executed on queues of all types.

> The memory objects in system memory heaps probably can be cleared faster by the CPU than the GPU.

Image Clears

Image clears are optimized operations to set a clear value to all elements of an aspect or set of aspects in the image. Both target and non-target image clears are supported by calling `grCmdClearColorImage()` or `grCmdClearColorImageRaw()`. Depth-stencil targets can be cleared by calling `grCmdClearDepthStencil()`. These clear operations for target images are only available in universal command buffers. Non-target color images can also be cleared in compute command buffers.

Before a color image or depth-stencil clear operation is performed, an application should ensure the image is in the `GR_IMAGE_STATE_CLEAR` state by issuing an appropriate resource preparation command.

The granularity of clears for non-target images is a subresource. For target images, the granularity depends on the GPU capabilities and the number of unique clear colors per image.

If `multiColorTargetClears` in GPU properties reports `GR_FALSE`, only a single clear color (or a single set of depth and stencil clear values) can be used per target image. In that case, the whole image first is cleared to a clear color, and then subsequently parts of the image are cleared to exactly the same color. If application would like to use a different clear color, the whole target image must be cleared. Clearing the image to multiple values on GPUs that do not support that capability produces undefined results.

> When only a subset of a resource that needs to be cleared is smaller than the allowed granularity, or multiple clear values per image need to be used, but they are not supported by the GPU, an application should use the graphics or compute pipeline for the purpose of image clears by rendering a constant shaded rectangle covering the cleared area.

Clearing an image with `grCmdClearColorImage()` automatically performs value conversion on the application-provided floating point color values, as is appropriate for the image format used. These clears are only allowed for formats that have a `GR_FORMAT_CONVERSION` capability flag exposed. For the sRGB formats, the clear color is specified in a linear space, which is appropriately converted by the Mantle driver to the sRGB color space. Raw clears perform no format conversion and are available for all image formats. The provided clear data are directly stored regardless of the format's numeric type (including sRGB formats). The `grCmdClearColorImageRaw()` function takes a number of least significant bits from per-channel unsigned integer color values as appropriate for the image format bit depth and stores them in the channels that are present in the format. The order of color channel data specified for clear functions is R, G, B, A.

To clear a depth-stencil image, an application uses `grCmdClearDepthStencil()` with specified depth and stencil clear values. The decision to clear depth or stencil parts of the image is made according to provided subresource range aspects. If the application wants to clear both depth and stencil, it needs to provide separate subresource ranges for depth and stencil aspects. The ranges for depth and stencil aspects are fully independent, and it is not required to specify the matching ranges of depth and stencil subresource in one clear call. It is also allowed to clear depth and stencil separately.

> ! For performance reasons, it is advised to clear depth and stencil in the same operation with matching subresource ranges.

Before clearing a resource, an application must ensure it is not bound to a command buffer state in the command buffer where it is cleared. If necessary, a resource could be rebound again after the clear and appropriate preparation operations. Clearing a resource while it is bound to a GPU state causes undefined results in subsequent rendering operations.

Multisampled Image Resolves

An application could implement its own resolve operations using shaders, but for convenience, an optimized resolve implementation is provided in Mantle. The built-in implementation understands all sample counts and is guaranteed to work on a variety of supported formats. Images of integer numeric formats cannot be uses for resolves. The resolve operation could be recorded into a command buffer using `grCmdResolveImage()`. The built-in resolve operation can only be executed on universal queue.

The source image must be a 2D multisampled image and the destination must be a single sample image. The formats of the source and destination image subresources should match; otherwise, the results of the resolve operation are undefined. It is not necessary to cover the whole destination subresource with the resolve rectangle – an application can perform partial subresource resolves.

The resolve operation is also supported for depth-stencil images, in which case it is performed by copying the first sample from the target image to the destination image.

Before a resolve operation can take a place, the source and destination image subresources must be processed using appropriate preparation commands, designating them as resolve source and destination using the `GR_IMAGE_STATE_RESOLVE_SOURCE` and `GR_IMAGE_STATE_RESOLVE_DESTINATION` image states, respectively. At the time of a resource resolve execution, the color target and depth-stencil view of the source and destination resources must not be bound in a command buffer, otherwise the rendering that follows the resolve causes undefined behavior. The application should rebind target views that are based on images that are used as a source or destination of the resolve operation.

Image Samplers

Sampler objects, represented in Mantle by the `GR_SAMPLER` handle, describe how images are processed (e.g., filtered, converted, and so on) on a texture fetch operation. A sampler object is created by calling grCreateSampler().

The core Mantle specification defines a limited support for border colors available to application when the GR_TEX_ADDRESS_CLAMP_BORDER addressing mode is used. Available border colors are specified using GR_BORDER_COLOR_TYPE. More options for border colors are exposed through special extensions, such as Border Color Palette Extension.

Resource Shader Binding

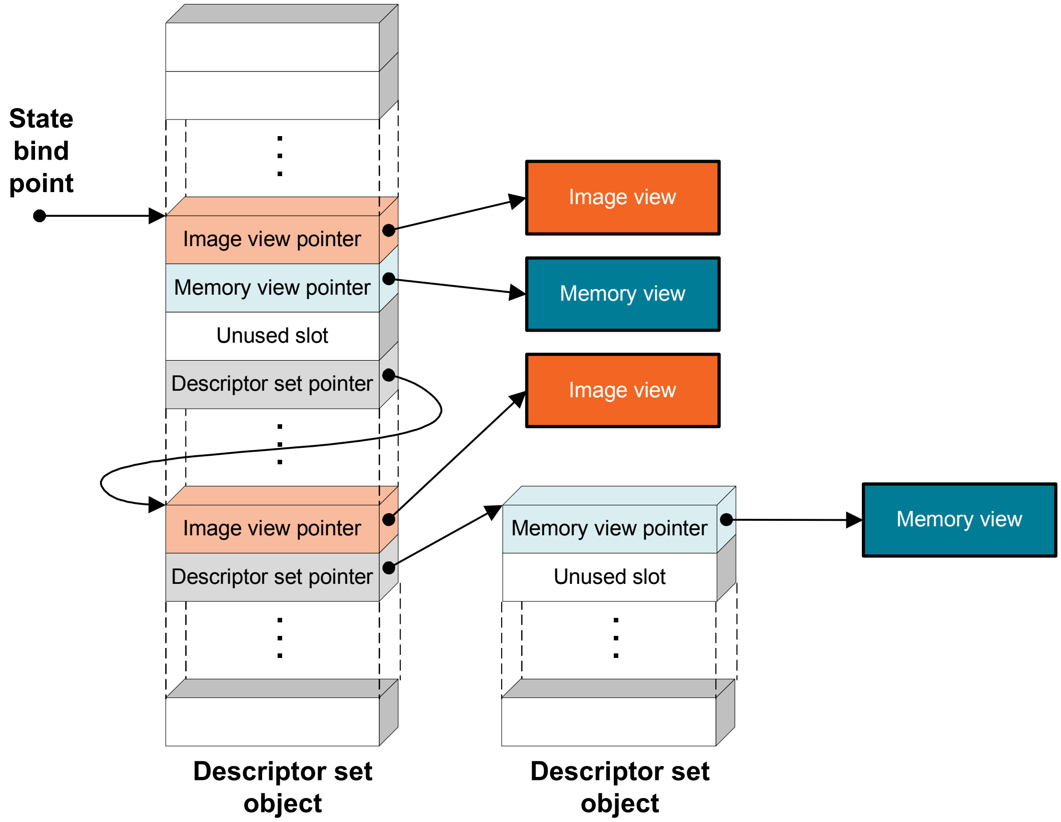
Shader resources, such as memory views and images, as well as the sampler object references, are not directly bound to pipeline shaders. They are grouped into monolithic *descriptor sets* that are bound to a command buffer state. Pipeline Resource Access discusses in greater detail how descriptor sets are mapped to shaders and bound to the state. In addition to descriptor sets, an application can use the Dynamic Memory View.

Descriptor Sets

A *descriptor set* is a special state object that conceptually can be viewed as an array of shader resource or sampler object descriptors or pointers to other descriptor sets. A portion of the descriptor set is bound to the command buffer state to be accessed by the shaders of the currently active pipeline. A descriptor set is created by calling grCreateDescriptorSet().

There could be several descriptor sets available to the pipeline. Shader resources and samplers referenced in descriptor sets are shared by all shaders forming a pipeline. The number of descriptor sets that can be bound to a command buffer state can be queried from physical GPU properties, but it is guaranteed to be at least 2. Additionally, more descriptor sets can be accessed hierarchically through the descriptor sets directly bound to the pipeline. An example of a descriptor set and its bindings is shown in Figure 11.

Figure 11. Descriptor set binding example



Mantle imposes no limits on the size of the descriptor set or the total number of created descriptor sets, provided they fit in memory. An application can create larger descriptor sets than necessary for a given pipeline, sub-allocate a range of slots, and bind descriptor set ranges to a pipeline with an offset. The ability to create large descriptor sets and sub-allocate descriptor set chunks provides a potential tradeoff between memory usage and complexity of descriptor set management.

When a descriptor set is created and its memory is bound, the contents of a descriptor set are not initialized. An application should explicitly initialize a descriptor set by binding shader resources and samplers or by clearing descriptor set slots as described in Descriptor Set Updates.

There are many strategies for organizing shader resources in descriptor sets, which provide a wide range of CPU and GPU performance tradeoffs. One example of such a strategy is to divide sampler and resource objects into separate descriptor sets: one dedicated to resources and another for samplers (i.e., for simplicity of object management). Another strategy is to mix resources and samplers in the same descriptor set, but group them into descriptor sets according to the frequency of update. For example, one descriptor set could be dedicated for frequently changing memory views and images. Using multiple directly bound descriptor sets provides a lot of freedom in managing resources and samplers for shader access.

Descriptor Set Updates

Descriptor sets can be initially constructed and later updated by an application outside of command buffer execution. Sets can be updated multiple times; however, when updating, an application should ensure they are not currently used for rendering.

An update for a descriptor set is initiated by calling `grBeginDescriptorSetUpdate()`, followed by calls to one of the following functions to update ranges of descriptor set slots with objects that need to be bound to them – `grAttachMemoryViewDescriptors()` for memory views, `grAttachImageViewDescriptors()` for image views, `grAttachSamplerDescriptors()` for samplers, and `grAttachNestedDescriptors()` for building hierarchical descriptor sets. After an update is complete, an application calls `grEndDescriptorSetUpdate()`. Failure to match `grBeginDescriptorSetUpdate()` with a call to `grEndDescriptorSetUpdate()`, or performing an update without beginning the update, results in undefined behavior.

Listing 15. Example of creating and updating descriptor set

```
GR_RESULT result;
// Create descriptor set with 100 slots
GR_DESCRIPTOR_SET_CREATE_INFO descSetInfo = {};
descSetInfo.slots = 100;
GR_DESCRIPTOR_SET descSet = GR_NULL_HANDLE;
result = grCreateDescriptorSet(device, &descSetInfo, &descSet);
// Descriptor set needs memory bound before it can be updated
CreateAndBindDescriptorSetMemory(device, descSet);
// Begin update
grBeginDescriptorSetUpdate(descSet);
// Bind image to slot 0 for compute write access
GR_IMAGE_VIEW_ATTACH_INFO imgViewAttach = {};
imgViewAttach.view  = view;
imgViewAttach.state = GR_IMAGE_STATE_COMPUTE_SHADER_READ_WRITE;
grAttachImageViewDescriptors(descSet, 0, 1, &imgViewAttach);
// Bind memory view to slot 1
GR_MEMORY_VIEW_ATTACH_INFO memViewAttach = {};
memViewAttach.mem    = mem;
memViewAttach.offset = 0;
memViewAttach.range  = 96;
memViewAttach.stride = 16;
memViewAttach.format.channelFormat = GR_CH_FMT_UNDEFINED;
memViewAttach.format.numericFormat = GR_NUM_FMT_UNDEFINED;
memViewAttach.state   = GR_MEMORY_STATE_COMPUTE_SHADER_READ_ONLY;
grAttachMemoryViewDescriptors(descSet, 1, 1, &memViewAttach);
// Bind nested sampler set (starting at slot 5) to slot 2
GR_DESCRIPTOR_SET_ATTACH_INFO nestedAttach = {};
nestedAttach.descriptorSet = descSet;
nestedAttach.slotOffset    = 5;
grAttachNestedDescriptors(descSet, 2, 1, &nestedAttach);
// Bind sampler to slot 5
grAttachSamplerDescriptors(descSet, 5, 1, &sampler);
// End update
grEndDescriptorSetUpdate(descSet);
```

Image objects cannot be directly bound to resource descriptor sets; image views are used instead. An image view always references the most recent memory association of the parent image object. Binding an image to a descriptor set takes a snapshot of the memory association as it was defined at the time of the binding. Later changes to the image's memory binding are not reflected in previously built descriptor sets. The memory for shader access is bound as described in Memory Views.

> Calls to the grBeginDescriptorSetUpdate() map descriptor set memory with the purpose of updating descriptor data on subsequent grAttach*() calls. If the memory object bound to the descriptor set resides in local video memory while being referenced in queued command buffers, the memory object cannot be safely mapped on Windows® 7-8.1 platforms and calls to grBeginDescriptorSetUpdate() lead to undefined results.

To create complex descriptor set hierarchies as shown in Figure 11, descriptor set ranges are hierarchically bound to slots of other descriptor sets. It is allowed to reference descriptor sets hierarchically within the same descriptor set.

The descriptor set update operation produces undefined results if the application attempts to bind a sampler or shader resource to a slot that does not exist in a descriptor set.

To reset a range of descriptor set slots to an unbound state, an application calls `grClearDescriptorSetSlots()`. There is no requirement for clearing descriptor set slots before binding new objects, but it could be useful for assisting in debugging an unexpected behavior related to bound descriptor set objects.

> Each individual descriptor set update might be fairly CPU-heavy, due to a memory mapping operation on a call to grBeginDescriptorSetUpdate() and memory unmapping on a call to grEndDescriptorSetUpdate(). In the case of heavy dynamic descriptor set updates, it is recommended to create larger descriptor sets and use them as pools of descriptor slots in ranges that are individually bound to the GPU state. In the case of a large descriptor set used as a pool, only a single set of grBeginDescriptorSetUpdate() and grEndDescriptorSetUpdate() calls per large descriptor set should be necessary.

An application can create and initialize descriptor set objects ahead of time or it can update them on the fly as necessary. Ranges of descriptor set slots must not be updated if they are referenced in command buffers scheduled for execution. An application is responsible for tracking the lifetime of descriptor sets and their slot reuse.

Chapter V.

State, Shaders, and Pipelines

Mantle State Overview

The configuration of the GPU device and how rendering happens is described by the state data. State is specified by binding various state objects and setting state values in command buffers. When command buffer recording starts, all GPU state is undefined and an application should explicitly initialize all relevant state before the first draw or dispatch call. Failing to bind all required state produces undefined results. State is persistent only within the boundaries of a command buffer. For performance reasons, the application should avoid binding state redundantly.

The compute command buffers have only one instance of the GPU state – compute. The universal command buffers have two separate GPU states for tracking compute and graphics related state information.

Static vs. Dynamic State

Conceptually there are several types of state data in Mantle – the *dynamic state* that is a configurable part of the state that is set in command buffers, and the *static state* used for the pipeline construction. The dynamic state is represented by state block objects, pipelines objects, and others.

Table 13 provides a summary of the dynamic state that can be bound or set in command buffers for graphics and compute operations.

Table 13. Summary of dynamic command buffer state

Dynamic state type	Graphics operations	Compute operations
Index data	YES	NO
Pipeline	YES	YES
Descriptor sets	YES	YES
Dynamic memory view	YES	YES
Render targets	YES	NO
Rasterizer state	YES	NO
Viewport and scissor state	YES	NO
Color blender state	YES	NO
Depth-stencil state	YES	NO
Multisampling state	YES	NO

Static state in graphic pipelines is discussed in Graphics Pipeline State.

Dynamic State Objects

The dynamic state is represented by *state objects*. The state objects are created by the driver from the application provided state descriptions and are referenced using handles. There are separate state objects for different fixed function blocks. The following types of dynamic state objects exist in Mantle:

- Rasterizer state (`GR_RASTER_STATE_OBJECT`)
- Viewport and scissor state (`GR_VIEWPORT_STATE_OBJECT`)
- Color blender state (`GR_COLOR_BLEND_STATE_OBJECT`)
- Depth-stencil state (`GR_DEPTH_STENCIL_STATE_OBJECT`)
- Multisampling state (`GR_MSAA_STATE_OBJECT`)

Mantle API specifies a set of matching bind points that state objects blocks can be attached to using `grCmdBindStateObject()`. All state bind points must have dynamic state objects bound for rendering operations. The state specified in the state blocks has to match pipeline expectations at the draw time.

Rasterizer State

The rasterizer state object is represented by the GR_RASTER_STATE_OBJECT handle. It describes primitive screen space orientation and rasterization rules, as well as specifies used depth bias. The raster state object is created by calling grCreateRasterState(). The rasterizer state is bound to the GR_STATE_BIND_RASTER binding point.

Listing 16. Example of creating rasterizer state

```
GR_RESULT result;
// Setup rasterizer state
GR_RASTER_STATE_CREATE_INFO rasterInfo = {};
rasterInfo.cullMode             = GR_CULL_NONE;
rasterInfo.fillMode             = GR_FILL_SOLID;
rasterInfo.frontFace            = GR_FRONT_FACE_CW;
rasterInfo.depthBias            = 0;
rasterInfo.depthBiasClamp       = 0.0f;
rasterInfo.slopeScaledDepthBias = 0.0f;
// Create rasterizer state object
GR_RASTER_STATE_OBJECT rasterState = GR_NULL_HANDLE;
result = grCreateRasterState(device, &rasterInfo, &rasterState);
```

Viewport and Scissor State

The viewport state object is represented by the GR_VIEWPORT_STATE_OBJECT handle. It describes viewports used for rendering and optional scissors corresponding to the viewports. The viewport state object is created by calling grCreateViewportState(). The viewport state is bound to the GR_STATE_BIND_VIEWPORT binding point.

Listing 17. Example of creating viewport state

```
GR_RESULT result;
// Setup viewport state
GR_VIEWPORT_STATE_CREATE_INFO viewportInfo = {};
viewportInfo.viewportCount          = 1;
viewportInfo.scissorEnable          = GR_FALSE;
viewportInfo.viewports[0].originX  = 0;
viewportInfo.viewports[0].originY  = 0;
viewportInfo.viewports[0].width    = 1000;
viewportInfo.viewports[0].height   = 800;
viewportInfo.viewports[0].minDepth = 0.0f;
viewportInfo.viewports[0].maxDepth = 1.0f;
// Create viewport state
GR_VIEWPORT_STATE_OBJECT viewportState = GR_NULL_HANDLE;
result = grCreateViewportState(device, &viewportInfo, &viewportState);
```

Color Blender State

The color blender state object is represented by the GR_COLOR_BLEND_STATE_OBJECT handle. It describes the color blending state for the pipelines that enable blending operations. The color blender state is created by calling `grCreateColorBlendState()`. The color blender state is bound to the `GR_STATE_BIND_COLOR_BLEND` binding point.

A blender state defined to use the second pixel shader output is considered to be the *dual source blender state*. Dual-source blending is specified by one of the following blend values:

- `GR_BLEND_SRC1_COLOR`
- `GR_BLEND_ONE_MINUS_SRC1_COLOR`
- `GR_BLEND_SRC1_ALPHA`
- `GR_BLEND_ONE_MINUS_SRC1_ALPHA`

A blender state object with dual-source blending must only be used with pipelines enabling dual source blend.

Listing 18. Example of creating viewport state

```
GR_RESULT result;
// Setup color blend state state
GR_COLOR_BLEND_STATE_CREATE_INFO cbInfo = {};
// Enable color blend for the first target only
cbInfo.target[0].blendEnable    = GR_TRUE;
cbInfo.target[0].srcBlendColor  = GR_BLEND_SRC_ALPHA;
cbInfo.target[0].destBlendColor = GR_BLEND_ONE_MINUS_SRC_ALPHA;
cbInfo.target[0].blendFuncColor = GR_BLEND_FUNC_ADD;
cbInfo.target[0].srcBlendAlpha  = GR_BLEND_SRC_ALPHA;
cbInfo.target[0].destBlendAlpha = GR_BLEND_ONE_MINUS_SRC_ALPHA;
cbInfo.target[0].blendFuncAlpha = GR_BLEND_FUNC_ADD;
for (GR_UINT i = 1; i < GR_MAX_COLOR_TARGETS; i++)
{
    cbInfo.target[i].blendEnable    = GR_FALSE;
    cbInfo.target[i].srcBlendColor  = GR_BLEND_ONE;
    cbInfo.target[i].destBlendColor = GR_BLEND_ZERO;
    cbInfo.target[i].blendFuncColor = GR_BLEND_FUNC_ADD;
    cbInfo.target[i].srcBlendAlpha  = GR_BLEND_ONE;
    cbInfo.target[i].destBlendAlpha = GR_BLEND_ZERO;
    cbInfo.target[i].blendFuncAlpha = GR_BLEND_FUNC_ADD;
}
cbInfo.blendConst[0] = 1.0f;
cbInfo.blendConst[1] = 1.0f;
cbInfo.blendConst[2] = 1.0f;
cbInfo.blendConst[3] = 1.0f;
// Create color blend state state
GR_COLOR_BLEND_STATE_OBJECT cbState = GR_NULL_HANDLE;
result = grCreateColorBlendState(device, &cbInfo, &cbState);
```

The blend enable specified in the color blender state for each color target must match the blend state defined in the pipelines with which it is used. Mismatches between pipeline declarations and actually bound blender state objects causes undefined results.

Depth-stencil State

The depth-stencil state object is represented by the `GR_DEPTH_STENCIL_STATE_OBJECT` handle. It describes depth-stencil test operations in the graphics pipeline. The depth-stencil state is created by calling `grCreateDepthStencilState()`. The depth-stencil state is bound to the `GR_STATE_BIND_DEPTH_STENCIL` binding point.

Multisampling State

The multisampling state object is represented by the `GR_MSAA_STATE_OBJECT` handle. It describes the *multisample anti-aliasing* (MSAA) options for the graphics rendering. The multisampling state is created by calling `grCreateMsaaState()`. The multisampling state is bound to `GR_STATE_BIND_MSAA` binding point.

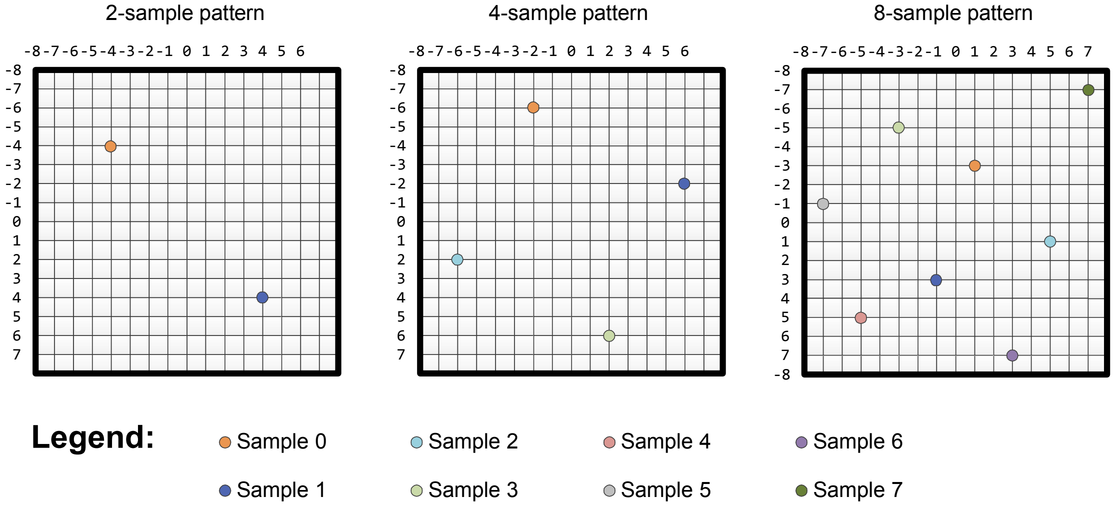
Specifying one sample in a multisampling state disables multisampling. A valid multisampling state must be bound even when rendering to single sampled images. The sampling rates defined in the multisampling state are uniform throughout the graphics pipeline. For more control of multisampling, the Advanced Multisampling Extension could be used.

Using multisampling state objects that have a different sample pattern or different configuration for rendering to the same set of color or depth-stencil targets produces an undefined result.

Default Sample Patterns

The sample patterns for multisampled images in Mantle are well defined and match the standard DirectX sample patterns for ease of portability. In Mantle, the application cannot query sample positions for the rasterizer or images from within a shader, but rather should rely on the knowledge of the patterns. Figure 12 defines default sample patterns in Mantle for 2-sample, 4-samples, and 8-samples.

Figure 12. Default sample patterns



SHADERS

Shader objects are used to represent code executing on programmable pipeline stages. The input shaders in Mantle are specified in binary *intermediate language* (IL) format. The currently supported intermediate language is a subset of *AMD IL*. The shaders can be developed in IL assembly or high-level languages and compiled off-line to a binary IL. The Mantle API can be considered language agnostic, as it could support other IL options in the future, provided that they expose a full shader feature set required by Mantle.

Shader objects, represented by `GR_SHADER` handles, are not directly used for rendering and are never bound to a command buffer state. Their only purpose is to serve as helper objects for pipeline creation. During the pipeline creation, shaders are converted to native GPU *shader instruction set architecture (ISA)* along with the relevant shader state. Once a pipeline is formed from the shader objects, the shader objects can be destroyed since the pipeline contains its own compiled and optimized shader representation. Shader objects help to reduce pipeline construction time when the same shader is used in multiple pipelines. Some of the compilation and pre-linking steps can be performed by the Mantle driver only once on the shader object construction, instead of during each pipeline creation. Since shaders are not directly used by the GPU, they never require GPU video memory binding.

A shader object for any shader stage is created by calling `grCreateShader()`.

PIPELINES

The Mantle API supports two principal types of pipelines – compute and graphics. In the future, more types of pipelines could be added to support new GPU architectures. All of the pipeline objects in Mantle, regardless of their type, are represented by the GR_PIPELINE handle. There are separate pipeline creation functions for different pipeline types.

The compute pipeline represents a compute shader operation. The graphics pipeline encapsulates the fixed function state and shader-based stages, all linked together into a special monolithic state object. It defines the communication between the pipeline stages and the flow of data within a graphics pipeline for rendering operations. Linking the whole pipeline together allows the optimization of shaders based on their input/outputs and eliminates expensive draw time state validation. This monolithic pipeline representation is bound to the GPU state in command buffers just like any other dynamic state.

Currently, the majority of developers create many thousands of different shaders and experience difficulties in managing this shader variety. In fact, shader management has been identified by many developers as one of their top problems. Given the combinatorial explosion that can otherwise occur, Mantle's programming model is designed with the expectation that future applications create a moderate number of linked pipelines (possibly hundreds or low thousands) to cover a variety of rendering scenarios and rely more on *uber-shaders* and data-driven approaches to manage the variety of rendering options.

Compute Pipelines

The compute pipeline encapsulates a compute shader and is created by calling grCreateComputePipeline() with a compute shader object handle in the pipeline creation parameters. It is invalid to specify GR_NULL_HANDLE for the compute shader.

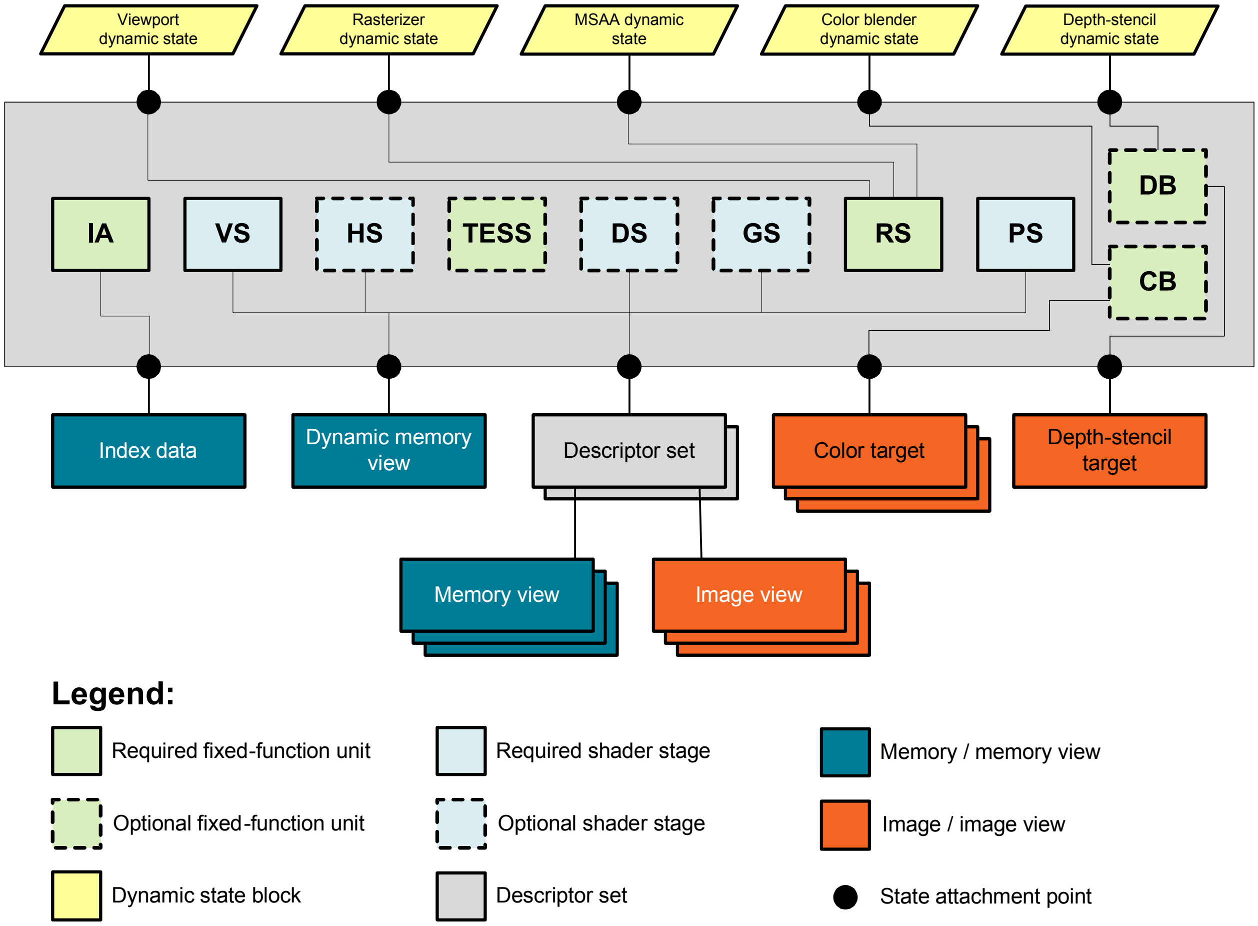
Listing 19. Example of creating a compute pipeline

```
GR_RESULT result;
// Create compute shader
GR_SHADER_CREATE_INFO shaderInfo = {};
shaderInfo.codeSize = sizeof(ShaderBytecode);
shaderInfo.pCode    = ShaderBytecode;
shaderInfo.flags    = 0;
GR_SHADER shader = GR_NULL_HANDLE;
result = grCreateShader(device, &shaderInfo, &shader);
// Setup resource mapping in descriptor sets
GR_DESCRIPTOR_SLOT_INFO resMapping[2] = {};
// Input image in IL slot t0
resMapping[0].slotObjectType    = GR_SLOT_SHADER_RESOURCE;
resMapping[0].shaderEntityIndex = 0;
// Output image in IL slot u0
resMapping[1].slotObjectType    = GR_SLOT_SHADER_UAV;
resMapping[1].shaderEntityIndex = 0;
// Setup pipeline info
GR_COMPUTE_PIPELINE_CREATE_INFO pipeInfo = {};
pipeInfo.cs.shader = shader;
pipeInfo.cs.descriptorSetMapping[0].descriptorCount = 2;
pipeInfo.cs.descriptorSetMapping[0].pDescriptorInfo = resMapping;
pipeInfo.cs.descriptorSetMapping[1].descriptorCount = 0;
pipeInfo.cs.descriptorSetMapping[1].pDescriptorInfo = NULL;
pipeInfo.cs.linkConstBufferCount = 0;
pipeInfo.cs.pLinkConstBufferInfo = NULL;
// Dynamic memory view in IL slot t1
pipeInfo.cs.dynamicMemoryViewMapping.slotObjectType = GR_SLOT_SHADER_RESOURCE;
pipeInfo.cs.dynamicMemoryViewMapping.shaderEntityIndex = 1;
pipeInfo.flags = 0;
// Create pipeline
GR_PIPELINE pipeline = GR_NULL_HANDLE;
result = grCreateComputePipeline(device, &pipeInfo, &pipeline);
// Shader no longer needed after pipeline creation
grDestroyObject(shader);
```

Graphics Pipelines

The graphics pipeline is created by calling `grCreateGraphicsPipeline()` according to the shader objects and the fixed function pipeline static state specified at creation time. An example of a full graphics pipeline configuration and its bound state is shown in Figure 13.

Figure 13. Graphics pipeline and its state



The nomenclature for shaders and fixed function blocks from the pipeline diagram are explained in Table 14.

Table 14. Graphics pipeline stages

Stage	Type	Description
IA	Fixed function	Input assembler
VS	Shader	Vertex shader
HS	Shader	Hull shader
TESS	Fixed function	Tessellator
DS	Shader	Domain shader
GS	Shader	Geometry shader
RS	Fixed function	Rasterizer

Stage	Type	Description
PS	Shader	Pixel shader
DB	Fixed function	Depth-stencil test and output
CB	Fixed function	Color blender and output

The following are the rules for building valid graphics pipelines:

- a vertex shader is always required, while other shaders might be optional, depending on pipeline configuration
- a pixel shader is always required for color output and blending, but is optional for depth-only rendering
- both hull and domain shaders must be present at the same time to enable tessellation

The presence of the shader stage in a pipeline is indicated by specifying a valid shader object. The application uses the `GR_NULL_HANDLE` value to indicate the shader stage is not needed. The presence of some of the fixed function stages in the pipeline is implicitly derived from enabled shaders and provided state. For example, the fixed function tessellator is always present when the pipeline has valid hull and domain shaders.

The following table lists the most common examples of valid graphics pipeline configurations.

Table 15. Examples of valid graphics pipeline configurations

Pipeline configuration	Description
IA-VS-RS-DB	Depth-stencil only rendering pipeline
IA-VS-RS-PS-DB	Depth/stencil only rendering pipeline with pixel shader (e.g., using pixel shader for alpha test)
IA-VS-RS-PS-CB	Color only rendering pipeline
IA-VS-RS-PS-DB-CB	Color and depth-stencil rendering pipeline
IA-VS-GS-RS-PS-DB-CB	Rendering pipeline with geometry shader
IA-VS-HS-TESS-DS-RS-PS-DB-CB	Rendering pipeline with tessellation
IA-VS-HS-TESS-DS-GS-RS-PS-DB-CB	Rendering pipeline with tessellation and geometry shader

Other pipeline configurations are possible, as long as they follow the rules outlined in this section of the document.

Graphics Pipeline Operation

In the Mantle environment, rendering is initiated by draw operations from a command buffer. Depending on the topology specified in a pipeline, the type of draw operation, and presence of index data, the vertex IDs are determined and provided to vertex shader threads. The vertex shader explicitly fetches vertices from bound resources or generates vertex data analytically, performs necessary computations, and outputs data. The vertex shader outputs are automatically forwarded to subsequent stages in the pipeline. Optionally, geometry could be further processed by tessellator and geometry shaders before it is rasterized. After geometry is rasterized, it is processed by the pixel shader and optionally forwarded to the color and depth fixed function units for processing and output.

Vertex Fetch in Graphics Pipeline

Unlike other APIs, Mantle does not provide special handling for vertex buffers and does not implicitly fetch vertex data before it is passed to the vertex shader. It is an application's responsibility to treat vertex buffers as any other memory views and generate vertex shaders to fetch them.

The vertex or index offset, as well the instance offset, in the case of instanced rendering, are passed as inputs to the vertex shader to compute a proper vertex ID.

Graphics Pipeline State

Each of the fixed-function stages of the pipeline has the static part of the state that is included in the pipeline.

Input Assembler Static State

The input assembler static state for the graphics pipeline is specified using the `GR_PIPELINE_IA_STATE` structure. The state includes information about primitive topology and vertex reuse.

The `GR_TOPOLOGY_PATCH` primitive topology is only valid for tessellation pipelines. Likewise, non-patch topologies cannot be used with tessellation pipelines. Mismatching primitive topology and tessellation fails graphics pipeline creation.

An application can use `disableVertexReuse` in the input assembler state to indicate that post-transform vertex reuse should be disabled. Normally vertex reuse should be enabled for better performance; however, there might be cases where more predictable execution of vertex or geometry shader is needed. This setting is just a hint and does not guarantee vertex reuse. Under some circumstances, vertex reuse might be disabled by the driver, even if the application allows it.

Vertex reuse should generally be disabled if any of the vertex shader, the geometry shader, or the tessellation shaders, write data out to memory or images.

Tessellator Static State

The tessellator static state for the graphics pipeline is specified using the GR_PIPELINE_TESS_STATE structure. The state includes information about the tessellation patches.

The tessellator static state is only used when valid hull and domain shaders are specified in the graphics pipeline. The patchControlPoints parameter is used to define the number of control points used by the pipeline. The number of control points must be greater than 0 and less than or equal to 32 when tessellation is enabled. It must be zero when tessellation is disabled.

The tessellator state includes the optimization hint (optimalTessFactor) that indicates for which tessellation factor to optimize the pipeline. For example, if an average tessellation factor for a set of objects rendering with the pipeline is expected to be around 7.0, the application could specify that value in optimalTessFactor. If the application is unsure of the optimal tessellation factor for the pipeline, the value should be left at zero.

Rasterizer Static State

The rasterizer static state for the graphics pipeline is specified using the GR_PIPELINE_RS_STATE structure. The state includes information about the depth clip mode.

Depth-stencil Static State

The depth-stencil static state for the graphics pipeline is specified using the GR_PIPELINE_DB_STATE structure. The state includes information about the depth-stencil target format that is used with the pipeline.

The pipeline depth-stencil format specification must match the actual the depth-stencil target format bound at draw time. Mismatch of the depth-stencil target and pipeline format expectations results in undefined behavior. If no depth-stencil is bound for rendering, the pipeline should specify the GR_CH_FMT_UNDEFINED channel format and the GR_NUM_FMT_UNDEFINED numeric format.

Color Output and Blender Static State

The color output and blender static state for the graphics pipeline is specified using the GR_PIPELINE_CB_STATE structure. The state includes information about color target formats, blending, and other color output options.

The blend enable and the color target format specified at pipeline creation must match the formats of the color target views bound at draw time. Mismatch of target formats or blend enable flags results in undefined rendering. If a target is not bound at draw time, its write mask has to be set to zero and the pipeline should specify GR_CH_FMT_UNDEFINED as the channel format and

GR_NUM_FMT_UNDEFINED as the numeric format for the target. For a valid color target output, the write mask should contain only channels present in the format.

When dual source blending is enabled (see Color Blender State), only a single color target can be specified and it must have blend enabled. A dynamic blender state object with dual source blending modes should only be used with pipelines enabling dual source blending. Any mismatch between the dual source blending pipeline declaration and the bound blender state object causes undefined results.

GR_LOGIC_OP_COPY is the default logic operation, choosing the unmodified source value. When the logic op is non-default, blending must be disabled for all color render targets. The logic operation may only be non-default on targets of GR_NUM_FMT_UINT and GR_NUM_FMT_SINT numeric formats, other formats fail pipeline creation.

Listing 20. Example of creating a graphics pipeline

```
GR_RESULT result;
// Setup resource mapping for vertex fetch
GR_DESCRIPTOR_SLOT_INFO resMapping = {};
resMapping.slotObjectType    = GR_SLOT_SHADER_RESOURCE;
resMapping.shaderEntityIndex = 0;
GR_GRAPHICS_PIPELINE_CREATE_INFO pipelineInfo = {};
// Vertex shader stage info
pipelineInfo.vs.shader = compiledVertexShader;
pipelineInfo.vs.descriptorSetMapping[0].descriptorCount = 1;
pipelineInfo.vs.descriptorSetMapping[0].pDescriptorInfo = &resMapping;
pipelineInfo.vs.dynamicMemoryViewMapping.slotObjectType = GR_SLOT_UNUSED;
// Pixel shader stage info
pipelineInfo.ps.shader = compiledPixelShader;
pipelineInfo.ps.dynamicMemoryViewMapping.slotObjectType = GR_SLOT_UNUSED;
// Fixed function state setup
pipelineInfo.iaState.topology                   = GR_TOPOLOGY_TRIANGLE_LIST;
pipelineInfo.rsState.depthClipEnable            = GR_FALSE;
pipelineInfo.cbState.logicOp                    = GR_LOGIC_OP_COPY;
pipelineInfo.cbState.target[0].blendEnable      = GR_FALSE;
pipelineInfo.cbState.target[0].channelWriteMask = 0xF;
pipelineInfo.cbState.target[0].format.channelFormat = GR_CH_FMT_R8G8B8A8;
pipelineInfo.cbState.target[0].format.numericFormat = GR_NUM_FMT_UNORM;
pipelineInfo.dbState.format.channelFormat       = GR_CH_FMT_R32G8;
pipelineInfo.dbState.format.numericFormat       = GR_NUM_FMT_DS;
// Create pipeline
GR_PIPELINE pipeline = GR_NULL_HANDLE;
result = grCreateGraphicsPipeline(device, &pipelineInfo, &pipeline);
```

“Re-Z” Support for Pixel Shaders

Whenever possible, depth-stencil test and update operations are performed in the graphics pipeline before pixel shader invocation to avoid its execution for pixels that should be discarded. This early depth and stencil processing (also known as *early-Z*) cannot be performed in cases when pixel shader exports new depth value, writes directly to images or memory, and when it discards pixels, such as for implementing alpha testing. For the latter case, some GPUs could support *re-Z* mode when pixel depth and stencil are conservatively tested before the pixel shader, but the actual update of depth buffer is performed after pixel shader. The re-Z operation could have performance penalty for small pixel shaders and is recommended only in case of fairly complex pixel processing.

The re-Z operation can be enabled by setting the GR_SHADER_CREATE_ALLOW_RE_Z flag at shader creation time. This flag is just a hint and is only applied when pixel shader does not request early-Z mode in the shader code. Applying re-Z flag to the pixel shader that requests early-Z has no effect on rendering. The flag is only available for pixel shaders and causes shader creation error for all other shader types.

Graphics Pipeline Shader Linking

Shaders in the pipeline are linked through matching the shader input and output registers by index. There is no semantic matching and it is a responsibility of the high-level compiler, or IL translator, or an application to guarantee compatible shader inputs and outputs.

Pipeline Serialization

For large and complex shaders, the shader compilation and pipeline construction could be quite a lengthy process. To avoid this costly pipeline construction every time an application links a pipeline, Mantle allows applications to save the pre-compiled pipelines as opaque binary objects and later load them back. An application only needs to incur a one-time pipeline construction cost on the first application run or even at application installation time. It is the application's responsibility to implement a pipeline cache and save/load binary pipeline objects.

A pipeline is saved to memory by calling grStorePipeline(). Before calling grStorePipeline(), the application should initialize the available data buffer size in the location pointed to by pDataSize. Upon completion, that location contains the amount of data stored in the buffer. To determine the exact buffer requirements, an application can call the grStorePipeline() function with NULL value in pData. The grStorePipeline() function fails if insufficient data buffer space is specified.

A pipeline object is loaded from memory with grLoadPipeline(). On loading a pipeline object, the driver performs a hardware and driver version compatibility check. If the versions of the current hardware and the driver do not match those of the saved pipeline, the pipeline load fails. The application is required to gracefully handle the failed pipeline loads and recreate the pipelines from scratch.

A pipeline can be saved and loaded with debug infrastructure enabled, which keeps internal data pertaining to debugging and validation in the serialized pipeline object. These versions of pipeline objects are intended for debugging only and cannot be loaded when validation is disabled. Mismatching debug capabilities of pipelines with validation currently enabled on device results in error.

Constant-Based Pipeline Compilation

There are some cases when it is not desirable to use uber-shaders for performance reasons and an application prefers to create variety of slightly specialized shaders. One way to implement such a variety of shader pipelines would be to pre-compile all possible shader versions off-line and use them for pipeline creation. The constant-based pipeline compilation feature available in Mantle reduces the need for off-line creation of a large number of similar shaders and simplifies the application's task of managing shaders when constructing pipelines.

The application is able to build uber-shaders with some constants that are not known at shader compilation time and are provided at the pipeline linkage time. Mantle uses constant buffer facilities available in shader IL to designate shader data that would be specified at pipeline linkage. The IL constant buffers in Mantle shaders can only be used for this purpose; an application must use conventional memory views for passing run-time data to shaders. Multiple link constant buffers per shader can be used, and each shader in a pipeline could have its own set of link time constant buffers. The constant data layout provided at pipeline link time must match the shader expectations, and all of the shader referenced constant data must be available for linking. Failing to match constant data layout or to provide sufficient amount of data results in undefined behavior.

The link time constants are specified per shader stage as a part of the `GR_PIPELINE_SHADER` structure when creating the pipeline.

Pipeline Binding

A pipeline object is bound to one of the pipeline bind points in the command buffer state by calling the `grCmdBindPipeline()` function. The pipeline bind point is specified in the `pipelineBindPoint` parameter and must match the creation type of the pipeline object being bound. Compute command buffers can only have compute pipelines bound and universal command buffers can have both graphics and pipeline bound.

As soon as a new pipeline object is bound within a command buffer, it remains in effect until another pipeline is bound or the command buffer is terminated. A pipeline object can be explicitly unbound by using `GR_NULL_HANDLE` for the pipeline parameter, leaving the pipeline in an undefined state. Pipeline unbinding is optional and should mainly be used for debugging.

PIPELINE RESOURCE ACCESS

Pipeline shaders access shader resources specified in descriptor sets bound to the command buffer state at the time of executing a draw or dispatch command. Additionally, a dynamic memory view can be used for buffer-like access to memory. The expected descriptor set layout and its mapping to shader resources is specified by the application at pipeline creation time.

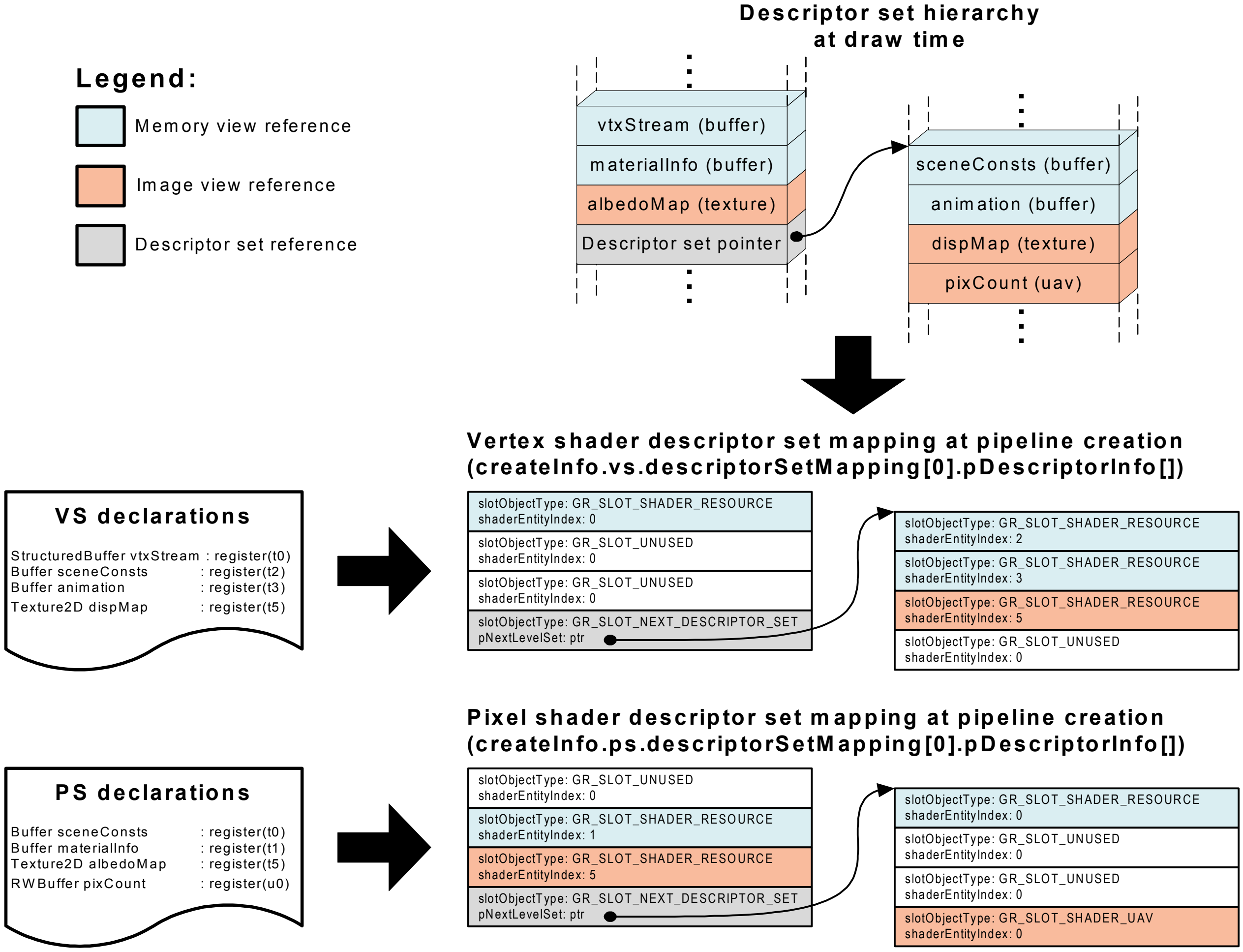
PIPELINE RESOURCE MAPPING

At the same time, the shaders themselves use the flat resource addressing scheme with different resource namespaces for distinct resource usages [e.g., read-only resources, read-write resources (a.k.a. unordered access views or UAVs), constant buffers], as specified in the IL definition, and have these separate namespaces for each pipeline shader. To reconcile these differences, mapping of shader resources and samplers to the descriptor sets is performed at pipeline construction time by means of the descriptor set remapping structures. If no mapping is specified, the pipeline creation fails. The mapping has to be provided for all resources that are declared by a given IL shader for all active shader stages. Even if it is known that resource access could be optimized out by the Mantle driver, it has to be present in remapping data if it is declared in IL. Failing to specify all shader resource mappings to the expected descriptor set hierarchy results in a pipeline creation failure.

The resource remapping structure can be different per shader stage, such that different shader resource slots can be mapped to the same descriptor set slot in a descriptor set hierarchy. If multiple shaders in a pipeline resolve to the same resource, their resource type expectations must match, otherwise pipeline creation fails. For example, a cubemap image from a pixel shader cannot be aliased to a resource slot that is expected to provide a buffer reference for a vertex shader.

The CPU side structures, used to describe descriptor set mapping, closely follow the desired descriptor set hierarchy as is referenced by the GPU. Each of the descriptor set slots in a bound range is represented by a structure describing the shader IL object type and the shader resource slot to which it maps. If a descriptor set element does not map to any IL shader resource, it must have the `GR_SLOT_UNUSED` object type specified. An indirection to the next level of descriptor set hierarchy is specified by using the `GR_SLOT_NEXT_DESCRIPTOR_SET` object type and a pointer to an array of next level descriptor set elements. A shader resource slot can be referenced only once in the whole descriptor set hierarchy. Specifying multiple references to a resource slot produce undefined results.

Figure 14. Example of hierarchical descriptor set mapping


Descriptor set hierarchy at draw time
Legend:
Memory view reference
Image view reference
Descriptor set reference
vtxStream (buffer)
materialInfo (buffer)
albedoMap (texture)
Descriptor set pointer
sceneConsts (buffer)
animation (buffer)
dispMap (texture)
pixCount (uav)
Vertex shader descriptor set mapping at pipeline creation (createInfo.vs.descriptorSetMapping[0].pDescriptorInfo[])
VS declarations
slotObjectType: GR_SLOT_SHADER_RESOURCE
shaderEntityIndex: 0
slotObjectType: GR_SLOT_UNUSED
shaderEntityIndex: 0
slotObjectType: GR_SLOT_UNUSED
shaderEntityIndex: 0
slotObjectType: GR_SLOT_NEXT_DESCRIPTOR_SET
pNextLevelSet: ptr
slotObjectType: GR_SLOT_SHADER_RESOURCE
shaderEntityIndex: 2
slotObjectType: GR_SLOT_SHADER_RESOURCE
shaderEntityIndex: 3
slotObjectType: GR_SLOT_SHADER_RESOURCE
shaderEntityIndex: 5
slotObjectType: GR_SLOT_UNUSED
shaderEntityIndex: 0
Pixel shader descriptor set mapping at pipeline creation (createInfo.ps.descriptorSetMapping[0].pDescriptorInfo[])
PS declarations
slotObjectType: GR_SLOT_UNUSED
shaderEntityIndex: 0
slotObjectType: GR_SLOT_SHADER_RESOURCE
shaderEntityIndex: 1
slotObjectType: GR_SLOT_SHADER_RESOURCE
shaderEntityIndex: 5
slotObjectType: GR_SLOT_NEXT_DESCRIPTOR_SET
pNextLevelSet: ptr
slotObjectType: GR_SLOT_SHADER_RESOURCE
shaderEntityIndex: 0
slotObjectType: GR_SLOT_UNUSED
shaderEntityIndex: 0
slotObjectType: GR_SLOT_UNUSED
shaderEntityIndex: 0
slotObjectType: GR_SLOT_SHADER_UAV
shaderEntityIndex: 0

```
StructuredBuffer vtxStream : register(t0)
Buffer sceneConsts         : register(t2)
Buffer animation           : register(t3)
Texture2D dispMap          : register(t5)
```

```
Buffer sceneConsts         : register(t0)
Buffer materialInfo        : register(t1)
Texture2D albedoMap        : register(t5)
RWBuffer pixCount          : register(u0)
```

Figure 14 shows an advanced example of the descriptor set remapping structures for a pipeline consisting of vertex and pixel shaders and a two level resource descriptor set hierarchy. An application should ensure there are no circular dependencies in the remapping structure or a soft hang in the driver might occur.

Listing 21. Resource mapping for vertex shader in the example above

```
// Resource mapping for nested desciptor set
GR_DESCRIPTOR_SLOT_INFO slotsVsNested[4] = {};
slotsVsNested[0].slotObjectType    = GR_SLOT_SHADER_RESOURCE;
slotsVsNested[0].shaderEntityIndex = 2;
slotsVsNested[1].slotObjectType    = GR_SLOT_SHADER_RESOURCE;
slotsVsNested[1].shaderEntityIndex = 3;
slotsVsNested[2].slotObjectType    = GR_SLOT_SHADER_RESOURCE;
slotsVsNested[2].shaderEntityIndex = 5;
slotsVsNested[3].slotObjectType    = GR_SLOT_UNUSED;
slotsVsNested[3].shaderEntityIndex = 0
// Nested descriptor set setup
GR_DESCRIPTOR_SET_MAPPING descSetVsNested = {};
mapVs1.descriptorCount = 4;
mapVs1.pDescriptorInfo = slotsVsNested;
// Resource mapping for vertex shader descriptor set
GR_DESCRIPTOR_SLOT_INFO slotsVs[4] = {};
slotsVs[0].slotObjectType    = GR_SLOT_SHADER_RESOURCE;
slotsVs[0].shaderEntityIndex = 0;
slotsVs[1].slotObjectType    = GR_SLOT_UNUSED;
slotsVs[1].shaderEntityIndex = 0;
slotsVs[2].slotObjectType    = GR_SLOT_UNUSED;
slotsVs[2].shaderEntityIndex = 0;
slotsVs[3].slotObjectType    = GR_SLOT_NEXT_DESCRIPTOR_SET;
slotsVs[3].pNextLevelSet     = &descSetVsNested;
// Descriptor set setup for vertex shader
GR_DESCRIPTOR_SET_MAPPING descSetVs = {};
descSetVs.descriptorCount = 4;
descSetVs.pDescriptorInfo = slotsVs;
```

Listing 22. Resource mapping for pixel shader in example above

```
// Resource mapping for nested desciptor set
GR_DESCRIPTOR_SLOT_INFO slotsPsNested[4] = {};
slotsPsNested[0].slotObjectType    = GR_SLOT_SHADER_RESOURCE;
slotsPsNested[0].shaderEntityIndex = 2;
slotsPsNested[1].slotObjectType    = GR_SLOT_UNUSED;
slotsPsNested[1].shaderEntityIndex = 0;
slotsPsNested[2].slotObjectType    = GR_SLOT_UNUSED;
slotsPsNested[2].shaderEntityIndex = 0;
slotsPsNested[3].slotObjectType    = GR_SLOT_SHADER_UAV;
slotsPsNested[3].shaderEntityIndex = 0
// Nested descriptor set setup
GR_DESCRIPTOR_SET_MAPPING descSetPsNested = {};
descSetPsNested.descriptorCount = 4;
descSetPsNested.pDescriptorInfo = slotsPsNested;
// Resource mapping for pixel shader descriptor set
GR_DESCRIPTOR_SLOT_INFO slotsPs[4] = {};
slotsPs[0].slotObjectType    = GR_SLOT_UNUSED;
slotsPs[0].shaderEntityIndex = 0;
slotsPs[1].slotObjectType    = GR_SLOT_SHADER_RESOURCE;
slotsPs[1].shaderEntityIndex = 1;
slotsPs[2].slotObjectType    = GR_SLOT_SHADER_RESOURCE;
slotsPs[2].shaderEntityIndex = 5;
slotsPs[3].slotObjectType    = GR_SLOT_NEXT_DESCRIPTOR_SET;
slotsPs[3].pNextLevelSet     = &descSetPsNested;
// Descriptor set setup for pixel shader
GR_DESCRIPTOR_SET_MAPPING descSetPs = {};
descSetPs.descriptorCount = 4;
descSetPs.pDescriptorInfo = slotsPs;
```

Descriptor Set Binding

Descriptor sets are bound to command buffer state using `grCmdBindDescriptorSet()`. There are separate descriptor sets for each pipeline type – graphics and compute. The pipeline bind point specified in `grCmdBindDescriptorSet()` indicates whether the descriptor sets should be available to the graphics or compute pipeline. For command buffers to be executed on compute queues, the only valid pipeline type option is `GR_PIPELINE_BIND_POINT_COMPUTE`.

Specifying `GR_NULL_HANDLE` for the descriptor set object unbinds the previously bound descriptor set. Failing to bind a descriptor set hierarchy that is compatible with the pipeline shader requirements produces undefined results.

Dynamic Memory View

One of the memory views referenced in a pipeline shader can be mapped to a *dynamic memory view*. The dynamic memory view can be used to send some frequently changing constants and other data to pipeline shaders without a need to manage descriptor sets. The dynamic memory view is directly bound to the command buffer state for a given pipeline type by describing the view defined by the GR_MEMORY_VIEW_ATTACH_INFO structure passed to the grCmdBindDynamicMemoryView() function. Use of the dynamic memory view is optional, but is highly encouraged.

The resource mapping for dynamic memory view is specified individually per shader stage. The dynamic memory view can be mapped to an IL resource slot or IL UAV slot by specifying the shader object type as GR_SLOT_SHADER_RESOURCE or GR_SLOT_SHADER_UAV, respectively. If a shader does not need to use a dynamic memory view, the shader object type in the mapping must be set to GR_SLOT_UNUSED.

The same shader resource cannot be specified in dynamic memory view mapping and descriptor set mapping of pipeline configuration info in the same shader. Specifying resource in both mappings fails the pipeline creation. However, multiple shaders within a pipeline might map resources differently.

It is invalid to specify dynamic memory view mapping for a shader resource slot that is used for non-buffer shader resources. Failure to match shader resource type produces undefined results.

Chapter VI.

Multi-device Operation

Overview

Mantle empowers applications to explicitly control multi-GPU operation and enables highly flexible and sophisticated solutions that could go far beyond *alternate frame rendering* (AFR) functionality. At the API level, each Mantle capable GPU in a system is presented as an independent device that is managed by an application. The GPUs that are part of the *linked adapter* in Windows®, such as in the case of AMD CrossFire™, are also presented in Mantle as separate devices, but with extra multi-device features.

The following features are exposed by the Mantle API for implementing multi-device functionality at the application level:

- Device discovery and identification
- Memory sharing
- Synchronization object sharing
- Peer-to-peer transfers
- Composition and cross-device presentation

This chapter focuses on multi-device operation in the Windows® OS environment.

Multi-device Configurations

Mantle supports many different platforms that range from a single GPU to various combinations of multiple GPUs. An application should detect available GPUs and determine the most appropriate GPU or set of GPUs according to the GPU capabilities, as well as the application requirements and applicable algorithms. The following provides a reference of most common configurations an application could target with Mantle.

Symmetric AMD CrossFire™ Configurations

A typical *symmetric CrossFire™ configuration* contains 2 to 4 GPUs of the same generation, features the same capabilities and performance. All GPUs in the enabled CrossFire™ configuration support a vast majority of advanced multi-device capabilities, such as peer-to-peer transfers, memory sharing, and so on. The support of AMD CrossFire™ technology is enabled in the AMD Catalyst™ control panel. If AMD CrossFire™ technology is disabled, none of the advanced multi-device capabilities might be available and that configuration can be classified as "Other Multi-device Configurations".

Asymmetric AMD CrossFire™ Configurations

Some of the GPUs of different capabilities and performance characteristics can be paired in *asymmetric AMD CrossFire™ configurations*. These are the most typical for laptop configurations with APU and discrete GPU. Generally, the performance ratio of discrete GPU to APU do not exceed 2:1 or 3:1. Just like symmetric AMD CrossFire™ technology, this type of configuration has to be enabled in the AMD Catalyst™ control panel to support advanced multi-device features.

Asymmetric Configurations not Supported by AMD CrossFire™

Some platforms might support discrete GPU and APU configurations with performance ratios exceeding 3:1 value. While these configurations are not classified as AMD CrossFire™ technology, they can expose some of the advanced multi-device capabilities.

Other Multi-device Configurations

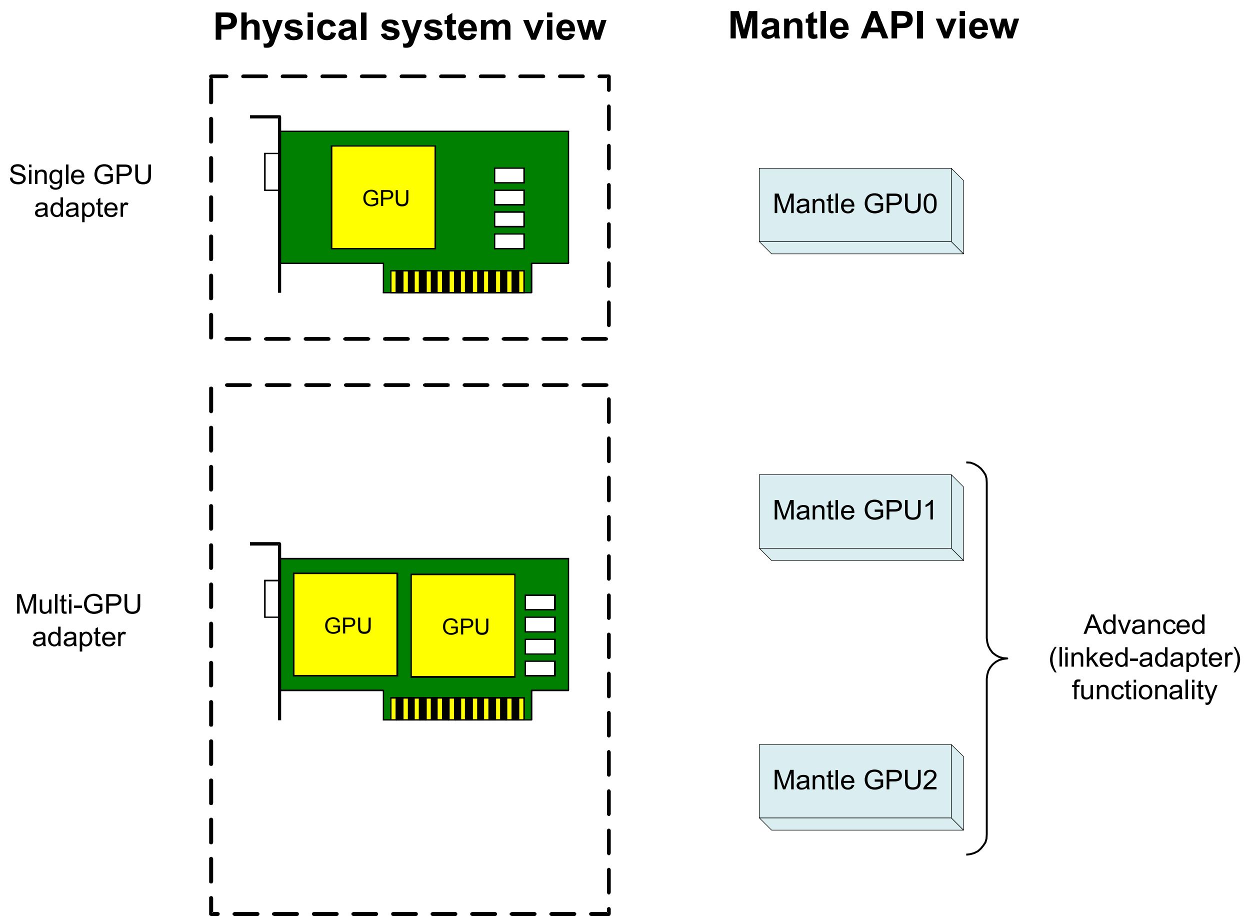
When AMD CrossFire™ technology is disabled in the AMD Catalyst™ control panel, or is unavailable for other reasons, multi-device capabilities might be limited or not available at all.

Multiple Devices

The overview of GPU device discovery and initialization was covered in GPU Identification and Initialization. Several additional aspects of device discovery have to be considered in the case of multiple Mantle GPUs. First, if multiple Mantle-capable GPU devices are present in the system, the application must decide which GPU or multiple GPUs are the best choice for executing rendering or other operations, and how to split workloads across devices, should it choose to target rendering on multiple GPUs. Second, if multiple Mantle GPUs are parts of the *linked adapter*, an application must discover what advanced multi-device functionality is available in AMD CrossFire™ configurations.

Figure 15 shows an example of a system with 2 graphic boards – one single GPU and another dual-GPU linked adapter (AMD CrossFire™ graphics board).

Figure 15. Multi-device configuration



GPU Device Selection

When deciding what GPU to use for rendering or other operations, an application looks at a number of different factors:

- Display connectivity
- General GPU capabilities
- Type of GPU
- Performance
- Multi-device capabilities

The discovery of display connectivity is addressed in Chapter IX. Window System Interface for Windows. In addition to display connectivity, WSI extension reports what displays can be used for *cross-device presentation*.

The general GPU capabilities and performance are reported by the Mantle core API using the `grGetGpuInfo()` function, as described in GPU Identification and Initialization. Along with that information, the device compatibility information allows applications to decide how to implement multi-device operation in the best possible way.

There are two aspects to device compatibility. The first aspect is matching of GPU features and image quality. The second aspect is the ability to use advanced multi-device functionality, which allows sharing of memory and synchronization objects, as well as compositing of displayable output. Not all GPUs or GPU combinations could expose these extra features. The multi-device compatibility can be queried with the `grGetMultiGpuCompatibility()` function. The compatibility information is returned in the `GR_GPU_COMPATIBILITY_INFO` structure containing various compatibility flags.

> In Windows® OS, the advanced multi-device features are only available when the AMD CrossFire™ technology mode is enabled in the AMD Catalyst™ Control Center.

Any devices created on compatible GPUs are considered compatible devices, inheriting the compatibility flags of the physical GPUs.

Image Quality Matching

Different generations of GPUs might produce images of slightly different quality. In particular, texture filtering is one area that is under constant improvement, both in terms of quality and performance. When using the alternate frame rendering mode, it is important to produce images of similar quality on the alternating frames.

If GPUs expose the `GR_GPU_COMPAT_IQ_MATCH` flag in multi-device capability info, they can be configured to produce similar image quality at device creation time by specifying `GR_DEVICE_CREATE_MGPU_IQ_MATCH` in device creation flags on all compatible GPUs. The Mantle driver attempts to match rendering quality between the supported GPUs as much as possible.

Sharing Memory between GPUs

The memory object residing in some of the non-local memory heaps can be made shareable across devices if the GPUs have the `GR_GPU_COMPAT_SHARED_MEMORY` flags set in the GPU compatibility information. A shared memory object is created on one device and opened on any other compatible device. Only the memory object associated with a particular device must be used, and it is not allowed to directly share memory object handles across devices.

There are several parts to enabling memory sharing across multiple Mantle devices:

- Discovery of heaps for shared memory
- Creation of shared memory object on one device
- Opening of shared memory object on another device

Discovery of Shareable Heaps

Memory heaps that could be used for creating shared memory objects are identified by the `GR_MEMORY_HEAP_FLAG_SHAREABLE` flag reported in heap properties. See GPU Memory Heaps for information on heap properties. If no heaps expose `GR_MEMORY_HEAP_FLAG_SHAREABLE`, the shared memory objects cannot be created. For devices with compatible memory capabilities, it is guaranteed that at least one heap is shareable.

Shared Memory Creation

A *shared memory object* is created just like any other regular memory objects using the `grAllocMemory()` function. A shared memory object is marked by the `GR_MEMORY_ALLOC_SHAREABLE` flag in its creation information and can only be created in heaps marked by the `GR_MEMORY_HEAP_FLAG_SHAREABLE` heap property flag.

A shared memory object created on one device can be opened on another compatible device using `grOpenSharedMemory()`. The shared memory object cannot be opened on the device on which it was created.

The opened memory object is associated with memory heaps of the second device equivalent to the heaps used for original shared object creation on the first device. Either device can be used for creating a shared memory object. The shared memory object created on the first device and opened on the second is functionally equivalent to the memory object created on the second device and opened on the first.

Opened memory objects have some limitations. They cannot have priority changed and they cannot be used for virtual allocation remapping.

Once no longer needed, opened memory objects are destroyed with `grFreeMemory()`. An opened memory object cannot be used once its corresponding shared memory object is freed, thus the shared memory object should not be freed until any of the devices stop using the corresponding opened memory objects.

Shared Images

The image data located in shared memory objects can be made shareable across multiple compatible devices by using *shared images*. The shared images are created on both devices with exactly the same creation parameters that include the `GR_IMAGE_CREATE_SHAREABLE` image creation flag. Then these images must be bound to a shared and opened memory object at the same offset. Shared images can only be used when the `GR_GPU_COMPAT_ASIC_FEATURES` flag is reported in GPU compatibility information.

Queue Semaphore Sharing

Queue semaphores can be made shareable across devices if the GPUs have the `GR_GPU_COMPAT_SHARED_SYNC` flags set in the GPU compatibility information. A shared semaphore should be created on one device and opened on other compatible devices. Only the semaphore object associated with the particular device can be used, and it is not allowed to directly share semaphore object handles across devices.

There are several parts to enabling creation of shared semaphores across multiple Mantle devices:

- Creation of shared queue semaphores on one device
- Opening of shared queue semaphores on another device

Shared Semaphore Creation

The shared queue semaphores is created just like any other regular semaphores using the `grCreateQueueSemaphore()` function. The shared queue semaphore object is marked by `GR_SEMAPHORE_CREATE_SHAREABLE` in object creation data. A shared queue semaphore behaves just like a regular queue semaphore object, but it could be signaled/waited on by queues from other compatible devices through their opened semaphore objects.

A shared queue semaphore created on one device can be opened on another compatible device using `grOpenSharedQueueSemaphore()`.

The shared semaphore cannot be opened on the device on which it was created. Just like with any other Mantle object, an application must query memory requirements for opened semaphore objects.

Either device can be used for creating a shared semaphore. The shared semaphore created on the first device and opened on the second is functionally equivalent to the semaphore created on the second device and opened on the first.

Once no longer needed, opened semaphores are destroyed with `grDestroyObject()`. An opened semaphore cannot be used, once a corresponding shared semaphore is destroyed. Thus, the shared semaphore must not be destroyed while any of corresponding opened semaphores are used on any of the devices.

Peer-to-peer Transfers

The memory and image objects data residing on a different GPU cannot be accessed by directly referencing their handles since only objects local to the device can be used for the GPU access. For optimal copying of image and other data between GPUs, an application uses *peer-to-peer write transfers*. These allow direct device-to-device writes over the PCIe bus without intermediate storage of data in system memory. It is not allowed to peer-to-peer read memory across GPUs. Mantle supports peer-to-peer transfers between GPUs if the `GR_GPU_COMPAT_PEER_WRITE_TRANSFER` flag is reported in GPU compatibility information.

There are several parts to enabling peer-to-peer transfers across multiple Mantle devices:

- Creation of *proxy* peer memory and optionally image objects on one of the devices, representing those objects from another device
- Executing transfers between memory or image local to the device and a peer memory or image

If an application wants to transfer memory from GPU0 to GPU1, it should create a proxy peer memory object on GPU0 for the target memory destination from GPU1. Then, it should transfer data on GPU0 using the proxy peer memory as a copy operation destination.

Opening Peer Memory

A memory object created on one device can be opened on another compatible device for peer access using `grOpenPeerMemory()`. A peer memory object cannot be opened on the device on which it was originally created. The original memory object has to be a real allocation.

Peer memory objects have some limitations. They cannot have priority changed, cannot be mapped, and they shouldn't be used for virtual allocation remapping. They should only be used as a destination for memory transfers.

Once no longer needed, peer memory objects are destroyed with `grFreeMemory()`. An opened peer memory object must be freed before a corresponding original memory object is freed. An original memory object should not be freed while any devices use corresponding peer memory objects for transfers.

Opening Peer Images

An image object created on one device can be opened on another compatible device for peer access using `grOpenPeerImage()`.

The `grOpenPeerImage()` function returns a peer image and a peer memory object to which the peer image is bound at the time of opening it. A valid memory object has to be bound to an original image before using it for opening the peer image, and the memory binding cannot be changed until associated peer images and memory objects are destroyed. A peer image object cannot be opened on the device on which it was originally created.

If both GPUs involved in a peer transfer have a `GR_GPU_COMPAT_ASIC_FEATURES` compatibility flag set, the peer transfer destination image can use `GR_OPTIMAL_TILING` tiling, otherwise, only `GR_LINEAR_TILING` must be used for the destination image.

Peer memory objects returned by `grOpenPeerImage()` have limitations regarding their use. These memory objects must only be used for memory references in command buffers that perform peer-to-peer image transfers. Peer images cannot be rebound to other memory objects.

Once no longer needed, peer images are destroyed with `grDestroyObject()`. An opened peer image object must be destroyed before a corresponding original image object is destroyed. An original image object must not be destroyed while any devices use corresponding peer image objects for transfers. The memory objects returned for peer images should not be freed by the application and are automatically disposed of by the driver on peer image destruction.

Peer Transfer Execution

The peer memory or image object should only be used as a destination for peer-to-peer transfers. They should not be used for any other purpose, such as binding as shader resources, etc.

> ! For performance and power efficiency reasons, it is recommended to use DMA queues for peer-to-peer transfers whenever possible.

Before a peer transfer can take place, the source and destination memory or images have to be transferred to `GR_MEMORY_STATE_DATA_TRANSFER` and `GR_IMAGE_STATE_DATA_TRANSFER` states. Specialized data transfer states cannot be used for peer transfers. Peer images cannot be in a queue-specific data transfer state. The state transitions for peer transfer have to be performed on devices owning the original memory objects or images. There is no need to prepare peer objects as they inherit the state of the original objects.

Compositing and Cross-device Presentation

Some multi-device Mantle configurations might include the display compositing capabilities for automatically transferring and displaying images between the GPUs using the dedicated hardware. The hardware compositor in Mantle is abstracted with cross-device presentation functionality.

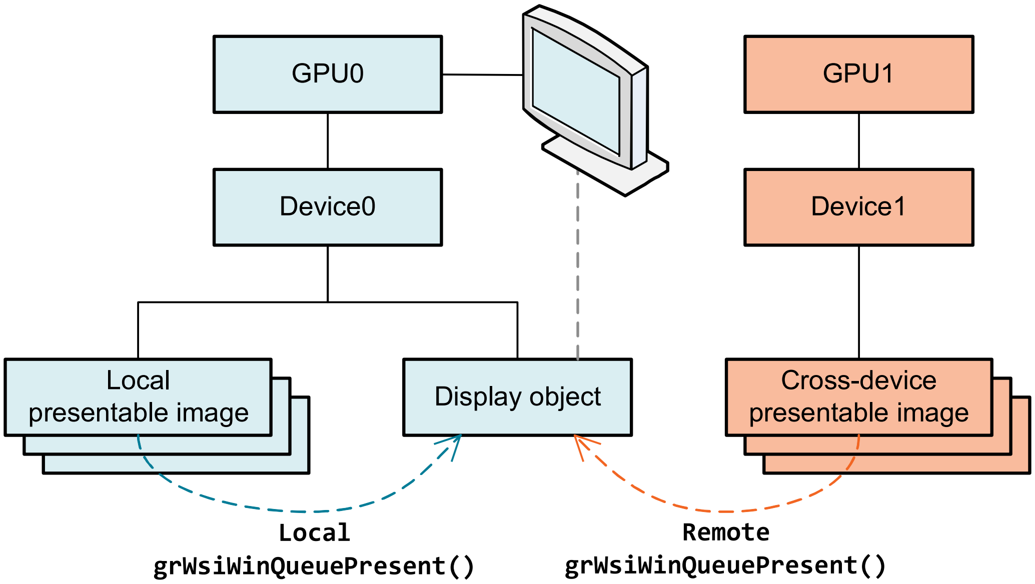
The automatic cross-device presentation is only available on compatible devices and only in full screen mode. In a windowed mode, it is an application's responsibility to transfer, composite, and present rendered images across the GPUs. In some display modes, the automatic cross-device presentation might not be available due to hardware compositor restrictions.

The cross-device presentation is based on the following steps:

- Discovering devices capable of sharing displays
- Checking if desired display modes supports cross-device presentation
- Creating special presentable images local to each of the compatible devices
- Presenting from compatible devices to a shared display

Figure 16 shows a conceptual diagram of cross-device presentation in a multi-device configuration with a single logical Mantle display.

Figure 16. Conceptual view of cross-device presentation


GPU0
GPU1
Device0
Device1
Local presentable image
Display object
Cross-device presentable image
Local grWsiWinQueuePresent()
Remote grWsiWinQueuePresent()

Discovering Cross-device Display Capabilities

An application detects if a GPU can present to a display from another GPU by examining `GR_GPU_COMPAT_SHARED_GPU0_DISPLAY` and `GR_GPU_COMPAT_SHARED_GPU1_DISPLAY` compatibility flags. If neither flag is not present, a *software compositing* implemented by an application should be used. If cross-device presentation is supported, an application should further check if it is available for a particular display mode. A per-mode support is reported in the `crossDisplayPresent` member of the `GR_WSI_WIN_DISPLAY_MODE` structure returned by `grWsiWinGetDisplayModeList()`.

In the case of software compositing, an application needs to transfer the final image across devices and present it locally on the desired device through a standard WSI full screen presentation mechanism (see Presentation).

Cross-device Presentable Images

Cross-device presentable images created by the WSI could be used for cross-device presentation. Any presentable image created for a display that belongs to another device is assumed to be cross-device presentation compatible. Cross-device presentable image creation fails if hardware compositing between devices is not available for the requested resolution. In case a presentable image cannot be created, an application falls back to a software compositing.

If multiple Mantle display objects are present in the system, it is an application's responsibility to split rendering on a per-display basis and manage separate presentable images for each of the displays.

Cross-device Presentation

From the application's perspective, the cross-device presentation is performed just like in a single device scenario. If there are multiple shared displays in a system, multiple presentation calls should be made – one per display.

Cross-device presentable images must only be presented from the device on which they were created. If the display associated with a presentable image is a display from another device, the presentation must only be performed in full screen mode. An attempt to present across devices in windowed mode fails.

If at any time cross-device presentation fails, it is required to switch to the application implemented software compositing fallback that transfers the presentable image to the device with the display attached and presents it locally.

Chapter VII.

Debugging and Validation Layer

The debug features are fundamental to the successful use of the Mantle API due to its lower-level nature – there are a lot of features that might be challenging to get right in Mantle without proper debugging and validation support. Additionally, for performance reasons, Mantle drivers perform only a very limited set of checks under normal circumstances, so it becomes even more important to validate the application operation with Mantle debug options enabled.

The Mantle debug infrastructure is layered on top of the core Mantle implementation and is enabled by specifying a debug flag at device creation time. The debug infrastructure provides a variety of additional checks and options to validate the use of the Mantle API and facilitate debugging of intermittent issues. The layered implementation allows significantly reducing the cost of debugging in release builds of the application.

Debug Device Initialization

The debugging and profiling infrastructure can be enabled on a per device basis by specifying the `GR_DEVICE_CREATE_VALIDATION` flag at device creation. Additionally, a maximum validation level that can be enabled at run time is specified at device creation. Without the `GR_DEVICE_CREATE_VALIDATION` flag, the maximum debug level has to be set to `GR_VALIDATION_LEVEL_0`.

Validation Levels

The debugging infrastructure is capable of detecting a variety of errors and suboptimal performance conditions, ranging from invalid function parameters to issues with object and memory dependencies. The cost of the error checking can also vary from very lightweight operations to some really expensive and thorough checking. To provide control over the performance and safety tradeoffs, Mantle introduces a concept of validation levels. Lower validation levels perform relatively lightweight checks, while higher levels perform increasingly more expensive validation.

There are two parts to specifying a desired validation level. First, the maximum validation level that can later be enabled has to be specified at device creation time. Setting the maximum validation level does not perform the validation, but internally enables tracking of additional object meta-data that are required for the validation at that level. This internal tracking introduces some additional CPU overhead, and the maximum validation level should be only as high as you actually intend to validate at run-time. Requesting higher than necessary maximum validation level has a higher impact on performance.

The second part is actually enabling a particular level of validation at run-time by calling `grDbgSetValidationLevel()`.

Setting the validation level is not a thread-safe operation. Additionally, when changing the validation level, an application should ensure it is not in the middle of building any command buffers. Switching the validation level while constructing command buffers leads to undefined results.

> Since higher validation level used at run-time causes bigger performance impact, it is recommended to avoid running with high validation levels if performing performance profiling. Validation should not be enabled in the publicly available builds of your application.

It is invalid to set the validation level higher than the maximum level specified at device creation, and the function call fails in that case. A particular level of validation implies that all lower-level validations are also performed. See `GR_VALIDATION_LEVEL` for description of various validation levels.

Debugger Callback

When running with the debugging infrastructure enabled and an error or a warning condition is encountered, the error or warning message could be logged to debug output. Additionally, an application or debugging tools could register a debug message callback function to be notified about the error or warning condition. The callbacks are globally registered across all devices enumerated by the Mantle environment and multiple callbacks can be simultaneously registered.

For example, an application could independently register a callback, as well as the debugger could register its own callback function. If multiple callback functions are registered, their execution order is not defined.

An application registers a debug message callback by calling `grDbgRegisterMsgCallback()`. The callback function is an application's function defined by the `GR_DBG_MSG_CALLBACK_FUNCTION` type. A callback function provided by an application must be re-entrant, as it might be simultaneously called from multiple threads and on multiple devices. It is allowed to register a debug message callback before Mantle is initialized.

When it no longer needs to receive debug messages, an application unregisters the callback with `grDbgUnregisterMsgCallback()`. These functions are valid even when debug features are not enabled on a device; however, only functions related to device creation and ICD loader operation generate callback messages and message filtering is not available.

These debugger callback handling functions are not thread safe. If an error occurs inside of the `grDbgRegisterMsgCallback()` or `grDbgUnregisterMsgCallback()` functions, an error code is returned, but it is not reported back to an application via a callback.

Debug Message Filtering

Sometimes the volume of error or warning messages can be overwhelming and an application might chose to temporarily ignore some of them during a debugging session. For example, during development, one might want to temporarily disregard specific performance warnings. An application filters the messages by calling `grDbgSetMessageFilter()`. Previously disabled messages are re-enabled at any time by specifying `GR_DBG_MSG_FILTER_NONE`. This function is only valid for devices created with debug features enabled.

The debug message filtering function is not thread safe. If an error occurs inside of the function, an error code is returned, but it is not reported back to an application via a callback. The errors generated by the ICD loader cannot be filtered.

> Debug message filtering should be considered a special debug feature that should be carefully used only when absolutely necessary during development and debugging. It should not be used when validating an application for correctness.

Object Debug Data

The validation infrastructure provides a wealth of debugging information to assist tools and applications with debugging and analysis of rendering. The following information can be retrieved:

- Application set object tags
- Internal debug and validation information

Object Tagging

When the debug infrastructure is enabled, an application can *tag* any Mantle object other than the GR_PHYSICAL_GPU by attaching a binary data structure containing application specific object information. One use of such annotations could be for identifying the objects reported by the debug infrastructure to an application on the debug callback execution. When the debug infrastructure is disabled, tagging functionality has no effect.

An application tags an object with its custom data by calling grDbgSetObjectTag(). Specifying a NULL pointer for the tag data removes any previously set application data. Only one tag can be attached to an object at any given time. The tag data are copied by the Mantle driver when grDbgSetObjectTag() is called.

To retrieve a previously set object tag, an application calls grGetObjectInfo() with the GR_DBG_DATA_OBJECT_TAG debug data type.

Generic Object Information

An application retrieves object type for any API object other than the GR_PHYSICAL_GPU by calling grGetObjectInfo() with the GR_DBG_DATA_OBJECT_TYPE information type. The returned data are defined by GR_DBG_OBJECT_TYPE. Once the object type is known, the object creation information can be retrieved by calling grGetObjectInfo() with the GR_DBG_DATA_OBJECT_CREATE_INFO information type. The type of the returned object creation information for core API objects is described in Table 16 and for objects defined in various extensions in Table 17. Please note that some object types do not support reporting of creation time information.

Table 16. Returned core object creation information

Object type	Returned information
GR_DBG_OBJECT_DEVICE	GR_DEVICE_CREATE_INFO followed by additional data
GR_DBG_OBJECT_QUEUE	N/A
GR_DBG_OBJECT_GPU_MEMORY	GR_MEMORY_ALLOC_INFO
GR_DBG_OBJECT_IMAGE	GR_IMAGE_CREATE_INFO
GR_DBG_OBJECT_IMAGE_VIEW	GR_IMAGE_VIEW_CREATE_INFO
GR_DBG_OBJECT_COLOR_TARGET_VIEW	GR_COLOR_TARGET_VIEW_CREATE_INFO
GR_DBG_OBJECT_DEPTH_STENCIL_VIEW	GR_DEPTH_STENCIL_VIEW_CREATE_INFO
GR_DBG_OBJECT_SHADER	GR_SHADER_CREATE_INFO followed by shader code
GR_DBG_OBJECT_GRAPHICS_PIPELINE	GR_GRAPHICS_PIPELINE_CREATE_INFO followed by additional data
GR_DBG_OBJECT_COMPUTE_PIPELINE	GR_COMPUTE_PIPELINE_CREATE_INFO followed by additional data
GR_DBG_OBJECT_SAMPLER	GR_SAMPLER_CREATE_INFO
GR_DBG_OBJECT_DESCRIPTOR_SET	GR_DESCRIPTOR_SET_CREATE_INFO
GR_DBG_OBJECT_VIEWPORT_STATE	GR_VIEWPORT_STATE_CREATE_INFO
GR_DBG_OBJECT_RASTER_STATE	GR_RASTER_STATE_CREATE_INFO
GR_DBG_OBJECT_MSAA_STATE	GR_MSAA_STATE_CREATE_INFO
GR_DBG_OBJECT_COLOR_BLEND_STATE	GR_COLOR_BLEND_STATE_CREATE_INFO
GR_DBG_OBJECT_DEPTH_STENCIL_STATE	GR_DEPTH_STENCIL_STATE_CREATE_INFO
GR_DBG_OBJECT_CMD_BUFFER	GR_CMD_BUFFER_CREATE_INFO
GR_DBG_OBJECT_FENCE	GR_FENCE_CREATE_INFO
GR_DBG_OBJECT_QUEUE_SEMAPHORE	GR_QUEUE_SEMAPHORE_CREATE_INFO
GR_DBG_OBJECT_EVENT	GR_EVENT_CREATE_INFO
GR_DBG_OBJECT_QUERY_POOL	GR_QUERY_POOL_CREATE_INFO
GR_DBG_OBJECT_SHARED_GPU_MEMORY	GR_MEMORY_OPEN_INFO

Object type	Returned information
GR_DBG_OBJECT_SHARED_QUEUE_SEMAPHORE	GR_QUEUE_SEMAPHORE_OPEN_INFO
GR_DBG_OBJECT_PEER_GPU_MEMORY	GR_PEER_MEMORY_OPEN_INFO
GR_DBG_OBJECT_PEER_IMAGE	GR_PEER_IMAGE_OPEN_INFO
GR_DBG_OBJECT_PINNED_GPU_MEMORY	GR_SIZE
GR_DBG_OBJECT_INTERNAL_GPU_MEMORY	N/A

Creation data for graphics and compute pipelines can only be retrieved for explicitly created pipeline objects. Creation information for pipelines loaded with grLoadPipeline() cannot be retrieved.

Table 17. Returned extension object creation information

Object type	Returned information
GR_WSI_WIN_DBG_OBJECT_DISPLAY	N/A
GR_WSI_WIN_DBG_OBJECT_PRESENTABLE_IMAGE	GR_WSI_WIN_PRESENTABLE_IMAGE_CREATE_INFO
GR_EXT_DBG_OBJECT_BORDER_COLOR_PALETTE	GR_BORDER_COLOR_PALETTE_CREATE_INFO
GR_EXT_DBG_OBJECT_ADVANCED_MSAA_STATE	GR_ADVANCED_MSAA_STATE_CREATE_INFO
GR_EXT_DBG_OBJECT_FMASK_IMAGE_VIEW	GR_FMASK_IMAGE_VIEW_CREATE_INFO

For creation data of variable size, an application should first determine the returned data size by calling grGetObjectInfo() with pData set to NULL.

Memory Object Information

An application retrieves internal debug information for various GPU memory objects by calling grGetObjectInfo() with an appropriate information type. Specifying the GR_DBG_DATA_MEMORY_OBJECT_LAYOUT information type returns a list of structures describing memory binding for API objects associated with the GPU memory object, while the GR_DBG_DATA_MEMORY_OBJECT_STATE information type returns a list of memory regions and their current memory state. The returned data are defined in GR_DBG_MEMORY_OBJECT_LAYOUT and GR_DBG_MEMORY_OBJECT_STATE accordingly. Memory state information is only available when validation level is set to GR_VALIDATION_LEVEL_3 or higher.

Command Buffer Trace

An application retrieves the API command trace recorded in a command buffer by calling `grGetObjectInfo()` with the `GR_DBG_DATA_CMD_BUFFER_API_TRACE` information type. The returned data are an array of API command structures. Each command structure starts with a `GR_DBG_OP_HEADER` header, followed by variable size payload according to the command token type. The end of the command buffer is marked by the `GR_DBG_OP_CMD_BUFFER_END` token.

Whenever an API object is referenced in recorded command buffer, the validation layer records the object's memory binding at the time of the command buffer recording. This memory binding information can be used by the application for playback and validation purposes.

Queue Semaphore State

To prevent hardware deadlocks, the Mantle driver stalls submission and other queue operations if semaphore is waited on before an application issues an appropriate signal command. To help debugging a case of software deadlock due to a queue semaphore waited on in the driver, an application retrieves semaphore status by calling `grGetObjectInfo()` with the `GR_DBG_DATA_SEMAPHORE_IS_BLOCKED` information type. The returned data type is `GR_BOOL`. This operation cannot be used to determine the actual hardware status of the queue semaphore.

Command Buffer Markers

For debugging and inspection purposes, an application could retrieve a list of API operations recorded in a command buffer, as described in Internal Debug and Validation Information. To aid with command buffer inspection and add some context to recorded commands, an application could record command buffer markers – arbitrary strings that have a meaning to an application or tools. These markers have no effect on the actual content of the command buffer data executed by the GPU, and with the validation layer enabled, the markers are just kept along with other CPU side meta-data for command buffers. The command buffer markers can be inserted using the `grCmdDbgMarkerBegin()` and `grCmdDbgMarkerEnd()` functions.

Tag and Marker Formatting Convention

Object tags and command buffer markers can follow a defined convention to be debug tools "friendly." If the object tag or marker starts with a non-text character, the information is not understood by the tools that are only looking for text information. Additionally, special markup data can be embedded in tags and markers to control the display of objects in tools. Table 18 and Table 19 list examples of markup for tags and markers.

Table 18. Examples of object tags

Tag data	Tool behavior
0, "Don't show this tag"	This tag is not interpreted by the tool
"My object"	This tag is interpreted and displayed by the tool
"My object<color>55bbcc</color>"	This tag is interpreted and displayed by the tool using the preferred color, if possible

Table 19. Examples of command buffer markers

Tag data	Tool behavior
0, "Don't show this marker"	This marker is not interpreted by the tool
"My marker"	This marker is interpreted and displayed by the tool
"My marker" “<a href=http://my.link.com">Mesh</a>	This marker is interpreted and displayed by the tool along with the hyper-linked user text

Debug Infrastructure Settings

The debug infrastructure has various settings that can be used during debugging to force specific Mantle driver and GPU behaviors. Some of the options are set globally for all devices and some are set per device. The per-device settings functionality is only available on devices created with debug features enabled. The global optional settings are configured using `grDbgSetGlobalOption()` and per-device settings are configured with `grDbgSetDeviceOption()`. These functions are not thread safe. If an error occurs inside of these functions, an error code is returned, but it is not reported back to an application via a callback.

Chapter VIII.

Mantle Extension Mechanism

The Mantle API provides a common feature set that can be supported by multiple GPU generations on different platforms. The optional features and capabilities for different platforms or different GPUs, as well as other new and experimental functionality could be exposed through the extension mechanism without providing a new revision of the API.

The extensions can be broadly broken into these categories:

- Platform specific extensions, such as windowing system bindings, other API interoperability, etc.
- GPU-specific extensions

The platform specific extension entry points are supplied by the ICD loader library, while the entry points for ASIC specific extensions are supported by additional extension libraries.

Logically, the additions in the extension could be grouped in the following functional areas:

- New API functions not used for building command buffers
- New API functions used for building command buffers
- New run-time behavior without changes to the API functions
- New shader ILs and new IL instructions

The extension functionality can only be used if an extension is requested to be enabled at device creation time. The GPU-specific extension entry points from additional extension libraries can only be used with API objects that belong to devices created for physical GPUs exposing the particular extension. A particular extension's functionality can only be used if the extension is supported.

Calling extension functions or using structures and enumeration types and values defined by an unsupported extension or an extension that was not registered at device creation produces undefined results and returns an `GR_ERROR_INVALID_EXTENSION` error code where applicable.

Extension Discovery

All extensions are named for discoverability and referencing purposes using null-terminated strings. Support for a particular extension is queried on a physical GPU device by calling `grGetExtensionSupport()` with an extension name as a parameter. The extension name used is case sensitive. If the extension is not supported, the `GR_UNSUPPORTED` return code is returned.

AMD Extension Library

All AMD specific extensions are provided in the *AMD extension library* (AXL), which is shipped as a part of the AMD Mantle driver. The application must ensure the AXL version to which it is built is compatible with the version reported by `grGetExtensionLibraryVersion()`. The library is compatible when its major version is the same and minor version is greater than or equal to the version used by the application.

The extension library version included with the driver is guaranteed to match the driver functionality. If the application redistributes the AXL independently from the rest of the Mantle driver, it needs to ensure the Mantle driver supports the AXL version used by the application. The range of supported AXL versions is retrieved for the physical GPU with the `GR_EXT_INFO_TYPE_PHYSICAL_GPU_SUPPORTED_AXL_VERSION` information type used with `grGetGpuInfo()`.

Chapter IX.

Window System Interface for Windows®

Extension Overview

The *window system interface* (WSI) extension provides interoperability with the Microsoft Windows® windowing system. It implements the mechanism through which images rendered by the Mantle API can be presented to visible windows. Unlike some other presentation APIs, this extension focuses only on presentation functionality, and leaves other OS-dependent features, such as cross-process resource sharing, to other parts of the API.

Much like the core Mantle API, this extension exposes a powerful, lower-level interface that pushes responsibility to the application in order to reduce the driver's software overhead.

The WSI extension supports displaying rendered images to the screen, either in windowed or fullscreen mode. The application creates a set of images compatible with the destination window, and manages a swap chain by presenting one of these images each frame. Presentable images are created by calling `grWsiWinCreatePresentableImage()`, and can be used as standard Mantle images with some restrictions.

Applications running in fullscreen mode require additional setup, but have access to features such as efficient presents via page flipping, programmable gamma ramp support, and stereoscopic display support. Functions are provided to query displays attached to a Mantle device, create images representing a swap chain which are compatible with a particular display, take exclusive ownership of a display, control a display's resolution, etc. A fullscreen application is responsible for taking fullscreen ownership of a display before performing presentation on that display.

Extension Discovery

Support for the WSI for Windows extension is queried by calling grGetExtensionSupport() with the string "GR_WSI_WINDOWS". Applications should only expect this extension to be available on platforms running Microsoft Windows® 7 or later; other platforms return GR_UNSUPPORTED.

A Mantle device that intends to use this extension must include "GR_WSI_WINDOWS" in the ppEnabledExtensionNames list at creation. The extension functions themselves are exported by the loader DLL (mantle32.dll and mantle64.dll), like core API functions.

Windowed Mode Overview

In windowed mode, an application creates a set of presentable images as described in Presentable Images and presents them to a window using *windowed mode presentation*.

Fullscreen Mode Overview

In fullscreen mode application queries and configures available displays attached to a Mantle device. Once an appropriate mode is selected, an application creates a set of presentable images compatible with the fullscreen presentation and enters a fullscreen mode. Once in a fullscreen mode, an application presents the images using *fullscreen mode presentation*.

Display Objects

A *display object* represents a display connected to a GPU. An application can retrieve a list of displays attached to a device by calling grWsiWinGetDisplays(). Each GR_WSI_WIN_DISPLAY object returned by the driver represents a single logical display[1]. When no longer used by the application, display objects should be destroyed by calling grDestroyObject(). Once the device is destroyed, attempting to use an associated display results in undefined behavior.

An application can query display properties by calling grGetObjectInfo() with an information type of GR_WSI_WIN_INFO_TYPE_DISPLAY_PROPERTIES. The display properties are returned in the GR_WSI_WIN_DISPLAY_PROPERTIES structure. This information should be used for selecting displays, correctly dealing with display orientation, as well as determining displays' spacial relationship to each other.

[1] On multi-monitor (i.e., Eyefinity platforms), one logical display may correspond to multiple physical displays that are configured to collectively display one contiguous primary surface.

An application can also query the list of modes supported by a display with `grWsiWinGetDisplayModeList()`. The mode information returned in `GR_WSI_WIN_DISPLAY_MODE` structure is used by the application to determine availability of fullscreen resolutions, support of stereo 3D and cross-device presentation for multi-device configurations.

Display objects may become invalid in some cases, such as if a display is removed from the Windows® desktop. In such cases, functions accepting a display argument return `GR_ERROR_DISPLAY_REMOVED` error code, and the application must destroy and recreate any objects associated with the invalidated displays.

Fullscreen Exclusive Mode

Before an application can enter a fullscreen mode, it needs to create a set of *presentable images* compatible with fullscreen presentation to form a fullscreen swap chain. The specified images must be created via `grWsiWinCreatePresentableImage()` by specifying the `GR_WSI_WIN_IMAGE_CREATE_FULLSCREEN_PRESENT` flag and must be associated with this display (see Presentable Images). After these images are created, the application switches to desired display mode with `ChangeDisplaySettings()` or `ChangeDisplaySettingsEx()`. Then the application enters a fullscreen exclusive mode by calling `grWsiWinTakeFullscreenOwnership()`. The function accepts an image object, which should be one of the images intended to be used for presentation. On successful execution, the display performs a mode set to match the specified image and the Mantle application takes exclusive ownership of the screen. Efficient *fullscreen presentation* may only be performed once exclusive ownership is established.

The application relinquishes fullscreen exclusive mode by calling `grWsiWinReleaseFullscreenOwnership()`. Applications must release fullscreen ownership and restore previous display mode before destroying an associated device. Furthermore, the application must respond to losing focus (on `WM_KILLFOCUS` events) by releasing fullscreen ownership and retaking fullscreen ownership when appropriate (on a subsequent `WM_SETFOCUS` event).

> Taking or releasing fullscreen ownership does not change the display mode. The application is responsible for changing the display mode before entering fullscreen and restoring the previous display mode after releasing the fullscreen ownership.
>
> Switching display mode might take some time. Taking the fullscreen ownership fails if the display mode switch has not completed. The application is advised to continue attempting to enter fullscreen after it issued a mode switch.

An application that wishes to use fullscreen presentation on multiple displays takes ownership of all displays and creates a separate swap chain image for each target display.

Gamma Ramp Control

When an application has exclusive ownership of a display, it may override the display's gamma correction properties. The application specifies its desired gamma ramp with a description of a step-wise linear function. The function is described by a set of gamma corrected output values for a set of system-defined input control points, and an optional scale and bias that are applied post-conversion. After entering fullscreen exclusive mode on a display, the application queries the gamma capabilities of that display by calling grGetObjectInfo() with GR_WSI_WIN_INFO_TYPE_GAMMA_RAMP_CAPABILITIES information type. The gamma ramp capabilities are reported in the GR_WSI_WIN_GAMMA_RAMP_CAPABILITIES structure. The function fails if display is invalid or it is not in a fullscreen exclusive mode. The application should re-query the capabilities any time it re-enters fullscreen exclusive mode, since the capabilities (min/max value and control points) may vary based on the display mode.

The application specifies a new gamma ramp by calling grWsiWinSetGammaRamp(). The number of gamma curve control points, their positions, as well as the scale and offset values, should match the reported gamma ramp capabilities. The gamma ramp setting fails if the display is invalid, or if the display is not in fullscreen exclusive mode. If the capabilities report scale and offset are not supported, then the application must always set scale to 1.0 and offset to 0.0. The application must ensure that all values in gamma curve values are between the minConvertedValue and maxConvertedValue reported in the capabilities.

The gamma ramp is reset when exiting fullscreen exclusive mode. It is up to the application to restore it when returning to fullscreen exclusive mode.

CPU/Display Coordination

A couple functions are provided to allow the application to coordinate work it is doing on the CPU with the timing of a particular display.

The grWsiWinWaitForVerticalBlank() function waits for the vertical blanking interval to occur on the specified display then returns. Calling this before present can be used to prevent tearing artifacts in windowed applications. The function returns GR_SUCCESS if it successfully waited for a vertical blank on the specified display, or an error otherwise.

For more detailed coordination between the CPU and display, the application can query the line currently being scanned out by a particular display by calling grWsiWinGetScanLine(). The value returned at the pScanLine location is the current scan line. A value of -1 indicates the display is currently in its vertical blanking period.

The CPU/display coordination is always available in fullscreen mode. In windowed mode, this coordination is only available when extended display properties, queried with the

GR_WSI_WIN_INFO_TYPE_EXTENDED_DISPLAY_PROPERTIES info type, report such capability with GR_WSI_WIN_WINDOWED_VBLANK_WAIT and GR_WSI_WIN_WINDOWED_GET_SCANLINE flags.

Presentation

Presentation is the process by which applications can display the contents of an image on the screen, either in a window or fullscreen.

Presentable Images

Presentable images are special images used as a present source. They must be created by calling grWsiWinCreatePresentableImage(). Presentable images have some implicit properties relative to standard images created with grCreateImage(). By definition, presentable images have a 2D image type, optimal tiling, a depth of 1, 1 mipmap level, and are single sampled. *Fullscreen stereo images* have an implicit array size of 2; all other presentable images have an implicit array size of 1.

Presentable images can have their format overwritten in image or color target views since an equivalent of the GR_IMAGE_CREATE_VIEW_FORMAT_CHANGE flag is implicitly set on presentable image creation.

It is up to the application to ensure that the specified image properties are compatible with the intended present target. Windowed swap chain images must match the width and height of the target window, and the application should recreate the swap chain in response to window re-size (on WM_SIZE message) if necessary. Fullscreen swap chain images must be compatible with a mode returned by grWsiWinGetDisplayModeList() for the specified display.

Unlike standard Mantle resources, the driver automatically allocates a GR_GPU_MEMORY object for presentable images, returned in pMem. There are a number of restrictions on presentable images and their associated memory objects:

- The returned memory object for presentable image is only valid for specifying memory references at command buffer submission (i.e., grQueueSubmit() or grQueueSetGlobalMemReferences()). It may not be mapped, attached to other device objects, etc.
- The grFreeMemory() function must not be called on this memory object; it is implicitly destroyed when the presentable image is destroyed.
- The image is attached to the returned memory object for its lifetime, and it is invalid to bind a new memory object via grBindObjectMemory().

Query Queue Support

Various queues of a Mantle device may not be able to support present operations due to platform or hardware restrictions. The application should check support on a particular queue by calling `grGetObjectInfo()` with the information type set to `GR_WSI_WIN_INFO_TYPE_QUEUE_PROPERTIES`. The extension queue properties are returned in `GR_WSI_WIN_QUEUE_PROPERTIES` structure. Windowed and fullscreen mode presentation capabilities are reported separately. An application must inspect queue properties to determine what queues can be used for presentation. At least one of the queues is guaranteed to support windowed mode presentation. Support for the fullscreen presentation and window presentation on other queues is optional.

> ! Present operations might optionally be available on all queue types. For example, in certain configurations, the DMA queue and timer queue might be able to support fullscreen presentation.

Present

The application calls `grWsiWinQueuePresent()` to display the contents of a presentable image in a window or in fullscreen mode.

In order to use fullscreen presentation functionality, the application must have fullscreen exclusive ownership of the display at the time of presentation. The fullscreen related presentation flag defined by `GR_WSI_WIN_PRESENT_FLAGS` must be zero for windowed applications.

Mantle's queue semaphore support can be used to ensure proper ordering when presenting images rendered by a different queue. If multiple queues support present, it is preferred to present on a queue that was used last to update or prepare the presentable image. If the application needs to coordinate CPU/GPU work after present completion, it should submit a command buffer with a fence after present and use that fence for synchronization.

If destination window during windowed presentation is occluded, `grWsiWinQueuePresent()` returns a `GR_WSI_WIN_PRESENT_OCCLUDED` result code, which should be properly handled by the application.

> ! The DirectX® runtime automatically updates render target views that reference a swap chain resource to point at the next resource in the chain after a present. This is not the case in Mantle and the application should "rotate" presentable images that comprise the swap chain.

Display Rotation for Present

In the windowed mode, the driver automatically rotates images to match any target display's rotation as part of its windowed presentation. Fullscreen applications are responsible for performing this rotation on their own before presenting. That is, if the display information indicates a display is rotated, the application must account for the fact that the end user perceives whatever image they generate at that rotation angle. Presentable images for rotated displays must have extents equal to the "identity rotation" resolution.

> When running in fullscreen mode with rotated displays, the application should implement a proper presentable surface rotation support instead of relying on automatic rotation in borderless windowed mode. The fullscreen presentation of application rotated presentable images is more efficient than windowed mode with automatic rotation handling.

Presentable Image Preparation

As described in Resource States and Preparation, Mantle applications must track the usage state of each image and notify the runtime on every state transition. The image state is extended with two new states for the support of image presentation: `GR_WSI_WIN_IMAGE_STATE_PRESENT_WINDOWED` for using image as a source for windowed presentation and `GR_WSI_WIN_IMAGE_STATE_PRESENT_FULLSCREEN` for using image as a source for fullscreen presentation.

The applications must ensure that any image used as a source by `grWsiWinQueuePresent()` is in the appropriate state by first calling `grCmdPrepareImages()`. The specified state must match the presentation mode of the subsequent present. This preparation gives the runtime an opportunity to reformat the image, flush dirty caches, etc., in order to make the image presentable. These internal requirements may be different depending on whether the image will be presented to the desktop in windowed mode or read directly by the display engine in fullscreen mode. Before an image can be used again for rendering or other operations, it should be transitioned from the presentation source state to a state appropriate for the next operation.

Presentation Queue Limit

There is a limit to how many frames can be queued in the system. By default, that limit is set to 3 frames. If necessary, an application adjusts that limit to the desired value using `grWsiWinSetMaxQueuedFrames()`. Specifying zero value for a queued frame limit resets the limit to the default value.

When the presentation limit is adjusted in a multi-device configuration, all devices that are used for alternate frame rendering should have the same presentation queue limit specified. In a multi-device configurations, the presentation queue limit is not per device, but is rather global for the system.

Querying Presentable Image Properties

When implementing frame capture and debug tools that intercept present operations, it might be necessary to know some information about presentable images. If API interception starts in the middle of an application run, the frame capture or debug tools might not be able to intercept presentable image creation and need alternative means of retrieving information about those objects. The presentable image information can be queried on presentable images by calling `grGetObjectInfo()` with the information type parameter set to `GR_WSI_WIN_INFO_TYPE_PRESENTABLE_IMAGE_PROPERTIES`. The presentable image information is returned in the `GR_WSI_WIN_PRESENTABLE_IMAGE_PROPERTIES` structure.

CHAPTER X.

DMA QUEUE EXTENSION

EXTENSION OVERVIEW

The DMA queue extension provides an efficient method for asynchronous memory and image copy between the CPU and GPU, as well as between peer GPUs in multi-device configurations. It allows maximizing PCIe bandwidth at high power efficiency. Without this extension, the universal or asynchronous compute queues could be used for transfers, which could result in lower over-the-bus transfer bandwidth and higher GPU power requirements.

The DMA queue extension does not expose any new functions and relies on the core API functions for memory and image transfers. This allows creating more portable code that can be easily re-targeted to support different queue types.

EXTENSION DISCOVERY

Support for this extension is queried by calling `grGetExtensionSupport()` with the string "`GR_DMA_QUEUE`". If an extension is available, the physical GPU queue information reports the DMA queue type. A Mantle device that intends to use this extension must include "`GR_DMA_QUEUE`" in the `ppEnabledExtensionNames` list at creation.

DMA Queue Type

The extension defines a new queue type enumeration value: GR_EXT_QUEUE_DMA. The DMA queue is requested on device creation and is later retrieved from the device by calling grGetDeviceQueue(). The DMA queue generally behaves just like a regular queue with respect to command buffer construction, submission, synchronization, and querying its capabilities.

Memory and Image State

Memory and image must be in the GR_MEMORY_STATE_DATA_TRANSFER, GR_MEMORY_STATE_DATA_TRANSFER_SOURCE, GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION, GR_IMAGE_STATE_DATA_TRANSFER, GR_IMAGE_STATE_DATA_TRANSFER_SOURCE, or GR_IMAGE_STATE_DATA_TRANSFER_DESTINATION state, as appropriate, before performing data transfer with DMA queue.

Some limited state transitions can be performed on a DMA queue. For example, one could transition between the GR_MEMORY_STATE_DATA_TRANSFER and GR_MEMORY_STATE_WRITE_TIMESTAMP states, or the GR_IMAGE_STATE_DATA_TRANSFER_SOURCE and GR_IMAGE_STATE_DATA_TRANSFER_DESTINATION states. When interacting with other queues, the preparations should follow the rules outlined in Multi-Queue Considerations.

DMA Command Buffer Building Functions

The command buffers build for the DMA queue support only a subset of functions targeted primarily at memory and image transfers. The following functions are supported:

- grCmdCopyMemory()
- grCmdCopyImage()
- grCmdCopyMemoryToImage()
- grCmdCopyImageToMemory()
- grCmdUpdateMemory()
- grCmdFillMemory()
- grCmdSetEvent()
- grCmdResetEvent()
- grCmdWriteTimestamp()[2]
- grCmdDbgMarkerBegin()
- grCmdDbgMarkerEnd()

Functional Limitations

The DMA extension provides a feature set close to the hardware DMA capabilities, and thus it directly exposes hardware limitations and peculiarities to the application.

The DMA engine is designed for maximizing the PCIe bandwidth and is an efficient mechanism for transferring resources across the bus between the CPU and GPU, as well as between multiple GPUs that are capable of peer-to-peer transfers. The DMA engine should not generally be used for local to local video memory transfers as it might not be able to saturate GPU memory bandwidth.

The functional limitations specific to the DMA queue extension are additional to the rules of the core Mantle API. Attempts to use DMA engine outside of the prescribed limitations results in undefined behavior.

[2] The timestamp functionality is available only if the DMA queue properties indicate timestamp support (reported in GR_PHYSICAL_GPU_QUEUE_PROPERTIES).

General Limitations

- Maximum image size supported: 16384x16384x2048
- For compressed images from the DMA perspective, the texel size is a compression block size

grCmdCopyMemory Limitations

- Source memory alignment: 4 bytes
- Destination memory alignment: 4 bytes

grCmdCopyImage Limitations

For performing image-to-image copies, some requirements are different for different tiling modes of source and destination images.

- Source and destination images have to be created with the same bit depth and dimensions. Additionally, for copies between images with the `GR_OPTIMAL_TILING` tiling mode, both source and destination images must have identical creation parameters and only copies between identical subresources are permitted
- Only a raw data copy is supported, so no conversion between source and destination formats is supported
- For partial subresource copies of the `GR_LINEAR_TILING` source to the `GR_LINEAR_TILING` destination, the `GR_LINEAR_TILING` source to the `GR_OPTIMAL_TILING` destination, and the `GR_OPTIMAL_TILING` source to the `GR_LINEAR_TILING` destination, the image offset and copy rectangle size must be aligned to 4 bytes
- For partial subresource copies between images with the `GR_OPTIMAL_TILING` tiling mode, the image offset and copy rectangle size alignment must be multiple of 8 texels in X and Y directions
- For full subresource copies between subresources of equal dimensions (offset is zero and rectangle covers the whole subresource) there are no alignment restrictions on the subresource dimensions

grCmdCopyMemoryToImage Limitations

- Supported destination tiling: `GR_LINEAR_TILING`, `GR_OPTIMAL_TILING`
- Source memory alignment: 4 bytes
- Destination image offset and copy rectangle size alignment must be aligned to 4 bytes

grCmdCopyMemoryToImage Limitations

- Supported source tiling: `GR_LINEAR_TILING`, `GR_OPTIMAL_TILING`
- Source image offset and copy rectangle size alignment must be aligned to 4 bytes
- Destination memory alignment: 4 bytes

grCmdFillMemory Limitations

- Destination memory alignment: 4 bytes

grCmdUpdateMemory Limitations

- Destination memory alignment: 4 bytes

grCmdWriteTimestamp Limitations

- Supported only on *graphics core next* (GCN) architecture version 1.1 and newer. Support can be queried in `GR_PHYSICAL_GPU_QUEUE_PROPERTIES`
- Destination memory alignment: 32 bytes

Chapter XI.

Timer Queue Extension

Extension Overview

The timer queue extension adds support for timed delay injection into a special queue. Along with queue synchronization, this delay could be used for spacing out workloads, which is useful for implementing power efficient and consistent fame rate limiting and frame pacing in multi-device configurations.

The timer queue extension adds a new queue type and a special timing operation only available on the timer queue.

Extension Discovery

Support for this extension is queried by calling `grGetExtensionSupport()` with the string "`GR_TIMER_QUEUE`". If an extension is available, the physical GPU queue information reports the timer queue type. A Mantle device that intends to use this extension must include "`GR_TIMER_QUEUE`" in the `ppEnabledExtensionNames` list at creation.

Timer Queue Type

The extension defines a new queue type enumeration value: `GR_EXT_QUEUE_TIMER`. The timer queue is requested on device creation and is later retrieved from device by calling `grGetDeviceQueue()`. The timer queue provides a limited support for Mantle queue operations. Only the following operations are compatible with timer queues:

- `grQueueDelay()`
- `grSignalQueueSemaphore()`
- `grWaitQueueSemaphore()`

Since the timer queue does not support command buffer submission, no command buffers can be built for the timer queue.

Timer queue behaves differently from other queue types with respect to queue and device idle operations. Due to this limitation, `grQueueWaitIdle()`, and `grDeviceWaitIdle()` do not wait for completion of timer queue operations.

Timed Delays

Timed delays are added to the timer queue by calling `grQueueDelay()`. The delays, as well as the synchronization operations, are executed in the order in which they were issued to the queue. There could be at most one outstanding delay operation per timer queue at any time. Queuing more delays might result in skipping some of the delay operations.

> ! An application should avoid inserting very long delays, as they might interfere with command buffer scheduling.

Chapter XII.

Advanced Multisampling Extension

Extension Overview

The advanced multisampling extension exposes multisampling hardware capabilities beyond the core Mantle feature set. These features enable increased anti-aliasing quality and performance, giving the application tighter control over the tradeoff between the two. The key extension features include:

- **Enhanced quality anti-aliasing (EQAA)**: This feature allows the application to independently control the sample rate for rasterization, depth-stencil, and color. It further decouples the color buffer's sample rate from its fragment rate – the number of distinct color values stored per pixel. Relative to the core Mantle anti-aliasing support, this allows quality approaching a higher sample rate at the memory cost of a lower sample rate.
- **FMask image views**: This feature lets the application create a shader view of an image's FMask data. The shader can use the FMask view to read directly from compressed MSAA images, greatly reducing the cost of resource preparation at the expense of some extra work in the shader.
- **Custom sample positions**: This feature allows the application to specify custom sample locations patterns per pipeline. These patterns are defined spanning a 2x2 pixel (*pixel quad*) area.

The advanced multisampling extension exposes the features above by enabling the creation of EQAA images, extending the MSAA state object with additional control over sample rates and sample positions, exposing support for FMask image views, and exposing support for a shader IL instruction to read FMask from a shader.

Extension Discovery

Support for the advanced multisampling extension is queried by calling `grGetExtensionSupport()` with the string "`GR_ADVANCED_MSAA`". A Mantle device that intends to use this extension must include "`GR_ADVANCED_MSAA`" in the `ppEnabledExtensionNames` list at creation. The extension functions themselves are exported by the loader AMD extension library (mantleaxl32.dll or mantleaxl64.dll).

EQAA Images

The *enhanced quality anti-aliasing* (EQAA) feature allows a color target image to store coverage information for more *sample* positions than it has *fragments* (per-pixel unique color storage). When creating a color target image with `grCreateImage()`, a separate number of coverage samples and fragments can be specified by setting the value of samples in the `GR_IMAGE_CREATE_INFO` structure to the result of the `GR_EQAA_COLOR_TARGET_SAMPLES` macro. The values produced by `GR_EQAA_COLOR_TARGET_SAMPLES` are only valid when the advanced multisampling extension is enabled, otherwise an incorrect sample count is reported by Mantle.

This macro only creates a valid samples value for images that specify the `GR_IMAGE_USAGE_COLOR_TARGET` usage. Valid values for fragments are 1, 2, 4, and 8. Valid values for coverage samples are 1, 2, 4, 8, and 16. The value of coverage samples must be greater than or equal to the value of fragments.

Setting the number of samples in `GR_IMAGE_CREATE_INFO` to a standard Mantle supported value (1, 2, 4, or 8) results in a color target image with the same number of coverage samples and fragments, which effectively disables EQAA.

EQAA Resolve Behavior

For color target images with more coverage samples than fragments, it is possible for some coverage samples to have an *unknown* sample color. For example, if 3 primitives with different colors touch a pixel in an image with 4 coverage samples and 2 fragments, at least one sample has an unknown color (see Figure 18). When performing resolves with `grCmdResolveImage()`, unknown samples are ignored, and the result is a weighted average of the samples with a known fragment color.

EQAA Image View Behavior

A standard image view of an EQAA image only has access to the number of samples that is equal to the number of color fragments, since the color image only has storage for that many unique color values. In order to take advantage of the additional coverage information available for EQAA images, the application uses the FMask image view feature described in FMask Shader Access.

Advanced Multisampling State

In order to render using EQAA or with custom sample positions, the application must use an advanced multisampling state object created by calling `grCreateAdvancedMsaaState()`. The `GR_ADVANCED_MSAA_STATE_CREATE_INFO` includes the following additional information about the multisampling configuration:

- sample rate options for different parts of the pipeline
- advanced alpha-to-coverage controls
- custom sample pattern

Sample Rates

The advanced multisampling state allows the specify different sample rates for various portions of the graphics pipeline. The following sample rate controls are provided to the application:

- coverage samples
- pixel shader samples
- depth target samples
- color target samples

The number of *coverage samples* specified in the `coverageSamples` member of the `GR_ADVANCED_MSAA_STATE_CREATE_INFO` structure controls the sample rate of the rasterizer. The rasterizer sample rate must be greater than or equal to sample rates in all other parts of the pipeline. The valid values for coverage samples are: 1, 2, 4, 8, and 16.

The number of *pixel shader samples* specified in the `pixelShaderSamples` member of the `GR_ADVANCED_MSAA_STATE_CREATE_INFO` structure controls the pixel shader execution rate for pixel shaders, which use inputs that are evaluated per sample (i.e., an `SV_SampleIndex` input in DirectX® *high-level shader language* (HLSL) or an input using the sample interpolation modifier). The default Mantle implementation behaves as if the number of pixel shader samples is set to the value of samples. Adjusting this parameter in the advanced multisampling state allows a pipeline to get some benefit from supersampling without running at full sample rate. If the number of pixel shader samples is less than the color samples, then outputs are replicated as necessary to fill the output color samples. For example, if the number of coverage and color target samples is 8, and the number of pixel shader samples is 2, the pixel shader is executed twice. Four of the color samples are populated with a color from one pixel shader invocation, and another four color samples are populated with a color from another pixel shader invocation. The valid values for pixel shader samples are: 1, 2, 4, and 8.

The number of *depth target samples* specified in the `depthTargetSamples` member of the `GR_ADVANCED_MSAA_STATE_CREATE_INFO` structure controls the number of samples in the bound depth target. The value is ignored if no depth target is bound. The depth target sample rate must be less than or equal to the number of coverage samples. If the number of depth target samples is less than the number of color target samples, then some color samples do not have a corresponding depth value. Such *unanchored samples* attempt to approximate a depth value based on nearby samples, but may have incorrect depth test results. The valid values for depth target samples are: 1, 2, 4, and 8.

The number of *color target samples* specified in the `colorTargetSamples` member of the `GR_ADVANCED_MSAA_STATE_CREATE_INFO` structure controls the maximal number of coverage samples stored in any color target's FMask. The number of color target samples must be less than or equal to the number of coverage samples. The valid values for color target samples are: 1, 2, 4, 8, and 16.

Sample Clustering for Over-rasterization

An application can specify the number of *sample clusters* in a pixel for controlling *over-rasterization*. If any sample in a cluster is covered, then all samples in a cluster are marked as covered as well. For example, specifying a single sample cluster makes all samples appear covered if any of them are covered. The number of sample clusters is specified in the `sampleClusters` member of the `GR_ADVANCED_MSAA_STATE_CREATE_INFO` structure. The number of sample clusters must be less than or equal to the number of coverage samples. The valid values for sample clusters are: 1, 2, 4, and 8.

Alpha-to-coverage Controls

Advanced multisampling state objects also allow the application to take greater control of the hardware's alpha-to-coverage capabilities by specifying the number of alpha-to-coverage samples and controlling its dither behavior.

The number of *alpha-to-coverage samples* specified in the `alphaToCoverageSamples` member of the `GR_ADVANCED_MSAA_STATE_CREATE_INFO` structure controls how many samples of quality are generated when alpha-to-coverage is enabled. If the alpha-to-coverage sample count is less than depth target or color target sample count, the additional sample coverage values are extrapolated.

The number of alpha-to-coverage samples must be less than or equal to the number of coverage samples. The valid values for alpha-to-coverage samples are: 1, 2, 4, and 8.

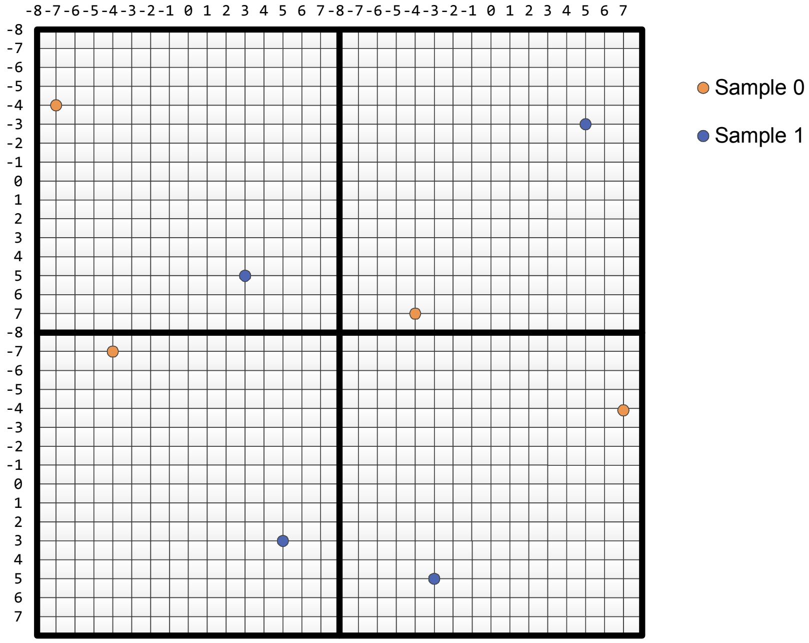
By default, the alpha-to-coverage implementation in Mantle dithers the generated coverage over a 2x2 pixel quad in order to more closely approximate the specified alpha coverage. Setting `disableAlphaToCoverageDither` in the `GR_ADVANCED_MSAA_STATE_CREATE_INFO` structure to `GR_TRUE` disables that dithering.

Custom Sample Patterns

In addition to detailed control over the sample rate of various portions of the graphics pipeline, the advanced multisampling state allows specifying custom sample patterns. A custom sample pattern defines each sample's position within a pixel. The pattern covers a 2x2 pixel area that is repeated over the entire viewport.

In order to enable a custom sample pattern, `customSamplePatternEnable` is set to `GR_TRUE` in `GR_ADVANCED_MSAA_STATE_CREATE_INFO`. The desired pattern should be specified in `customSamplePattern`. A separate pattern is specified for each of the 4 pixels in the quad, with each sample described by a coordinate in the [-8, 7] range. Figure 17 gives an example of 2-sample pattern spanning a 2 by 2 pixel area.

Figure 17: Example of custom sample pattern



This sample pattern is setup as follows:

Listing 23. Sample pattern setup

```
// Top-left quad pixel
pSamples->topLeft[0].x = -7;
pSamples->topLeft[0].y = -4;
pSamples->topLeft[1].x =  3;
pSamples->topLeft[1].y =  5;
// Top-right quad pixel
pSamples->topRight[0].x = -4;
pSamples->topRight[0].y =  7;
pSamples->topRight[1].x =  5;
pSamples->topRight[1].y = -3;
// Bottom-left quad pixel
pSamples->bottomLeft[0].x = -4;
pSamples->bottomLeft[0].y = -7;
pSamples->bottomLeft[1].x =  5;
pSamples->bottomLeft[1].y =  3;
// Bottom-right quad pixel
pSamples->bottomRight[0].x =  7;
pSamples->bottomRight[0].y = -4;
pSamples->bottomRight[1].x = -3;
pSamples->bottomRight[1].y =  5;
```

When running with EQAA scenarios, where different parts of the graphics pipeline are running at different sample rates, the best quality is achieved if the samples are ordered, such that each sample ***n*** is closer to sample ***n-floor(log2(n))*** than any earlier sample. For example:

- Sample 2 should be closer to sample 0 than sample 1
- Sample 3 should be closer to sample 1 than sample 0
- Sample 4 should be closer to sample 0 than samples 1 through 3
- Sample 5 should be closer to sample 1 than samples 0, 2, 3, or 4
- …
- Sample 15 should be closer to sample 7 than any other sample

Ordering the samples this way ensures that good results are achieved when imposing lower sample rates than the rasterizer uses. Regardless of the rasterizer sample rate, samples 0 and 1 should form a good 2-sample pattern, samples 0 through 3 should form a good 4-sample pattern, etc.

For example, if the rasterizer runs at the rate of 16 samples, and the depth test runs at 2 samples, the stored depth values are only stored for samples 0 and 1; it is important that the chosen samples represent the overall pixel as well as possible. In such a case, depth testing needs to be performed on samples where an exact depth value was not stored, and the ordering described above allows the hardware to quickly pick the closest neighbor sample from which to generate an approximate depth value.

If the application wants all pixels to have the same sample pattern, it should specify the same pattern in `topLeft`, `topRight`, `bottomLeft`, and `bottomRight` in `GR_MSAA_QUAD_SAMPLE_PATTERN`.

Multisampled Image FMask

The term *FMask* refers to a multisampled image meta-data surface which is used to track MSAA surface compression data in the GCN architecture. The MSAA surface only stores unique colors (*fragments*), while FMask contains coverage information (per sample pointers to unique fragments). This compression scheme functions on a per-pixel level.

The advanced multisampling extension allows an application to create an FMask image view of an image to improve the performance and quality of shader access of MSAA images. Performance can be improved since preparing an image for FMask-based shader access is significantly cheaper than preparing an image for direct shader access. Quality can be improved since the application can now get complete coverage information for color targets with more samples than fragments.

An application typically uses FMask in the shader to read the FMask coverage data, extract the fragment index for a sample of interest, and subsequently read the fragment's color data.

FMask Image Views

The FMask image data are not treated as a standalone subresource, and may only be accessed through FMask image views. It can be thought of as a companion to the image; any action to a subresource (clear, preparation, etc.) in that image also implicitly updates that subresource's FMask meta-data.

The application creates an FMask image view by calling `grCreateFmaskImageView()`. The specified parent image must have more than 1 sample and support the `GR_IMAGE_USAGE_COLOR_TARGET` and `GR_IMAGE_USAGE_SHADER_ACCESS_READ` usages.

Once created, FMask image views can be attached to a descriptor set by calling `grAttachImageViewDescriptors()`, just like a regular image view. FMask image views can only be read in a shader with the `LOAD_FPTR` IL instruction, described in FMask Shader Access.

> The FMask view is bound completely independently from the regular image view, which in the presence of FMask access contains color fragment data.

FMask Preparation

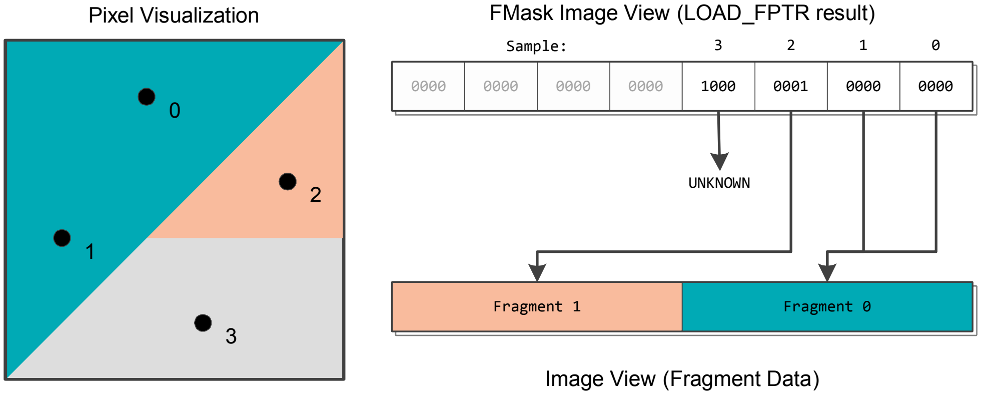
The image state is extended with two new states for supporting FMask read access from graphics and compute shaders: GR_EXT_IMAGE_STATE_GRAPHICS_SHADER_FMASK_LOOKUP and GR_EXT_IMAGE_STATE_COMPUTE_SHADER_FMASK_LOOKUP. These states are used for access of both FMask view and the image fragments.

FMask Shader Access

Shader access to FMask and image fragment data is performed through appropriate image views. Both of these views have to be attached to descriptor sets with the GR_EXT_IMAGE_STATE_GRAPHICS_SHADER_FMASK_LOOKUP or GR_EXT_IMAGE_STATE_COMPUTE_SHADER_FMASK_LOOKUP state, depending on the pipeline type.

FMask image views can only be accessed in the shader via a special AMD IL instruction: LOAD_FPTR. This instruction returns the fragment indices for each sample packed into a 32-bit (up to 8 samples) or 64-bit (for 16 samples) unsigned integer mask. Each fragment index is 4 bits wide, with samples 0-7 returned in the red channel and samples 8-15 returned in the green channel. They are packed starting with the low bits, such that sample 0 is in bits [3:0] of the red channel, sample 1 is in bits [7:4] of the red channel, sample 8 is in bits [3:0] of the green channel, and sample 15 is in bits [31:28] of the green channel. The fragment index references a fragment color in the image view for the sample.

Figure 18. FMask data example



If the FMask is loaded for a cleared descriptor set slot, the LOAD_FPTR instruction returns *identity* fragment mapping value, which is 0x876543210 for up to 8 samples and 0xfedcba9876543210 for 16 samples.

If the fragment pointer is greater than or equal to the number of fragments, the sample's color is *unknown*. This can occur in EQAA images with more samples than fragments, and the application should take care to deal with the unknown value in some manner, possibly ignoring that sample (e.g., when filtering multiple samples), and instead choosing a nearby sample's fragment.

This following snippet of shader IL code uses the LOAD_FPTR instruction to compute the average color of two samples in a 2 sample, 2 fragment image.

Listing 24. Example of FMask access in shader IL

```
; DCL texture coordinate interpolant.
dcl_input_usage(generic) v0
; DCL base MSAA image (contains fragment data).
dclpt_type(2dmsaa)_coordmode(normalized)_stage(0)
; DCL FMask.
dcl_resource_id(1)_type(2d)_fmtx(uint)_fmty(uint)_fmtz(uint)_fmtw(uint)
; Convert float interpolant to integer pixel position.
ftou r0, v0.xy00
; Load FMask data for this pixel.
load_fptr_resource(1) r3.x___, r0
dcl_literal l0, 0x00000000, 0x0000000F, 0x00000004, 0x00000008
; Extract sample 0's fragment index from the load_fptr result.
and  r4, r3, l0.y
; Extract sample 1's fragment index from the load_fptr result.
ishr r5, r3, l0.z
and  r5, r5, l0.y
; Fetch sample 0's color using fragment index.
mov  r0.__zw, r4.00xx ; sample 0 index
texldms_stage(0)_mag(point)_min(point)_xoffset(0)_yoffset(0) r1, r0
; Fetch sample 1's color using fragment index.
mov  r0.__zw, r5.00xx ; sample 1 index
texldms_stage(0)_mag(point)_min(point)_xoffset(0)_yoffset(0) r2, r0
; Store average sample color in r0.
add_d2 r0, r1, r2
```

In many multisample scenarios, it is common for most pixels in a frame to have all samples be the same color. A shader can capitalize on this by creating fast shader paths for a LOAD_FPTR result of 0 (all fragment indices refer to sample 0).

CHAPTER XIII.

BORDER COLOR PALETTE EXTENSION

EXTENSION OVERVIEW

Samplers that specify a clamp-to-border addressing mode cause texture fetch operations to return a constant color if the relevant texture coordinate is clamped. The core Mantle API only supports a small, fixed set of border colors: white, transparent black, and opaque black. The border color palette extension allows an application to specify arbitrary border colors by managing and referencing a *border color palette*.

A border color palette consists of one or more 4-component, 32-bit floating point RGBA tuples which are set to arbitrary color values by the application. Once its contents are fully initialized, the palette is bound and referenced by samplers.

At sampler creation, the extension allows the application to specify a border color index. If such a sampler clamps a texture fetch to the border, the fetch returns the color from the specified entry in the currently bound border color palette.

EXTENSION DISCOVERY

Support for the border color palette extension is queried by calling `grGetExtensionSupport()` with the string "`GR_BORDER_COLOR_PALETTE`". A Mantle device that intends to use this extension must include "GR_BORDER_COLOR_PALETTE" in the ppEnabledExtensionNames list at creation. The extension functions themselves are exported by the AMD extension library (mantleaxl32.dll or mantleaxl64.dll).

Querying Queue Support

The border color palette capabilities of a Mantle implementation may vary per command queue, and must be queried using grGetObjectInfo() on each desired queue with an information type of GR_EXT_INFO_TYPE_QUEUE_BORDER_COLOR_PALETTE_PROPERTIES. The border color palette queue properties are returned in the GR_BORDER_COLOR_PALETTE_PROPERTIES structure.

A reported maximum palette size of zero indicates that the queue does not support border color palettes at all. Queues that support border color palettes are guaranteed support at least 256 entries. The absolute maximum palette size for any queue is 4096 entries.

Palette Management

A border color palette is created by calling grCreateBorderColorPalette() with a desired palette size. The palette size cannot be larger than the maximum reported palette size for the queue that is used with the created palette. Multiple border color palettes can be created by the application.

As with all Mantle objects, the application must query the palette for its GPU memory requirements, and, if necessary, bind an appropriate memory object to it. The contents of the palette are undefined when a new memory object is bound. Colors for each entry in the palette can be specified by calling grUpdateBorderColorPalette(). This function takes an offset to the first entry, a count of entries to update, and a pointer to the new color data. The color entries are specified as four consecutive floats per entry in R, G, B, A order. Before updating palette colors, the application should ensure the palette is not currently used for rendering operations. The update fails if a valid memory object is not bound or if the update goes past the end of the palette.

Palette Binding

A palette can be bound to the command buffer state by calling grCmdBindBorderColorPalette() during command buffer building. Separate palettes can be bound for each pipeline type – compute and graphics – as specified by the pipelineBindPoint parameter. It is valid to bind the same palette for multiple pipeline bind points.

Once bound, the palette acts as the source for any sampler for that pipeline type in that command buffer that clamps-to-border and specifies a palette border color index.

Like all command buffer functions, illegal grCmdBindBorderColorPalette() calls produce undefined results. The application must ensure the palette is not larger than the maximum supported border color palette size reported for the relevant queue. Results are also undefined if samplers reference palette indices beyond the extents of the bound palette.

Sampler Border Color Palette Support

The core Mantle API allows samplers to specify a clamp-to-border addressing mode and choose one of three possible border colors for clamped fetches: white, transparent black, and opaque black. With the border color palette, an arbitrary color can be specified in a sampler by referencing one of the border color palette slots.

An index into a border color palette is specified during sampler creation, by setting the borderColor member of the GR_SAMPLER_CREATE_INFO structure to a value returned by the GR_EXT_BORDER_COLOR_TYPE_PALETTE_ENTRY macro. The the GR_EXT_BORDER_COLOR_TYPE_PALETTE_ENTRY macro encodes a palette index into a valid enumeration value.

If a texture fetch using such a sampler is clamped to the border color due to the addressing modes, the fetch returns the color at the specified index in the currently bound palette. It is up to the application to ensure a valid palette is bound, and that each entry is set to the desired color.

The border color supports only a limited set of texture channel swizzles. The following table describes a set of channel swizzles for different formats compatible with border color.

Table 20. Format swizzles compatible with border color

Channels in the format	R channel swizzle	G channel swizzle	B channel swizzle	A channel swizzle
1	R	0 or 1	0 or 1	0 or 1
	0 or 1	0 or 1	0 or 1	R
	R	R	R	0 or 1
	R	R	R	R
2	R	G	0 or 1	0 or 1
	G	R	0 or 1	0 or 1
	R	R	R	G
	G	G	G	R

Channels in the format	R channel swizzle	G channel swizzle	B channel swizzle	A channel swizzle
	R	0 or 1	0 or 1	G
3	R	G	B	0 or 1
	B	G	R	0 or 1
4	R	G	B	A
	B	G	R	A
	R	G	B	0 or 1
	B	G	R	0 or 1
	G	B	A	R
	G	B	A	0 or 1
	A	B	G	R
	A	B	G	0 or 1

Using any other channel swizzle with border color produces undefined results.

CHAPTER XIV.

OCCLUSION QUERY DATA COPY EXTENSION

EXTENSION OVERVIEW

The occlusion query data copy extension provides an efficient method for accessing occlusion query data using the GPU without involving the CPU. The occlusion query data can be directly copied to a memory location where it can be accessible by a shader, used for control flow, and so on. Without this extension, the occlusion query result has to be queried on the CPU by calling the `grGetQueryPoolResults()` function and then uploaded into GPU memory.

EXTENSION DISCOVERY

Support for the occlusion query data extension is queried by calling `grGetExtensionSupport()` with the string "`GR_COPY_OCCLUSION_DATA`". A Mantle device that intends to use this extension must include "`GR_COPY_OCCLUSION_DATA`" in the `ppEnabledExtensionNames` list at creation. The extension functions themselves are exported by the AMD extension library (mantleaxl32.dll or mantleaxl64.dll).

This extension functionality is only supported by universal command buffers.

Copying Occlusion Results

An application copies the occlusion query result to GPU memory by calling the grCmdCopyOcclusionData() function. The result is a 64-bit unsigned integer for each occlusion query slot. The copy can only be performed on query pools created for occlusion queries. The application must make sure all queries are properly terminated before their results are copied to memory. The copy operation allows adding the occlusion query data to the current values stored in memory by setting the accumulateData parameter to GR_TRUE. The accumulation of occlusion results is performed separately for each memory location corresponding to the query slot. The destination memory for occlusion data copy has to be 4-byte aligned and in the GR_EXT_MEMORY_STATE_COPY_OCCLUSION_DATA state.

Chapter XV.

GPU Timestamp Calibration Extension

Extension Overview

The GPU timestamp calibration extension provides a reasonably accurate mechanism for synchronizing current GPU timestamps with the CPU clock. This allows applications and tools to synchronize CPU and GPU execution timelines and even synchronize timestamps between multiple GPUs by matching timestamps from different GPUs to a common CPU clock.

Additionally, this extension provides a mechanism to retrieve the current GPU timestamp value outside of a command buffer.

Extension Discovery

Support for the GPU timestamps calibration extension is queried by calling `grGetExtensionSupport()` with the string "`GR_GPU_TIMESTAMP_CALIBRATION`". A Mantle device that intends to use this extension must include "`GR_GPU_TIMESTAMP_CALIBRATION`" in the `ppEnabledExtensionNames` list at creation. The extension function is exported by the AMD extension library (mantleaxl32.dll or mantleaxl64.dll).

Calibrating GPU Timestamps

An application calibrates the GPU timestamps by correlating them with a high-precision CPU timer, represented by the *performance counter* in the Windows® OS. The calibration is performed by calling `grCalibrateGpuTimestamp()`. The result is a current 64-bit GPU timestamp and a 64-bit CPU time corresponding to the time the GPU timestamp was generated. The reported GPU timestamp is compatible with the timestamp value written to memory by `grCmdWriteTimestamp()`, while the CPU time is compatible with the results of `QueryPerformanceCounter()` in the Windows® OS.

Tha application is expected to re-calibrate GPU timestamps once a frame or at some other small regular interval, since with time, the CPU and GPU clocks could diverge. On the other hand, the recalibration should not be performed too frequently, as it might increase CPU overhead.

Chapter XVI.

Command Buffer Control Flow Extension

Extension Overview

The command buffer control flow extension adds occlusion and memory-based predication, as well as control flow constructs to the Mantle command buffers. The control flow is evaluated at command buffer execution time and requires no extra CPU intervention for operation. Along with synchronization primitives and on-the-fly GPU resource manipulation, this extension allows applications to implement powerful control logic for conditional rendering and compute execution. For example, the control flow driven by occlusion queries allows applications to implement sophisticated occlusion control to complement predicated rendering functionality.

Extension Discovery

Support for the GPU timestamps calibration extension is queried by calling `grGetExtensionSupport()` with the string "`GR_CONTROL_FLOW`". A Mantle device that intends to use this extension must include "`GR_CONTROL_FLOW`" in the `ppEnabledExtensionNames` list at creation. The extension functions are exported by the AMD extension library (mantleaxl32.dll or mantleaxl64.dll).

CONTROL FLOW OPERATION

The extension adds two separate features that could be used in a complementary fashion to enable conditional execution of GPU commands: a predication of certain GPU operations based on occlusion query data and memory value, and control flow structures for selective execution of blocks of operations recorded in command buffers. In addition to conditional execution, the control flow constructs also include loops.

The control flow support is optional and might be available only on a subset of the queues. An application queries control flow support for each of the queues to determine its availability. The control flow statements can be nested up to the depth reported by the queue. The nested loops and conditionals count against the common nesting limit. It is allowed for different queue types to expose a different maximum level of nesting on the same GPU.

There is very important restriction related to the interaction of command buffer control flow with the command buffer state. In Mantle the command buffer state (e.g., bound pipeline, descriptor sets, applicable state blocks) has to be fully qualified and deterministic for all draw and dispatch calls inside of control flow statements and immediately following them. If any command buffer state is changed inside of the control flow, that state becomes undefined when exiting the control flow scope and must be explicitly rebound before the subsequent draw or dispatch call. For example, binding a new pipeline inside of the control flow scope forces the application to bind a pipeline before the first draw following the control flow statement. A requirement to bind the changed state after control flow guarantees the state to be fully qualified at all times. Failure to define the complete and deterministic state at the draw time results in undefined results.

An application must be very careful about the control flow interaction with the memory and image preparation operations – all branches should ensure the correct state is provided through appropriate preparations. In general, it is advised to avoid putting preparation calls inside of the control flow statements, unless absolutely necessary. The most notable exception would be the preparation of the memory locations used for the control flow – they need to be prepared accordingly if data for control flow are changed inside of the control flow blocks.

There might be some additional CPU and GPU overhead from the command buffer control operation, so it should not be overused; however, it is expected that control flow can enable new usage scenarios and efficient rendering algorithms that could offset this extra cost.

QUERYING SUPPORT

Predication and control flow is not guaranteed to be available on all queues. The feature support can be queried on a per queue basis using the `grGetObjectInfo()` function with the `GR_EXT_INFO_TYPE_QUEUE_CONTROL_FLOW_PROPERTIES` information type parameter. The control flow queue properties are returned in the `GR_QUEUE_CONTROL_FLOW_PROPERTIES` structure.

If control flow is not supported, the reported maximum control flow nesting level is zero and none of the supported control flow operation flags are set. The reported maximal nesting limit is applicable to both the conditional statements and loops.

Predication

Execution of draws, dispatches, and resource copy operations can be predicated using occlusion query results or contents of memory. Both of these methods of predication are fully independent and can be used at the same time.

The following command buffer operations are predicated:

- `grCmdDraw()`
- `grCmdDrawIndexed()`
- `grCmdDrawIndirect()`
- `grCmdDrawIndexedIndirect()`
- `grCmdDispatch()`
- `grCmdDispatchIndirect()`
- `grCmdCopyMemory()`
- `grCmdCopyImage()`
- `grCmdCopyMemoryToImage()`
- `grCmdCopyImageToMemory()`
- `grCmdUpdateMemory()`
- `grCmdSetEvent()`
- `grCmdResetEvent()`
- `grCmdMemoryAtomic()`

All other command buffer operations are unaffected by predication.

Occlusion-Based Predication

The supported command buffer operations can be predicated based on occlusion query results. The occlusion predication is set by calling the `grCmdSetOcclusionPredication()` method within a command buffer. The occlusion predication state is set if the occlusion results match the provided condition (visible or invisible). The `waitResults` argument could be used to specify if the queue should stall and wait for the results of occlusion query to become available. The predication result of multiple occlusion queries can be accumulated by specifying the `GR_TRUE` value in `accumulateData`. Specifying the `GR_FALSE` value sets predication according to a newly provided occlusion query.

To explicitly reset the predication state, an application calls `grCmdResetOcclusionPredication()` within a command buffer. There is no persistence of predicated state between the command buffers, and at the end of command buffer execution, the predication is implicitly cleared.

Memory Value Based Predication

The supported command buffer operations can be predicated based on a value stored in GPU memory. The memory predication is set by calling the `grCmdSetMemoryPredication()` method within a command buffer.

The 64-bit value stored at the memory location specified by `mem` and `offset` is used to set predication. Further execution of applicable command buffer operations is skipped when the 64-bit memory value is equal to zero at the time of setting the memory predication during the command buffer execution. The commands are executed when the value in memory is equal to one. Any other values produce undefined results. The provided memory location has to be 4-byte aligned and the memory range has to be in the `GR_EXT_MEMORY_STATE_CMD_CONTROL` state. There is only one memory predication active at a time, and subsequent calls to `grCmdSetMemoryPredication()` change the predication value.

To reset predication state and disable memory predication, an application calls `grCmdResetMemoryPredication()` within a command buffer. There is no persistence of the predicated state between the command buffers, and at the end of command buffer execution, the predication is implicitly cleared.

Conditional Command Buffer Execution

A section of a recorded command buffer is conditionally executed based on the results of an execution time comparison of values in memory objects against literal values specified in conditional statements.

The conditional execution block is opened with the `grCmdIf()` function and closed with the `grCmdEndIf()` function during command buffer recording. An alternative block of actions to be executed is specified by using the `grCmdElse()` function between `grCmdIf()` and `grCmdEndIf()`. Use of the `grCmdElse()` function is optional.

Commands between the `grCmdIf()` and `grCmdEndIf()` or `grCmdElse()` calls are executed only if the condition specified in `grCmdIf()` evaluates to true at execution time. Similarly, the commands between the grCmdElse() and grCmdEndIf() calls are executed only if the condition is false. The condition is based on comparison of the 64-bit value coming from memory, masked with the mask value using the bit-wise AND operation, and the literal data value, according to the specified comparison function. The memory location for the condition must be 4-byte aligned. The mask is applied to both the memory data and a literal value. The comparison functions GR_COMPARE_NEVER and GR_COMPARE_ALWAYS are not available for conditional statement evaluation.

When using conditional statements, an application must ensure the memory range is in the GR_EXT_MEMORY_STATE_CMD_CONTROL memory state using an appropriate preparation command.

An application could nest conditional statements up to the limit specified in the control flow queue properties. Failing to properly terminate all conditional statements results in unsuccessful command buffer building. An attempt to execute the unsuccessfully built command buffer results in undefined behavior.

Loops in Command Buffers

Similarly to conditional statements, an application can add looping constructs to the command buffers. All commands between the grCmdWhile() and grCmdEndWhile() functions are executed while a condition evaluates to true.

A conditional statement for the loop is specified similarly to how it is set in the grCmdIf() function. The comparison functions GR_COMPARE_NEVER and GR_COMPARE_ALWAYS are not available for loop condition evaluation.

When using loops in command buffers, an application must ensure the memory range is in the GR_EXT_MEMORY_STATE_CMD_CONTROL memory state using an appropriate preparation command.

The loop statements can be nested inside of the conditional statements and other loops; however, an application must ensure that loops are properly terminated and do not cross conditional statement boundaries. Failing to properly terminate all loops results in unsuccessful command buffer building. An attempt to execute the unsuccessfully built command buffer results in undefined behavior.

Special care must be taken to ensure a command buffer loop can terminate in a finite amount of time and that it does not depend on execution of the other queues. Looping in a command buffer to wait on a result from a second command buffer might cause a deadlock if the second command buffer has not been scheduled for execution at the time of performing a loop. Unless queue semaphores are used to guarantee the order of command buffer execution, command buffers on other queues might not be scheduled during execution of the command buffer containing a loop.

Memory State for Control Flow

Memory range used for memory-based predication, conditional execution, or command buffer loops has to be in the GR_EXT_MEMORY_STATE_CMD_CONTROL state, as described above. While memory is in this state, it is equivalent to GR_MEMORY_STATE_DATA_TRANSFER for the purpose of the CPU access. This means an application can change a memory value used for control flow with a CPU while it is in the GR_EXT_MEMORY_STATE_CMD_CONTROL state. However, if the memory location is changed with the GPU, a transition to and from the GR_MEMORY_STATE_DATA_TRANSFER memory state must be performed.

Chapter XVII.

Resource State Access Extension

Extension Overview

Memory and image states in Mantle cover a wide range of use scenarios, often with the same state covering multiple GPU engines or CPU accessing memory and image resources. While this reduced set of states is easy and convenient for applications to use, more optimal behavior can be derived from additional specification of resource access clients (i.e., CPU or GPU engines). This extension allows applications to provide additional access information in addition to state for the most optimal Mantle operation.

Extension Discovery

Support for the GPU timestamps calibration extension is queried by calling `grGetExtensionSupport()` with the string "`GR_RESOURCE_STATE_ACCESS`". A Mantle device that intends to use this extension must include "`GR_RESOURCE_STATE_ACCESS`" in the `ppEnabledExtensionNames` list at creation. There are no new API entry points defined in this extension.

Extended Memory and Image States

Memory and image states specified by GR_MEMORY_STATE and GR_IMAGE_STATE are extended with client access flags defined in GR_EXT_ACCESS_CLIENT. A convenient macro GR_EXT_ACCESS_STATE is provided to combine memory or image state with client access flags. Multiple client access flags can be specified together by using *bitwise OR* operation. For example, allowing access from DMA or compute queues can be specified by combining GR_EXT_ACCESS_DMA_QUEUE and GR_EXT_ACCESS_COMPUTE_QUEUE flags. These extended memory and image states can be used for state transitions, as well as for image and memory view binding. State specified at resource bind time should match memory or image state precisely at the time of use, including the access client flags.

When performing image cloning operations, both source and destination extended image state can be used, as long as they match expected image usage and client access flags for source and destination images specify universal queue only.

While resource state access extension is enabled, regular memory and image states can still be used. They are equivalent to specifying a GR_EXT_ACCESS_DEFAULT value and might be less efficient than specifying a minimal set of required access flags. When memory or image state is explicitly reset or image has a new memory binding, the default client access is equivalent to GR_EXT_ACCESS_DEFAULT.

A minimal set of required access client flags should be used to guarantee the optimal performance of resource preparations, as well as their access by the GPU.

A general set of permitted combinations of client access flags and resource states are described in Table 21 and Table 22. Specifying disallowed combinations of client access flags and states results in undefined behavior.

Table 21. Permitted extended memory states

Memory state	Universal queue	Compute queue	DMA queue	Timer queue	CPU access
GR_MEMORY_STATE_DATA_TRANSFER	X	X	X		X
GR_MEMORY_STATE_GRAPHICS_SHADER_READ_ONLY	X				
GR_MEMORY_STATE_GRAPHICS_SHADER_WRITE_ONLY	X				
GR_MEMORY_STATE_GRAPHICS_SHADER_READ_WRITE	X				
GR_MEMORY_STATE_COMPUTE_SHADER_READ_ONLY	X	X			
GR_MEMORY_STATE_COMPUTE_SHADER_WRITE_ONLY	X	X			
GR_MEMORY_STATE_COMPUTE_SHADER_READ_WRITE	X	X			
GR_MEMORY_STATE_MULTI_USE_READ_ONLY	X	X			
GR_MEMORY_STATE_INDEX_DATA	X				
GR_MEMORY_STATE_INDIRECT_ARG	X	X			
GR_MEMORY_STATE_WRITE_TIMESTAMP	X	X	X		
GR_MEMORY_STATE_QUEUE_ATOMIC	X	X			
GR_MEMORY_STATE_DISCARD					
GR_MEMORY_STATE_DATA_TRANSFER_SOURCE	X	X	X		
GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION	X	X	X		
GR_EXT_MEMORY_STATE_COPY_OCCLUSION_DATA	X				
GR_EXT_MEMORY_STATE_CMD_CONTROL	X	X	X		

Table 22. Permitted extended image states

Image state	Universal queue	Compute queue	DMA queue	Timer queue	CPU access
GR_IMAGE_STATE_DATA_TRANSFER	X	X	X		X
GR_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY	X				
GR_IMAGE_STATE_GRAPHICS_SHADER_WRITE_ONLY	X				
GR_IMAGE_STATE_GRAPHICS_SHADER_READ_WRITE	X				
GR_IMAGE_STATE_COMPUTE_SHADER_READ_ONLY	X	X			
GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY	X	X			
GR_IMAGE_STATE_COMPUTE_SHADER_READ_WRITE	X	X			
GR_IMAGE_STATE_MULTI_SHADER_READ_ONLY	X	X			
GR_IMAGE_STATE_TARGET_AND_SHADER_READ_ONLY	X				
GR_IMAGE_STATE_UNINITIALIZED					
GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL	X				
GR_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL	X				
GR_IMAGE_STATE_CLEAR	X	X			
GR_IMAGE_STATE_RESOLVE_SOURCE	X				
GR_IMAGE_STATE_RESOLVE_DESTINATION	X				
GR_IMAGE_STATE_DISCARD					
GR_IMAGE_STATE_DATA_TRANSFER_SOURCE	X	X	X		
GR_IMAGE_STATE_DATA_TRANSFER_DESTINATION	X	X	X		
GR_EXT_IMAGE_STATE_GRAPHICS_SHADER_FMASK_LOOKUP	X				
GR_EXT_IMAGE_STATE_COMPUTE_SHADER_FMASK_LOOKUP	X	X			
GR_WSI_WIN_IMAGE_STATE_PRESENT_WINDOWED	X	X	X	X	
GR_WSI_WIN_IMAGE_STATE_PRESENT_FULLSCREEN	X	X	X	X	

For presentation states `GR_WSI_WIN_IMAGE_STATE_PRESENT_WINDOWED` and `GR_WSI_WIN_IMAGE_STATE_PRESENT_FULLSCREEN`, allowed client access flags must match queue presentation capabilities. For example, if in a windowed mode an image can be presented only on universal queue, only the `GR_EXT_ACCESS_UNIVERSAL_QUEUE` access flag can be specified along with the GR_WSI_WIN_IMAGE_STATE_PRESENT_WINDOWED state. See Query Queue Support for more details on queue presentation capabilities. In case a timer queue is used by the application and it supports presentation operations, the GR_EXT_ACCESS_TIMER_QUEUE access flag can be used. The GR_EXT_ACCESS_TIMER_QUEUE access flag cannot be used for any other purpose.

To ensure image data are preserved on memory rebinding as described in Invariant Image Data, an image cannot be in the extended image state at the time of rebinding. In other words, on rebinding the same memory, the data are guaranteed to be preserved in the GR_IMAGE_STATE_DATA_TRANSFER state only when the client access mask is set to GR_EXT_ACCESS_DEFAULT.

Chapter XVIII.

Mantle API Reference

Functions

Initialization and Device Functions

grInitAndEnumerateGpus

Initializes the Mantle runtime and enumerates the handles of all Mantle-capable physical GPUs present in the system. Each GPU is reported separately for multi-GPU boards. This function is also used to re-enumerate GPUs after receiving a GR_ERROR_DEVICE_LOST error code.

```
GR_RESULT grInitAndEnumerateGpus(
    const GR_APPLICATION_INFO* pAppInfo,
    const GR_ALLOC_CALLBACKS* pAllocCb,
    GR_UINT* pGpuCount,
    GR_PHYSICAL_GPU gpus[GR_MAX_PHYSICAL_GPUS]);
```

Parameters

pAppInfo

[in] Application information provided to the Mantle drivers. See GR_APPLICATION_INFO.

pAllocCb

[in] Optional system memory alloc/free function callbacks. Can be NULL. See GR_ALLOC_CALLBACKS.

pGpuCount

[out] Count of available Mantle GPUs.

gpus

[out] Handles of all available Mantle GPUs.

Returns

If successful, `grInitAndEnumerateGpus()` returns `GR_SUCCESS`, the number of available Mantle GPUs is written to the location specified by `pGpuCount`, and the list of GPU handles is written to `gpus`. The number of reported GPUs can be zero. Otherwise, it returns one of the following errors:

- `GR_ERROR_INITIALIZATION_FAILED` if the loader cannot load any Mantle ICDs
- `GR_ERROR_INCOMPATIBLE_DRIVER` if the loader loaded a Mantle ICD, but it is incompatible with the Mantle version supported by the loader
- `GR_ERROR_INVALID_POINTER` if `pGpuCount` is `NULL`
- `GR_ERROR_INVALID_POINTER` if application information is specified, but `pAppName` is `NULL`
- `GR_ERROR_INVALID_POINTER` if `pAllocCb` is not `NULL`, but one or more of the callback function pointers are `NULL`
- `GR_ERROR_INVALID_POINTER` if this function is called more than once and the function callback pointers are different from previous function invocations

Notes

An application can call this function multiple times if necessary. The first `grInitAndEnumerateGpus()` call loads and initializes the drivers; subsequent calls force the driver re-initialization. Before calling this function a second time, all devices and other Mantle objects must be destroyed by the application.

Thread safety

Not thread safe.

grGetGpuInfo

Retrieves specific information about a Mantle GPU. This function is called before device creation in order to select a suitable GPU.

```
GR_RESULT grGetGpuInfo(
    GR_PHYSICAL_GPU gpu,
    GR_ENUM infoType,
    GR_SIZE* pDataSize,
    GR_VOID* pData);
```

Parameters

gpu

Physical GPU device handle.

infoType

Type of information to retrieve. See GR_INFO_TYPE.

pDataSize

[in/out] Input value specifies the size in bytes of the pData output buffer; output value reports the number of bytes written to pData.

pData

[out] Device information structure. Can be NULL.

Returns

If successful, grGetGpuInfo() returns GR_SUCCESS and the queried info is written to the location specified by pData. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the gpu handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the gpu handle references an invalid object type
- GR_ERROR_INVALID_VALUE if infoType is not one of the supported values
- GR_ERROR_INVALID_POINTER if pDataSize is NULL
- GR_ERROR_INVALID_MEMORY_SIZE if pData is not NULL and the pDataSize input value is smaller than the size of the appropriate return data structure

Notes

If pData is NULL, the input pDataSize value does not matter and the function returns the expected data structure size in pDataSize.

GPU Properties

The GPU properties are retrieved with the GR_INFO_TYPE_PHYSICAL_GPU_PROPERTIES information type. Returned is the GR_PHYSICAL_GPU_PROPERTIES structure.

GPU Performance

The GPU performance properties are retrieved with the GR_INFO_TYPE_PHYSICAL_GPU_PERFORMANCE information type. Returned is the GR_PHYSICAL_GPU_PERFORMANCE structure.

Queue Properties

Queue properties are retrieved on physical GPUs with the GR_INFO_TYPE_PHYSICAL_GPU_QUEUE_PROPERTIES information type. Returned is a list of GR_PHYSICAL_GPU_QUEUE_PROPERTIES structures, one per queue type.

Thread safety

Not thread safe.

grCreateDevice

Creates a Mantle device object.

```
GR_RESULT grCreateDevice(
    GR_PHYSICAL_GPU gpu,
    const GR_DEVICE_CREATE_INFO* pCreateInfo,
    GR_DEVICE* pDevice);
```

Parameters

gpu

Physical GPU device handle.

pCreateInfo

[in] Device creation parameters. See `GR_DEVICE_CREATE_INFO`.

pDevice

[out] Device handle.

Returns

If successful, `grCreateDevice()` returns `GR_SUCCESS` and the created Mantle device handle is written to the location specified by `pDevice`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `gpu` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `gpu` handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pDevice` are `NULL`
- `GR_ERROR_INVALID_POINTER` if a non-zero number of extensions is specified and `pCreateInfo.ppEnabledExtensionNames` is `NULL`, or any extension name pointer is `NULL`
- `GR_ERROR_INVALID_VALUE` if the requested number of queues for each type is invalid
- `GR_ERROR_INVALID_VALUE` if the validation level is invalid; if no validation level is enabled, only `GR_VALIDATION_LEVEL_0` can be specified
- `GR_ERROR_INVALID_EXTENSION` if a requested extension is not supported
- `GR_ERROR_INVALID_FLAGS` if the creation flags are invalid
- `GR_ERROR_INITIALIZATION_FAILED` if the driver could not initialize the device object for internal reasons
- `GR_ERROR_DEVICE_ALREADY_CREATED` if a device instance is already active for the given physical GPU

Notes

`pCreateInfo.ppEnabledExtensionNames` pointer can be `NULL` if `pCreateInfo.extensionCount` is zero.

See `GR_VALIDATION_LEVEL` for a list of available validation levels.

At least one universal queue must be specified.

Thread safety

Not thread safe.

grDestroyDevice

Destroys a valid Mantle device.

```
GR_RESULT grDestroyDevice(
    GR_DEVICE device);
```

Parameters

device

Device handle.

Returns

`grDestroyDevice()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type

Notes

None.

Thread safety

Not thread safe.

Extension Discovery Functions

grGetExtensionSupport

Checks extension support by name.

```
GR_RESULT grGetExtensionSupport(
    GR_PHYSICAL_GPU gpu,
    const GR_CHAR* pExtName);
```

Parameters

gpu

Physical GPU device handle.

pExtName

[in] Extension name for which to check support.

Returns

`grGetExtensionSupport()` returns `GR_SUCCESS` if the function executed successfully and the specified extension is supported. If the function executed successfully, but the specified extension is not available, `GR_UNSUPPORTED` is returned. Otherwise, it returns one of the following errors:

- ▼ `GR_ERROR_INVALID_HANDLE` if the `gpu` handle is invalid
- ▼ `GR_ERROR_INVALID_OBJECT_TYPE` if the `gpu` handle references invalid object type
- ▼ `GR_ERROR_INVALID_POINTER` if `pExtName` is `NULL`

Notes

None.

Thread safety

Not thread safe.

Queue Functions

grGetDeviceQueue

Returns a queue handle for the specified queue type and ordinal.

```
GR_RESULT grGetDeviceQueue(
    GR_DEVICE device,
    GR_ENUM queueType,
    GR_UINT queueId,
    GR_QUEUE* pQueue);
```

Parameters

device

Device handle.

queueType

Queue type. See GR_QUEUE_TYPE.

queueId

Queue ordinal for the given queue type.

pQueue

[out] Queue handle.

Returns

If successful, grGetDeviceQueue() returns GR_SUCCESS and the queried queue handle is written to the location specified by pQueue. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_ORDINAL if the queue ordinal exceeds the number of queues requested at device creation
- GR_ERROR_INVALID_POINTER if pQueue is NULL
- GR_ERROR_INVALID_QUEUE_TYPE if queueType is invalid

Notes

None.

Thread safety

Not thread safe for calls referencing the same device object.

grQueueWaitIdle

Waits for a specific queue to complete execution of all submitted command buffers before returning to the application.

```
GR_RESULT grQueueWaitIdle(
    GR_QUEUE queue);
```

Parameters

queue

Queue handle.

Returns

`grQueueWaitIdle()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- ▼ `GR_ERROR_INVALID_HANDLE` if the `queue` handle is invalid
- ▼ `GR_ERROR_INVALID_OBJECT_TYPE` if the `queue` handle references an invalid object type

Notes

None.

Thread safety

Not thread safe for calls referencing the device object associated with the queue or any other objects associated with that device.

grDeviceWaitIdle

Waits for all queues associated with a device to complete execution of all submitted command buffers before returning to the application.

```
GR_RESULT grDeviceWaitIdle(
    GR_DEVICE device);
```

Parameters

device

Device handle.

Returns

`grDeviceWaitIdle()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- ▼ `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid
- ▼ `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type

Notes

None.

Thread safety

Not thread safe for calls referencing the same device object or any other objects associated with that device.

grQueueSubmit

Submits a command buffer to a queue for execution.

```
GR_RESULT grQueueSubmit(
    GR_QUEUE queue,
    GR_UINT cmdBufferCount,
    const GR_CMD_BUFFER* pCmdBuffers,
    GR_UINT memRefCount,
    const GR_MEMORY_REF* pMemRefs,
    GR_FENCE fence);
```

Parameters

queue

Queue handle.

cmdBufferCount

Number of command buffers to be submitted.

pCmdBuffers

[in] List of command buffer handles.

memRefCount

Number of memory object references for this command buffer (i.e., size of the `pMemRefs` array). Can be zero.

pMemRefs

[in] Array of memory reference descriptors. Can be `NULL` if `memRefCount` is zero. See `GR_MEMORY_REF`.

fence

Handle of the fence object to be associated with this submission (optional, can be `GR_NULL_HANDLE`).

Returns

grQueueSubmit() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the queue handle is invalid
- GR_ERROR_INVALID_HANDLE if any memory or command buffer object handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the queue handle references an invalid object type
- GR_ERROR_TOO_MANY_MEMORY_REFERENCES if the total number of memory references (queue global and per-command buffer) is too large
- GR_ERROR_INVALID_POINTER if pMemRefs is NULL when the memory reference count is greater than zero
- GR_ERROR_INVALID_POINTER if pCmdBuffers is NULL
- GR_ERROR_INVALID_VALUE if cmdBufferCount is zero
- GR_ERROR_INVALID_FLAGS if flags in pMemRefs[].flags are invalid
- GR_ERROR_INCOMPLETE_COMMAND_BUFFER if any of the submitted command buffers is not properly constructed
- GR_ERROR_INCOMPATIBLE_QUEUE if a command buffer is submitted with the wrong queue type

Notes

When a valid fence object is provided, the driver submits the fence after the last command buffer from the list executes.

Thread safety

Not thread safe for calls referencing the same queue object.

grQueueSetGlobalMemReferences

Sets a list of per-queue memory object references that persists across command buffer submissions. A snapshot of the current global queue memory reference list is taken at command buffer submission time. After submission, the global queue memory reference list can be changed for subsequent submissions without affecting previously queued submissions.

```
GR_RESULT grQueueSetGlobalMemReferences(
    GR_QUEUE queue,
    GR_UINT memRefCount,
    const GR_MEMORY_REF* pMemRefs);
```

Parameters

queue

Queue handle.

memRefCount

Number of global memory object references for this queue (i.e., size of the pMemRefs array). Can be zero.

pMemRefs

[in] Array of memory reference descriptors. See grQueueSubmit(). Can be NULL if memRefCount is zero. See GR_MEMORY_REF.

Returns

grQueueSetGlobalMemReferences() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the queue handle is invalid or if any memory object handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the queue handle references an invalid object type
- GR_ERROR_TOO_MANY_MEMORY_REFERENCES if the number of global memory references exceeds the limit supported by the device
- GR_ERROR_INVALID_POINTER if pMemRefs is NULL
- GR_ERROR_INVALID_FLAGS if flags in pMemRefs[].flags are invalid.

Notes

None.

Thread safety

Not thread safe for calls referencing the same queue object.

MEMORY MANAGEMENT FUNCTIONS

grGetMemoryHeapCount

Returns the number of GPU memory heaps for a Mantle device.

```
GR_RESULT grGetMemoryHeapCount(
    GR_DEVICE device,
    GR_UINT* pCount);
```

Parameters

device

Device handle.

pCount

[out] Number of GPU memory heaps.

Returns

If successful, `grGetMemoryHeapCount()` returns `GR_SUCCESS` and the number of GPU memory heaps is written to the location specified by `pCount`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pCount` is `NULL`

Notes

The number of heaps returned is guaranteed to be at least one.

Thread safety

Not thread safe for calls referencing the same device object.

grGetMemoryHeapInfo

Retrieves specific information about a GPU memory heap.

```
GR_RESULT grGetMemoryHeapInfo(
    GR_DEVICE device,
    GR_UINT heapId,
    GR_ENUM infoType,
    GR_SIZE* pDataSize,
    GR_VOID* pData);
```

Parameters

device

Device handle.

heapId

GPU memory heap ordinal up to the number of heaps reported by `grGetMemoryHeapCount()`.

infoType

Type of information to retrieve. See `GR_INFO_TYPE`.

pDataSize

[in/out] Input value specifies the size in bytes of the `pData` output buffer; output value reports the number of bytes written to `pData`.

pData

[out] Memory heap information structure.

Returns

If successful, `grGetMemoryHeapInfo()` returns `GR_SUCCESS` and the queried info is written to the location specified by pData. `Otherwise`, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if `infoType` is invalid
- `GR_ERROR_INVALID_ORDINAL` if the GPU memory heap ordinal is invalid
- `GR_ERROR_INVALID_POINTER` if `pDataSize` is `NULL`
- `GR_ERROR_INVALID_MEMORY_SIZE` if `pData` is not `NULL` and `pDataSize` input value is smaller than the size of the appropriate return data structure

Notes

If `pData` is `NULL`, the input `pDataSize` value does not matter and the function returns the expected data structure size in `pDataSize`.

For heaps not visible by the CPU (the `GR_MEMORY_HEAP_FLAG_CPU_VISIBLE` flag is not set), the CPU read and write performance ratings are zero.

Thread safety

Not thread safe for calls referencing the same device object.

grAllocMemory

Allocates GPU memory by creating a memory object.

```
GR_RESULT grAllocMemory(
    GR_DEVICE device,
    const GR_MEMORY_ALLOC_INFO* pAllocInfo,
    GR_GPU_MEMORY* pMem);
```

Parameters

device

Device handle.

pAllocInfo

[in] Creation data for the memory object. See `GR_MEMORY_ALLOC_INFO`.

pMem

[out] Memory object handle.

Returns

If successful, `grAllocMemory()` returns `GR_SUCCESS` and the handle of the created GPU memory object is written to the location specified by `pMem`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pAllocInfo` or `pMem` are `NULL`
- `GR_ERROR_INVALID_MEMORY_SIZE` if the allocation size is invalid
- `GR_ERROR_INVALID_ALIGNMENT` if the allocation alignment is invalid
- `GR_ERROR_INVALID_VALUE` if the priority value is invalid, or if `pAllocInfo.heapCount` is zero for real allocations or greater than zero for virtual allocations
- `GR_ERROR_INVALID_ORDINAL` if an invalid valid heap ordinal is specified or if a heap ordinal is used more than once
- `GR_ERROR_INVALID_FLAGS` if the flags are invalid or incompatible
- `GR_ERROR_OUT_OF_GPU_MEMORY` if memory object creation failed due to a lack of video memory
- `GR_ERROR_UNAVAILABLE` if attempting to create virtual allocation and memory virtual remapping functionality is unavailable

Notes

A particular heap ID may not appear in the heap list more than once. Real allocations must have at least one heap specified.

Virtual allocations must have zero heaps specified. Priority has no effect for virtual allocations.

Memory size is specified in bytes and must be a multiple of the heap's page size. If an allocation can be placed in multiple memory heaps, the largest page size should be used.

The optional memory allocation alignment is specified in bytes. When zero, the alignment is equal to the specified page size, otherwise it must be a multiple of the page size.

Thread safety

Thread safe.

grFreeMemory

Frees GPU memory and destroys the memory object. For pinned memory objects, the underlying system memory is unpinned.

```
GR_RESULT grFreeMemory(
    GR_GPU_MEMORY mem);
```

Parameters

mem

Memory object handle.

Returns

`grFreeMemory()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `mem` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `mem` handle references an invalid object type
- `GR_ERROR_UNAVAILABLE` if the `mem` handle references an internal memory object that cannot be freed

Notes

None.

Thread safety

Thread safe.

grSetMemoryPriority

Sets a new priority for the specified memory object.

```
GR_RESULT grSetMemoryPriority(
    GR_GPU_MEMORY mem,
    GR_ENUM priority);
```

Parameters

mem

Memory object handle.

priority

New priority for the memory object. See `GR_MEMORY_PRIORITY`.

Returns

`grSetMemoryPriority()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `mem` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `mem` handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if `priority` is invalid
- `GR_ERROR_UNAVAILABLE` if the memory object is not a real allocation

Notes

Not available for pinned and virtual memory objects.

Thread safety

Not thread safe for calls referencing the same memory object.

grMapMemory

Get a CPU pointer to the data contained in a memory object.

```
GR_RESULT grMapMemory(
    GR_GPU_MEMORY mem,
    GR_FLAGS flags,
    GR_VOID** ppData);
```

Parameters

mem

Memory object handle.

flags

Map flags, reserved.

ppData

[out] CPU pointer to the memory object data.

Returns

If successful, `grMapMemory()` returns `GR_SUCCESS` and a pointer to the CPU-accessible memory object data is written to the location specified by `ppData`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `mem` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `mem` handle references an invalid object type
- `GR_ERROR_INVALID_FLAGS` if the `flags` are invalid
- `GR_ERROR_INVALID_POINTER` if `ppData` is `NULL`
- `GR_ERROR_MEMORY_MAP_FAILED` if the memory object is busy and cannot be mapped by the OS
- `GR_ERROR_NOT_MAPPABLE` if the memory object cannot be mapped due to some of its heaps not having the CPU visible flag set

- GR_ERROR_UNAVAILABLE if the memory object is not a real allocation

Notes

Memory objects cannot be mapped multiple times concurrently. Mapping is not available for pinned and virtual memory objects.

Thread safety

Not thread safe for calls referencing the same memory object.

grUnmapMemory

Remove CPU access from a previously mapped memory object.

```
GR_RESULT grUnmapMemory(
    GR_GPU_MEMORY mem);
```

Parameters

mem

Memory object handle.

Returns

grUnmapMemory() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the mem handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the mem handle references an invalid object type
- GR_ERROR_UNAVAILABLE if the memory object is not a real allocation
- GR_ERROR_MEMORY_UNMAP_FAILED if the memory object cannot be unlocked by the OS

Notes

Not available for pinned and virtual memory objects.

Thread safety

Not thread safe for calls referencing the same memory object.

grRemapVirtualMemoryPages

Update memory mappings for a virtual allocation. The remapping is performed on a page boundary.

```
GR_RESULT grRemapVirtualMemoryPages(
    GR_DEVICE device,
    GR_UINT rangeCount,
    const GR_VIRTUAL_MEMORY_REMAP_RANGE* pRanges,
    GR_UINT preWaitSemaphoreCount,
    const GR_QUEUE_SEMAPHORE* pPreWaitSemaphores,
    GR_UINT postSignalSemaphoreCount,
    const GR_QUEUE_SEMAPHORE* pPostSignalSemaphores);
```

Parameters

device

Device handle.

rangeCount

Number of ranges to remap.

pRanges

[in] Array of memory range descriptors. See `GR_VIRTUAL_MEMORY_REMAP_RANGE`.

preWaitSemaphoreCount

Number of semaphores in `pPreWaitSemaphores`.

pPreWaitSemaphores

[in] Array of queue semaphores to wait on before performing memory remapping. Can be `NULL` if `preWaitSemaphoreCount` is zero.

postSignalSemaphoreCount

Number of semaphores in `pPostSignalSemaphores`.

pPostSignalSemaphores

[in] Array of queue semaphores to signal after performing memory remapping. Can be `NULL` if `postSignalSemaphoreCount` is zero.

Returns

`grRemapVirtualMemoryPages()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid
- `GR_ERROR_INVALID_HANDLE` if the `pRanges.virtualMem` handles are invalid or reference non-virtual allocations
- `GR_ERROR_INVALID_HANDLE` if the `pRanges.realMem` handles are invalid or reference non-real allocations

- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the page range for any of the memory objects is invalid
- `GR_ERROR_INVALID_POINTER` if `pRanges` is a `NULL` pointer
- `GR_ERROR_INVALID_POINTER` if `pPreWaitSemaphores` is a `NULL` pointer, but `preWaitSemaphoreCount` is greater than zero
- `GR_ERROR_INVALID_POINTER` if `pPostSignalSemaphores` is a `NULL` pointer, but `postSignalSemaphoreCount` is greater than zero
- `GR_ERROR_UNAVAILABLE` if memory virtual remapping functionality is unavailable

Notes

It is valid to specify a `GR_NULL_HANDLE` object handle in `pRanges.realMem` – it unmaps pages in the specified range. Remapping is not available for pinned and real memory objects.

Thread safety

Not thread safe.

grPinSystemMemory

Pins a system memory region and creates a Mantle memory object representing it. Pinned memory objects are freed via `grFreeMemory()` like regular memory objects.

```
GR_RESULT grPinSystemMemory(
    GR_DEVICE device,
    const GR_VOID* pSysMem,
    GR_SIZE memSize,
    GR_GPU_MEMORY* pMem);
```

Parameters

device

Device handle.

pSysMem

[in] Pointer to the system memory region to pin. Must be aligned to a page boundary of the heap marked with the `GR_MEMORY_HEAP_FLAG_HOLDS_PINNED` flag.

memSize

Size of the system memory region to pin. Must be aligned to a page boundary of the heap marked with the `GR_MEMORY_HEAP_FLAG_HOLDS_PINNED` flag.

pMem

[out] Memory object handle.

Returns

If successful, grPinSystemMemory() returns GR_SUCCESS and a GPU memory object handle representing the pinned memory is written to the location specified by pMem. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_OUT_OF_MEMORY if memory object creation failed due to an inability to pin memory
- GR_ERROR_INVALID_MEMORY_SIZE if memSize is not page size aligned
- GR_ERROR_INVALID_POINTER if pMem is NULL, or pSysMem is NULL, or pSysMem is not page size aligned
- GR_ERROR_UNAVAILABLE if memory pinning functionality is unavailable

Notes

None.

Thread safety

Thread safe.

Generic API Object Management Functions

grDestroyObject

Destroys an API object. Not applicable to devices, queues, or memory objects.

```
GR_RESULT grDestroyObject(
    GR_OBJECT object);
```

Parameters

object

API object handle.

Returns

`grDestroyObject()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `object` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `object` handle references an invalid object type

Notes

None.

Thread safety

Thread safe.

grGetObjectInfo

Retrieves info such as memory requirements for a given API object. Not applicable to physical GPU objects.

```
GR_RESULT grGetObjectInfo(
    GR_BASE_OBJECT object,
    GR_ENUM infoType,
    GR_SIZE* pDataSize,
    GR_VOID* pData);
```

Parameters

object

API object handle.

infoType

Type of object information to retrieve; valid values vary by object type. See `GR_INFO_TYPE`.

pDataSize

[in/out] Input value specifies the size in bytes of the pData output buffer; output value reports the number of bytes written to pData.

pData

[out] Object info structure. Can be NULL.

Returns

If successful, grGetObjectInfo() returns GR_SUCCESS and the queried API object information is written to the location specified by pData. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the object handle is invalid
- GR_ERROR_INVALID_VALUE if infoType is invalid for the given object type or debug information is requested without enabling the validation layer
- GR_ERROR_INVALID_MEMORY_SIZE if pData is not NULL and pDataSize input value is smaller than the size of the appropriate return data structure
- GR_ERROR_INVALID_POINTER if pDataSize is NULL.
- GR_ERROR_UNAVAILABLE if running with the validation layer enabled, attempting to retrieve an object tag and the tag information is not attached to an object

Notes

If pData is NULL, the input pDataSize value does not matter and the function returns the expected data structure size in pDataSize.

Memory requirements

Memory requirements are retrieved with the GR_INFO_TYPE_MEMORY_REQUIREMENTS information type. The returned data are in the GR_MEMORY_REQUIREMENTS structure. Available for all objects types except physical GPU, device, queue, shader, and memory, which do not support memory binding.

heaps[] in GR_MEMORY_REQUIREMENTS stores the heap ordinals (same as those used with grGetMemoryHeapInfo()).

Not all objects have memory requirements, in which case it is valid for the requirements structure to return zero size and alignment, and no heaps. For objects with valid memory requirements, at least one valid heap is returned.

Thread safety

Thread safe.

grBindObjectMemory

Binds memory to an API object according to previously queried memory requirements. Not applicable to devices, queues, shaders, or memory objects. Specifying a GR_NULL_HANDLE memory object unbinds the currently bound memory from an object.

```
GR_RESULT grBindObjectMemory(
    GR_OBJECT object,
    GR_GPU_MEMORY mem,
    GR_GPU_SIZE offset);
```

Parameters

object

API object handle.

mem

Memory object handle to use for memory binding. Can be GR_NULL_HANDLE.

offset

Byte offset into the memory object.

Returns

grBindObjectMemory() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the object handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the referenced object does not supports memory binding – a device, queue, shader or memory object
- GR_ERROR_UNAVAILABLE if binding non-image type object to virtual allocation
- GR_ERROR_INVALID_ALIGNMENT if the offset does not match the alignment requirements for the object
- GR_ERROR_INVALID_MEMORY_SIZE if the API object memory size at the given offset is not completely within the memory object range
- GR_ERROR_UNAVAILABLE if the object does not have any memory requirements

Notes

Binding memory to objects other than images automatically initializes the object memory as necessary. Image objects used as color or depth-stencil targets have to be explicitly initialized in command buffers using a grCmdPrepareImages() command to transition them from GR_IMAGE_STATE_UNINITIALIZED to an appropriate image state.

Device, queue, shader, and memory objects do not support memory binding.

Binding memory to an object automatically unbinds any previously bound memory. There is no need to bind `GR_NULL_HANDLE` memory to an object to explicitly unbind previously bound memory before binding a new memory.

This call is invalid on objects that have no memory requirements, even if binding `GR_NULL_HANDLE` memory.

Virtual memory objects can only be used for binding image objects.

Thread safety

Not thread safe for calls referencing the same API object.

Image and Sampler Functions

grGetFormatInfo

Retrieves resource format information and capabilities.

```
GR_RESULT grGetFormatInfo(
    GR_DEVICE device,
    GR_FORMAT format,
    GR_ENUM infoType,
    GR_SIZE* pDataSize,
    GR_VOID* pData);
```

Parameters

device

Device handle.

format

Resource format. See GR_FORMAT.

infoType

Type of format information to retrieve. See GR_INFO_TYPE.

pDataSize

[in/out] Input value specifies the size in bytes of the pData output buffer; output value reports the number of bytes written to pData.

pData

[out] Format information structure. Can be NULL.

Returns

If successful, grGetFormatInfo() returns GR_SUCCESS and the queried format info is written to the location specified by pData. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_VALUE if infoType is invalid
- GR_ERROR_INVALID_FORMAT if format is invalid
- GR_ERROR_INVALID_POINTER if pDataSize is NULL
- GR_ERROR_INVALID_MEMORY_SIZE if pData is not NULL and pDataSize input value is smaller than the size of the appropriate return data structure

Notes

If `pData` is `NULL`, the input `pDataSize` value does not matter and the function returns the expected data structure size in `pDataSize`.

Currently only `GR_INFO_TYPE_FORMAT_PROPERTIES` information type is valid. The returned information is in `GR_FORMAT_PROPERTIES`.

It is allowed for some channel and numeric format combinations to expose no capabilities. The format is not illegal from the API perspective, but it cannot really be used for anything.

Thread safety

Thread safe.

grCreateImage

Creates a 1D, 2D or 3D image object.

```
GR_RESULT grCreateImage(
    GR_DEVICE device,
    const GR_IMAGE_CREATE_INFO* pCreateInfo,
    GR_IMAGE* pImage);
```

Parameters

device

Device handle.

pCreateInfo

[in] Image creation info. See `GR_IMAGE_CREATE_INFO`.

pImage

[out] Image object handle.

Returns

If successful, `grCreateImage()` returns `GR_SUCCESS` and the created image object handle is written to the location specified by `pImage`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the image type or tiling type is invalid
- `GR_ERROR_INVALID_VALUE` if the image dimensions are invalid for a given image type
- `GR_ERROR_INVALID_VALUE` if for compressed formats, the image dimensions aren’t multiples of the compression block size
- `GR_ERROR_INVALID_VALUE` if the number of samples is invalid for the given image type and format

- GR_ERROR_INVALID_VALUE if MSAA is enabled (has samples > 1) for images that don't have a color target or depth-stencil usage flag set
- GR_ERROR_INVALID_VALUE if MSAA images have more than 1 mipmap level
- GR_ERROR_INVALID_VALUE if the array size is zero or greater than supported for 1D or 2D images, or the arrays size isn't equal to 1 for 3D images
- GR_ERROR_INVALID_VALUE if the size of the mipmap chain is invalid for the given image type and dimensions
- GR_ERROR_INVALID_POINTER if pCreateInfo or pImage is NULL
- GR_ERROR_INVALID_FORMAT if the format doesn't match usage flags
- GR_ERROR_INVALID_FORMAT if a compressed format is used with a 1D image type
- GR_ERROR_INVALID_FLAGS if invalid image creation flags or image usage flags are specified
- GR_ERROR_INVALID_FLAGS if color target and depth-stencil flags are set together
- GR_ERROR_INVALID_FLAGS if the color target flag is set for 1D images
- GR_ERROR_INVALID_FLAGS if the depth-stencil flag is set for non-2D images

Notes

The GR_IMAGE_USAGE_COLOR_TARGET and GR_IMAGE_USAGE_DEPTH_STENCIL flags are mutually exclusive. Depth-stencil images must be 2D, color target images must be 2D or 3D.

The number of mipmap level is specified explicitly and should always be greater than or equal to 1.

The number of samples greater than 1 is only available for 2D images and only for formats that support multisampling.

The array size must be 1 for 3D images.

Images with more than 1 sample (MSAA images) must have only 1 mipmap level.

1D images ignore height and depth parameters, 2D images ignore depth parameter.

For compressed images, the dimensions are specified in texels and the top-most mipmap level dimensions must be a multiple of the compression block size.

Thread safety

Thread safe.

grGetImageSubresourceInfo

Retrieves information about an image subresource.

```
GR_RESULT grGetImageSubresourceInfo(
    GR_IMAGE image,
    const GR_IMAGE_SUBRESOURCE* pSubresource,
    GR_ENUM infoType,
    GR_SIZE* pDataSize,
    GR_VOID* pData);
```

Parameters

image

Image handle.

pSubresource

Pointer to subresource ID to retrieve information about. See GR_IMAGE_SUBRESOURCE.

infoType

Type of information to retrieve. See GR_INFO_TYPE.

pDataSize

[in/out] Input value specifies the size in bytes of the pData output buffer; output value reports the number of bytes written to pData.

pData

[out] Subresource information structure. Can be NULL.

Returns

If successful, grGetImageSubresourceInfo() returns GR_SUCCESS and the queried subresource information is written to the location specified by pData. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the image handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the image handle references an invalid object type
- GR_ERROR_INVALID_VALUE if infoType is invalid
- GR_ERROR_INVALID_ORDINAL if the subresource ID ordinal is invalid
- GR_ERROR_INVALID_POINTER if pDataSize is NULL
- GR_ERROR_INVALID_MEMORY_SIZE if pData isn't NULL and pDataSize input value is smaller than the size of the appropriate return data structure

Notes

If pData is NULL, the input pDataSize value does not matter and the function returns the expected data structure size in pDataSize.

The internal subresource memory layout is returned by querying subresource properties with the GR_INFO_TYPE_SUBRESOURCE_LAYOUT information type. The returned data are in the GR_SUBRESOURCE_LAYOUT structure. The offset returned in the layout structure is relative to the beginning of the memory range associated with the image object.

Depending on the internal memory organization of the image, some image subresources may alias to the same offset.

For opaque images, the returned pitch values are zero.

Thread safety

Thread safe.

grCreateSampler

Creates a sampler object.

```
GR_RESULT grCreateSampler(
    GR_DEVICE device,
    const GR_SAMPLER_CREATE_INFO* pCreateInfo,
    GR_SAMPLER* pSampler);
```

Parameters

device

Device handle.

pCreateInfo

[in] Sampler creation info. See GR_SAMPLER_CREATE_INFO.

pSampler

[out] Sampler object handle.

Returns

If successful, grCreateSampler() returns GR_SUCCESS and the handle of the created sampler object is written to the location specified by pSampler. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_VALUE if any filtering, addressing modes or comparison function are invalid
- GR_ERROR_INVALID_VALUE if the border color value is invalid

- GR_ERROR_INVALID_VALUE if the anisotropy or level-of-detail (LOD) bias value is outside of the allowed range
- GR_ERROR_INVALID_VALUE if the min/max LOD values are outside of the allowed range or the max LOD is smaller than min LOD
- GR_ERROR_INVALID_POINTER if pCreateInfo or pSampler is NULL

Notes

Min/max LOD and mipmap LOD bias values are specified in floating point with the value of 0.0 corresponding to the largest mipmap level, 1.0 corresponding to the next mipmap level, and so on. The valid range for min/max LOD is [0..16] and for mipmap LOD bias the valid range is [-16..16].

The valid range for max anisotropy values is [1..16].

More texture filter modes can be added to reflect hardware capabilities.

Thread safety

Thread safe.

Image View Functions

grCreateImageView

Creates an image representation that can be bound to the graphics or compute pipeline for shader read or write access.

```
GR_RESULT grCreateImageView(
    GR_DEVICE device,
    const GR_IMAGE_VIEW_CREATE_INFO* pCreateInfo,
    GR_IMAGE_VIEW* pView);
```

Parameters

device

Device handle.

pCreateInfo

[in] View creation info. See `GR_IMAGE_VIEW_CREATE_INFO`.

pView

[out] Image view handle.

Returns

If successful, `grCreateImageView()` returns `GR_SUCCESS` and the handle of the created image view object is written to the location specified by `pView`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid or the image handle in `pCreateInfo` is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the view type or image aspect is invalid
- `GR_ERROR_INVALID_VALUE` if the channel swizzle value is invalid or the swizzle refers to a channel not present in the given format
- `GR_ERROR_INVALID_VALUE` if the color image aspect is specified for a depth/stencil image
- `GR_ERROR_INVALID_VALUE` if the depth image aspect is specified for an image that doesn't have depth
- `GR_ERROR_INVALID_VALUE` if the stencil image aspect is specified for an image that doesn't have stencil
- `GR_ERROR_INVALID_VALUE` if the number of array slices is zero or a range of slices starting from the base is greater than what is available in the image object
- `GR_ERROR_INVALID_VALUE` if the base mipmap level is invalid for the given image object

- GR_ERROR_INVALID_POINTER if pCreateInfo or pView is NULL
- GR_ERROR_INVALID_FORMAT if the view format is incompatible with the image format (has a different bit-depth)
- GR_ERROR_INVALID_FORMAT if the view format is incompatible with the image usage flags or image aspect
- GR_ERROR_INVALID_IMAGE if the image object doesn't have the shader read or write flags set
- GR_ERROR_INVALID_IMAGE if the view type is incompatible with the given image object type
- GR_ERROR_INVALID_IMAGE if a cubemap view is created for an image object that doesn't have square 2D slices
- GR_ERROR_INVALID_IMAGE if a cubemap view is created for an MSAA image

Notes

The view references a subset of mipmap levels and image array slices or the whole image.

The range of mipmap levels is truncated to the available dimensions of the image object.

Thread safety

Thread safe.

grCreateColorTargetView

Creates an image representation that can be bound to the graphics pipeline state for color render target writes.

```
GR_RESULT grCreateColorTargetView(
    GR_DEVICE device,
    const GR_COLOR_TARGET_VIEW_CREATE_INFO* pCreateInfo,
    GR_COLOR_TARGET_VIEW* pView);
```

Parameters

device

Device handle.

pCreateInfo

[in] Color target view creation info. See GR_COLOR_TARGET_VIEW_CREATE_INFO.

pView

[out] Color render target view handle.

Returns

If successful, `grCreateColorTargetView()` returns `GR_SUCCESS` and the handle of the created color target view object is written to the location specified by `pView`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid or the image handle in pCreateInfo is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the base slice is invalid for the given image object and view type
- `GR_ERROR_INVALID_VALUE` if the number of array slices is zero or the range of slices is greater than what is available in the image object
- `GR_ERROR_INVALID_VALUE` if the mipmap level is invalid for the given image object
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pView` is `NULL`
- `GR_ERROR_INVALID_IMAGE` if the image object doesn't have the color target access flag set

Notes

None.

Thread safety

Thread safe.

grCreateDepthStencilView

Creates an image representation that can be bound to the graphics pipeline state as a depth-stencil target.

```
GR_RESULT grCreateDepthStencilView(
    GR_DEVICE device,
    const GR_DEPTH_STENCIL_VIEW_CREATE_INFO* pCreateInfo,
    GR_DEPTH_STENCIL_VIEW* pView);
```

Parameters

device

Device handle.

pCreateInfo

[in] Depth-stencil target view creation info. See `GR_DEPTH_STENCIL_VIEW_CREATE_INFO`.

pView

[out] Depth-stencil target view handle.

Returns

If successful, `grCreateDepthStencilView()` returns `GR_SUCCESS` and the handle of the created depth-stencil target view object is written to the location specified by pView. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid or the image handle in pCreateInfo is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the base slice is invalid for the given image object and view type
- `GR_ERROR_INVALID_VALUE` if the number of array slices is zero or the range of slices is greater than what is available in the image object
- `GR_ERROR_INVALID_VALUE` if the mipmap level is invalid for the given image object
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pView` is `NULL`
- `GR_ERROR_INVALID_IMAGE` if the image object doesn't have the depth-stencil target access flag set
- `GR_ERROR_INVALID_IMAGE` if the image object doesn't have the appropriate depth or stencil aspect for read-only depth or stencil flag
- `GR_ERROR_INVALID_FLAGS` if the view creation flags are invalid

Notes

None.

Thread safety

Thread safe.

Shader and Pipeline Functions

grCreateShader

Creates a shader object for the specified source IL.

```
GR_RESULT grCreateShader(
    GR_DEVICE device,
    const GR_SHADER_CREATE_INFO* pCreateInfo,
    GR_SHADER* pShader);
```

Parameters

device

Device handle.

pCreateInfo

[in] Shader creation info. See `GR_SHADER_CREATE_INFO`.

pShader

[out] Shader handle.

Returns

If successful, `grCreateShader()` returns `GR_SUCCESS` and the handle of the created shader object is written to the location specified by `pShader`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if code size is zero
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo`, or `pShader`, or code pointer is `NULL`
- `GR_ERROR_UNSUPPORTED_SHADER_IL_VERSION` if shader IL version is not supported
- `GR_ERROR_BAD_SHADER_CODE` if an unknown shader type or inconsistent shader code is detected
- `GR_ERROR_INVALID_FLAGS` if flags are invalid

Notes

Pre-processes shader IL and performs rudimentary validation of the correctness of shader IL code.

Thread safety

Thread safe.

grCreateGraphicsPipeline

Creates a graphics pipeline object.

```
GR_RESULT grCreateGraphicsPipeline(
    GR_DEVICE device,
    const GR_GRAPHICS_PIPELINE_CREATE_INFO* pCreateInfo,
    GR_PIPELINE* pPipeline);
```

Parameters

device

Device handle.

pCreateInfo

[in] Pipeline creation info. See `GR_GRAPHICS_PIPELINE_CREATE_INFO`.

pPipeline

[out] Pipeline handle.

Returns

If successful, `grCreateGraphicsPipeline()` returns `GR_SUCCESS` and the handle of the created graphics pipeline object is written to the location specified by `pPipeline`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `device` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `device` handle references an invalid object type
- `GR_ERROR_INVALID_HANDLE` if the vertex shader handle is invalid
- `GR_ERROR_INVALID_HANDLE` if either hull or domain shader handle is `GR_NULL_HANDLE` (both have to be either valid or `GR_NULL_HANDLE occurs`)
- `GR_ERROR_INVALID_VALUE` if the topology value is invalid
- `GR_ERROR_INVALID_VALUE` if the logic operation in the color output and blender state is valid
- `GR_ERROR_INVALID_VALUE` if the primitive type is invalid for the given pipeline configuration
- `GR_ERROR_INVALID_VALUE` if the number of control points is invalid for the tessellation pipeline
- `GR_ERROR_INVALID_VALUE` if the logic operation is enabled while some of the color targets enable blending
- `GR_ERROR_INVALID_VALUE` if the dual source blend enable doesn't match expectations for the color target and blend enable setup

- GR_ERROR_INVALID_VALUE if for any of the active shader stages the pLinkConstBufferInfo pointer isn't consistent with the linkConstBufferCount value (the pLinkConstBufferInfo pointer should be valid only if linkConstBufferCount is greater than zero)
- GR_ERROR_INVALID_VALUE if for any of the active shader stages the dynamic data mapping slot object type is invalid (should be either unused, or resource, or UAV)
- GR_ERROR_INVALID_VALUE if the link time constant buffer size or ID is invalid
- GR_ERROR_INVALID_FORMAT if the specified depth-stencil format is invalid
- GR_ERROR_INVALID_FORMAT if the specified color target format is invalid
- GR_ERROR_INVALID_FORMAT if blending is enabled, but the color target format is not compatible with blending
- GR_ERROR_INVALID_FORMAT if a logic op is enabled, but an incompatible format is used
- GR_ERROR_INVALID_POINTER if pCreateInfo or pPipeline is NULL
- GR_ERROR_INVALID_POINTER if the link time constant data pointer is NULL
- GR_ERROR_UNSUPPORTED_SHADER_IL_VERSION if an incorrect shader type is used in any of the shader stages
- GR_ERROR_INVALID_FLAGS if the flags are invalid
- GR_ERROR_INVALID_VALUE if the descriptor set slot type is invalid
- GR_ERROR_INVALID_DESCRIPTOR_SET_DATA if the descriptor set data are invalid or inconsistent

Notes

If the pipeline does not use a color target, the target's format must be undefined and color write mask must be zero.

If the pipeline doesn't use depth-stencil, the depth-stencil's format must be undefined.

Thread safety

Thread safe.

grCreateComputePipeline

Creates a compute pipeline object.

```
GR_RESULT grCreateComputePipeline(
    GR_DEVICE device,
    const GR_COMPUTE_PIPELINE_CREATE_INFO* pCreateInfo,
    GR_PIPELINE* pPipeline);
```

Parameters

device

Device handle.

pCreateInfo

[in] Pipeline creation info. See GR_COMPUTE_PIPELINE_CREATE_INFO.

pPipeline

[out] Compute pipeline handle.

Returns

If successful, grCreateComputePipeline() returns GR_SUCCESS and the handle of the created compute pipeline object is written to the location specified by pPipeline. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_HANDLE if the compute shader handle is invalid
- GR_ERROR_INVALID_VALUE if the pLinkConstBufferInfo pointer isn't consistent with the linkConstBufferCount value (the pLinkConstBufferInfo pointer should be valid only if the linkConstBufferCount is greater than zero)
- GR_ERROR_INVALID_VALUE if the dynamic data mapping slot object type is invalid (should be either unused, resource, or UAV)
- GR_ERROR_INVALID_VALUE if the link time constant buffer size or ID is invalid
- GR_ERROR_INVALID_POINTER if pCreateInfo or pPipeline is NULL
- GR_ERROR_INVALID_POINTER if the link time constant data pointer is NULL
- GR_ERROR_UNSUPPORTED_SHADER_IL_VERSION if an incorrect shader type is used in any of the shader stages
- GR_ERROR_INVALID_FLAGS if the flags are invalid
- GR_ERROR_INVALID_DESCRIPTOR_SET_DATA if descriptor set data are invalid or inconsistent.

Notes

None.

Thread safety

Thread safe.

grStorePipeline

Stores an internal binary pipeline representation to a region of CPU memory.

```
GR_RESULT grStorePipeline(
    GR_PIPELINE pipeline,
    GR_SIZE* pDataSize,
    GR_VOID* pData);
```

Parameters

pipeline

Pipeline handle.

pDataSize

[in/out] Input value specifies the size in bytes of the `pData` output buffer; output value reports the number of bytes written to `pData`.

pData

[out] Internal binary pipeline representation. Can be `NULL`.

Returns

If successful, `grStorePipeline()` returns `GR_SUCCESS` and the internal binary pipeline representation is stored to the location specified by `pData`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the pipeline handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the pipeline handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pDataSize` is `NULL`
- `GR_ERROR_INVALID_MEMORY_SIZE` if `pData` isn't `NULL` and `pDataSize` input value is smaller than the size of the appropriate return data structure

Notes

If `pData` is `NULL`, the input `pDataSize` value does not matter and the function returns the expected pipeline data size in `pDataSize`.

Thread safety

Thread safe.

grLoadPipeline

Creates a pipeline object from an internal binary representation. Only works when the GPU and driver version match those used when grStorePipeline() generated the binary representation.

```
GR_RESULT grLoadPipeline(
    GR_DEVICE device,
    GR_SIZE dataSize,
    const GR_VOID* pData,
    GR_PIPELINE* pPipeline);
```

Parameters

device

Device handle.

dataSize

Data size for internal binary pipeline representation.

pData

[in] Internal binary pipeline representation as generated by grStorePipeline().

pPipeline

[out] Pipeline handle.

Returns

If successful, grLoadPipeline() returns GR_SUCCESS and the handle of the created pipeline object is written to the location specified by pPipeline. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_MEMORY_SIZE if dataSize value does not match the expected pipeline data size
- GR_ERROR_INVALID_POINTER if pData is NULL
- GR_ERROR_INCOMPATIBLE_DEVICE if the device is incompatible with the GPU device where the binary pipeline was saved
- GR_ERROR_INCOMPATIBLE_DRIVER if the driver version is incompatible with the one used for saving the binary pipeline
- GR_ERROR_BAD_PIPELINE_DATA if invalid pipeline code is detected

Notes

None.

Thread safety

Thread safe.

Descriptor Set Functions

grCreateDescriptorSet

Creates a descriptor set object.

```
GR_RESULT grCreateDescriptorSet(
    GR_DEVICE device,
    const GR_DESCRIPTOR_SET_CREATE_INFO* pCreateInfo,
    GR_DESCRIPTOR_SET* pDescriptorSet);
```

Parameters

device

Device handle.

pCreateInfo

[in] Descriptor set creation info. See `GR_DESCRIPTOR_SET_CREATE_INFO`.

pDescriptorSet

[out] Descriptor set object handle.

Returns

If successful, `grCreateDescriptorSet()` returns `GR_SUCCESS` and the handle of the created descriptor set object is written to the location specified by `pDescriptorSet`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the number of slots is greater than the max allowed descriptor set size
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pDescriptorSet` is `NULL`

Notes

None.

Thread safety

Thread safe.

grBeginDescriptorSetUpdate

Initiates a descriptor set update. Should be called before updating descriptor set slots using `grAttachSamplerDescriptors()`, `grAttachImageViewDescriptors()`, `grAttachMemoryViewDescriptors()`, `grAttachNestedDescriptors()`, or `grClearDescriptorSetSlots()`.

```
GR_VOID grBeginDescriptorSetUpdate(
    GR_DESCRIPTOR_SET descriptorSet);
```

Parameters

descriptorSet

Descriptor set handle.

Returns

None.

Notes

For performance reasons, this function does not perform any sanity checking.

Thread safety

Not thread safe for calls referencing the same descriptor set object.

grEndDescriptorSetUpdate

Ends a descriptor set update. Should be called after updating descriptor set slots.

```
GR_VOID grEndDescriptorSetUpdate(
    GR_DESCRIPTOR_SET descriptorSet);
```

Parameters

descriptorSet

Descriptor set handle.

Returns

None.

Notes

For performance reasons, this function does not perform any sanity checking.

Thread safety

Not thread safe for calls referencing the same descriptor set object.

grAttachSamplerDescriptors

Updates a range of descriptor set slots with sampler objects.

```
GR_VOID grAttachSamplerDescriptors(
    GR_DESCRIPTOR_SET descriptorSet,
    GR_UINT startSlot,
    GR_UINT slotCount,
    const GR_SAMPLER* pSamplers);
```

Parameters

descriptorSet

Descriptor set handle.

startSlot

First descriptor set slot in a range to update.

slotCount

Number of descriptor set slots to update.

pSamplers

[in] Array of sampler object handles.

Returns

None.

Notes

For performance reasons, this function does not perform any sanity checking.

Thread safety

Not thread safe for calls referencing the same descriptor set object.

grAttachImageViewDescriptors

Updates a range of descriptor set slots with image view objects.

```
GR_VOID grAttachImageViewDescriptors(
    GR_DESCRIPTOR_SET descriptorSet,
    GR_UINT startSlot,
    GR_UINT slotCount,
    const GR_IMAGE_VIEW_ATTACH_INFO* pImageViews);
```

Parameters

descriptorSet

Descriptor set handle.

startSlot

First descriptor set slot in a range to update.

slotCount

Number of descriptor set slots to update.

pImageViews

[in] Array of image view object handles and attachment properties. See `GR_IMAGE_VIEW_ATTACH_INFO`.

Returns

None.

Notes

For performance reasons, this function does not perform any sanity checking.

Thread safety

Not thread safe for calls referencing the same descriptor set object.

grAttachMemoryViewDescriptors

Updates a range of descriptor set slots with memory views.

```
GR_VOID grAttachMemoryViewDescriptors(
    GR_DESCRIPTOR_SET descriptorSet,
    GR_UINT startSlot,
    GR_UINT slotCount,
    const GR_MEMORY_VIEW_ATTACH_INFO* pMemViews);
```

Parameters

descriptorSet

Descriptor set handle.

startSlot

First descriptor set slot in a range to update.

slotCount

Number of descriptor set slots to update.

pMemViews

[in] Array of memory views. See `GR_MEMORY_VIEW_ATTACH_INFO`.

Returns

None.

Notes

For performance reasons, this function does not perform any sanity checking.

Thread safety

Not thread safe for calls referencing the same descriptor set object.

grAttachNestedDescriptors

Updates a range of descriptor set slots with nested references to other descriptor sets.

```
GR_VOID grAttachNestedDescriptors(
    GR_DESCRIPTOR_SET descriptorSet,
    GR_UINT startSlot,
    GR_UINT slotCount,
    const GR_DESCRIPTOR_SET_ATTACH_INFO* pNestedDescriptorSets);
```

Parameters

descriptorSet

Descriptor set handle.

startSlot

First descriptor set slot in a range to update.

slotCount

Number of descriptor set slots to update.

pNestedDescriptorSets

[in] Array of nested descriptor sets and attachment point offsets. See `GR_DESCRIPTOR_SET_ATTACH_INFO`.

Returns

None.

Notes

For performance reasons, this function does not perform any sanity checking.

Thread safety

Not thread safe for calls referencing the same descriptor set object.

grClearDescriptorSetSlots

Resets a range of descriptor set slots to a cleared state.

```
GR_VOID grClearDescriptorSetSlots(
    GR_DESCRIPTOR_SET descriptorSet,
    GR_UINT startSlot,
    GR_UINT slotCount);
```

Parameters

descriptorSet

Descriptor set handle.

startSlot

First descriptor set slot in a range to update.

slotCount

Number of descriptor set slots to update.

Returns

None.

Notes

For performance reasons, this function does not perform any sanity checking.

Thread safety

Not thread safe for calls referencing the same descriptor set object.

State Object Functions

grCreateViewportState

Creates a viewport state object.

```
GR_RESULT grCreateViewportState(
    GR_DEVICE device,
    const GR_VIEWPORT_STATE_CREATE_INFO* pCreateInfo,
    GR_VIEWPORT_STATE_OBJECT* pState);
```

Parameters

device

Device handle.

pCreateInfo

[in] Viewport state object creation info. See `GR_VIEWPORT_STATE_CREATE_INFO`.

pState

[out] Viewport state object handle.

Returns

If successful, `grCreateViewportState()` returns `GR_SUCCESS` and the handle of the created viewport state object is written to the location specified by `pState`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if an invalid number of viewports is specified
- `GR_ERROR_INVALID_VALUE` if any viewport parameters are outside of the valid range
- `GR_ERROR_INVALID_VALUE` if scissors are enabled and any scissor parameters are outside of the valid range
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pState` is `NULL`

Notes

Viewport offset is valid in the [-32768..32768] range and scissor offset is valid in the [0..32768] range. Both the viewport and scissor sizes cannot exceed 32768 pixels.

Thread safety

Thread safe.

grCreateRasterState

Creates a rasterizer state object.

```
GR_RESULT grCreateRasterState(
    GR_DEVICE device,
    const GR_RASTER_STATE_CREATE_INFO* pCreateInfo,
    GR_RASTER_STATE_OBJECT* pState);
```

Parameters

device

Device handle.

pCreateInfo

[in] Rasterizer state object creation info. See `GR_RASTER_STATE_CREATE_INFO`.

pState

[out] Rasterizer state object handle.

Returns

If successful, `grCreateRasterState()` returns `GR_SUCCESS` and the handle of the created rasterizer state object is written to the location specified by `pState`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the fill mode or cull mode is invalid
- `GR_ERROR_INVALID_VALUE` if the face orientation is invalid
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pState` is `NULL`

Notes

None.

Thread safety

Thread safe.

grCreateColorBlendState

Creates a color blender state object.

```
GR_RESULT grCreateColorBlendState(
    GR_DEVICE device,
    const GR_COLOR_BLEND_STATE_CREATE_INFO* pCreateInfo,
    GR_COLOR_BLEND_STATE_OBJECT* pState);
```

Parameters

device

Device handle.

pCreateInfo

[in] Blender state object creation info. See GR_COLOR_BLEND_STATE_CREATE_INFO.

pState

[out] Blender state object handle.

Returns

If successful, grCreateColorBlendState() returns GR_SUCCESS and the handle of the created color blend state object is written to the location specified by pState. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_VALUE if the source or destination color/alpha blend operation is invalid for color targets that have blending enabled
- GR_ERROR_INVALID_VALUE if the color/alpha blend function is invalid for color targets that have blending enabled
- GR_ERROR_INVALID_VALUE if an unsupported blend function is used with a dual source blend
- GR_ERROR_INVALID_POINTER if pCreateInfo or pState is NULL

Notes

None.

Thread safety

Thread safe.

grCreateDepthStencilState

Creates a depth-stencil state object.

```
GR_RESULT grCreateDepthStencilState(
    GR_DEVICE device,
    const GR_DEPTH_STENCIL_STATE_CREATE_INFO* pCreateInfo,
    GR_DEPTH_STENCIL_STATE_OBJECT* pState);
```

Parameters

device

Device handle.

pCreateInfo

[in] Depth-stencil state object creation info. See GR_DEPTH_STENCIL_STATE_CREATE_INFO.

pState

[out] Depth-stencil state object handle.

Returns

If successful, grCreateDepthStencilState() returns GR_SUCCESS and the handle of the created depth-stencil state object is written to the location specified by pState. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_VALUE if the depth function is invalid
- GR_ERROR_INVALID_VALUE if the depth bounds feature is enabled and the depth range is invalid
- GR_ERROR_INVALID_VALUE if any stencil operation is invalid
- GR_ERROR_INVALID_POINTER if pCreateInfo or pState is NULL

Notes

None.

Thread safety

Thread safe.

grCreateMsaaState

Creates multisampling state object.

```
GR_RESULT grCreateMsaaState(
    GR_DEVICE device,
    const GR_MSAA_STATE_CREATE_INFO* pCreateInfo,
    GR_MSAA_STATE_OBJECT* pState);
```

Parameters

device

Device handle.

pCreateInfo

[in] Multisampling state object creation info. See GR_MSAA_STATE_CREATE_INFO.

pState

[out] Multisampling state object handle.

Returns

If successful, grCreateMsaaState() returns GR_SUCCESS and the handle of the created MSAA state object is written to the location specified by pState. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_VALUE if the number of samples is unsupported
- GR_ERROR_INVALID_POINTER if pCreateInfo or pState is NULL

Notes

None.

Thread safety

Thread safe.

Query and Synchronization Function

grCreateQueryPool

Creates a query object pool of the specified size.

```
GR_RESULT grCreateQueryPool(
    GR_DEVICE device,
    const GR_QUERY_POOL_CREATE_INFO* pCreateInfo,
    GR_QUERY_POOL* pQueryPool);
```

Parameters

device

Device handle.

pCreateInfo

[in] Query pool object creation info. See `GR_QUERY_POOL_CREATE_INFO`.

pQueryPool

[out] Query pool handle.

Returns

If successful, `grCreateQueryPool()` returns `GR_SUCCESS` and the handle of the created query pool object is written to the location specified by `pQueryPool`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the query type is invalid
- `GR_ERROR_INVALID_VALUE` if the number of slots is zero
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pQueryPool` is `NULL`

Notes

None.

Thread safety

Thread safe.

grGetQueryPoolResults

Retrieves query results from a query pool. Multiple consecutive query results can be retrieved with one function call.

```
GR_RESULT grGetQueryPoolResults(
    GR_QUERY_POOL queryPool,
    GR_UINT startQuery,
    GR_UINT queryCount,
    GR_SIZE* pDataSize,
    GR_VOID* pData);
```

Parameters

queryPool

Query pool handle.

startQuery

Start of query pool slot range for which data are retrieved.

queryCount

Consecutive number of query slots in a range for which data are retrieved.

pDataSize

[in/out] Input value specifies the size in bytes of the `pData` output buffer; output value reports the number of bytes written to `pData`.

pData

[out] Query results. Can be `NULL`.

Returns

If the function is successful and all query slot information is available, `grGetQueryPoolResults()` returns `GR_SUCCESS` and the query results are written to the location specified by `pData`. If the function executed successfully and any of the requested query slots do not have results available, the function returns `GR_NOT_READY`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `queryPool` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `queryPool` handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the range of queries defined by `startQuery` and `queryCount` is invalid for the given query pool
- `GR_ERROR_INVALID_POINTER` if `pDataSize` is `NULL`
- `GR_ERROR_INVALID_MEMORY_SIZE` if `pData` isn't `NULL` and `pDataSize` input value is smaller than the size of the appropriate return data
- `GR_ERROR_MEMORY_NOT_BOUND` if query pool requires GPU memory, but it wasn't bound

Notes

If pData is NULL, the input pDataSize value does not matter and the function returns the expected data structure size in pDataSize.

Occlusion query results

For occlusion queries, the results are returned as an array of 64-bit unsigned integers, one per query pool slot.

Pipeline statistics results

For pipeline statistics queries, the results are returned as an array of the GR_PIPELINE_STATISTICS_DATA structures, one per query pool slot.

Thread safety

Not thread safe for calls referencing the same query pool object.

grCreateFence

Creates a GPU execution fence object.

```
GR_RESULT grCreateFence(
    GR_DEVICE device,
    const GR_FENCE_CREATE_INFO* pCreateInfo,
    GR_FENCE* pFence);
```

Parameters

device

Device handle.

pCreateInfo

[in] Fence object creation info. See GR_FENCE_CREATE_INFO.

pFence

[out] Fence object handle.

Returns

If successful, grCreateFence() returns GR_SUCCESS and the handle of the created fence object is written to the location specified by pFence. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_FLAGS if the flags are invalid
- GR_ERROR_INVALID_POINTER if pCreateInfo or pFence is NULL

Notes

The fence creation flags are currently reserved.

Thread safety

Thread safe.

grGetFenceStatus

Retrieves the status of a fence object.

```
GR_RESULT grGetFenceStatus(
    GR_FENCE fence);
```

Parameters

fence

Fence object handle.

Returns

If the function call is successful and the fence has been reached, `grGetFenceStatus()` returns `GR_SUCCESS`. If the function call is successful, but the fence hasn't been reached, the function returns `GR_NOT_READY`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the fence handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the fence handle references an invalid object type
- `GR_ERROR_UNAVAILABLE` if the fence hasn't been submitted by the application
- `GR_ERROR_MEMORY_NOT_BOUND` if fence requires GPU memory, but it wasn't bound

Notes

The fence has to be submitted at least once to return a non-error result.

Thread safety

Thread safe.

grWaitForFences

Stalls the current thread until any or all of the fences have been reached by GPU.

```
GR_RESULT grWaitForFences(
    GR_DEVICE device,
    GR_UINT fenceCount,
    const GR_FENCE* pFences,
    GR_BOOL waitAll,
    GR_FLOAT timeout);
```

Parameters

device

Device handle.

fenceCount

Number of fences to wait for.

pFences

[in] Array of fence object handles.

waitAll

Wait behavior: if `GR_TRUE`, wait for completion of all fences in the provided list; if `GR_FALSE`, wait for completion of any of the provided fences.

timeout

Wait timeout in seconds.

Returns

If the function executed successfully and the fences have been reached, `grWaitForFences()` returns `GR_SUCCESS`. If the function executed successfully, but the fences haven't been reached before the timeout, the function returns `GR_TIMEOUT`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_HANDLE` if any of the fence handles are invalid
- `GR_ERROR_INVALID_VALUE` if the fence count is zero
- `GR_ERROR_INVALID_POINTER` if pFences is NULL
- `GR_ERROR_UNAVAILABLE` if any of the fences haven't been submitted by the application

Notes

All fences have to be submitted at least once to return a non-error result. Returns `GR_TIMEOUT` if the required fences have not completed after timeout seconds have passed. Using zero timeout value returns immediately and can be used to determine if all required fences have been completed.

Thread safety

Thread safe.

grCreateQueueSemaphore

Creates a counting semaphore object to be used for GPU queue synchronization.

```
GR_RESULT grCreateQueueSemaphore(
    GR_DEVICE device,
    const GR_QUEUE_SEMAPHORE_CREATE_INFO* pCreateInfo,
    GR_QUEUE_SEMAPHORE* pSemaphore);
```

Parameters

device

Device handle.

pCreateInfo

[in] Queue semaphore object creation info. See `GR_QUEUE_SEMAPHORE_CREATE_INFO`.

pSemaphore

[out] Queue semaphore object handle.

Returns

If successful, `grCreateQueueSemaphore()` returns `GR_SUCCESS` and the handle of the created queue semaphore object is written to the location specified by `pSemaphore`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the initial semaphore count is invalid
- `GR_ERROR_INVALID_FLAGS` if the flags are invalid
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pSemaphore` is `NULL`

Notes

The semaphore creation flags are currently reserved.

The initial semaphore count must be in the range [0..31].

Thread safety

Thread safe.

grSignalQueueSemaphore

Inserts a semaphore signal into a GPU queue.

```
GR_RESULT grSignalQueueSemaphore(
    GR_QUEUE queue,
    GR_QUEUE_SEMAPHORE semaphore);
```

Parameters

queue

Queue handle.

semaphore

Queue semaphore to signal.

Returns

`grSignalQueueSemaphore()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the queue or semaphore handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the queue handle references an invalid object type

Notes

None.

Thread safety

Not thread safe for calls referencing the same queue or semaphore object.

grWaitQueueSemaphore

Inserts a semaphore wait into a GPU queue.

```
GR_RESULT grWaitQueueSemaphore(
    GR_QUEUE queue,
    GR_QUEUE_SEMAPHORE semaphore);
```

Parameters

queue

Queue handle.

semaphore

Queue semaphore to wait on.

Returns

`grWaitQueueSemaphore()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the queue or semaphore handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the queue handle references an invalid object type

Notes

None.

Thread safety

Not thread safe for calls referencing the same queue or semaphore object.

grCreateEvent

Creates a GPU event object that can be set and reset by the CPU directly or by the GPU via command buffers.

```
GR_RESULT grCreateEvent(
    GR_DEVICE device,
    const GR_EVENT_CREATE_INFO* pCreateInfo,
    GR_EVENT* pEvent);
```

Parameters

device

Device handle.

pCreateInfo

[in] Event object creation info. See `GR_EVENT_CREATE_INFO`.

pEvent

[out] Event object handle.

Returns

If successful, `grCreateEvent()` returns `GR_SUCCESS` and the handle of the create event object is written to the location specified by `pEvent`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_FLAGS` if the flags are invalid
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pEvent` is `NULL`

Notes

The event creation flags are currently reserved.

The event is in the reset state at creation.

Thread safety

Thread safe.

grGetEventStatus

Retrieves the status of an event object.

```
GR_RESULT grGetEventStatus(
    GR_EVENT event);
```

Parameters

event

Event object handle.

Returns

If the function executed successfully and the event is set, grGetEventStatus() returns GR_EVENT_SET. If the function executed successfully and the event is reset, the function returns GR_EVENT_RESET. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the event handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the event handle references an invalid object type
- GR_ERROR_MEMORY_NOT_BOUND if event requires GPU memory, but it wasn't bound

Notes

None.

Thread safety

Not thread safe for calls accessing the same event object.

grSetEvent

Sets an event object's status from the CPU.

```
GR_RESULT grSetEvent(
    GR_EVENT event);
```

Parameters

event

Event object handle.

Returns

grSetEvent() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the event handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the event handle references an invalid object type
- GR_ERROR_MEMORY_NOT_BOUND if event requires GPU memory, but it wasn't bound

Notes

None.

Thread safety

Not thread safe for calls accessing the same event object.

grResetEvent

Resets an event object’s status from the CPU.

```
GR_RESULT grResetEvent(
    GR_EVENT event);
```

Parameters

event

Event object handle.

Returns

`gResetEvent()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the event handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the event handle references an invalid object type
- `GR_ERROR_MEMORY_NOT_BOUND` if event requires GPU memory, but it wasn’t bound

Notes

None.

Thread safety

Not thread safe for calls accessing the same event object.

Multi-device Management Functions

grGetMultiGpuCompatibility

Retrieves information about Mantle GPU compatibility features.

```
GR_RESULT grGetMultiGpuCompatibility(
    GR_PHYSICAL_GPU gpu0,
    GR_PHYSICAL_GPU gpu1,
    GR_GPU_COMPATIBILITY_INFO* pInfo);
```

Parameters

gpu0

First physical GPU handle.

gpu1

Second physical GPU handle.

pInfo

[out] Multi-GPU compatibility info. See GR_GPU_COMPATIBILITY_INFO.

Returns

If successful, grGetMultiGpuCompatibility() returns GR_SUCCESS and multi-GPU compatibility information. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the gpu0 or gpu1 handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the gpu0 or gpu1 handle references an invalid object type
- GR_ERROR_INVALID_POINTER if pInfo is NULL

Notes

None.

Thread safety

Not thread safe.

grOpenSharedMemory

Opens a previously created GPU memory object for sharing on another device.

```
GR_RESULT grOpenSharedMemory(
    GR_DEVICE device,
    const GR_MEMORY_OPEN_INFO* pOpenInfo,
    GR_GPU_MEMORY* pMem);
```

Parameters

device

Device handle.

pOpenInfo

[in] Data for opening a shared memory. See `GR_MEMORY_OPEN_INFO`.

pMem

[out] Shared memory object handle.

Returns

If successful, `grOpenSharedMemory()` returns `GR_SUCCESS` and the handle of the shared GPU memory object is written to the location specified by `pMem`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pOpenInfo` or `pMem` are `NULL`
- `GR_ERROR_INVALID_HANDLE` if `pOpenInfo->sharedMem` handle is invalid
- `GR_ERROR_NOT_SHAREABLE` if `pOpenInfo->sharedMem` was not marked as shareable at creation

Notes

None.

Thread safety

Thread safe.

grOpenSharedQueueSemaphore

Opens a previously created queue semaphore object for sharing on another device.

```
GR_RESULT grOpenSharedQueueSemaphore(
    GR_DEVICE device,
    const GR_QUEUE_SEMAPHORE_OPEN_INFO* pOpenInfo,
    GR_QUEUE_SEMAPHORE* pSemaphore);
```

Parameters

device

Device handle.

pOpenInfo

[in] Data for opening a shared queue semaphore. See `GR_QUEUE_SEMAPHORE_OPEN_INFO`.

pSemaphore

[out] Shared queue semaphore handle.

Returns

If successful, `grOpenSharedQueueSemaphore()` returns `GR_SUCCESS` and the handle of the shared queue semaphore object is written to the location specified by `pSemaphore`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pOpenInfo` or `pSemaphore` are `NULL`
- `GR_ERROR_INVALID_HANDLE` if `pOpenInfo->sharedSemaphore` handle is invalid
- `GR_ERROR_NOT_SHAREABLE` if `pOpenInfo->sharedSemaphore` was not marked as shareable at creation

Notes

None.

Thread safety

Thread safe.

grOpenPeerMemory

Opens a previously created GPU memory object for peer access on another device.

```
GR_RESULT grOpenPeerMemory(
    GR_DEVICE device,
    const GR_PEER_MEMORY_OPEN_INFO* pOpenInfo,
    GR_GPU_MEMORY* pMem);
```

Parameters

device

Device handle.

pOpenInfo

[in] Data for opening a peer memory. See `GR_PEER_MEMORY_OPEN_INFO`.

pMem

[out] Peer access memory object handle.

Returns

If successful, `grOpenPeerMemory()` returns `GR_SUCCESS` and the handle of the peer access memory object is written to the location specified by `pMem`. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_POINTER if pOpenInfo or pMem are NULL
- GR_ERROR_INVALID_HANDLE if pOpenInfo->originalMem handle is invalid

Notes

None.

Thread safety

Thread safe.

grOpenPeerImage

Opens a previously created image object for peer access on another device.

```
GR_RESULT grOpenPeerImage(
    GR_DEVICE device,
    const GR_PEER_IMAGE_OPEN_INFO* pOpenInfo,
    GR_IMAGE* pImage,
    GR_GPU_MEMORY* pMem);
```

Parameters

device

Device handle.

pOpenInfo

[in] Data for opening a peer image. See GR_PEER_IMAGE_OPEN_INFO.

pImage

[out] Peer access image object handle.

pMem

[out] Memory object handle for peer access image.

Returns

If successful, grOpenPeerImage() returns GR_SUCCESS, the handle of the peer access image object is written to the location specified by pImage, and the memory object handle for the peer access image object is written to the location specified by pMem. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_POINTER if pOpenInfo, or pImage. or pMem are NULL
- GR_ERROR_INVALID_HANDLE if pOpenInfo->originalImage handle is invalid

Notes

None.

Thread safety

Thread safe.

Command Buffer Management Functions

grCreateCommandBuffer

Creates a command buffer object that supports submission to a specific queue type.

```
GR_RESULT grCreateCommandBuffer(
    GR_DEVICE device,
    const GR_CMD_BUFFER_CREATE_INFO* pCreateInfo,
    GR_CMD_BUFFER* pCmdBuffer);
```

Parameters

device

Device handle.

pCreateInfo

[in] Command buffer creation info. See GR_CMD_BUFFER_CREATE_INFO.

pCmdBuffer

[out] Command buffer object handle.

Returns

If successful, grCreateCommandBuffer() returns GR_SUCCESS and the handle of the created command buffer object is written to the location specified by pCmdBuffer. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_POINTER if pCreateInfo or pCmdBuffer is NULL
- GR_ERROR_INVALID_FLAGS if the flags are invalid
- GR_ERROR_INVALID_QUEUE_TYPE if the queue is invalid or not supported by the device

Notes

The command buffer creation flags are currently reserved.

Thread safety

Thread safe.

grBeginCommandBuffer

Resets the command buffer's previous contents and state, then puts it in a *building state* allowing new command buffer data to be recorded.

```
GR_RESULT grBeginCommandBuffer(
    GR_CMD_BUFFER cmdBuffer,
    GR_FLAGS flags);
```

Parameters

cmdBuffer

Command buffer handle.

flags

Command buffer recording flags. See GR_CMD_BUFFER_BUILD_FLAGS.

Returns

grBeginCommandBuffer() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the cmdBuffer handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the cmdBuffer handle references an invalid object type
- GR_ERROR_INVALID_FLAGS if the flags are invalid
- GR_ERROR_INCOMPLETE_COMMAND_BUFFER if the command buffer is already in the building state

Notes

The command buffer begin flags are currently reserved.

Trying to start command buffer recording on a buffer that was not properly terminated returns GR_ERROR_INCOMPLETE_COMMAND_BUFFER. In that case, the application should explicitly reset the command buffer by calling grResetCommandBuffer().

Thread safety

Not thread safe for calls referencing the same command buffer object.

grEndCommandBuffer

Completes recording of a command buffer in the *building state*.

```
GR_RESULT grEndCommandBuffer(
    GR_CMD_BUFFER cmdBuffer);
```

Parameters

cmdBuffer

Command buffer handle.

Returns

`grEndCommandBuffer()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `cmdBuffer` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `cmdBuffer` handle references an invalid object type
- `GR_ERROR_INCOMPLETE_COMMAND_BUFFER` if the command buffer is not in the building state
- `GR_ERROR_BUILDING_COMMAND_BUFFER` if some error has occurred during the command buffer building and the command buffer cannot be used for submission

Notes

Trying to end a command buffer that isn't in a building state returns `GR_ERROR_INCOMPLETE_COMMAND_BUFFER`. In that case, the application should explicitly reset the command buffer by calling `grResetCommandBuffer()`.

Thread safety

Not thread safe for calls referencing the same command buffer object.

grResetCommandBuffer

Explicitly resets a command buffer and releases any internal resources associated with it. Must be used to reset command buffers that have previously reported a `GR_ERROR_INCOMPLETE_COMMAND_BUFFER` error.

```
GR_RESULT grResetCommandBuffer(
    GR_CMD_BUFFER cmdBuffer);
```

Parameters

cmdBuffer

Command buffer handle.

Returns

`grResetCommandBuffer()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the `cmdBuffer` handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the `cmdBuffer` handle references an invalid object type

Notes

None.

Thread safety

Not thread safe for calls referencing the same command buffer object.

Command Buffer Building Functions

General notes:

For performance reasons, all command building function do not perform a lot of sanity checking. If something goes wrong, the call gets dropped or produces undefined results.

General thread safety:

Not thread safe for calls referencing the same command buffer object.

grCmdBindPipeline

Binds a graphics or compute pipeline to the current command buffer state.

```
GR_VOID grCmdBindPipeline(
    GR_CMD_BUFFER cmdBuffer,
    GR_ENUM pipelineBindPoint,
    GR_PIPELINE pipeline);
```

Parameters

cmdBuffer

Command buffer handle.

pipelineBindPoint

Pipeline binding point (graphics or compute). See `GR_PIPELINE_BIND_POINT`.

pipeline

Pipeline object handle.

Notes

None.

grCmdBindStateObject

Binds a state object to the current command buffer state.

```
GR_VOID grCmdBindStateObject(
    GR_CMD_BUFFER cmdBuffer,
    GR_ENUM stateBindPoint,
    GR_STATE_OBJECT state);
```

Parameters

cmdBuffer

Command buffer handle.

stateBindPoint

State bind point. See `GR_STATE_BIND_POINT`.

pipeline

State object handle.

Notes

None.

grCmdBindDescriptorSet

Binds a descriptor set to the current command buffer state making it accessible from either the graphics or compute pipeline.

```
GR_VOID grCmdBindDescriptorSet(
    GR_CMD_BUFFER cmdBuffer,
    GR_ENUM pipelineBindPoint,
    GR_UINT index,
    GR_DESCRIPTOR_SET descriptorSet,
    GR_UINT slotOffset);
```

Parameters

cmdBuffer

Command buffer handle.

pipelineBindPoint

Pipeline type to make the descriptor set available to. See `GR_PIPELINE_BIND_POINT`.

index

Descriptor set bind point index.

descriptorSet

Descriptor set handle.

slotOffset

Descriptor set slot offset to use for binding.

Notes

None.

grCmdBindDynamicMemoryView

Binds a dynamic memory view to the current command buffer state making it accessible from either the graphics or compute pipeline.

```
GR_VOID grCmdBindDynamicMemoryView(
    GR_CMD_BUFFER cmdBuffer,
    GR_ENUM pipelineBindPoint,
    const GR_MEMORY_VIEW_ATTACH_INFO* pMemView);
```

Parameters

cmdBuffer

Command buffer handle.

pipelineBindPoint

Pipeline type to make the dynamic memory view available to. See `GR_PIPELINE_BIND_POINT`.

pMemView

[in] Memory view description. See `GR_MEMORY_VIEW_ATTACH_INFO`. Can be `NULL`.

Notes

The dynamic memory view behaves similarly to a memory view bound through the descriptor set hierarchy.

Specifying `NULL` in `pMemView` unbinds currently bound dynamic memory view.

grCmdBindIndexData

Binds draw index data to the current command buffer state.

```
GR_VOID grCmdBindIndexData(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY mem,
    GR_GPU_SIZE offset,
    GR_ENUM indexType);
```

Parameters

cmdBuffer

Command buffer handle.

mem

Memory object containing index data. Can be `GR_NULL_HANDLE`.

offset

Byte offset within a memory object to the first index.

indexType

Data type of the indices. See `GR_INDEX_TYPE`.

Notes

The memory offset must be index element size aligned – 2-byte aligned for 16-bit indices or 4-byte aligned for 32-bit indices.

Memory regions containing indices needs to be in the GR_MEMORY_STATE_INDEX_DATA state before being bound as index data.

Specifying GR_NULL_HANDLE for memory object unbinds currently bound index data.

grCmdBindTargets

Binds color and depth-stencil targets to the current command buffer state. The current image state for targets is also specified at bind time.

```
GR_VOID grCmdBindTargets(
    GR_CMD_BUFFER cmdBuffer,
    GR_UINT colorTargetCount,
    const GR_COLOR_TARGET_BIND_INFO* pColorTargets,
    const GR_DEPTH_STENCIL_BIND_INFO* pDepthTarget);
```

Parameters

cmdBuffer

Command buffer handle.

colorTargetCount

Number of color render targets to be bound.

pColorTargets

[in] Array of color render target view handles. See GR_COLOR_TARGET_BIND_INFO. Can be NULL if colorTargetCount is zero.

pDepthTarget

[in] Depth-stencil view handle. See GR_DEPTH_STENCIL_BIND_INFO. Can be NULL.

Notes

The valid states for binding color targets and depth-stencil target are GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL and GR_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL. Additionally depth-stencil target can be in GR_IMAGE_STATE_TARGET_AND_SHADER_READ_ONLY for either depth or stencil aspects.

Specifying NULL pDepthTarget unbinds previously bound depth-stencil target.

Specifying NULL pColorTargets and zero colorTargetCount unbinds all previously bound color targets. Specifying GR_NULL_HANDLE for any of the color target view objects also unbinds previously bound color target.

grCmdPrepareMemoryRegions

Specifies memory region state transition for a given list of memory objects.

```
GR_VOID grCmdPrepareMemoryRegions(
    GR_CMD_BUFFER cmdBuffer,
    GR_UINT transitionCount,
    const GR_MEMORY_STATE_TRANSITION* pStateTransitions);
```

Parameters

cmdBuffer

Command buffer handle.

transitionCount

Number of memory regions that need to perform a state transition.

pStateTransitions

[in] Array of structures describing each memory region state transition. See `GR_MEMORY_STATE_TRANSITION`.

Notes

Specifying a memory range (or some portion of a range) multiple times produces undefined results.

Memory range that has not been used before is assumed to be in the `GR_MEMORY_STATE_DATA_TRANSFER` state.

grCmdPrepareImages

Specifies image state transitions for a given list of image resources.

```
GR_VOID grCmdPrepareImages(
    GR_CMD_BUFFER cmdBuffer,
    GR_UINT transitionCount,
    const GR_IMAGE_STATE_TRANSITION* pStateTransitions);
```

Parameters

cmdBuffer

Command buffer handle.

transitionCount

Number of image resources that need to perform a state transition.

pStateTransitions

[in] Array of structures describing each image state transition. See `GR_IMAGE_STATE_TRANSITION`.

Notes

Specifying a subresource multiple times in one image preparation call produces undefined results.

When memory is bound to an image object used as a render target or a depth-stencil, the resource state is implicitly set to `GR_IMAGE_STATE_UNINITIALIZED`, and it needs to be transitioned to a proper state before its first use. All other image objects implicitly receive the `GR_IMAGE_STATE_DATA_TRANSFER` state on memory bind.

grCmdDraw

Draws instanced, non-indexed geometry using the current graphics state.

```
GR_VOID grCmdDraw(
    GR_CMD_BUFFER cmdBuffer,
    GR_UINT firstVertex,
    GR_UINT vertexCount,
    GR_UINT firstInstance,
    GR_UINT instanceCount);
```

Parameters

cmdBuffer

Command buffer handle.

firstVertex

Offset to the first vertex.

vertexCount

Number of vertices per instance to draw.

firstInstance

Offset to the first instance.

instanceCount

Number of instances to draw.

Notes

None.

grCmdDrawIndexed

Draws instanced, indexed geometry using the current graphics state.

```
GR_VOID grCmdDrawIndexed(
    GR_CMD_BUFFER cmdBuffer,
    GR_UINT firstIndex,
    GR_UINT indexCount,
    GR_INT vertexOffset,
    GR_UINT firstInstance,
    GR_UINT instanceCount);
```

Parameters

cmdBuffer

Command buffer handle.

firstIndex

Offset to the first index.

indexCount

Number of indices per instance to draw.

vertexOffset

Vertex offset to be added to each vertex index.

firstInstance

Offset to the first instance.

instanceCount

Number of instances to draw.

Notes

None.

grCmdDrawIndirect

Draws instanced, non-indexed geometry using the current graphics state. The draw arguments come from data stored in GPU memory.

```
GR_VOID grCmdDrawIndirect(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY mem,
    GR_GPU_SIZE offset);
```

Parameters

cmdBuffer

Command buffer handle.

mem

Memory object containing the draw argument data.

offset

Byte offset from the beginning of the memory object to the draw argument data.

Notes

Draw argument data offset in memory must be 4-byte aligned. The layout of the argument data is defined in `GR_DRAW_INDIRECT_ARG`.

The memory range used for draw arguments needs to be in the `GR_MEMORY_STATE_INDIRECT_ARG` state before using it as argument data.

grCmdDrawIndexedIndirect

Draws instanced, indexed geometry using the current graphics state. The draw arguments come from data stored in GPU memory.

```
GR_VOID grCmdDrawIndexedIndirect(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY mem,
    GR_GPU_SIZE offset);
```

Parameters

cmdBuffer

Command buffer handle.

mem

Memory object containing the draw argument data.

offset

Byte offset from the beginning of the memory object to the draw argument data.

Notes

Draw argument data offset in the memory must be 4-byte aligned. The layout of the argument data is defined in `GR_DRAW_INDEXED_INDIRECT_ARG`.

The memory range used for draw arguments needs to be in the `GR_MEMORY_STATE_INDIRECT_ARG` state before using it as argument data.

grCmdDispatch

Dispatches a compute workload of the given dimensions using the current compute state.

```
GR_VOID grCmdDispatch(
    GR_CMD_BUFFER cmdBuffer,
    GR_UINT x,
    GR_UINT y,
    GR_UINT z);
```

Parameters

cmdBuffer

Command buffer handle.

x

Thread groups to dispatch in the X dimension.

y

Thread groups to dispatch in the Y dimension.

z

Thread groups to dispatch in the Z dimension.

Notes

The thread group size is defined in the compute shader.

grCmdDispatchIndirect

Dispatches a compute workload using the current compute state. The dimensions of the workload come from data stored in GPU memory.

```
GR_VOID grCmdDispatchIndirect(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY mem,
    GR_GPU_SIZE offset);
```

Parameters

cmdBuffer

Command buffer handle.

mem

Memory object containing the dispatch arguments.

offset

Byte offset from the beginning of the memory object to the dispatch argument data.

Notes

The thread group size is defined in the compute shader.

The dispatch argument data offset in the memory object must be 4-byte aligned. The layout of the argument data is defined in GR_DISPATCH_INDIRECT_ARG.

The memory range used for dispatch arguments needs to be in the GR_MEMORY_STATE_INDIRECT_ARG state before using it as argument data.

grCmdCopyMemory

Copies multiple regions from one GPU memory object to another.

```
GR_VOID grCmdCopyMemory(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY srcMem,
    GR_GPU_MEMORY destMem,
    GR_UINT regionCount,
    const GR_MEMORY_COPY* pRegions);
```

Parameters

cmdBuffer

Command buffer handle.

srcMem

Source memory object.

destMem

Destination memory object.

regionCount

Number of regions for the copy operation.

pRegions

[in] Array of copy region descriptors. See GR_MEMORY_COPY.

Notes

None of the destination regions are allowed to overlap with each other or with source regions. Overlapping any of them produces undefined results.

For performance reasons, it is preferred to align offsets and copy sizes to 4-byte boundaries.

Both the source and destination memory regions must be in the GR_MEMORY_STATE_DATA_TRANSFER or an appropriate specialized data transfer state before performing a copy operation.

grCmdCopyImage

Copies multiple regions from one image to another.

```
GR_VOID grCmdCopyImage(
    GR_CMD_BUFFER cmdBuffer,
    GR_IMAGE srcImage,
    GR_IMAGE destImage,
    GR_UINT regionCount,
    const GR_IMAGE_COPY* pRegions);
```

Parameters

cmdBuffer

Command buffer handle.

srcImage

Source image handle.

destImage

Destination image handle.

regionCount

Number of regions for the copy operation.

pRegions

[in] Array of copy region descriptors. See `GR_IMAGE_COPY`.

Notes

The source and destination subresources are not allowed to be the same. Overlapping any of the source and destination subresources produces undefined copy results. Additionally, destination subresources cannot be present more than once per `grCmdCopyImage()` function call.

The source and destination formats do not have to match; appropriate format conversion is performed automatically if image and destination formats support conversion, which is indicated by the `GR_FORMAT_CONVERSION` format capability flag. Format conversions cannot be performed for compressed image formats. For resources with multiple aspects, each aspect format is determined according to the subresource. When either the source or destination image format does not have a `GR_FORMAT_CONVERSION` flag, the pixel size must match, and a raw image data copy is performed. For compressed images, the compression block size is used as a pixel size.

For compressed images, the image extents are specified in compression blocks.

The source and destination images must to be of the same type (1D, 2D, or 3D).

The MSAA source and destination images must have the same number of samples.

Both the source and destination images must be in the GR_IMAGE_STATE_DATA_TRANSFER or an appropriate specialized data transfer state before performing a copy operation.

grCmdCopyMemoryToImage

Copies data directly from a GPU memory object to an image.

```
GR_VOID grCmdCopyMemoryToImage(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY srcMem,
    GR_IMAGE destImage,
    GR_UINT regionCount,
    const GR_MEMORY_IMAGE_COPY* pRegions);
```

Parameters

cmdBuffer

Command buffer handle.

srcMem

Source memory object.

destImage

Destination image handle.

regionCount

Number of regions for the copy operation.

pRegions

[in] Array of copy region descriptors. See GR_MEMORY_IMAGE_COPY.

Notes

For compressed images, the image extents are specified in compression blocks.

The size of the data copied from memory is implicitly derived from the extents.

The destination memory offset has to be aligned to the smaller of the copied texel size or the 4-byte boundary. The destination subresources cannot be present more than once per grCmdCopyMemoryToImage function call.

The source memory regions need to be in the GR_MEMORY_STATE_DATA_TRANSFER or GR_MEMORY_STATE_DATA_TRANSFER_SOURCE state, and the destination image needs to be in the GR_IMAGE_STATE_DATA_TRANSFER or GR_IMAGE_STATE_DATA_TRANSFER_DESTINATION state before performing a copy operation.

grCmdCopyImageToMemory

Copies data from an image directly to a GPU memory object.

```
GR_VOID grCmdCopyImageToMemory(
    GR_CMD_BUFFER cmdBuffer,
    GR_IMAGE srcImage,
    GR_GPU_MEMORY destMem,
    GR_UINT regionCount,
    const GR_MEMORY_IMAGE_COPY* pRegions);
```

Parameters

cmdBuffer

Command buffer handle.

srcImage

Source image handle.

destMem

Destination memory object.

regionCount

Number of regions for the copy operation.

pRegions

[in] Array of copy region descriptors. See GR_MEMORY_IMAGE_COPY.

Notes

For compressed images, the image extents are specified in compression blocks.

The size of the data copied to memory is implicitly derived from the extents.

The destination memory offset has to be aligned to the smaller of the copied texel size or the 4-byte boundary.

The destination memory regions need to be in the GR_MEMORY_STATE_DATA_TRANSFER or GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION state, and the source image needs to be in the GR_IMAGE_STATE_DATA_TRANSFER or GR_IMAGE_STATE_DATA_TRANSFER_SOURCE state before performing a copy operation.

grCmdResolveImage

Resolves multiple rectangles from a multisampled resource to a single sampled-resource.

```
GR_VOID grCmdResolveImage(
    GR_CMD_BUFFER cmdBuffer,
    GR_IMAGE srcImage,
    GR_IMAGE destImage,
    GR_UINT regionCount,
    const GR_IMAGE_RESOLVE* pRegions);
```

Parameters

cmdBuffer

Command buffer handle.

srcImage

Source image handle.

destImage

Destination image handle.

regionCount

Number of regions for the resolve operation.

pRegions

[in] Array of resolve region descriptors. See `GR_IMAGE_RESOLVE`.

Notes

The source image has to be a 2D multisampled image and the destination must be a single sample image. The formats of the source and destination images should match.

For depth-stencil images the resolve is performed by copying the first sample from the target image to the destination image.

The destination subresources cannot be present more than once in an array of regions.

The source image must be in the `GR_IMAGE_STATE_RESOLVE_SOURCE` state and the destination image must be in the `GR_IMAGE_STATE_RESOLVE_DESTINATION` state before performing a resolve operation.

grCmdCloneImageData

Clones data of one image object to another while preserving the image state. The source and destination images must be created with identical creation parameters and have memory bound.

```
GR_VOID grCmdCloneImageData(
    GR_CMD_BUFFER cmdBuffer,
    GR_IMAGE srcImage,
    GR_ENUM srcImageState,
    GR_IMAGE destImage,
    GR_ENUM destImageState);
```

Parameters

cmdBuffer

Command buffer handle.

srcImage

Source image handle.

srcImageState

Source image state before cloning. See GR_IMAGE_STATE.

destImage

Destination image handle.

destImageState

Destination image state before cloning. See GR_IMAGE_STATE.

Notes

Both the source and destination image have to be created with GR_IMAGE_CREATE_CLONEABLE flag.

Both resources can be in any state before the cloning operation. After the cloning operation, the source image state is left intact and the destination image state becomes the same as the source.

The clone operation clones all subresources. All subresources of the source image have to be in the same state. All subresources of the destination image have to be in the same state. A mismatch of subresource state produces undefined results.

grCmdUpdateMemory

Directly updates a GPU memory object with a small amount of host data.

```
GR_VOID grCmdUpdateMemory(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY destMem,
    GR_GPU_SIZE destOffset,
    GR_GPU_SIZE dataSize,
    const GR_UINT32* pData);
```

Parameters

cmdBuffer

Command buffer handle.

destMem

Destination memory object handle.

destOffset

Offset into the destination memory object.

dataSize

Data size in bytes.

pData

[in] Data to write into the memory object.

Notes

The memory region needs to be in the `GR_MEMORY_STATE_DATA_TRANSFER` or `GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION` state before updating its data.

The GPU memory offset and data size must be 4-byte aligned.

The amount of data must be less than or equal to what is reported in the physical GPU properties – see GPU Identification and Initialization.

grCmdFillMemory

Fills a range of GPU memory object with provided 32-bit data.

```
GR_VOID grCmdFillMemory(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY destMem,
    GR_GPU_SIZE destOffset,
    GR_GPU_SIZE fillSize,
    GR_UINT32 data);
```

Parameters

cmdBuffer

Command buffer handle.

destMem

Destination memory object handle.

destOffset

Offset into the destination memory object.

fillSize

Fill memory range in bytes.

data

Value to fill the memory object with.

Notes

The memory region needs to be in the `GR_MEMORY_STATE_DATA_TRANSFER` or `GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION` state before updating its data.

The GPU memory offset and data size must be 4-byte aligned.

grCmdClearColorImage

Clears a color image to a color specified in floating point format.

```
GR_VOID grCmdClearColorTarget(
    GR_CMD_BUFFER cmdBuffer,
    GR_IMAGE image,
    const GR_FLOAT color[4],
    GR_UINT rangeCount,
    const GR_IMAGE_SUBRESOURCE_RANGE* pRanges);
```

Parameters

cmdBuffer

Command buffer handle.

image

Image handle.

color

Clear color in floating point format.

rangeCount

Number of subresource ranges to clear.

pRanges

[in] Array of subresource ranges. See `GR_IMAGE_SUBRESOURCE_RANGE`.

Notes

For images of `GR_NUM_FMT_UNORM` type, the color values must be in the [0..1] range. For images of `GR_NUM_FMT_SNORM` type, the color values must be in the [-1..1] range.

For images of `GR_NUM_FMT_UINT` type, the floating point color is rounded down to an integer value.

Specifying a clear value outside of the range representable by an image format produces undefined results.

All image subresources have to be in the `GR_IMAGE_STATE_CLEAR` state before performing a clear operation.

grCmdClearColorImageRaw

Clears a color image to a color specified with raw data bits.

```
GR_VOID grCmdClearColorImageRaw(
    GR_CMD_BUFFER cmdBuffer,
    GR_IMAGE image,
    const GR_UINT32 color[4],
    GR_UINT rangeCount,
    const GR_IMAGE_SUBRESOURCE_RANGE* pRanges);
```

Parameters

cmdBuffer

Command buffer handle.

image

Image handle.

color

Raw clear color value in integer format.

rangeCount

Number of subresource ranges to clear.

pRanges

[in] Array of subresource ranges. See `GR_IMAGE_SUBRESOURCE_RANGE`.

Notes

The lowest bits of the clear color (number of bits depending on format) are stored in the cleared image per channel.

All image subresources have to be in the `GR_IMAGE_STATE_CLEAR` state before performing a clear operation.

grCmdClearDepthStencil

Clears a depth-stencil image to the specified clear values.

```
GR_VOID grCmdClearDepthStencil(
    GR_CMD_BUFFER cmdBuffer,
    GR_IMAGE image,
    GR_FLOAT depth,
    GR_UINT8 stencil,
    GR_UINT rangeCount,
    const GR_IMAGE_SUBRESOURCE_RANGE* pRanges);
```

Parameters

cmdBuffer

Command buffer handle.

image

Image handle.

depth

Depth clear value.

stencil

Stencil clear values.

rangeCount

Number of subresource ranges to clear.

pRanges

[in] Array of subresource ranges. See GR_IMAGE_SUBRESOURCE_RANGE.

Notes

All image subresources have to be in the GR_IMAGE_STATE_CLEAR state before performing a clear operation.

grCmdSetEvent

Sets an event object from a command buffer when all previous work completes.

```
GR_VOID grCmdSetEvent(
    GR_CMD_BUFFER cmdBuffer,
    GR_EVENT event);
```

Parameters

cmdBuffer

Command buffer handle.

event

Event handle.

Notes

None.

grCmdResetEvent

Resets an event object from a command buffer when all previous work completes.

```
GR_VOID grCmdResetEvent(
    GR_CMD_BUFFER cmdBuffer,
    GR_EVENT event);
```

Parameters

cmdBuffer

Command buffer handle.

event

Event handle.

Notes

None.

grCmdMemoryAtomic

Performs a 32-bit or 64-bit memory atomic operation consistently with atomics in the shaders.

```
GR_VOID grCmdMemoryAtomic(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY destMem,
    GR_GPU_SIZE destOffset,
    GR_UINT64 srcData,
    GR_ENUM atomicOp);
```

Parameters

cmdBuffer

Command buffer handle.

destMem

Memory object.

destOffset

Byte offset into the destination memory object.

srcData

Source data to use for atomic operation.

atomicOp

Atomic operation type. See `GR_ATOMIC_OP`.

Notes

The data size (32-bits or 64-bits) is determined by the operation type. For 32-bit atomics only, the lower 32-bits of `srcData` is used.

The destination GPU memory offset must be 4-byte aligned for 32-bit atomics, and 8-byte aligned for 64-bit atomics.

The memory range must be in the `GR_MEMORY_STATE_QUEUE_ATOMIC` state before performing an atomic operation.

grCmdBeginQuery

Starts query operation for the given slot of a query pool.

```
GR_VOID grCmdBeginQuery(
    GR_CMD_BUFFER cmdBuffer,
    GR_QUERY_POOL queryPool,
    GR_UINT slot,
    GR_FLAGS flags);
```

Parameters

cmdBuffer

Command buffer handle.

queryPool

Query pool handle.

slot

Query pool slot to start query.

flags

Flags controlling query execution. See `GR_QUERY_CONTROL_FLAGS`.

Notes

The query slot must have been previously cleared with `grCmdResetQueryPool()` before starting the query operation.

grCmdEndQuery

Stops query operation for the given slot of a query pool.

```
GR_VOID grCmdEndQuery(
    GR_CMD_BUFFER cmdBuffer,
    GR_QUERY_POOL queryPool,
    GR_UINT slot);
```

Parameters

cmdBuffer

Command buffer handle.

queryPool

Query pool handle.

slot

Query pool slot to stop query.

Notes

Should only be called after `grCmdBeginQuery()` was issued on the query slot.

grCmdResetQueryPool

Resets a range of query slots in a query pool. A query slot must be reset each time before the query can be started to generate meaningful results.

```
GR_VOID grCmdResetQueryPool(
    GR_CMD_BUFFER cmdBuffer,
    GR_QUERY_POOL queryPool,
    GR_UINT startQuery,
    GR_UINT queryCount);
```

Parameters

cmdBuffer

Command buffer handle.

queryPool

Query pool handle.

startQuery

Fist query pool slot to reset.

queryCount

Number of query slots to reset.

Notes

None.

grCmdWriteTimestamp

Writes a top or bottom of pipe 64-bit timestamp to a memory location.

```
GR_VOID grCmdWriteTimestamp(
    GR_CMD_BUFFER cmdBuffer,
    GR_ENUM timestampType,
    GR_GPU_MEMORY destMem,
    GR_GPU_SIZE destOffset);
```

Parameters

cmdBuffer

Command buffer handle.

timestampType

Timestamp type. See `GR_TIMESTAMP_TYPE`.

destMem

Destination memory object.

destOffset

Byte offset in the memory object to the timestamp data.

Notes

The memory needs to be in the GR_MEMORY_STATE_WRITE_TIMESTAMP state before writing the timestamp.

The destination memory address must be 8-byte aligned.

grCmdInitAtomicCounters

Loads atomic counter with provided values.

```
GR_VOID grCmdInitAtomicCounters(
    GR_CMD_BUFFER cmdBuffer,
    GR_ENUM pipelineBindPoint,
    GR_UINT startCounter,
    GR_UINT counterCount,
    const GR_UINT32* pData);
```

Parameters

cmdBuffer

Command buffer handle.

pipelineBindPoint

Pipeline type to load atomic counters for. See GR_PIPELINE_BIND_POINT.

startCounter

First atomic counter slot to load.

counterCount

Number of atomic counter slots to load.

pData

[in] The counter data.

Notes

Each counter has a 32-bit value, each of which is consecutively loaded from provided system memory.

grCmdLoadAtomicCounters

Loads atomic counter values from a memory location.

```
GR_VOID grCmdLoadAtomicCounters(
    GR_CMD_BUFFER cmdBuffer,
    GR_ENUM pipelineBindPoint,
    GR_UINT startCounter,
    GR_UINT counterCount,
    GR_GPU_MEMORY srcMem,
    GR_GPU_SIZE srcOffset);
```

Parameters

cmdBuffer

Command buffer handle.

pipelineBindPoint

Pipeline type to load atomic counters for. See `GR_PIPELINE_BIND_POINT`.

startCounter

First atomic counter slot to load.

counterCount

Number of atomic counter slots to load.

srcMem

Source memory object.

srcOffset

Byte offset in the memory object to the beginning of the counter data.

Notes

The memory must be in the `GR_MEMORY_STATE_DATA_TRANSFER` state before loading atomic counter data.

Each counter has a 32-bit value, each of which is consecutively loaded from memory.

The source memory offset must be 4-byte aligned.

grCmdSaveAtomicCounters

Saves current atomic counter values to a memory location.

```
GR_VOID grCmdSaveAtomicCounters(
    GR_CMD_BUFFER cmdBuffer,
    GR_ENUM pipelineBindPoint,
    GR_UINT startCounter,
    GR_UINT counterCount,
    GR_GPU_MEMORY destMem,
    GR_GPU_SIZE destOffset);
```

Parameters

cmdBuffer

Command buffer handle.

pipelineBindPoint

Pipeline type to save atomic counters for. See `GR_PIPELINE_BIND_POINT`.

startCounter

First atomic counter slot to save.

counterCount

Number of atomic counter slots to save.

destMem

Destination memory object.

destOffset

Byte offset in the memory object to the beginning of the counter data.

Notes

The memory must be in the `GR_MEMORY_STATE_DATA_TRANSFER` state before saving atomic counter data.

Each counter has a 32-bit value, each of which is consecutively stored to GPU memory.

The destination memory offset must be 4-byte aligned.

ENUMERATIONS

GR_ATOMIC_OP

Defines a memory atomic operation that can be performed from command buffers.

```
typedef enum _GR_ATOMIC_OP
{
    GR_ATOMIC_ADD_INT32     = 0x2d00,
    GR_ATOMIC_SUB_INT32     = 0x2d01,
    GR_ATOMIC_MIN_UINT32    = 0x2d02,
    GR_ATOMIC_MAX_UINT32    = 0x2d03,
    GR_ATOMIC_MIN_SINT32    = 0x2d04,
    GR_ATOMIC_MAX_SINT32    = 0x2d05,
    GR_ATOMIC_AND_INT32     = 0x2d06,
    GR_ATOMIC_OR_INT32      = 0x2d07,
    GR_ATOMIC_XOR_INT32     = 0x2d08,
    GR_ATOMIC_INC_UINT32    = 0x2d09,
    GR_ATOMIC_DEC_UINT32    = 0x2d0a,
    GR_ATOMIC_ADD_INT64     = 0x2d0b,
    GR_ATOMIC_SUB_INT64     = 0x2d0c,
    GR_ATOMIC_MIN_UINT64    = 0x2d0d,
    GR_ATOMIC_MAX_UINT64    = 0x2d0e,
    GR_ATOMIC_MIN_SINT64    = 0x2d0f,
    GR_ATOMIC_MAX_SINT64    = 0x2d10,
    GR_ATOMIC_AND_INT64     = 0x2d11,
    GR_ATOMIC_OR_INT64      = 0x2d12,
    GR_ATOMIC_XOR_INT64     = 0x2d13,
    GR_ATOMIC_INC_UINT64    = 0x2d14,
    GR_ATOMIC_DEC_UINT64    = 0x2d15,
} GR_ATOMIC_OP;
```

Values

GR_ATOMIC_ADD_INT32

destData = destData + srcData

GR_ATOMIC_SUB_INT32

destData = destData - srcData

GR_ATOMIC_MIN_UINT32

destData = (srcData < destData) ? srcData : destData, unsigned

GR_ATOMIC_MAX_UINT32

destData = (srcData > destData) ? srcData : destData, unsigned

GR_ATOMIC_MIN_SINT32

destData = (srcData < destData) ? srcData : destData, signed

GR_ATOMIC_MAX_SINT32

destData = (srcData > destData) ? srcData : destData, signed

GR_ATOMIC_AND_INT32

destData = srcData & destData

GR_ATOMIC_OR_INT32

destData = srcData | destData

GR_ATOMIC_XOR_INT32

destData = srcData ^ destData

GR_ATOMIC_INC_UINT32

destData = (destData >= srcData) ? 0 : (destData + 1), unsigned

GR_ATOMIC_DEC_UINT32

destData = ((destData == 0) || (destData > srcData)) ? srcData : (destData - 1), unsigned

GR_ATOMIC_ADD_INT64

destData = destData + srcData

GR_ATOMIC_SUB_INT64

destData = destData - srcData

GR_ATOMIC_MIN_UINT64

destData = (srcData < destData) ? srcData : destData, unsigned

GR_ATOMIC_MAX_UINT64

destData = (srcData > destData) ? srcData : destData, unsigned

GR_ATOMIC_MIN_SINT64

destData = (srcData < destData) ? srcData : destData, signed

GR_ATOMIC_MAX_SINT64

destData = (srcData > destData) ? srcData : destData, signed

GR_ATOMIC_AND_INT64

destData = srcData & destData

GR_ATOMIC_OR_INT64

destData = srcData | destData

GR_ATOMIC_XOR_INT64

destData = srcData ^ destData

GR_ATOMIC_INC_UINT64

destData = (destData >= srcData) ? 0 : (destData + 1) , unsigned

GR_ATOMIC_DEC_UINT64

destData = ((destData == 0) || (destData > srcData)) ? srcData : (destData - 1) , unsigned

GR_BORDER_COLOR_TYPE

The border color type specifies what color is fetched in the GR_TEX_ADDRESS_CLAMP_BORDER texture addressing mode for coordinates outside of the range [0..1].

```
typedef enum _GR_BORDER_COLOR_TYPE
{
    GR_BORDER_COLOR_WHITE             = 0x1c00,
    GR_BORDER_COLOR_TRANSPARENT_BLACK = 0x1c01,
    GR_BORDER_COLOR_OPAQUE_BLACK      = 0x1c02,
} GR_BORDER_COLOR_TYPE;
```

Values

GR_BORDER_COLOR_WHITE

White (1.0, 1.0, 1.0, 1.0)

GR_BORDER_COLOR_TRANSPARENT_BLACK

Transparent black (0.0, 0.0, 0.0, 0.0)

GR_BORDER_COLOR_OPAQUE_BLACK

Opaque black (0.0, 0.0, 0.0, 1.0)

GR_BLEND

Blend factors define how source and destination parts of the blend equation are computed.

```
typedef enum _GR_BLEND
{
    GR_BLEND_ZERO                          = 0x2900,
    GR_BLEND_ONE                           = 0x2901,
    GR_BLEND_SRC_COLOR                     = 0x2902,
    GR_BLEND_ONE_MINUS_SRC_COLOR           = 0x2903,
    GR_BLEND_DEST_COLOR                    = 0x2904,
    GR_BLEND_ONE_MINUS_DEST_COLOR          = 0x2905,
    GR_BLEND_SRC_ALPHA                     = 0x2906,
    GR_BLEND_ONE_MINUS_SRC_ALPHA           = 0x2907,
    GR_BLEND_DEST_ALPHA                    = 0x2908,
    GR_BLEND_ONE_MINUS_DEST_ALPHA          = 0x2909,
    GR_BLEND_CONSTANT_COLOR                = 0x290a,
    GR_BLEND_ONE_MINUS_CONSTANT_COLOR      = 0x290b,
    GR_BLEND_CONSTANT_ALPHA                = 0x290c,
    GR_BLEND_ONE_MINUS_CONSTANT_ALPHA      = 0x290d,
    GR_BLEND_SRC_ALPHA_SATURATE            = 0x290e,
    GR_BLEND_SRC1_COLOR                    = 0x290f,
    GR_BLEND_ONE_MINUS_SRC1_COLOR          = 0x2910,
    GR_BLEND_SRC1_ALPHA                    = 0x2911,
    GR_BLEND_ONE_MINUS_SRC1_ALPHA          = 0x2912,
} GR_BLEND;
```

Values

GR_BLEND_ZERO

Blend factor is set to black color (0,0,0,0).

GR_BLEND_ONE

Blend factor is set to white color (1,1,1,1).

GR_BLEND_SRC_COLOR

Blend factor is set to the source color coming from a pixel shader (RGB).

GR_BLEND_ONE_MINUS_SRC_COLOR

Blend factor is set to the inverted source color coming from a pixel shader (1-RGB).

GR_BLEND_DEST_COLOR

Blend factor is set to the destination color coming from a target image (RGB).

GR_BLEND_ONE_MINUS_DEST_COLOR

Blend factor is set to the inverted destination color coming from a target image (RGB).

GR_BLEND_SRC_ALPHA

Blend factor is set to the source alpha coming from a pixel shader (A).

GR_BLEND_ONE_MINUS_SRC_ALPHA

Blend factor is set to the inverted source alpha coming from a pixel shader (1-A).

GR_BLEND_DEST_ALPHA

Blend factor is set to the destination alpha coming from a target image (A).

GR_BLEND_ONE_MINUS_DEST_ALPHA

Blend factor is set to the inverted destination alpha coming from a target image (1-A).

GR_BLEND_CONSTANT_COLOR

Blend factor is set to the constant color specified in blend state (blendConstRGB).

GR_BLEND_ONE_MINUS_CONSTANT_COLOR

Blend factor is set to the inverted constant color specified in blend state (1-blendConstRGB).

GR_BLEND_CONSTANT_ALPHA

Blend factor is set to the constant alpha specified in blend state (blendConstA).

GR_BLEND_ONE_MINUS_CONSTANT_ALPHA

Blend factor is set to the inverted constant alpha specified in blend state (1-blendConstA).

GR_BLEND_SRC_ALPHA_SATURATE

Blend factor is set to the source alpha coming from a pixel shader (A) clamped to an inverted destination alpha coming from a target image.

GR_BLEND_SRC1_COLOR

Blend factor is set to the second source color coming from a pixel shader (RGB1). Used for dual source blend mode.

GR_BLEND_ONE_MINUS_SRC1_COLOR

Blend factor is set to the inverted second source color coming from a pixel shader (1-RGB1). Used for dual source blend mode.

GR_BLEND_SRC1_ALPHA

Blend factor is set to the second source alpha coming from a pixel shader (A1). Used for dual source blend mode.

GR_BLEND_ONE_MINUS_SRC1_ALPHA

Blend factor is set to the inverted second source alpha coming from a pixel shader (1-A1). Used for dual source blend mode.

GR_BLEND_FUNC

Defines blend function in a blend equation.

```
typedef enum _GR_BLEND_FUNC
{
    GR_BLEND_FUNC_ADD               = 0x2a00,
    GR_BLEND_FUNC_SUBTRACT          = 0x2a01,
    GR_BLEND_FUNC_REVERSE_SUBTRACT  = 0x2a02,
    GR_BLEND_FUNC_MIN               = 0x2a03,
    GR_BLEND_FUNC_MAX               = 0x2a04,
} GR_BLEND_FUNC;
```

Values

GR_BLEND_FUNC_ADD

Add source and destination parts of a blend equation.

GR_BLEND_FUNC_SUBTRACT

Subtract destination part of a blend equation from source.

GR_BLEND_FUNC_REVERSE_SUBTRACT

Subtract source part of a blend equation from destination.

GR_BLEND_FUNC_MIN

Compute minimum of source and destination parts of a blend equation.

GR_BLEND_FUNC_MAX

Compute maximum of source and destination parts of a blend equation.

GR_CHANNEL_FORMAT

Defines an image and memory view channel format.

```
typedef enum _GR_CHANNEL_FORMAT
{
    GR_CH_FMT_UNDEFINED     = 0,
    GR_CH_FMT_R4G4          = 1,
    GR_CH_FMT_R4G4B4A4      = 2,
    GR_CH_FMT_R5G6B5        = 3,
    GR_CH_FMT_B5G6R5        = 4,
    GR_CH_FMT_R5G5B5A1      = 5,
    GR_CH_FMT_R8            = 6,
    GR_CH_FMT_R8G8          = 7,
    GR_CH_FMT_R8G8B8A8      = 8,
    GR_CH_FMT_B8G8R8A8      = 9,
    GR_CH_FMT_R10G11B11     = 10,
    GR_CH_FMT_R11G11B10     = 11,
    GR_CH_FMT_R10G10B10A2   = 12,
    GR_CH_FMT_R16           = 13,
    GR_CH_FMT_R16G16        = 14,
    GR_CH_FMT_R16G16B16A16  = 15,
    GR_CH_FMT_R32           = 16,
    GR_CH_FMT_R32G32        = 17,
    GR_CH_FMT_R32G32B32     = 18,
    GR_CH_FMT_R32G32B32A32  = 19,
    GR_CH_FMT_R16G8         = 20,
    GR_CH_FMT_R32G8         = 21,
    GR_CH_FMT_R9G9B9E5      = 22,
    GR_CH_FMT_BC1           = 23,
    GR_CH_FMT_BC2           = 24,
    GR_CH_FMT_BC3           = 25,
    GR_CH_FMT_BC4           = 26,
    GR_CH_FMT_BC5           = 27,
    GR_CH_FMT_BC6U          = 28,
    GR_CH_FMT_BC6S          = 29,
    GR_CH_FMT_BC7           = 30,
} GR_CHANNEL_FORMAT;
```

Values

GR_CH_FMT_UNDEFINED

An undefined channel format.

GR_CH_FMT_R4G4

A channel format of R4G4.

GR_CH_FMT_R4G4B4A4

A channel format of R4G4B4A4.

GR_CH_FMT_R5G6B5

A channel format of R5G6B5.

GR_CH_FMT_B5G6R5

A channel format of B5G6R5.

GR_CH_FMT_R5G5B5A1.

A channel format of R5G5B5A1.

GR_CH_FMT_R8

A channel format of R5G5B5A1.

GR_CH_FMT_R8G8

A channel format of R8G8.

GR_CH_FMT_R8G8B8A8

A channel format of R8G8B8A8.

GR_CH_FMT_B8G8R8A8

A channel format of B8G8R8A8.

GR_CH_FMT_R10G11B11

A channel format of R10G11B11.

GR_CH_FMT_R11G11B10

A channel format of R11G11B10.

GR_CH_FMT_R10G10B10A2

A channel format of R10G10B10A2.

GR_CH_FMT_R16

A channel format of R16.

GR_CH_FMT_R16G16

A channel format of R16G16.

GR_CH_FMT_R16G16B16A16

A channel format of R16G16B16A16.

GR_CH_FMT_R32

A channel format of R32.

GR_CH_FMT_R32G32

A channel format of R32G32.

GR_CH_FMT_R32G32B32

A channel format of R32G32B32.

GR_CH_FMT_R32G32B32A32

A channel format of R32G32B32A32.

GR_CH_FMT_R16G8

A channel format of R16G8.

GR_CH_FMT_R32G8

A channel format of R32G8.

GR_CH_FMT_R9G9B9E5

A channel format of R9G9B9E5.

GR_CH_FMT_BC1

A channel format of BC1.

GR_CH_FMT_BC2

A channel format of BC2.

GR_CH_FMT_BC3

A channel format of BC3.

GR_CH_FMT_BC4

A channel format of BC4.

GR_CH_FMT_BC5

A channel format of BC5.

GR_CH_FMT_BC6U

A channel format of BC6U.

GR_CH_FMT_BC6S

A channel format of BC6S.

GR_CH_FMT_BC7

A channel format of BC7.

GR_CHANNEL_SWIZZLE

Channel swizzle defines remapping of texture channels in image views.

```
typedef enum _GR_CHANNEL_SWIZZLE
{
    GR_CHANNEL_SWIZZLE_ZERO = 0x1800,
    GR_CHANNEL_SWIZZLE_ONE  = 0x1801,
    GR_CHANNEL_SWIZZLE_R    = 0x1802,
    GR_CHANNEL_SWIZZLE_G    = 0x1803,
    GR_CHANNEL_SWIZZLE_B    = 0x1804,
    GR_CHANNEL_SWIZZLE_A    = 0x1805,
} GR_CHANNEL_SWIZZLE;
```

Values

GR_CHANNEL_SWIZZLE_ZERO

Image fetch returns zero value.

GR_CHANNEL_SWIZZLE_ONE

Image fetch returns a value of one.

GR_CHANNEL_SWIZZLE_R

Maps image data to red channel.

GR_CHANNEL_SWIZZLE_G

Maps image data to green channel.

GR_CHANNEL_SWIZZLE_B

Maps image data to blue channel.

GR_CHANNEL_SWIZZLE_A

Maps image data to alpha channel.

GR_COMPARE_FUNC

A comparison function determines how a condition that compares two values is evaluated. For depth and stencil comparison, the first value comes from source data and the second value comes from destination data.

```
typedef enum _GR_COMPARE_FUNC
{
    GR_COMPARE_NEVER            = 0x2500,
    GR_COMPARE_LESS             = 0x2501,
    GR_COMPARE_EQUAL            = 0x2502,
    GR_COMPARE_LESS_EQUAL       = 0x2503,
    GR_COMPARE_GREATER          = 0x2504,
    GR_COMPARE_NOT_EQUAL        = 0x2505,
    GR_COMPARE_GREATER_EQUAL    = 0x2506,
    GR_COMPARE_ALWAYS           = 0x2507,
} GR_COMPARE_FUNC;
```

Values

GR_COMPARE_NEVER

Function never passes the comparison.

GR_COMPARE_LESS

The comparison passes if the first value is less than the second value.

GR_COMPARE_EQUAL

The comparison passes if the first value is equal to the second value.

GR_COMPARE_LESS_EQUAL

The comparison passes if the first value is less than or equal the second value.

GR_COMPARE_GREATER

The comparison passes if the first value is greater than the second value.

GR_COMPARE_NOT_EQUAL

The comparison passes if the first value is not equal to the second value.

GR_COMPARE_GREATER_EQUAL

The comparison passes if the first value is greater than or equal to the second value.

GR_COMPARE_ALWAYS

Function always passes the comparison.

GR_CULL_MODE

Defines triangle facing direction used for primitive culling.

```
typedef enum _GR_CULL_MODE
{
    GR_CULL_NONE        = 0x2700,
    GR_CULL_FRONT       = 0x2701,
    GR_CULL_BACK        = 0x2702,
} GR_CULL_MODE;
```

Values

GR_CULL_NONE

Always draw geometry.

GR_CULL_FRONT

Cull front-facing triangles.

GR_CULL_BACK

Cull back-facing triangles.

GR_DESCRIPTOR_SET_SLOT_TYPE

Defines a type of object expected by a shader in a descriptor slot.

```
typedef enum _GR_DESCRIPTOR_SET_SLOT_TYPE
{
    GR_SLOT_UNUSED              = 0x1900,
    GR_SLOT_SHADER_RESOURCE     = 0x1901,
    GR_SLOT_SHADER_UAV          = 0x1902,
    GR_SLOT_SHADER_SAMPLER      = 0x1903,
    GR_SLOT_NEXT_DESCRIPTOR_SET = 0x1904,
} GR_DESCRIPTOR_SET_SLOT_TYPE;
```

Values

GR_SLOT_UNUSED

The descriptor set slot is not used by the shader.

GR_SLOT_SHADER_RESOURCE

The descriptor set slot maps to a “t#” shader resource.

GR_SLOT_SHADER_UAV

The descriptor set slot maps to a “u#” shader UAV resource.

GR_SLOT_SHADER_SAMPLER

The descriptor set slot maps to a sampler.

GR_SLOT_NEXT_DESCRIPTOR_SET

The descriptor set stores a pointer to the next level of a nested descriptor set.

GR_FACE_ORIENTATION

Defines front-facing triangle orientation to be used for culling.

```
typedef enum _GR_FACE_ORIENTATION
{
    GR_FRONT_FACE_CCW   = 0x2800,
    GR_FRONT_FACE_CW    = 0x2801,
} GR_FACE_ORIENTATION;
```

Values

GR_FRONT_FACE_CCW

A triangle is front-facing if vertices are oriented counter-clockwise.

GR_FRONT_FACE_CW

A triangle is front-facing if vertices are oriented clockwise.

GR_FILL_MODE

Defines triangle rendering mode.

```
typedef enum _GR_FILL_MODE
{
    GR_FILL_SOLID       = 0x2600,
    GR_FILL_WIREFRAME   = 0x2601,
} GR_FILL_MODE;
```

Values

GR_FILL_SOLID

Draws filled triangles.

GR_FILL_WIREFRAME

Draws triangles as wire-frame.

GR_HEAP_MEMORY_TYPE

Defines the type of memory heap.

```
typedef enum _GR_HEAP_MEMORY_TYPE
{
    GR_HEAP_MEMORY_OTHER    = 0x2f00,
    GR_HEAP_MEMORY_LOCAL    = 0x2f01,
    GR_HEAP_MEMORY_REMOTE   = 0x2f02,
    GR_HEAP_MEMORY_EMBEDDED = 0x2f03,
} GR_HEAP_MEMORY_TYPE;
```

Values

GR_HEAP_MEMORY_OTHER

Heap memory type that does not belong to any other category.

GR_HEAP_MEMORY_LOCAL

Heap represents local video memory.

GR_HEAP_MEMORY_REMOTE

Heap represents remote (non-local) video memory.

GR_HEAP_MEMORY_EMBEDDED

Heap represents memory physically connected to the GPU (e.g., on-chip memory).

GR_IMAGE_ASPECT

Image aspect defines what components of the image object are referenced: color, depth, or stencil.

```
typedef enum _GR_IMAGE_ASPECT
{
    GR_IMAGE_ASPECT_COLOR   = 0x1700,
    GR_IMAGE_ASPECT_DEPTH   = 0x1701,
    GR_IMAGE_ASPECT_STENCIL = 0x1702,
} GR_IMAGE_ASPECT;
```

Values

GR_IMAGE_ASPECT_COLOR

Color components of the image.

GR_IMAGE_ASPECT_DEPTH

Depth component of the image.

GR_IMAGE_ASPECT_STENCIL

Stencil component of the image.

GR_IMAGE_STATE

The image state defines how the GPU expects to use a range of image subresources.

```
typedef enum _GR_IMAGE_STATE
{
    GR_IMAGE_STATE_DATA_TRANSFER                = 0x1300,
    GR_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY    = 0x1301,
    GR_IMAGE_STATE_GRAPHICS_SHADER_WRITE_ONLY   = 0x1302,
    GR_IMAGE_STATE_GRAPHICS_SHADER_READ_WRITE   = 0x1303,
    GR_IMAGE_STATE_COMPUTE_SHADER_READ_ONLY     = 0x1304,
    GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY    = 0x1305,
    GR_IMAGE_STATE_COMPUTE_SHADER_READ_WRITE    = 0x1306,
    GR_IMAGE_STATE_MULTI_SHADER_READ_ONLY       = 0x1307,
    GR_IMAGE_STATE_TARGET_AND_SHADER_READ_ONLY  = 0x1308,
    GR_IMAGE_STATE_UNINITIALIZED                = 0x1309,
    GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL = 0x130a,
    GR_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL = 0x130b,
    GR_IMAGE_STATE_CLEAR                        = 0x130c,
    GR_IMAGE_STATE_RESOLVE_SOURCE               = 0x130d,
    GR_IMAGE_STATE_RESOLVE_DESTINATION          = 0x130e,
    GR_IMAGE_STATE_DISCARD                      = 0x131f,
    GR_IMAGE_STATE_DATA_TRANSFER_SOURCE         = 0x1310,
    GR_IMAGE_STATE_DATA_TRANSFER_DESTINATION    = 0x1311,
} GR_IMAGE_STATE;
```

Values

GR_IMAGE_STATE_DATA_TRANSFER

Range of image subresources is accessible by the CPU for data transfer or can be copied by the GPU.

GR_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY

Range of image subresources can be used as a read-only image view in the graphics pipeline.

GR_IMAGE_STATE_GRAPHICS_SHADER_WRITE_ONLY

Range of image subresources can be used as a write-only image view in the graphics pipeline.

GR_IMAGE_STATE_GRAPHICS_SHADER_READ_WRITE

Range of image subresources can be used as a read or write image view in the graphics pipeline.

GR_IMAGE_STATE_COMPUTE_SHADER_READ_ONLY

Range of image subresources can be used as a read-only image view in the compute pipeline.

GR_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY

Range of image subresources can be used as a write-only image view in the compute pipeline.

GR_IMAGE_STATE_COMPUTE_SHADER_READ_WRITE

Range of image subresources can be used as a read or write image view in the graphics pipeline.

GR_IMAGE_STATE_MULTI_SHADER_READ_ONLY

Range of image subresources can be simultaneously used as a read-only image view in both the graphics and compute pipelines.

GR_IMAGE_STATE_TARGET_AND_SHADER_READ_ONLY

Range of image subresources can be simultaneously used as read-only depth or stencil target and as a read-only image view in the graphics pipeline.

GR_IMAGE_STATE_UNINITIALIZED

Range of image subresources in depth-stencil or color target images is assumed to be in an undefined state. `GR_IMAGE_STATE_UNINITIALIZED` is the default state for all target image subresources after binding the target image to a new memory location. The state cannot be used for any operation.

GR_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL

Range of image subresources is intended to be used as a color or depth-stencil target. The image is optimized for rendering.

GR_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL

Range of image subresources is intended to be used as a color or depth-stencil target. The image is optimized for shader access.

GR_IMAGE_STATE_CLEAR

Range of image subresources can be used for image clears.

GR_IMAGE_STATE_RESOLVE_SOURCE

Range of image subresources can be used as a source for resolve operation.

GR_IMAGE_STATE_RESOLVE_DESTINATION

Range of image subresources can be used as a destination for resolve operation.

GR_IMAGE_STATE_DISCARD

Range of image subresources is in invalid state until they are transitioned to a valid state.

GR_IMAGE_STATE_DATA_TRANSFER_SOURCE

Range of image subresources can be used as a source for the GPU copies.

GR_IMAGE_STATE_DATA_TRANSFER_DESTINATION

Range of image subresources can be used as a destination for the GPU copies.

GR_IMAGE_TILING

Image tiling defines internal texel layout in memory.

```
typedef enum _GR_IMAGE_TILING
{
    GR_LINEAR_TILING    = 0x1500,
    GR_OPTIMAL_TILING   = 0x1501,
} GR_IMAGE_TILING;
```

Values

GR_LINEAR_TILING

Images with linear tiling are stored linearly in memory with device specific pitch.

GR_OPTIMAL_TILING

Images with optimal tiling have device-optimal texel layout in memory.

GR_IMAGE_TYPE

Image type defines image dimensionality and organization of subresources.

```
typedef enum _GR_IMAGE_TYPE
{
    GR_IMAGE_1D          = 0x1400,
    GR_IMAGE_2D          = 0x1401,
    GR_IMAGE_3D          = 0x1402,
} GR_IMAGE_TYPE;
```

Values

GR_IMAGE_1D

The image is a 1D texture or 1D texture array.

GR_IMAGE_2D

The image is a 2D texture or 2D texture array.

GR_IMAGE_3D

The image is a 3D texture.

GR_IMAGE_VIEW_TYPE

Defines image view type for shader image access.

```
typedef enum _GR_IMAGE_VIEW_TYPE
{
    GR_IMAGE_VIEW_1D        = 0x1600,
    GR_IMAGE_VIEW_2D        = 0x1601,
    GR_IMAGE_VIEW_3D        = 0x1602,
    GR_IMAGE_VIEW_CUBE      = 0x1603,
} GR_IMAGE_VIEW_TYPE;
```

Values

GR_IMAGE_VIEW_1D

The image view is a 1D texture or 1D texture array.

GR_IMAGE_VIEW_2D

The image view is a 2D texture or 2D texture array.

GR_IMAGE_VIEW_3D

The image view is a 3D texture.

GR_IMAGE_VIEW_CUBE

The image view is a cubemap texture or a cubemap texture array.

GR_INDEX_TYPE

Index type defines size of the index elements.

```
typedef enum _GR_INDEX_TYPE
{
    GR_INDEX_16          = 0x2100,
    GR_INDEX_32          = 0x2101,
} GR_INDEX_TYPE;
```

Values

GR_INDEX_16

The index data are 16-bits per index.

GR_INDEX_32

The index data are 32-bits per index.

GR_INFO_TYPE

Defines types of information that can be retrieved from different objects.

```
typedef enum _GR_INFO_TYPE
{
    GR_INFO_TYPE_PHYSICAL_GPU_PROPERTIES            = 0x6100,
    GR_INFO_TYPE_PHYSICAL_GPU_PERFORMANCE           = 0x6101,
    GR_INFO_TYPE_PHYSICAL_GPU_QUEUE_PROPERTIES      = 0x6102,
    GR_INFO_TYPE_PHYSICAL_GPU_MEMORY_PROPERTIES     = 0x6103,
    GR_INFO_TYPE_PHYSICAL_GPU_IMAGE_PROPERTIES      = 0x6104,
    GR_INFO_TYPE_MEMORY_HEAP_PROPERTIES             = 0x6200,
    GR_INFO_TYPE_FORMAT_PROPERTIES                  = 0x6300,
    GR_INFO_TYPE_SUBRESOURCE_LAYOUT                 = 0x6400,
    GR_INFO_TYPE_MEMORY_REQUIREMENTS                = 0x6800,
    GR_INFO_TYPE_PARENT_DEVICE                      = 0x6801,
    GR_INFO_TYPE_PARENT_PHYSICAL_GPU                = 0x6802,
} GR_INFO_TYPE;
```

Values

GR_INFO_TYPE_PHYSICAL_GPU_PROPERTIES

Retrieves physical GPU information with `grGetGpuInfo()`.

GR_INFO_TYPE_PHYSICAL_GPU_PERFORMANCE

Retrieves physical GPU performance information with `grGetGpuInfo()`.

GR_INFO_TYPE_PHYSICAL_GPU_QUEUE_PROPERTIES

Retrieves information about all queues available in a physical GPU with `grGetGpuInfo()`.

GR_INFO_TYPE_PHYSICAL_GPU_MEMORY_PROPERTIES

Retrieves information about memory management capabilities for a physical GPU with `grGetGpuInfo()`.

GR_INFO_TYPE_PHYSICAL_GPU_IMAGE_PROPERTIES

Retrieves information about image capabilities for a physical GPU with `grGetGpuInfo()`.

GR_INFO_TYPE_MEMORY_HEAP_PROPERTIES

Retrieves GPU memory heap information with `grGetMemoryHeapInfo()`.

GR_INFO_TYPE_FORMAT_PROPERTIES

Retrieves information on format properties with `grGetFormatInfo()`.

GR_INFO_TYPE_SUBRESOURCE_LAYOUT

Retrieves information about image subresource layout with `grGetImageSubresourceInfo()`.

GR_INFO_TYPE_MEMORY_REQUIREMENTS

Retrieves information about object GPU memory requirements with `grGetObjectInfo()`. Valid for all object types that can have memory requirements.

GR_INFO_TYPE_PARENT_DEVICE

Retrieves parent device handle for API objects with `grGetObjectInfo()`.

GR_INFO_TYPE_PARENT_PHYSICAL_GPU

Retrieves a parent physical GPU handle for the Mantle device with `grGetObjectInfo()`.

GR_LOGIC_OP

Defines a logical operation applied between the color coming from pixel shader and the value in the target image.

```
typedef enum _GR_LOGIC_OP
{
    GR_LOGIC_OP_COPY              = 0x2c00,
    GR_LOGIC_OP_CLEAR             = 0x2c01,
    GR_LOGIC_OP_AND               = 0x2c02,
    GR_LOGIC_OP_AND_REVERSE       = 0x2c03,
    GR_LOGIC_OP_AND_INVERTED      = 0x2c04,
    GR_LOGIC_OP_NOOP              = 0x2c05,
    GR_LOGIC_OP_XOR               = 0x2c06,
    GR_LOGIC_OP_OR                = 0x2c07,
    GR_LOGIC_OP_NOR               = 0x2c08,
    GR_LOGIC_OP_EQUIV             = 0x2c09,
    GR_LOGIC_OP_INVERT            = 0x2c0a,
    GR_LOGIC_OP_OR_REVERSE        = 0x2c0b,
    GR_LOGIC_OP_COPY_INVERTED     = 0x2c0c,
    GR_LOGIC_OP_OR_INVERTED       = 0x2c0d,
    GR_LOGIC_OP_NAND              = 0x2c0e,
    GR_LOGIC_OP_SET               = 0x2c0f,
} GR_LOGIC_OP;
```

Values

GR_LOGIC_OP_COPY

Writes the value coming from pixel shader. `GR_LOGIC_OP_COPY` effectively disables logic operations.

GR_LOGIC_OP_CLEAR

Writes the zero value.

GR_LOGIC_OP_AND

Performs a logical AND between the value coming from pixel shader and the destination value.

GR_LOGIC_OP_AND_REVERSE

Performs a logical AND between the value coming from pixel shader and the inverse of the destination value.

GR_LOGIC_OP_AND_INVERTED

Performs a logical AND between the inverse of value coming from pixel shader and the destination value.

GR_LOGIC_OP_NOOP

Preserves the original target value.

GR_LOGIC_OP_XOR

Performs a logical XOR between the value coming from pixel shader and the destination value.

GR_LOGIC_OP_OR

Performs a logical OR between the value coming from pixel shader and the destination value.

GR_LOGIC_OP_NOR

Performs a logical NOR between the value coming from pixel shader and the destination value.

GR_LOGIC_OP_EQUIV

Performs an equivalency test between the value coming from pixel shader and the destination value.

GR_LOGIC_OP_INVERT

Writes the inverted destination value.

GR_LOGIC_OP_OR_REVERSE

Performs a logical OR between the value coming from pixel shader and the inverse of the destination value.

GR_LOGIC_OP_COPY_INVERTED

Writes the inverted value coming from pixel shader.

GR_LOGIC_OP_OR_INVERTED

Performs a logical OR between the inverse of value coming from pixel shader and the destination value.

GR_LOGIC_OP_NAND

Performs a logical AND between the value coming from pixel shader and the destination value.

GR_LOGIC_OP_SET

Writes a value with all bits set to 1.

GR_MEMORY_PRIORITY

GPU memory object priority that provides a hint to the GPU memory manager regarding how hard it should try to keep allocation in a preferred heap.

```
typedef enum _GR_MEMORY_PRIORITY
{
    GR_MEMORY_PRIORITY_NORMAL    = 0x1100,
    GR_MEMORY_PRIORITY_HIGH      = 0x1101,
    GR_MEMORY_PRIORITY_LOW       = 0x1102,
    GR_MEMORY_PRIORITY_UNUSED    = 0x1103,
    GR_MEMORY_PRIORITY_VERY_HIGH = 0x1104,
    GR_MEMORY_PRIORITY_VERY_LOW  = 0x1105,
} GR_MEMORY_PRIORITY;
```

Values

GR_MEMORY_PRIORITY_NORMAL

Normal GPU memory object priority.

GR_MEMORY_PRIORITY_HIGH

High GPU memory object priority. Should be used for storing performance critical resources, such as render targets, depth buffers, and write accessible images.

GR_MEMORY_PRIORITY_LOW

Low GPU memory object priority. Should be used for infrequently accessed resources that generally do not require a lot of memory bandwidth.

GR_MEMORY_PRIORITY_UNUSED

GPU memory priority for marking memory objects that are not a part of the working set. Should only be set for memory allocations that do not contain any used resources.

GR_MEMORY_PRIORITY_VERY_HIGH

Highest GPU memory object priority. Should be used for storing performance critical resources, such as high-priority render targets, depth buffers, and write accessible images.

GR_MEMORY_PRIORITY_VERY_LOW

Lowest GPU memory object priority. Should be used for lowest priority infrequently accessed resources that generally do not require a lot of memory bandwidth.

GR_MEMORY_STATE

The memory state defines how the GPU expects to use a range of memory.

```
typedef enum _GR_MEMORY_STATE
{
    GR_MEMORY_STATE_DATA_TRANSFER               = 0x1200,
    GR_MEMORY_STATE_GRAPHICS_SHADER_READ_ONLY   = 0x1201,
    GR_MEMORY_STATE_GRAPHICS_SHADER_WRITE_ONLY  = 0x1202,
    GR_MEMORY_STATE_GRAPHICS_SHADER_READ_WRITE  = 0x1203,
    GR_MEMORY_STATE_COMPUTE_SHADER_READ_ONLY    = 0x1204,
    GR_MEMORY_STATE_COMPUTE_SHADER_WRITE_ONLY   = 0x1205,
    GR_MEMORY_STATE_COMPUTE_SHADER_READ_WRITE   = 0x1206,
    GR_MEMORY_STATE_MULTI_USE_READ_ONLY         = 0x1207,
    GR_MEMORY_STATE_INDEX_DATA                  = 0x1208,
    GR_MEMORY_STATE_INDIRECT_ARG                = 0x1209,
    GR_MEMORY_STATE_WRITE_TIMESTAMP             = 0x120a,
    GR_MEMORY_STATE_QUEUE_ATOMIC                = 0x120b,
    GR_MEMORY_STATE_DISCARD                     = 0x120c,
    GR_MEMORY_STATE_DATA_TRANSFER_SOURCE        = 0x120d,
    GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION   = 0x120e,
} GR_MEMORY_STATE;
```

Values

GR_MEMORY_STATE_DATA_TRANSFER

Memory range is accessible by the CPU for data transfer, or it can be copied to and from by the GPU.

GR_MEMORY_STATE_GRAPHICS_SHADER_READ_ONLY

Memory range can be used as a read-only memory view in the graphics pipeline.

GR_MEMORY_STATE_GRAPHICS_SHADER_WRITE_ONLY

Memory range can be used as a write-only memory view in the graphics pipeline.

GR_MEMORY_STATE_GRAPHICS_SHADER_READ_WRITE

Memory range can be used as a read or write memory view in the graphics pipeline.

GR_MEMORY_STATE_COMPUTE_SHADER_READ_ONLY

Memory range can be used as a read-only memory view in the compute pipeline.

GR_MEMORY_STATE_COMPUTE_SHADER_WRITE_ONLY

Memory range can be used as a write-only memory view in the compute pipeline.

GR_MEMORY_STATE_COMPUTE_SHADER_READ_WRITE

Memory range can be used as a read or write memory view in the compute pipeline.

GR_MEMORY_STATE_MULTI_USE_READ_ONLY

Memory range can be simultaneously used as a read-only memory view in the graphics or compute pipelines, or as index data, or as indirect arguments for draws and dispatches.

GR_MEMORY_STATE_INDEX_DATA

Memory range can be used by the graphics pipeline as index data.

GR_MEMORY_STATE_INDIRECT_ARG

Memory range can be used as arguments for indirect draw or dispatch operations.

GR_MEMORY_STATE_WRITE_TIMESTAMP

Memory range can be used as destination for writing GPU timestamps.

GR_MEMORY_STATE_QUEUE_ATOMIC

Memory range can be used for performing queue atomic operations.

GR_MEMORY_STATE_DISCARD

Memory range state should not be tracked and is invalid until it is transitioned to a valid state.

GR_MEMORY_STATE_DATA_TRANSFER_SOURCE

Memory range can be used as a source for GPU copies.

GR_MEMORY_STATE_DATA_TRANSFER_DESTINATION

Memory range can be used as a destination for GPU copies.

GR_NUM_FORMAT

Defines an image and memory view number format.

```
typedef enum _GR_NUM_FORMAT
{
    GR_NUM_FMT_UNDEFINED    = 0,
    GR_NUM_FMT_UNORM        = 1,
    GR_NUM_FMT_SNORM        = 2,
    GR_NUM_FMT_UINT         = 3,
    GR_NUM_FMT_SINT         = 4,
    GR_NUM_FMT_FLOAT        = 5,
    GR_NUM_FMT_SRGB         = 6,
    GR_NUM_FMT_DS           = 7,
} GR_NUM_FORMAT;
```

Values

GR_NUM_FMT_UNDEFINED

An undefined number format.

GR_NUM_FMT_UNORM

An unsigned normalized integer format.

GR_NUM_FMT_SNORM

A signed normalized integer format.

GR_NUM_FMT_UINT

An unsigned integer format.

GR_NUM_FMT_SINT

A signed integer format.

GR_NUM_FMT_FLOAT

A floating-point format.

GR_NUM_FMT_SRGB

An unsigned normalized sRGB integer format.

GR_NUM_FMT_DS

A depth-stencil format.

GR_PHYSICAL_GPU_TYPE

Defines the physical GPU type.

```
typedef enum _GR_PHYSICAL_GPU_TYPE
{
    GR_GPU_TYPE_OTHER                = 0x3000,
    GR_GPU_TYPE_INTEGRATED           = 0x3001,
    GR_GPU_TYPE_DISCRETE             = 0x3002,
    GR_GPU_TYPE_VIRTUAL              = 0x3003,
} GR_PHYSICAL_GPU_TYPE;
```

Values

GR_GPU_TYPE_OTHER

The GPU type that does not belong to any other category.

GR_GPU_TYPE_INTEGRATED

An integrated GPU, which is part of the APU.

GR_GPU_TYPE_DISCRETE

A discrete GPU.

GR_GPU_TYPE_VIRTUAL

A virtual GPU.

GR_PIPELINE_BIND_POINT

The pipeline bind point.

```
typedef enum _GR_PIPELINE_BIND_POINT
{
    GR_PIPELINE_BIND_POINT_COMPUTE  = 0x1e00,
    GR_PIPELINE_BIND_POINT_GRAPHICS = 0x1e01,
} GR_PIPELINE_BIND_POINT;
```

Values

GR_PIPELINE_BIND_POINT_COMPUTE

The bind point for compute pipelines.

GR_PIPELINE_BIND_POINT_GRAPHICS

The bind point for graphics pipelines.

GR_PRIMITIVE_TOPOLOGY

Primitive topology determines the type of the graphic primitives and vertex ordering for rendered geometry.

```
typedef enum _GR_PRIMITIVE_TOPOLOGY
{
    GR_TOPOLOGY_POINT_LIST          = 0x2000,
    GR_TOPOLOGY_LINE_LIST           = 0x2001,
    GR_TOPOLOGY_LINE_STRIP          = 0x2002,
    GR_TOPOLOGY_TRIANGLE_LIST       = 0x2003,
    GR_TOPOLOGY_TRIANGLE_STRIP      = 0x2004,
    GR_TOPOLOGY_RECT_LIST           = 0x2005,
    GR_TOPOLOGY_QUAD_LIST           = 0x2006,
    GR_TOPOLOGY_QUAD_STRIP          = 0x2007,
    GR_TOPOLOGY_LINE_LIST_ADJ       = 0x2008,
    GR_TOPOLOGY_LINE_STRIP_ADJ      = 0x2009,
    GR_TOPOLOGY_TRIANGLE_LIST_ADJ   = 0x200a,
    GR_TOPOLOGY_TRIANGLE_STRIP_ADJ  = 0x200b,
    GR_TOPOLOGY_PATCH               = 0x200c,
} GR_PRIMITIVE_TOPOLOGY;
```

Values

GR_TOPOLOGY_POINT_LIST

Input geometry is a list of points.

GR_TOPOLOGY_LINE_LIST

Input geometry is a list of lines.

GR_TOPOLOGY_LINE_STRIP

Input geometry is a line strip.

GR_TOPOLOGY_TRIANGLE_LIST

Input geometry is a list of triangles.

GR_TOPOLOGY_TRIANGLE_STRIP

Input geometry is a triangle strip.

GR_TOPOLOGY_RECT_LIST

Input geometry is a list of screen-aligned, non-clipped rectangles defined by three vertices.

GR_TOPOLOGY_QUAD_LIST

Input geometry is a list of quads.

GR_TOPOLOGY_QUAD_STRIP

Input geometry is a quad strip.

GR_TOPOLOGY_LINE_LIST_ADJ

Input geometry is a list of lines with adjacency information.

GR_TOPOLOGY_LINE_STRIP_ADJ

Input geometry is a line strip with adjacency information.

GR_TOPOLOGY_TRIANGLE_LIST_ADJ

Input geometry is a list of triangles with adjacency information.

GR_TOPOLOGY_TRIANGLE_STRIP_ADJ

Input geometry is a triangle strip with adjacency information.

GR_TOPOLOGY_PATCH

Input geometry is a list of tessellated patches.

GR_QUERY_TYPE

The types of GPU queries.

```
typedef enum _GR_QUERY_TYPE
{
    GR_QUERY_OCCLUSION              = 0x1a00,
    GR_QUERY_PIPELINE_STATISTICS    = 0x1a01,
} GR_QUERY_TYPE;
```

Values

GR_QUERY_OCCLUSION

An occlusion query counts a number of samples that pass depth and stencil tests.

GR_QUERY_PIPELINE_STATISTICS

A pipeline statistics query counts a number of processed elements at different stages in a pipeline.

GR_QUEUE_TYPE

The GPU queue type.

```
typedef enum _GR_QUEUE_TYPE
{
    GR_QUEUE_UNIVERSAL = 0x1000,
    GR_QUEUE_COMPUTE   = 0x1001,
} GR_QUEUE_TYPE;
```

Values

GR_QUEUE_UNIVERSAL

A universal pipeline queue capable of executing graphics and compute workloads.

GR_QUEUE_COMPUTE

A compute only pipeline queue.

GR_STATE_BIND_POINT

The bind points for the dynamic fixed-function state.

```
typedef enum _GR_STATE_BIND_POINT
{
    GR_STATE_BIND_VIEWPORT      = 0x1f00,
    GR_STATE_BIND_RASTER        = 0x1f01,
    GR_STATE_BIND_DEPTH_STENCIL = 0x1f02,
    GR_STATE_BIND_COLOR_BLEND   = 0x1f03,
    GR_STATE_BIND_MSAA          = 0x1f04,
} GR_STATE_BIND_POINT;
```

Values

GR_STATE_BIND_VIEWPORT

Bind point for a viewport and scissor dynamic state.

GR_STATE_BIND_RASTER

Bind point for a rasterizer dynamic state.

GR_STATE_BIND_DEPTH_STENCIL

Bind point for a depth-stencil dynamic state.

GR_STATE_BIND_COLOR_BLEND

Bind point for a color blender dynamic state.

GR_STATE_BIND_MSAA

Bind point for a multisampling dynamic state.

GR_STENCIL_OP

Defines a stencil operation performed during a stencil test.

```
typedef enum _GR_STENCIL_OP
{
    GR_STENCIL_OP_KEEP      = 0x2b00,
    GR_STENCIL_OP_ZERO      = 0x2b01,
    GR_STENCIL_OP_REPLACE   = 0x2b02,
    GR_STENCIL_OP_INC_CLAMP = 0x2b03,
    GR_STENCIL_OP_DEC_CLAMP = 0x2b04,
    GR_STENCIL_OP_INVERT    = 0x2b05,
    GR_STENCIL_OP_INC_WRAP  = 0x2b06,
    GR_STENCIL_OP_DEC_WRAP  = 0x2b07,
} GR_STENCIL_OP;
```

Values

GR_STENCIL_OP_KEEP

Keeps the stencil unchanged.

GR_STENCIL_OP_ZERO

Sets the stencil data to zero.

GR_STENCIL_OP_REPLACE

Sets the stencil data to a reference value.

GR_STENCIL_OP_INC_CLAMP

Increments the stencil data and clamps the result.

GR_STENCIL_OP_DEC_CLAMP

Decrements the stencil data and clamps the result.

GR_STENCIL_OP_INVERT

Inverts the stencil data.

GR_STENCIL_OP_INC_WRAP

Increments the stencil data and wraps the result.

GR_STENCIL_OP_DEC_WRAP

Decrements the stencil data and wraps the result.

GR_SYSTEM_ALLOC_TYPE

Defines the system memory allocation type reported in allocator callback.

```
typedef enum _GR_SYSTEM_ALLOC_TYPE
{
    GR_SYSTEM_ALLOC_API_OBJECT      = 0x2e00,
    GR_SYSTEM_ALLOC_INTERNAL        = 0x2e01,
    GR_SYSTEM_ALLOC_INTERNAL_TEMP   = 0x2e02,
    GR_SYSTEM_ALLOC_INTERNAL_SHADER = 0x2e03,
    GR_SYSTEM_ALLOC_DEBUG           = 0x2e04,
} GR_SYSTEM_ALLOC_TYPE;
```

Values

GR_SYSTEM_ALLOC_API_OBJECT

The allocation is used for an API object or for other data that share the lifetime of an API object.

GR_SYSTEM_ALLOC_INTERNAL

The allocation is used for an internal structure that driver expects to be relatively long-lived.

GR_SYSTEM_ALLOC_INTERNAL_TEMP

The allocation is used for an internal structure that driver expects to be short-lived. A general lifetime expectancy for this allocation type is the duration of an API call.

GR_SYSTEM_ALLOC_INTERNAL_SHADER

The allocation is used for an internal structure used for shader compilation that driver expects to be short-lived. A general lifetime expectancy for this allocation type is the duration of pipeline creation call.

GR_SYSTEM_ALLOC_DEBUG

The allocation is used for validation layer internal data other than API objects.

GR_TEX_ADDRESS

Texture address mode determines how texture coordinates outside of texture boundaries are interpreted.

```
typedef enum _GR_TEX_ADDRESS
{
    GR_TEX_ADDRESS_WRAP         = 0x2400,
    GR_TEX_ADDRESS_MIRROR       = 0x2401,
    GR_TEX_ADDRESS_CLAMP        = 0x2402,
    GR_TEX_ADDRESS_MIRROR_ONCE  = 0x2403,
    GR_TEX_ADDRESS_CLAMP_BORDER = 0x2404,
} GR_TEX_ADDRESS;
```

Values

GR_TEX_ADDRESS_WRAP

Repeats the texture in a given direction.

GR_TEX_ADDRESS_MIRROR

Mirrors the texture in a given direction by flipping the texture at every other coordinate interval.

GR_TEX_ADDRESS_CLAMP

Clamps the texture to the last edge pixel.

GR_TEX_ADDRESS_MIRROR_ONCE

Mirrors the texture just once, then clamps it.

GR_TEX_ADDRESS_CLAMP_BORDER

Clamps the texture to the border color specified in the sampler.

GR_TEX_FILTER

The texture filter determines how sampled texture color is derived from neighboring texels.

```
typedef enum _GR_TEX_FILTER
{
    GR_TEX_FILTER_MAG_POINT_MIN_POINT_MIP_POINT     = 0x2340,
    GR_TEX_FILTER_MAG_LINEAR_MIN_POINT_MIP_POINT    = 0x2341,
    GR_TEX_FILTER_MAG_POINT_MIN_LINEAR_MIP_POINT    = 0x2344,
    GR_TEX_FILTER_MAG_LINEAR_MIN_LINEAR_MIP_POINT   = 0x2345,
    GR_TEX_FILTER_MAG_POINT_MIN_POINT_MIP_LINEAR    = 0x2380,
    GR_TEX_FILTER_MAG_LINEAR_MIN_POINT_MIP_LINEAR   = 0x2381,
    GR_TEX_FILTER_MAG_POINT_MIN_LINEAR_MIP_LINEAR   = 0x2384,
    GR_TEX_FILTER_MAG_LINEAR_MIN_LINEAR_MIP_LINEAR  = 0x2385,
    GR_TEX_FILTER_ANISOTROPIC                       = 0x238f,
} GR_TEX_FILTER;
```

Values

GR_TEX_FILTER_MAG_POINT_MIN_POINT_MIP_POINT

Point sample for magnification, point sample for minification, and point sample for mipmap level filtering

GR_TEX_FILTER_MAG_LINEAR_MIN_POINT_MIP_POINT

Linear interpolation for magnification, point sample for minification, and point sample for mipmap level filtering

GR_TEX_FILTER_MAG_POINT_MIN_LINEAR_MIP_POINT

Point sample for magnification, linear interpolation for minification, and point sample for mipmap level filtering

GR_TEX_FILTER_MAG_LINEAR_MIN_LINEAR_MIP_POINT

Linear interpolation for magnification, linear interpolation for minification, and point sample for mipmap level filtering

GR_TEX_FILTER_MAG_POINT_MIN_POINT_MIP_LINEAR

Point sample for magnification, point sample for minification, and linear interpolation for mipmap level filtering

GR_TEX_FILTER_MAG_LINEAR_MIN_POINT_MIP_LINEAR

Linear interpolation for magnification, point sample for minification, and linear interpolation for mipmap level filtering

GR_TEX_FILTER_MAG_POINT_MIN_LINEAR_MIP_LINEAR

Point sample for magnification, linear interpolation for minification, and linear interpolation for mipmap level filtering

GR_TEX_FILTER_MAG_LINEAR_MIN_LINEAR_MIP_LINEAR

Linear interpolation for magnification, linear interpolation for minification, linear interpolation for and mipmap level filtering

GR_TEX_FILTER_ANISOTROPIC

Anisotropic interpolation

GR_TIMESTAMP_TYPE

The GPU timestamp type determines where in a pipeline timestamps are generated.

```
typedef enum _GR_TIMESTAMP_TYPE
{
    GR_TIMESTAMP_TOP        = 0x1b00,
    GR_TIMESTAMP_BOTTOM     = 0x1b01,
} GR_TIMESTAMP_TYPE;
```

Values

GR_TIMESTAMP_TOP

Top-of-pipe timestamp is generated when draw or dispatch become active.

GR_TIMESTAMP_BOTTOM

Bottom-of-pipe timestamp is generated when draw or dispatch have finished execution.

GR_VALIDATION_LEVEL

Defines a level of validation.

```
typedef enum _GR_VALIDATION_LEVEL
{
    GR_VALIDATION_LEVEL_0   = 0x8000,
    GR_VALIDATION_LEVEL_1   = 0x8001,
    GR_VALIDATION_LEVEL_2   = 0x8002,
    GR_VALIDATION_LEVEL_3   = 0x8003,
    GR_VALIDATION_LEVEL_4   = 0x8004,
} GR_VALIDATION_LEVEL;
```

Values

GR_VALIDATION_LEVEL_0

At this validation level, trivial API checks are performed (e.g., checking function parameters). This is the default level of checks without the validation level. At this level command buffer, building is not validated.

GR_VALIDATION_LEVEL_1

Level 1 validation adds checks that do not require command buffer analysis or knowledge of the execution-time memory layout. At this level, command buffer building is partially validated.

GR_VALIDATION_LEVEL_2

Level 2 validation adds command buffer checks that depend on submission-time analysis of command buffer contents, but have no knowledge of the execution-time memory layout.

GR_VALIDATION_LEVEL_3

Level 3 validation adds checks that require relatively lightweight analysis of execution-time memory layout.

GR_VALIDATION_LEVEL_4

Level 4 validation adds checks that require full analysis of execution-time memory layout.

FLAGS

GR_CMD_BUFFER_BUILD_FLAGS

Optional hints to specify command buffer building optimizations.

```
typedef enum _GR_CMD_BUFFER_BUILD_FLAGS
{
    GR_CMD_BUFFER_OPTIMIZE_GPU_SMALL_BATCH       = 0x00000001,
    GR_CMD_BUFFER_OPTIMIZE_PIPELINE_SWITCH       = 0x00000002,
    GR_CMD_BUFFER_OPTIMIZE_ONE_TIME_SUBMIT       = 0x00000004,
    GR_CMD_BUFFER_OPTIMIZE_DESCRIPTOR_SET_SWITCH = 0x00000008,
} GR_CMD_BUFFER_BUILD_FLAGS;
```

Values

GR_CMD_BUFFER_OPTIMIZE_GPU_SMALL_BATCH

Optimize command buffer building for a large number of draw or dispatch operations that are GPU front-end limited. Optimization might increase CPU overhead during command buffer building.

GR_CMD_BUFFER_OPTIMIZE_PIPELINE_SWITCH

Optimize command buffer building for the case of frequent pipeline switching. Optimization might increase CPU overhead during command buffer building.

GR_CMD_BUFFER_OPTIMIZE_ONE_TIME_SUBMIT

Optimizes command buffer building for single command buffer submission. Command buffers built with this flag cannot be submitted more than once.

GR_CMD_BUFFER_OPTIMIZE_DESCRIPTOR_SET_SWITCH

Optimizes command buffer building for the case of frequent descriptor set switching. Optimization might increase CPU overhead during command buffer building.

GR_DEPTH_STENCIL_VIEW_CREATE_FLAGS

Depth-stencil view creation flags.

```
typedef enum _GR_DEPTH_STENCIL_VIEW_CREATE_FLAGS
{
    GR_DEPTH_STENCIL_VIEW_CREATE_READ_ONLY_DEPTH   = 0x00000001,
    GR_DEPTH_STENCIL_VIEW_CREATE_READ_ONLY_STENCIL = 0x00000002,
} GR_DEPTH_STENCIL_VIEW_CREATE_FLAGS;
```

Values

GR_DEPTH_STENCIL_VIEW_CREATE_READ_ONLY_DEPTH

Depth-stencil view has depth that is available for read-only access.

GR_DEPTH_STENCIL_VIEW_CREATE_READ_ONLY_STENCIL

Depth-stencil view has stencil that is available for read-only access.

GR_DEVICE_CREATE_FLAGS

Device creation flags.

```
typedef enum _GR_DEVICE_CREATE_FLAGS
{
    GR_DEVICE_CREATE_VALIDATION = 0x00000001,
} GR_DEVICE_CREATE_FLAGS;
```

Values

GR_DEVICE_CREATE_VALIDATION

Enables validation layer for the device.

GR_FORMAT_FEATURE_FLAGS

Format capability flags for images and memory views.

```
typedef enum _GR_FORMAT_FEATURE_FLAGS
{
    GR_FORMAT_IMAGE_SHADER_READ    = 0x00000001,
    GR_FORMAT_IMAGE_SHADER_WRITE   = 0x00000002,
    GR_FORMAT_IMAGE_COPY           = 0x00000004,
    GR_FORMAT_MEMORY_SHADER_ACCESS = 0x00000008,
    GR_FORMAT_COLOR_TARGET_WRITE   = 0x00000010,
    GR_FORMAT_COLOR_TARGET_BLEND   = 0x00000020,
    GR_FORMAT_DEPTH_TARGET         = 0x00000040,
    GR_FORMAT_STENCIL_TARGET       = 0x00000080,
    GR_FORMAT_MSAA_TARGET          = 0x00000100,
    GR_FORMAT_CONVERSION           = 0x00000200,
} GR_FORMAT_FEATURE_FLAGS;
```

Values

GR_FORMAT_IMAGE_SHADER_READ

Images of this format can be accessed in shaders for read operations.

GR_FORMAT_IMAGE_SHADER_WRITE

Images of this format can be accessed in shaders for write operations.

GR_FORMAT_IMAGE_COPY

Images of this format could be used as source or destination for image copy operations.

GR_FORMAT_MEMORY_SHADER_ACCESS

Memory views of this format can be accessed in shaders for read or write operations.

GR_FORMAT_COLOR_TARGET_WRITE

Images of this format can be used as color targets.

GR_FORMAT_COLOR_TARGET_BLEND

Images of this format can be used as blendable color targets.

GR_FORMAT_DEPTH_TARGET

Images of this format can be used as depth targets.

GR_FORMAT_STENCIL_TARGET

Images of this format can be used as stencil targets.

GR_FORMAT_MSAA_TARGET

Images of this format support multisampling.

GR_FORMAT_CONVERSION

Images of this format support format conversion on image copy operations.

GR_GPU_COMPATIBILITY_FLAGS

GPU compatibility flags for multi-device configurations.

```
typedef enum _GR_GPU_COMPATIBILITY_FLAGS
{
    GR_GPU_COMPAT_ASIC_FEATURES       = 0x00000001,
    GR_GPU_COMPAT_IQ_MATCH            = 0x00000002,
    GR_GPU_COMPAT_PEER_WRITE_TRANSFER = 0x00000004,
    GR_GPU_COMPAT_SHARED_MEMORY       = 0x00000008,
    GR_GPU_COMPAT_SHARED_SYNC         = 0x00000010,
    GR_GPU_COMPAT_SHARED_GPU0_DISPLAY = 0x00000020,
    GR_GPU_COMPAT_SHARED_GPU1_DISPLAY = 0x00000040,
} GR_GPU_COMPATIBILITY_FLAGS;
```

Values

GR_GPU_COMPAT_ASIC_FEATURES

GPUs have compatible ASIC features (exactly the same internal tiling, the same pipeline binary data, etc.).

GR_GPU_COMPAT_IQ_MATCH

GPUs can generate images with similar image quality.

GR_GPU_COMPAT_PEER_WRITE_TRANSFER

GPUs support peer-to-peer transfers over the PCIe.

GR_GPU_COMPAT_SHARED_MEMORY

GPUs can share some memory objects.

GR_GPU_COMPAT_SHARED_SYNC

GPUs can share queue semaphores.

GR_GPU_COMPAT_SHARED_GPU0_DISPLAY

GPU1 can create a presentable image on a display connected to GPU0.

GR_GPU_COMPAT_SHARED_GPU1_DISPLAY

GPU0 can create a presentable image on a display connected to GPU1.

GR_IMAGE_CREATE_FLAGS

Image creation flags.

```
typedef enum _GR_IMAGE_CREATE_FLAGS
{
    GR_IMAGE_CREATE_INVARIANT_DATA     = 0x00000001,
    GR_IMAGE_CREATE_CLONEABLE          = 0x00000002,
    GR_IMAGE_CREATE_SHAREABLE          = 0x00000004,
    GR_IMAGE_CREATE_VIEW_FORMAT_CHANGE = 0x00000008,
} GR_IMAGE_CREATE_FLAGS;
```

Values

GR_IMAGE_CREATE_INVARIANT_DATA

Images of exactly the same creation parameters are guaranteed to have consistent data layout.

GR_IMAGE_CREATE_CLONEABLE

Image can be used as a source or destination for cloning operation.

GR_IMAGE_CREATE_SHAREABLE

Image can be shared between compatible devices.

GR_IMAGE_CREATE_VIEW_FORMAT_CHANGE

Image can have its format changed in image or color target views.

GR_IMAGE_USAGE_FLAGS

Image usage flags.

```
typedef enum _GR_IMAGE_USAGE_FLAGS
{
    GR_IMAGE_USAGE_SHADER_ACCESS_READ  = 0x00000001,
    GR_IMAGE_USAGE_SHADER_ACCESS_WRITE = 0x00000002,
    GR_IMAGE_USAGE_COLOR_TARGET        = 0x00000004,
    GR_IMAGE_USAGE_DEPTH_STENCIL       = 0x00000008,
} GR_IMAGE_USAGE_FLAGS;
```

Values

GR_IMAGE_USAGE_SHADER_ACCESS_READ

Image will be bound to shaders for read access.

GR_IMAGE_USAGE_SHADER_ACCESS_WRITE

Image will be bound to shaders for write access. Only applies to direct image writes from shaders; it is not required for targets.

GR_IMAGE_USAGE_COLOR_TARGET

Image will be used as a color target. Used for color target output and blending.

GR_IMAGE_USAGE_DEPTH_STENCIL

Image will be used as a depth-stencil target.

GR_MEMORY_ALLOC_FLAGS

Memory allocation flags.

```
typedef enum _GR_MEMORY_ALLOC_FLAGS
{
    GR_MEMORY_ALLOC_VIRTUAL   = 0x00000001,
    GR_MEMORY_ALLOC_SHAREABLE = 0x00000002,
} GR_MEMORY_ALLOC_FLAGS;
```

Values

GR_MEMORY_ALLOC_VIRTUAL

Memory object represents a virtual allocation.

GR_MEMORY_ALLOC_SHAREABLE

Memory object can be shared between compatible devices.

GR_MEMORY_HEAP_FLAGS

GPU memory heap property flags.

```
typedef enum _GR_MEMORY_HEAP_FLAGS
{
    GR_MEMORY_HEAP_CPU_VISIBLE        = 0x00000001,
    GR_MEMORY_HEAP_CPU_GPU_COHERENT   = 0x00000002,
    GR_MEMORY_HEAP_CPU_UNCACHED       = 0x00000004,
    GR_MEMORY_HEAP_CPU_WRITE_COMBINED = 0x00000008,
    GR_MEMORY_HEAP_HOLDS_PINNED       = 0x00000010,
    GR_MEMORY_HEAP_SHAREABLE          = 0x00000020,
} GR_MEMORY_HEAP_FLAGS;
```

Values

GR_MEMORY_HEAP_CPU_VISIBLE

Memory heap is in a CPU address space and is CPU accessible through map mechanism.

GR_MEMORY_HEAP_CPU_GPU_COHERENT

Memory heap is cache coherent between the CPU and GPU.

GR_MEMORY_HEAP_CPU_UNCACHED

Memory heap is not cached by the CPU, but it could still be cached by the GPU.

GR_MEMORY_HEAP_CPU_WRITE_COMBINED

Memory heap is write-combined by the CPU.

GR_MEMORY_HEAP_HOLDS_PINNED

All pinned memory objects behave as if they were created in a heap marked with this flag. Only one heap has this flag set.

GR_MEMORY_HEAP_SHAREABLE

Memory heap can be used for memory objects that can be shared between multiple GPUs.

GR_MEMORY_PROPERTY_FLAGS

Flags for GPU memory system properties for the physical GPU.

```
typedef enum _GR_MEMORY_PROPERTY_FLAGS
{
    GR_MEMORY_MIGRATION_SUPPORT          = 0x00000001,
    GR_MEMORY_VIRTUAL_REMAPPING_SUPPORT = 0x00000002,
    GR_MEMORY_PINNING_SUPPORT            = 0x00000004,
    GR_MEMORY_PREFER_GLOBAL_REFS         = 0x00000008,
} GR_MEMORY_PROPERTY_FLAGS;
```

Values

GR_MEMORY_MIGRATION_SUPPORT

The GPU memory manager supports dynamic memory object migration.

GR_MEMORY_VIRTUAL_REMAPPING_SUPPORT

The GPU memory manager supports virtual memory remapping.

GR_MEMORY_PINNING_SUPPORT

The GPU memory manager supports pinning of system memory.

GR_MEMORY_PREFER_GLOBAL_REFS

When set, the application should prefer using global memory references instead of per command buffer memory references for CPU performance reasons.

GR_MEMORY_REF_FLAGS

Flags for GPU memory object references used for command buffer submission.

```
typedef enum _GR_MEMORY_REF_FLAGS
{
    GR_MEMORY_REF_READ_ONLY = 0x00000001,
} GR_MEMORY_REF_FLAGS;
```

Values

GR_MEMORY_REF_READ_ONLY

GPU memory object is only used for read-only access in the submitted command buffers.

GR_PIPELINE_CREATE_FLAGS

Pipeline creation flags.

```
typedef enum _GR_PIPELINE_CREATE_FLAGS
{
    GR_PIPELINE_CREATE_DISABLE_OPTIMIZATION = 0x00000001,
} GR_PIPELINE_CREATE_FLAGS;
```

Values

GR_PIPELINE_CREATE_DISABLE_OPTIMIZATION

Disables pipeline link-time optimizations. Should only be used for debugging.

GR_QUERY_CONTROL_FLAGS

Flags for controlling GPU query behavior.

```
typedef enum _GR_QUERY_CONTROL_FLAGS
{
    GR_QUERY_IMPRECISE_DATA = 0x00000001,
} GR_QUERY_CONTROL_FLAGS;
```

Values

GR_QUERY_IMPRECISE_DATA

Controls accuracy of query data collection. Available only for occlusion queries. If set, the occlusion query is guaranteed to return an imprecise non-zero value in case any of the samples pass a depth and stencil test. Using imprecise occlusion query results could improve rendering performance while an occlusion query is active.

GR_SEMAPHORE_CREATE_FLAGS

Queue semaphore creation flags.

```
typedef enum _GR_SEMAPHORE_CREATE_FLAGS
{
    GR_SEMAPHORE_CREATE_SHAREABLE = 0x00000001,
} GR_SEMAPHORE_CREATE_FLAGS;
```

Values

GR_SEMAPHORE_CREATE_SHAREABLE

Queue semaphore can be shared between compatible devices.

GR_SHADER_CREATE_FLAGS

Shader creation flags.

```
typedef enum _GR_SHADER_CREATE_FLAGS
{
    GR_SHADER_CREATE_ALLOW_RE_Z = 0x00000001,
} GR_SHADER_CREATE_FLAGS;
```

Values

GR_SHADER_CREATE_ALLOW_RE_Z

Pixel shader can have Re-Z enabled (applicable to pixel shaders only).

DATA STRUCTURES

GR_ALLOC_CALLBACKS

Application provided callbacks for system memory allocations inside of the Mantle driver.

```
typedef struct _GR_ALLOC_CALLBACKS
{
    GR_ALLOC_FUNCTION pfnAlloc;
    GR_FREE_FUNCTION  pfnFree;
} GR_ALLOC_CALLBACKS;
```

Members

pfnAlloc

[in] An allocation provided callback to allocate system memory inside the Mantle driver. See `GR_ALLOC_FUNCTION`.

pfnFree

[in] An application provided callback to free system memory inside the Mantle driver. See `GR_FREE_FUNCTION`.

GR_APPLICATION_INFO

Application identification information that can be communicated by the application to the driver.

```
typedef struct _GR_APPLICATION_INFO
{
    const GR_CHAR* pAppName;
    GR_UINT32      appVersion;
    const GR_CHAR* pEngineName;
    GR_UINT32      engineVersion;
    GR_UINT32      apiVersion;
} GR_APPLICATION_INFO;
```

Members

pAppName

[in] A string with the name of the applications.

appVersion

The version of the application encoded using `GR_MAKE_VERSION` macro.

pEngineName

[in] A string with the engine name.

engineVersion

The engine version encoded using the `GR_MAKE_VERSION` macro.

apiVersion

The API version to which the application is compiled; encoded using the `GR_MAKE_VERSION` macro.

GR_CHANNEL_MAPPING

Channel mapping for image views.

```
typedef struct _GR_CHANNEL_MAPPING
{
    GR_ENUM r;
    GR_ENUM g;
    GR_ENUM b;
    GR_ENUM a;
} GR_CHANNEL_MAPPING;
```

Members

r

Swizzle for red channel. See `GR_CHANNEL_SWIZZLE`.

g

Swizzle for green channel. See `GR_CHANNEL_SWIZZLE`.

b

Swizzle for blue channel. See `GR_CHANNEL_SWIZZLE`.

a

Swizzle for alpha channel. See `GR_CHANNEL_SWIZZLE`.

GR_CMD_BUFFER_CREATE_INFO

Command buffer creation information.

```
typedef struct _GR_CMD_BUFFER_CREATE_INFO
{
    GR_ENUM  queueType;
    GR_FLAGS flags;
} GR_CMD_BUFFER_CREATE_INFO;
```

Members

queueType

Queue type the command buffer is prepared for. See `GR_QUEUE_TYPE`.

flags

Reserved, must be zero.

GR_COLOR_BLEND_STATE_CREATE_INFO

Dynamic color blender state object creation information.

```
typedef struct _GR_COLOR_BLEND_STATE_CREATE_INFO
{
    GR_COLOR_TARGET_BLEND_STATE target[GR_MAX_COLOR_TARGETS];
    GR_FLOAT                    blendConst[4];
} GR_COLOR_BLEND_STATE_CREATE_INFO;
```

Members

target

Array of blender state per color target. See GR_COLOR_TARGET_BLEND_STATE.

blendConst

Constant color value to use for blending.

GR_COLOR_TARGET_BIND_INFO

Per color target information for binding it to command buffer state.

```
typedef struct _GR_COLOR_TARGET_BIND_INFO
{
    GR_COLOR_TARGET_VIEW view;
    GR_ENUM              colorTargetState;
} GR_COLOR_TARGET_BIND_INFO;
```

Members

view

Color target view to bind.

colorTargetState

Color target view image state at the draw time. See GR_IMAGE_STATE.

GR_COLOR_TARGET_BLEND_STATE

Per target dynamic color blender state object creation information.

```
typedef struct _GR_COLOR_TARGET_BLEND_STATE
{
    GR_BOOL blendEnable;
    GR_ENUM srcBlendColor;
    GR_ENUM destBlendColor;
    GR_ENUM blendFuncColor;
    GR_ENUM srcBlendAlpha;
    GR_ENUM destBlendAlpha;
    GR_ENUM blendFuncAlpha;
} GR_COLOR_TARGET_BLEND_STATE;
```

Members

blendEnable

Per color target blending operation enable.

srcBlendColor

Source part of the blend equation for color. See `GR_BLEND`.

destBlendColor

Destination part of the blend equation for color. See `GR_BLEND`.

blendFuncColor

Blend function for color. See `GR_BLEND_FUNC`.

srcBlendAlpha

Source part of the blend equation for alpha. See `GR_BLEND`.

destBlendAlpha

Destination part of the blend equation for alpha. See `GR_BLEND`.

blendFuncAlpha

Blend function for alpha. See `GR_BLEND_FUNC`.

GR_COLOR_TARGET_VIEW_CREATE_INFO

Color target view creation information.

```
typedef struct _GR_COLOR_TARGET_VIEW_CREATE_INFO
{
    GR_IMAGE  image;
    GR_FORMAT format;
    GR_UINT   mipLevel;
    GR_UINT   baseArraySlice;
    GR_UINT   arraySize;
} GR_COLOR_TARGET_VIEW_CREATE_INFO;
```

Members

image

Image for the view.

format

Format for the view. Has to be compatible with the image format. See GR_FORMAT.

mipLevel

Mipmap level to render.

baseArraySlice

First array slice for 2D array resources, or first depth slice for 3D image resources.

arraySize

Number of array slice for 2D array resources, or number of depth slices for 3D image resources.

GR_COMPUTE_PIPELINE_CREATE_INFO

Compute pipeline creation information.

```
typedef struct _GR_COMPUTE_PIPELINE_CREATE_INFO
{
    GR_PIPELINE_SHADER cs;
    GR_FLAGS           flags;
} GR_COMPUTE_PIPELINE_CREATE_INFO;
```

Members

cs

Compute shader information. See GR_PIPELINE_SHADER.

flags

Flags for pipeline creation. See GR_PIPELINE_CREATE_FLAGS.

GR_DEPTH_STENCIL_BIND_INFO

Depth-stencil target information for binding it to command buffer state.

```
typedef struct _GR_DEPTH_STENCIL_BIND_INFO
{
    GR_DEPTH_STENCIL_VIEW view;
    GR_ENUM               depthState;
    GR_ENUM               stencilState;
} GR_DEPTH_STENCIL_BIND_INFO;
```

Members

view

Depth-stencil view to bind.

depthState

Depth aspect target view image state at the draw time. See `GR_IMAGE_STATE`.

stencilState

Stencil aspect target view image state at the draw time. See `GR_IMAGE_STATE`.

GR_DEPTH_STENCIL_OP

Per face (front or back) stencil state for the dynamic depth-stencil state.

```
typedef struct _GR_DEPTH_STENCIL_OP
{
    GR_ENUM  stencilFailOp;
    GR_ENUM  stencilPassOp;
    GR_ENUM  stencilDepthFailOp;
    GR_ENUM  stencilFunc;
    GR_UINT8 stencilRef;
} GR_DEPTH_STENCIL_OP;
```

Members

stencilFailOp

Stencil operation to apply when stencil test fails. See `GR_STENCIL_OP`.

stencilPassOp

Stencil operation to apply when stencil and depth tests pass. See `GR_STENCIL_OP`.

stencilDepthFailOp

Stencil operation to apply when stencil test passes and depth test fails. See `GR_STENCIL_OP`.

stencilFunc

Stencil comparison function. See `GR_COMPARE_FUNC`.

stencilRef

Stencil reference value.

GR_DEPTH_STENCIL_STATE_CREATE_INFO

Dynamic depth-stencil state creation information.

```
typedef struct _GR_DEPTH_STENCIL_STATE_CREATE_INFO
{
    GR_BOOL             depthEnable;
    GR_BOOL             depthWriteEnable;
    GR_ENUM             depthFunc;
    GR_BOOL             depthBoundsEnable;
    GR_FLOAT            minDepth;
    GR_FLOAT            maxDepth;
    GR_BOOL             stencilEnable;
    GR_UINT8            stencilReadMask;
    GR_UINT8            stencilWriteMask;
    GR_DEPTH_STENCIL_OP front;
    GR_DEPTH_STENCIL_OP back;
} GR_DEPTH_STENCIL_STATE_CREATE_INFO;
```

Members

depthEnable

Enable depth testing.

depthWriteEnable

Enable depth writing.

depthFunc

Depth comparison function. See GR_COMPARE_FUNC.

depthBoundsEnable

Enable depth bounds.

minDepth

Minimal depth bounds value.

maxDepth

Maximum depth bounds value.

stencilEnable

Enable stencil testing.

stencilReadMask

Bitmask to apply to stencil reads.

stencilWriteMask

Bitmask to apply to stencil writes.

front

Stencil operations for front-facing geometry. See GR_DEPTH_STENCIL_OP.

back

Stencil operations for back-facing geometry. See GR_DEPTH_STENCIL_OP.

GR_DEPTH_STENCIL_VIEW_CREATE_INFO

Depth-stencil target view creation information.

```
typedef struct _GR_DEPTH_STENCIL_VIEW_CREATE_INFO
{
    GR_IMAGE image;
    GR_UINT  mipLevel;
    GR_UINT  baseArraySlice;
    GR_UINT  arraySize;
    GR_FLAGS flags;
} GR_DEPTH_STENCIL_VIEW_CREATE_INFO;
```

Members

image

Image for the view.

mipLevel

Mipmap level to render.

baseArraySlice

First array slice for 2D array resources, or first depth slice for 3D image resources.

arraySize

Number of array slice for 2D array resources, or number of depth slices for 3D image resources.

flags

Depth-stencil view flags. See GR_DEPTH_STENCIL_VIEW_CREATE_FLAGS.

GR_DESCRIPTOR_SET_ATTACH_INFO

Descriptor set range attachment info for building hierarchical descriptor sets.

```
typedef struct _GR_DESCRIPTOR_SET_ATTACH_INFO
{
    GR_DESCRIPTOR_SET descriptorSet;
    GR_UINT           slotOffset;
} GR_DESCRIPTOR_SET_ATTACH_INFO;
```

Members

descriptorSet

Descriptor set handle to use for binding.

slotOffset

The first slot in the descriptor set to be used for binding.

GR_DESCRIPTOR_SET_CREATE_INFO

Descriptor set creation information.

```
typedef struct _GR_DESCRIPTOR_SET_CREATE_INFO
{
    GR_UINT slots;
} GR_DESCRIPTOR_SET_CREATE_INFO;
```

Members

slots

Total number of resource slots in the descriptor set.

GR_DESCRIPTOR_SET_MAPPING

Descriptor set mapping for pipeline shaders. Provides association of descriptor sets to the shader resources. The structure represents the descriptor set layout that is used at the draw time. A separate mapping is provided for each shader in the pipeline.

```
typedef struct _GR_DESCRIPTOR_SET_MAPPING
{
    GR_UINT                        descriptorCount;
    const GR_DESCRIPTOR_SLOT_INFO* pDescriptorInfo;
} GR_DESCRIPTOR_SET_MAPPING;
```

Members

descriptorCount

Number of slots in a descriptor set that are available to the shader.

pDescriptorInfo

Array of descriptor slot mappings. See `GR_DESCRIPTOR_SLOT_INFO`.

GR_DESCRIPTOR_SLOT_INFO

Mapping of descriptor slot to the shader IL entities.

```
typedef struct _GR_DESCRIPTOR_SLOT_INFO
{
    GR_ENUM slotObjectType;
    union
    {
        GR_UINT                                  shaderEntityIndex;
        const struct _GR_DESCRIPTOR_SET_MAPPING* pNextLevelSet;
    };
} GR_DESCRIPTOR_SLOT_INFO;
```

Members

slotObjectType

The object type a pipeline expects to see in the descriptor set at the draw time. See `GR_DESCRIPTOR_SET_SLOT_TYPE`.

shaderEntityIndex

The shader entity index, if the slot object type references one of the shader entities.

pNextLevelSet

The pointer to the next level of descriptor set mapping information, if the slot object type references a nested descriptor set (for hierarchical descriptor sets). See `GR_DESCRIPTOR_SET_MAPPING`.

GR_DEVICE_CREATE_INFO

Device creation information.

```
typedef struct _GR_DEVICE_CREATE_INFO
{
    GR_UINT                            queueRecordCount;
    const GR_DEVICE_QUEUE_CREATE_INFO* pRequestedQueues;
    GR_UINT                            extensionCount;
    const GR_CHAR*const*               ppEnabledExtensionNames;
    GR_ENUM                            maxValidationLevel;
    GR_FLAGS                           flags;
} GR_DEVICE_CREATE_INFO;
```

Members

queueRecordCount

The number of queue initialization records.

pRequestedQueues

[in] An array of queue initialization records. See `GR_DEVICE_QUEUE_CREATE_INFO`. There could only be one record per queue type.

extensionCount

The number of extensions requested on device creation.

ppEnabledExtensionNames

[in] The array of strings with extension names the application would like to enable on the device.

maxValidationLevel

The maximum validation level that could be enabled on a device during application execution. See `GR_VALIDATION_LEVEL`. If validation is disabled, the only valid value is `GR_VALIDATION_LEVEL_0`.

flags

Device creation flags. See GR_DEVICE_CREATE_FLAGS.

GR_DEVICE_QUEUE_CREATE_INFO

Per-queue type initialization information specified on device creation.

```
typedef struct _GR_DEVICE_QUEUE_CREATE_INFO
{
    GR_ENUM queueType;
    GR_UINT queueCount;
} GR_DEVICE_QUEUE_CREATE_INFO;
```

Members

queueType

The type of queue to initialize on device creation. See GR_QUEUE_TYPE.

queueCount

The number of queues of a given type to initialize on device creation.

GR_DISPATCH_INDIRECT_ARG

Structure describing work dimensions for indirect dispatch.

```
typedef struct _GR_DISPATCH_INDIRECT_ARG
{
    GR_UINT32 x;
    GR_UINT32 y;
    GR_UINT32 z;
} GR_DISPATCH_INDIRECT_ARG;
```

Members

x

Number of thread groups in X direction.

y

Number of thread groups in Y direction.

z

Number of thread groups in Z direction.

GR_DRAW_INDEXED_INDIRECT_ARG

Structure describing work parameters for indirect indexed draw.

```
typedef struct _GR_DRAW_INDEXED_INDIRECT_ARG
{
    GR_UINT32 indexCount;
    GR_UINT32 instanceCount;
    GR_UINT32 firstIndex;
    GR_INT32  vertexOffset;
    GR_UINT32 firstInstance;
} GR_DRAW_INDEXED_INDIRECT_ARG;
```

Members

indexCount

Number of indices per instance.

instanceCount

Number of instances.

firstIndex

Index offset.

vertexOffset

Vertex offset.

firstInstance

Instance offset.

GR_DRAW_INDIRECT_ARG

Structure describing work parameters for indirect draw.

```
typedef struct _GR_DRAW_INDIRECT_ARG
{
    GR_UINT32 vertexCount;
    GR_UINT32 instanceCount;
    GR_UINT32 firstVertex;
    GR_UINT32 firstInstance;
} GR_DRAW_INDIRECT_ARG;
```

Members

vertexCount

Number of vertices per instance.

instanceCount

Number of instances.

firstVertex

First vertex offset.

firstInstance

First instance offset.

GR_DYNAMIC_MEMORY_VIEW_SLOT_INFO

Per shader mapping of dynamic memory view to shader entity.

```
typedef struct _GR_DYNAMIC_MEMORY_VIEW_SLOT_INFO
{
    GR_ENUM slotObjectType;
    GR_UINT shaderEntityIndex;
} GR_DYNAMIC_MEMORY_VIEW_SLOT_INFO;
```

Members

slotObjectType

The object type a pipeline expects to see in the descriptor set at the draw time. See `GR_DESCRIPTOR_SET_SLOT_TYPE`. Only `GR_SLOT_SHADER_RESOURCE` and `GR_SLOT_SHADER_UAV` values are valid for dynamic memory view.

shaderEntityIndex

The shader entity index.

GR_EVENT_CREATE_INFO

Event object creation information.

```
typedef struct _GR_EVENT_CREATE_INFO
{
    GR_FLAGS flags;
} GR_EVENT_CREATE_INFO;
```

Members

flags

Reserved, must be zero.

GR_EXTENT2D

The width and height for a 2D image region.

```
typedef struct _GR_EXTENT2D
{
    GR_INT width;
    GR_INT height;
} GR_EXTENT2D;
```

Members

width

The width for a 2D image.

height

The height for a 2D image.

GR_EXTENT3D

The width, height, and depth for a 3D image region.

```
typedef struct _GR_EXTENT3D
{
    GR_INT width;
    GR_INT height;
    GR_INT depth;
} GR_EXTENT3D;
```

Members

width

The width for a 3D image region.

height

The height for a 3D image region.

depth

The depth for a 3D image region.

GR_FENCE_CREATE_INFO

Fence object creation information.

```
typedef struct _GR_FENCE_CREATE_INFO
{
    GR_FLAGS flags;
} GR_FENCE_CREATE_INFO;
```

Members

flags

Reserved, must be zero.

GR_FORMAT

Image or memory view format.

```
typedef struct _GR_FORMAT
{
    GR_UINT32 channelFormat : 16;
    GR_UINT32 numericFormat : 16;
} GR_FORMAT;
```

Members

channelFormat

The channel format. See `GR_CHANNEL_FORMAT`.

numericFormat

The numeric format. See `GR_NUM_FORMAT`.

GR_FORMAT_PROPERTIES

Reported format properties for different tiling modes.

```
typedef struct _GR_FORMAT_PROPERTIES
{
    GR_FLAGS linearTilingFeatures;
    GR_FLAGS optimalTilingFeatures;
} GR_FORMAT_PROPERTIES;
```

Members

linearTilingFeatures

Format properties for images of linear tiling and memory views. See `GR_FORMAT_FEATURE_FLAGS`.

optimalTilingFeatures

Format properties for images of optimal tiling. See `GR_FORMAT_FEATURE_FLAGS`.

GR_GPU_COMPATIBILITY_INFO

Cross-GPU compatibility information.

```
typedef struct _GR_GPU_COMPATIBILITY_INFO
{
    GR_FLAGS compatibilityFlags;
} GR_GPU_COMPATIBILITY_INFO;
```

Members

compatibilityFlags

Cross-GPU compatibility flags. See `GR_GPU_COMPATIBILITY_FLAGS`.

GR_GRAPHICS_PIPELINE_CREATE_INFO

Graphics pipeline creation information.

```
typedef struct _GR_GRAPHICS_PIPELINE_CREATE_INFO
{
    GR_PIPELINE_SHADER     vs;
    GR_PIPELINE_SHADER     hs;
    GR_PIPELINE_SHADER     ds;
    GR_PIPELINE_SHADER     gs;
    GR_PIPELINE_SHADER     ps;
    GR_PIPELINE_IA_STATE   iaState;
    GR_PIPELINE_TESS_STATE tessState;
    GR_PIPELINE_RS_STATE   rsState;
    GR_PIPELINE_CB_STATE   cbState;
    GR_PIPELINE_DB_STATE   dbState;
    GR_FLAGS               flags;
} GR_GRAPHICS_PIPELINE_CREATE_INFO;
```

Members

vs

Vertex shader information. See GR_PIPELINE_SHADER.

hs

Hull shader information. See GR_PIPELINE_SHADER.

ds

Domain shader information. See GR_PIPELINE_SHADER.

gs

Geometry shader information. See GR_PIPELINE_SHADER.

ps

Pixel shader information. See GR_PIPELINE_SHADER.

iaState

Input assembler static pipeline state. See GR_PIPELINE_IA_STATE.

tessState

Tessellator static pipeline state. See GR_PIPELINE_TESS_STATE.

rsState

Rasterizer static pipeline state. See GR_PIPELINE_RS_STATE.

cbState

Color blender and output static pipeline state. See GR_PIPELINE_CB_STATE.

dbState

Depth-stencil static pipeline state. See GR_PIPELINE_DB_STATE.

flags

Pipeline creation flags. See GR_PIPELINE_CREATE_FLAGS.

GR_IMAGE_COPY

Image to image region copy description.

```
typedef struct _GR_IMAGE_COPY
{
    GR_IMAGE_SUBRESOURCE srcSubresource;
    GR_OFFSET3D          srcOffset;
    GR_IMAGE_SUBRESOURCE destSubresource;
    GR_OFFSET3D          destOffset;
    GR_EXTENT3D          extent;
} GR_IMAGE_COPY;
```

Members

srcSubresource

Source image subresource. See GR_IMAGE_SUBRESOURCE.

srcOffset

Texel offset in the source subresource. For compressed images use compression blocks instead of texels. See GR_OFFSET3D.

destSubresource

Destination image subresource. See GR_IMAGE_SUBRESOURCE.

destOffset

Texel offset in the destination subresource. For compressed images, use compression blocks instead of texels. See GR_OFFSET3D.

extent

Texel dimensions of the image region to copy. For compressed images, use compression blocks instead of texels. See GR_EXTENT3D.

GR_IMAGE_CREATE_INFO

Image creation information.

```
typedef struct _GR_IMAGE_CREATE_INFO
{
    GR_ENUM     imageType;
    GR_FORMAT   format;
    GR_EXTENT3D extent;
    GR_UINT     mipLevels;
    GR_UINT     arraySize;
    GR_UINT     samples;
    GR_ENUM     tiling;
    GR_FLAGS    usage;
    GR_FLAGS    flags;
} GR_IMAGE_CREATE_INFO;
```

Members

imageType

Image type (1D, 2D or 3D). See `GR_INDEX_TYPE`.

format

Image format. See `GR_FORMAT`.

extent

Image dimensions in texels. See `GR_EXTENT3D`.

mipLevels

Number of mipmap levels. Cannot be zero.

arraySize

Array size. Use value of one for non-array images. Cannot be zero.

samples

Number of coverage samples. Use value of one for non-multisampled images.

tiling

Image tiling. See `GR_IMAGE_TILING`.

usage

Image usage flags. See `GR_IMAGE_USAGE_FLAGS`.

flags

Image creation flags. See `GR_IMAGE_CREATE_FLAGS`.

GR_IMAGE_RESOLVE

Image resolve region description.

```
typedef struct _GR_IMAGE_RESOLVE
{
    GR_IMAGE_SUBRESOURCE srcSubresource;
    GR_OFFSET2D          srcOffset;
    GR_IMAGE_SUBRESOURCE destSubresource;
    GR_OFFSET2D          destOffset;
    GR_EXTENT2D          extent;
} GR_IMAGE_RESOLVE;
```

Members

srcSubresource

Subresource in multisampled source image. See `GR_IMAGE_SUBRESOURCE`.

srcOffset

Texel offset in the source subresource. See `GR_OFFSET2D`.

destSubresource

Subresource in non-multisampled destination image. See `GR_IMAGE_SUBRESOURCE`.

destOffset

Texel offset in the destination subresource. See `GR_OFFSET2D`.

extent

Texel dimensions of the image region to resolve. See `GR_EXTENT2D`.

GR_IMAGE_STATE_TRANSITION

Description of image state transition for a range of subresources.

```
typedef struct _GR_IMAGE_STATE_TRANSITION
{
    GR_IMAGE                   image;
    GR_ENUM                    oldState;
    GR_ENUM                    newState;
    GR_IMAGE_SUBRESOURCE_RANGE subresourceRange;
} GR_IMAGE_STATE_TRANSITION;
```

Members

image

Image object to use for state transition.

oldState

Previous image state. See `GR_IMAGE_STATE`.

newState

New image state. See GR_IMAGE_STATE.

subresourceRange

Images subresource range. See GR_IMAGE_SUBRESOURCE_RANGE.

GR_IMAGE_SUBRESOURCE

Image subresource identifier.

```
typedef struct _GR_IMAGE_SUBRESOURCE
{
    GR_ENUM aspect;
    GR_UINT mipLevel;
    GR_UINT arraySlice;
} GR_IMAGE_SUBRESOURCE;
```

Members

aspect

Image aspect the subresource belongs to. See GR_IMAGE_ASPECT.

mipLevel

Image mipmap level for the subresource.

arraySlice

Image array slice for the subresource.

GR_IMAGE_SUBRESOURCE_RANGE

Defines a range of subresources within an image aspect.

```
typedef struct _GR_IMAGE_SUBRESOURCE_RANGE
{
    GR_ENUM aspect;
    GR_UINT baseMipLevel;
    GR_UINT mipLevels;
    GR_UINT baseArraySlice;
    GR_UINT arraySize;
} GR_IMAGE_SUBRESOURCE_RANGE;
```

Members

aspect

Image aspect the subresource range belongs to. See GR_IMAGE_ASPECT.

baseMipLevel

Base image mipmap level for the subresource range.

mipLevels

Number of image mipmap levels in the subresource range. Use `GR_LAST_MIP_OR_SLICE` to specify the range of mipmap levels from `baseMipLevel` to the last one available in in the image.

baseArraySlice

Base image array slice for the subresource range.

arraySize

Number of image array slices in the subresource range. Use `GR_LAST_MIP_OR_SLICE` to specify the range of array slices from `baseArraySlice` to the last one available in in the image.

GR_IMAGE_VIEW_ATTACH_INFO

Image view description for attachment to descriptor set slots.

```
typedef struct _GR_IMAGE_VIEW_ATTACH_INFO
{
    GR_IMAGE_VIEW view;
    GR_ENUM       state;
} GR_IMAGE_VIEW_ATTACH_INFO;
```

Members

view

Image view object.

state

Image state for the view subresources at the draw time. See `GR_IMAGE_STATE`.

GR_IMAGE_VIEW_CREATE_INFO

Image view creation information.

```
typedef struct _GR_IMAGE_VIEW_CREATE_INFO
{
    GR_IMAGE                      image;
    GR_ENUM                       viewType;
    GR_FORMAT                     format;
    GR_CHANNEL_MAPPING            channels;
    GR_IMAGE_SUBRESOURCE_RANGE subresourceRange;
    GR_FLOAT                      minLod;
} GR_IMAGE_VIEW_CREATE_INFO;
```

Members

image

Image for the view.

viewType

View type matching the image topology. See GR_IMAGE_VIEW_TYPE.

format

Image format for the view; has to be compatible with the format of the image. See GR_FORMAT.

channels

Channel swizzle. See GR_CHANNEL_MAPPING.

subresourceRange

Contiguous range of subresources to use for the image view. See GR_IMAGE_SUBRESOURCE_RANGE.

minLod

Highest-resolution mipmap level available for access through the view.

GR_LINK_CONST_BUFFER

Constant data for link-time pipeline optimizations.

```
typedef struct _GR_LINK_CONST_BUFFER
{
    GR_UINT         bufferId;
    GR_SIZE         bufferSize;
    const GR_VOID* pBufferData;
} GR_LINK_CONST_BUFFER;
```

Members

bufferId

Constant buffer ID to match references in IL shader.

bufferSize

Constant buffer size in bytes (has to be a multiple of 16-bytes).

pBufferData

Pointer to application provided link time constant buffer data.

GR_MEMORY_ALLOC_INFO

GPU memory allocation information.

```
typedef struct _GR_MEMORY_ALLOC_INFO
{
    GR_GPU_SIZE size;
    GR_GPU_SIZE alignment;
    GR_FLAGS    flags;
    GR_UINT     heapCount;
    GR_UINT     heaps[GR_MAX_MEMORY_HEAPS];
    GR_ENUM     memPriority;
} GR_MEMORY_ALLOC_INFO;
```

Member

size

The size of the GPU memory allocation in bytes.

alignment

Optional GPU memory alignment in bytes. Must be multiple of the biggest page size.

flags

The flags for the memory allocation. See `GR_MEMORY_ALLOC_FLAGS`.

heapCount

The number of GPU memory heaps allowed for allocation placement.

heaps

An array of memory heap IDs allowed for allocation placement. The order of heap IDs defines preferred placement priority for the GPU memory heap selection.

memPriority

The memory priorities for the allocation at creation time. See `GR_MEMORY_PRIORITY`.

GR_MEMORY_COPY

Memory to memory copy region information.

```
typedef struct _GR_MEMORY_COPY
{
    GR_GPU_SIZE srcOffset;
    GR_GPU_SIZE destOffset;
    GR_GPU_SIZE copySize;
} GR_MEMORY_COPY;
```

Members

srcOffset

Byte offset in the source memory object.

destOffset

Byte offset in the destination memory object.

copySize

Copy region in bytes.

GR_MEMORY_HEAP_PROPERTIES

Memory heap properties.

```
typedef struct _GR_MEMORY_HEAP_PROPERTIES
{
    GR_ENUM     heapMemoryType;
    GR_GPU_SIZE heapSize;
    GR_GPU_SIZE pageSize;
    GR_FLAGS    flags;
    GR_FLOAT    gpuReadPerfRating;
    GR_FLOAT    gpuWritePerfRating;
    GR_FLOAT    cpuReadPerfRating;
    GR_FLOAT    cpuWritePerfRating;
} GR_MEMORY_HEAP_PROPERTIES;
```

Members

heapMemoryType

The GPU memory heap type. See `GR_HEAP_MEMORY_TYPE`.

heapSize

The size of the GPU memory heap in bytes.

pageSize

The page size the GPU memory heap in bytes.

flags

GPU memory heap property flags. See `GR_MEMORY_HEAP_FLAGS`.

gpuReadPerfRating

Relative heap performance rating for GPU reads.

gpuWritePerfRating

Relative heap performance rating for GPU writes.

cpuReadPerfRating

Relative heap performance rating for CPU reads.

cpuWritePerfRating

Relative heap performance rating for CPU writes.

GR_MEMORY_IMAGE_COPY

Memory to image and image to memory copy region description.

```
typedef struct _GR_MEMORY_IMAGE_COPY
{
    GR_GPU_SIZE          memOffset;
    GR_IMAGE_SUBRESOURCE imageSubresource;
    GR_OFFSET3D          imageOffset;
    GR_EXTENT3D          imageExtent;
} GR_MEMORY_IMAGE_COPY;
```

Members

memOffset

Byte offset in the memory object.

imageSubresource

Image subresource to use for copy. See GR_IMAGE_SUBRESOURCE.

imageOffset

Texel offset in the image subresource. For compressed images, use compression blocks instead of texels. See GR_OFFSET3D.

imageExtent

Texel dimensions of the image region to copy. For compressed images, use compression blocks instead of texels. See GR_EXTENT3D.

GR_MEMORY_OPEN_INFO

Parameters for opening shared GPU memory object on another device.

```
typedef struct _GR_MEMORY_OPEN_INFO
{
    GR_GPU_MEMORY sharedMem;
} GR_MEMORY_OPEN_INFO;
```

Members

sharedMem

The handle of a shared GPU memory object from another device to open.

GR_MEMORY_REF

Information about memory object reference in command buffer for submission.

```
typedef struct _GR_MEMORY_REF
{
    GR_GPU_MEMORY mem;
    GR_FLAGS      flags;
} GR_MEMORY_REF;
```

Members

mem

Memory object for the reference.

flags

Memory reference flags. See GR_MEMORY_REF_FLAGS.

GR_MEMORY_REQUIREMENTS

Memory binding requirements for an object.

```
typedef struct _GR_MEMORY_REQUIREMENTS
{
    GR_GPU_SIZE size;
    GR_GPU_SIZE alignment;
    GR_UINT     heapCount;
    GR_UINT     heaps[GR_MAX_MEMORY_HEAPS];
} GR_MEMORY_REQUIREMENTS;
```

Members

size

GPU memory size in bytes required for object storage.

alignment

Memory alignment in bytes.

heapCount

Number of valid entries returned in heaps array.

heaps

Array of returned heap IDs for all heaps that can be used for the object placement.

GR_MEMORY_STATE_TRANSITION

Defines memory state transition for a range of memory.

```
typedef struct _GR_MEMORY_STATE_TRANSITION
{
    GR_GPU_MEMORY mem;
    GR_ENUM       oldState;
    GR_ENUM       newState;
    GR_GPU_SIZE   offset;
    GR_GPU_SIZE   regionSize;
} GR_MEMORY_STATE_TRANSITION;
```

Members

mem

GPU memory object to use for state transition.

oldState

Previous memory state for the range. See GR_MEMORY_STATE.

newState

New memory state for the range. See GR_MEMORY_STATE.

offset

Byte offset within the GPU memory object that defines the beginning of the memory range for state transition.

regionSize

GPU memory region size in bytes to use for state transition.

GR_MEMORY_VIEW_ATTACH_INFO

Memory view description for attachment to descriptor set slots.

```
typedef struct _GR_MEMORY_VIEW_ATTACH_INFO
{
    GR_GPU_MEMORY mem;
    GR_GPU_SIZE   offset;
    GR_GPU_SIZE   range;
    GR_GPU_SIZE   stride;
    GR_FORMAT     format;
    GR_ENUM       state;
} GR_MEMORY_VIEW_ATTACH_INFO;
```

Members

mem

GPU memory object to use for memory view.

offset

Byte offset within the GPU memory object to the beginning of memory view.

range

Memory range in bytes for the memory view.

stride

Element stride for the memory view.

format

Optional format for typed memory views. See GR_FORMAT.

state

Current memory state for the memory view range. See GR_MEMORY_STATE.

GR_MSAA_STATE_CREATE_INFO

Dynamic multisampling state creation information.

```
typedef struct _GR_MSAA_STATE_CREATE_INFO
{
    GR_UINT        samples;
    GR_SAMPLE_MASK sampleMask;
} GR_MSAA_STATE_CREATE_INFO;
```

Members

samples

Number of samples.

sampleMask

Sample bit-mask. Determines which samples in color targets are updated. Lower bit represents sample zero.

GR_OFFSET2D

The 2D image coordinate offset for image manipulation.

```
typedef struct _GR_OFFSET2D
{
    GR_INT x;
    GR_INT y;
} GR_OFFSET2D;
```

Members

x

The x coordinate for the offset.

y

The y coordinate for the offset.

GR_OFFSET3D

The 3D image coordinate offset for image manipulation.

```
typedef struct _GR_OFFSET3D
{
    GR_INT x;
    GR_INT y;
    GR_INT z;
} GR_OFFSET3D;
```

Members

x

The x coordinate for the offset.

y

The y coordinate for the offset.

z

The z coordinate for the offset.

GR_PARENT_DEVICE

Information about the parent device for an API object.

```
typedef struct _GR_PARENT_DEVICE
{
    GR_DEVICE device;
} GR_PARENT_DEVICE;
```

Members

device

The handle of a parent device.

GR_PARENT_PHYSICAL_GPU

Information about parent physical GPU for a device object.

```
typedef struct _GR_PARENT_PHYSICAL_GPU
{
    GR_PHYSICAL_GPU gpu;
} GR_PARENT_PHYSICAL_GPU;
```

Members

gpu

The handle of a parent physical GPU object.

GR_PEER_IMAGE_OPEN_INFO

Parameters for opening image object on another device for peer-to-peer image transfers.

```
typedef struct _GR_PEER_IMAGE_OPEN_INFO
{
    GR_IMAGE originalImage;
} GR_PEER_IMAGE_OPEN_INFO;
```

Members

originalImage

The handle of an image object from another device to open for peer-to-peer image transfers.

GR_PEER_MEMORY_OPEN_INFO

Parameters for opening the GPU memory object on another device for peer-to-peer memory transfers.

```
typedef struct _GR_PEER_MEMORY_OPEN_INFO
{
    GR_GPU_MEMORY originalMem;
} GR_PEER_MEMORY_OPEN_INFO;
```

Members

originalMem

The handle of a GPU memory object from another device to open for peer-to-peer memory transfers.

GR_PHYSICAL_GPU_IMAGE_PROPERTIES

Image support capabilities of a physical GPU object.

```
typedef struct _GR_PHYSICAL_GPU_IMAGE_PROPERTIES
{
    GR_UINT     maxSliceWidth;
    GR_UINT     maxSliceHeight;
    GR_UINT     maxDepth;
    GR_UINT     maxArraySlices;
    GR_UINT     reserved1;
    GR_UINT     reserved2;
    GR_GPU_SIZE maxMemoryAlignment;
    GR_UINT32   sparseImageSupportLevel;
    GR_FLAGS    flags;
} GR_PHYSICAL_GPU_IMAGE_PROPERTIES;
```

Members

maxSliceWidth

Maximum image slice width in texels.

maxSliceHeight

Maximum image slice height in texels.

maxDepth

Maximum 3D image depth.

maxArraySlices

Maximum number of slices in an image array.

reserved1

Reserved.

reserved2

Reserved.

maxMemoryAlignment

Maximum memory alignment requirements any image can have in bytes.

sparseImageSupportLevel

Sparse image support level.

flags

Reserved.

GR_PHYSICAL_GPU_MEMORY_PROPERTIES

Memory management capabilities of a physical GPU object.

```
typedef struct _GR_PHYSICAL_GPU_MEMORY_PROPERTIES
{
    GR_FLAGS    flags;
    GR_GPU_SIZE virtualMemPageSize;
    GR_GPU_SIZE maxVirtualMemSize;
    GR_GPU_SIZE maxPhysicalMemSize;
} GR_PHYSICAL_GPU_MEMORY_PROPERTIES;
```

Members

flags

The GPU memory manager capability flags. See `GR_MEMORY_PROPERTY_FLAGS`.

virtualMemPageSize

The virtual memory page size for the GPU. Zero if virtual memory remapping is not supported.

maxVirtualMemSize

The upper bound of the address range available for creation of virtual memory objects. Zero if virtual memory remapping is not supported or if unknown.

maxPhysicalMemSize

The upper bound of all GPU accessible memory in the system. Zero if unknown.

GR_PHYSICAL_GPU_PERFORMANCE

Performance properties of a physical GPU object. Provides rough performance estimates for the GPU performance.

```
typedef struct _GR_PHYSICAL_GPU_PERFORMANCE
{
    GR_FLOAT maxGpuClock;
    GR_FLOAT aluPerClock;
    GR_FLOAT texPerClock;
    GR_FLOAT primsPerClock;
    GR_FLOAT pixelsPerClock;
} GR_PHYSICAL_GPU_PERFORMANCE;
```

Members

maxGpuClock

The maximum GPU engine clock in MHz.

aluPerClock

The maximum number of shader ALU operations per clock.

texPerClock

The maximum number of texture fetches per clock.

primsPerClock

The maximum number of processed geometry primitives per clock.

pixelsPerClock

The maximum number of processed pixels per clock.

GR_PHYSICAL_GPU_PROPERTIES

General properties of a physical GPU object.

```
typedef struct _GR_PHYSICAL_GPU_PROPERTIES
{
    GR_UINT32   apiVersion;
    GR_UINT32   driverVersion;
    GR_UINT32   vendorId;
    GR_UINT32   deviceId;
    GR_ENUM     gpuType;
    GR_CHAR     gpuName[GR_MAX_PHYSICAL_GPU_NAME];
    GR_UINT     maxMemRefsPerSubmission;
    GR_GPU_SIZE reserved;
    GR_GPU_SIZE maxInlineMemoryUpdateSize;
    GR_UINT     maxBoundDescriptorSets;
    GR_UINT     maxThreadGroupSize;
    GR_UINT64   timestampFrequency;
    GR_BOOL     multiColorTargetClears;
} GR_PHYSICAL_GPU_PROPERTIES;
```

Members

apiVersion

The Mantle API version supported by the GPU.

driverVersion

The driver version.

vendorId

The vendor ID of the GPU.

deviceId

The device ID of the GPU.

gpuType

The GPU type. See `GR_PHYSICAL_GPU_TYPE`.

gpuName

A string with the GPU description.

maxMemRefsPerSubmission

The maximum number of memory references per submission for the GPU.

reserved

Reserved.

maxInlineMemoryUpdateSize

The maximum inline memory update size for the GPU.

maxBoundDescriptorSets

The maximum number of bound descriptor sets for the GPU.

maxThreadGroupSize

The maximum compute thread group size for the GPU.

timestampFrequency

The timestamp frequency for the GPU in Hz.

multiColorTargetClears

A flag indicating support of multiple color target clears for the GPU.

GR_PHYSICAL_GPU_QUEUE_PROPERTIES

Queue type properties for a physical GPU.

```
typedef struct _GR_PHYSICAL_GPU_QUEUE_PROPERTIES
{
    GR_ENUM queueType;
    GR_UINT queueCount;
    GR_UINT maxAtomicCounters
    GR_BOOL supportsTimestamps;
} GR_PHYSICAL_GPU_QUEUE_PROPERTIES;
```

Members

queueType

The type of queue. See `GR_QUEUE_TYPE`.

queueCount

The maximum available queue count.

maxAtomicCounters

The maximum number of atomic counters available for the queues of the given type.

supportsTimestamps

The timestamps support flag for the queues of the given type.

GR_PIPELINE_CB_STATE

Static color blender and output state for pipeline.

```
typedef struct _GR_PIPELINE_CB_STATE
{
    GR_BOOL                     alphaToCoverageEnable;
    GR_BOOL                     dualSourceBlendEnable;
    GR_ENUM                     logicOp;
    GR_PIPELINE_CB_TARGET_STATE target[GR_MAX_COLOR_TARGETS];
} GR_PIPELINE_CB_STATE;
```

Members

alphaToCoverageEnable

Alpha to coverage enable.

dualSourceBlendEnable

The blend state used at the draw time specifies the dual source blend mode.

logicOp

Logic operation to perform. See `GR_LOGIC_OP`.

target

Per color target description of the state. See GR_PIPELINE_CB_TARGET_STATE.

GR_PIPELINE_CB_TARGET_STATE

Per color target description of the color blender and output state for pipeline.

```
typedef struct _GR_PIPELINE_CB_TARGET_STATE
{
    GR_BOOL   blendEnable;
    GR_FORMAT format;
    GR_UINT8  channelWriteMask;
} GR_PIPELINE_CB_TARGET_STATE;
```

Members

blendEnable

Blend enable for color target.

format

Color target format at the draw time. Should match the actual target format used for rendering. See GR_FORMAT.

channelWriteMask

Color target write mask. Each bit controls a color channel in R, G, B, A order, with bit 0 controlling the red channel and so on.

GR_PIPELINE_DB_STATE

Static depth-stencil state for pipeline.

```
typedef struct _GR_PIPELINE_DB_STATE
{
    GR_FORMAT format;
} GR_PIPELINE_DB_STATE;
```

Members

format

Depth-stencil target format at the draw time. Should match the actual depth-stencil format used for rendering. See GR_FORMAT.

GR_PIPELINE_IA_STATE

Static input assembler state for pipeline.

```
typedef struct _GR_PIPELINE_IA_STATE
{
    GR_ENUM topology;
    GR_BOOL disableVertexReuse;
} GR_PIPELINE_IA_STATE;
```

Members

topology

Primitive topology. See GR_PRIMITIVE_TOPOLOGY.

disableVertexReuse

Provides ability to disable vertex reuse in indexed draws when set to GR_TRUE (disables post-transform cache).

GR_PIPELINE_RS_STATE

Static rasterizer state for pipeline.

```
typedef struct _GR_PIPELINE_RS_STATE
{
    GR_BOOL depthClipEnable;
} GR_PIPELINE_RS_STATE;
```

Members

depthClipEnable

Depth clip functionality enable.

GR_PIPELINE_SHADER

Definition of the shader and its resource mappings to descriptor sets and dynamic memory view for programmable pipeline stages.

```
typedef struct _GR_PIPELINE_SHADER
{
    GR_SHADER                        shader;
    GR_DESCRIPTOR_SET_MAPPING        descriptorSetMapping[GR_MAX_DESCRIPTOR_SETS];
    GR_UINT                          linkConstBufferCount;
    const GR_LINK_CONST_BUFFER* pLinkConstBufferInfo;
    GR_DYNAMIC_MEMORY_VIEW_SLOT_INFO dynamicMemoryViewMapping;
} GR_PIPELINE_SHADER;
```

Members

shader

Shader object to be used for the pipeline stage.

descriptorSetMapping

Array of descriptor set mapping information. One entry per descriptor set bind point. See GR_DESCRIPTOR_SET_MAPPING.

linkConstBufferCount

Number of link-time constant buffers.

pLinkConstBufferInfo

Array of constant data structures. One constant data structure per link-time constant buffer. See GR_LINK_CONST_BUFFER.

dynamicMemoryViewMapping

Mapping of dynamic memory view to shader entity. See GR_DYNAMIC_MEMORY_VIEW_SLOT_INFO.

GR_PIPELINE_STATISTICS_DATA

Result of pipeline statistics query.

```
typedef struct _GR_PIPELINE_STATISTICS_DATA
{
    GR_UINT64 psInvocations;
    GR_UINT64 cPrimitives;
    GR_UINT64 cInvocations;
    GR_UINT64 vsInvocations;
    GR_UINT64 gsInvocations;
    GR_UINT64 gsPrimitives;
    GR_UINT64 iaPrimitives;
    GR_UINT64 iaVertices;
    GR_UINT64 hsInvocations;
    GR_UINT64 dsInvocations;
    GR_UINT64 csInvocations;
} GR_PIPELINE_STATISTICS_DATA;
```

Members

psInvocations

Pixel shader invocations.

cPrimitives

Clipper primitives.

cInvocations

Clipper invocations.

vsInvocations

Vertex shader invocations.

gsInvocations

Geometry shader invocations.

gsPrimitives

Geometry shader primitives.

iaPrimitives

Input primitives.

iaVertices

Input vertices.

hsInvocations

Hull shader invocations.

dsInvocations

Domain shader invocations.

csInvocations

Compute shader invocations.

GR_PIPELINE_TESS_STATE

Static tessellator state for pipeline.

```
typedef struct _GR_PIPELINE_TESS_STATE
{
    GR_UINT  patchControlPoints;
    GR_FLOAT optimalTessFactor;
} GR_PIPELINE_TESS_STATE;
```

Members

patchControlPoints

Number of control points per patch.

optimalTessFactor

Tessellation factor to optimize pipeline operation for.

GR_QUERY_POOL_CREATE_INFO

Query pool creation information.

```
typedef struct _GR_QUERY_POOL_CREATE_INFO
{
    GR_ENUM queryType;
    GR_UINT slots;
} GR_QUERY_POOL_CREATE_INFO;
```

Members

queryType

Type of the queries that are used with this query pool. Queries of only one type can be present in the query pool. See GR_QUERY_TYPE.

slots

Number of query slots in the pool.

GR_QUEUE_SEMAPHORE_CREATE_INFO

Queue semaphore creation information.

```
typedef struct _GR_QUEUE_SEMAPHORE_CREATE_INFO
{
    GR_UINT  initialCount;
    GR_FLAGS flags;
} GR_QUEUE_SEMAPHORE_CREATE_INFO;
```

Members

initialCount

Initial queue semaphore count. Value must be in [0..31] range.

flags

Semaphore creation flags. See GR_SEMAPHORE_CREATE_FLAGS.

GR_QUEUE_SEMAPHORE_OPEN_INFO

Parameters for opening a shared queue semaphore on another device.

```
typedef struct _GR_QUEUE_SEMAPHORE_OPEN_INFO
{
    GR_QUEUE_SEMAPHORE sharedSemaphore;
} GR_QUEUE_SEMAPHORE_OPEN_INFO;
```

Members

sharedSemaphore

The handle of a shared queue semaphore from another device to open.

GR_RASTER_STATE_CREATE_INFO

Dynamic rasterizer state creation information.

```
typedef struct _GR_RASTER_STATE_CREATE_INFO
{
    GR_ENUM  fillMode;
    GR_ENUM  cullMode;
    GR_ENUM  frontFace;
    GR_INT   depthBias;
    GR_FLOAT depthBiasClamp;
    GR_FLOAT slopeScaledDepthBias;
} GR_RASTER_STATE_CREATE_INFO;
```

Members

fillMode

Fill mode. See `GR_FILL_MODE`.

cullMode

Cull mode. See `GR_CULL_MODE`.

frontFace

Front face orientation. See `GR_FACE_ORIENTATION`.

depthBias

Value added to pixel depth.

depthBiasClamp

Maximum depth bias value.

slopeScaledDepthBias

Scale of the slope-based value added to pixel depth.

GR_RECT

A rectangle region for 2D image.

```
typedef struct _GR_RECT
{
    GR_OFFSET2D offset;
    GR_EXTENT2D extent;
} GR_RECT;
```

Members

offset

The rectangle region offset. See `GR_OFFSET2D`.

extent

The extent of the rectangle region. See GR_EXTENT2D.

GR_SAMPLER_CREATE_INFO

Sampler creation information.

```
typedef struct _GR_SAMPLER_CREATE_INFO
{
    GR_ENUM  filter;
    GR_ENUM  addressU;
    GR_ENUM  addressV;
    GR_ENUM  addressW;
    GR_FLOAT mipLodBias;
    GR_UINT  maxAnisotropy;
    GR_ENUM  compareFunc;
    GR_FLOAT minLod;
    GR_FLOAT maxLod;
    GR_ENUM  borderColor;
} GR_SAMPLER_CREATE_INFO;
```

Members

filter

Filtering to apply to texture fetches. See GR_TEX_FILTER.

addressU

Texture addressing mode for outside of the [0..1] range for U texture coordinate. See GR_TEX_ADDRESS.

addressV

Texture addressing mode for outside of the [0..1] range for V texture coordinate. See GR_TEX_ADDRESS.

addressW

Texture addressing mode for outside of the [0..1] range for W texture coordinate. See GR_TEX_ADDRESS.

mipLodBias

LOD bias.

maxAnisotropy

Anisotropy value clamp when filter mode is GR_TEX_FILTER_ANISOTROPIC.

compareFunc

Comparison function to apply to fetched data. See GR_COMPARE_FUNC.

minLod

Highest-resolution mipmap level available for access.

maxLod

Lowest-resolution mipmap level available for access; has to be greater or equal to `minLod`.

borderColor

One of predefined border color values (white, transparent black or opaque black). See `GR_BORDER_COLOR_TYPE`.

GR_SHADER_CREATE_INFO

Shader creation information.

```
typedef struct _GR_SHADER_CREATE_INFO
{
    GR_SIZE         codeSize;
    const GR_VOID*  pCode;
    GR_FLAGS        flags;
} GR_SHADER_CREATE_INFO;
```

Members

codeSize

Input shader code size in bytes.

pCode

Pointer to the input shader binary code.

flags

Shader creation flags. See `GR_SHADER_CREATE_FLAGS`.

GR_SUBRESOURCE_LAYOUT

Subresource layout returned for a subresource.

```
typedef struct _GR_SUBRESOURCE_LAYOUT
{
    GR_GPU_SIZE offset;
    GR_GPU_SIZE size;
    GR_GPU_SIZE rowPitch;
    GR_GPU_SIZE depthPitch;
} GR_SUBRESOURCE_LAYOUT;
```

Members

offset

Byte offset of the subresource data relative to the beginning of memory associated with an image object.

size

Subresource size in bytes.

rowPitch

Row pitch in bytes. For opaque resources reported pitch is zero.

depthPitch

Depth pitch for image arrays and 3D images in bytes. For opaque resources, reported pitch is zero.

GR_VIEWPORT

Defines dimensions of a viewport.

```
typedef struct _GR_VIEWPORT
{
    GR_FLOAT originX;
    GR_FLOAT originY;
    GR_FLOAT width;
    GR_FLOAT height;
    GR_FLOAT minDepth;
    GR_FLOAT maxDepth;
} GR_VIEWPORT;
```

Members

originX

The x coordinate for the origin of the viewport.

originY

The y coordinate for the origin of the viewport.

width

The width of the viewport.

height

The height of the viewport.

minDepth

The minimum depth value of the viewport. The valid range is [0..1].

maxDepth

The maximum depth value of the viewport. The valid range is [0..1]. The maximum viewport depth value has to be greater than minimum depth value.

GR_VIEWPORT_STATE_CREATE_INFO

Dynamic viewport and scissor state creation information.

```
typedef struct _GR_VIEWPORT_STATE_CREATE_INFO
{
    GR_UINT     viewportCount;
    GR_BOOL     scissorEnable;
    GR_VIEWPORT viewports[GR_MAX_VIEWPORTS];
    GR_RECT     scissors[GR_MAX_VIEWPORTS];
} GR_VIEWPORT_STATE_CREATE_INFO;
```

Members

viewportCount

Number of viewports.

scissorEnable

Scissor enable flag.

viewports

Array of viewports. See GR_VIEWPORT.

scissors

Array of scissors. See GR_RECT.

GR_VIRTUAL_MEMORY_REMAP_RANGE

Specified a range of pages in a virtual memory object for remapping to pages of real memory object.

```
typedef struct _GR_VIRTUAL_MEMORY_REMAP_RANGE
{
    GR_GPU_MEMORY virtualMem;
    GR_GPU_SIZE   virtualStartPage;
    GR_GPU_MEMORY realMem;
    GR_GPU_SIZE   realStartPage;
    GR_GPU_SIZE   pageCount;
} GR_VIRTUAL_MEMORY_REMAP_RANGE;
```

Members

virtualMem

A virtual memory object handle for page remapping.

virtualStartPage

First page of a virtual memory object in a remapped range.

realMem

Handle of a real memory object to which virtual memory object pages are remapped.

realStartPage

First page of a real memory object to which virtual memory pages are remapped.

pageCount

Number of pages in a range to remap.

CALLBACKS

GR_ALLOC_FUNCTION

Application callback to allocate a block of system memory.

```
typedef GR_VOID* (GR_STDCALL *GR_ALLOC_FUNCTION)(
    GR_SIZE size,
    GR_SIZE alignment,
    GR_ENUM allocType);
```

Parameters

size

System memory allocation size in bytes.

alignment

Allocation requirements in bytes.

allocType

System memory allocation type. See GR_SYSTEM_ALLOC_TYPE.

GR_FREE_FUNCTION

Application callback to free a block of system memory.

```
typedef GR_VOID (GR_STDCALL *GR_FREE_FUNCTION)(
    GR_VOID* pMem);
```

Parameters

pMem

System memory allocation to free. The allocation was previously created through the GR_ALLOC_FUNCTION callback.

Errors and Return Codes

GR_SUCCESS

The API call successfully completed.

GR_UNSUPPORTED

The API call successfully completed, but the requested feature is not available.

GR_NOT_READY

The API call successfully completed, but the result of the operation is not ready.

GR_TIMEOUT

The wait operation completed due to a timeout condition.

GR_EVENT_SET

The event is in the “set” state.

GR_EVENT_RESET

The event is in the “reset” state.

GR_ERROR_UNKNOWN

An unknown error was encountered during the operation.

GR_ERROR_UNAVAILABLE

The requested operation is unavailable at this time.

GR_ERROR_INITIALIZATION_FAILED

Cannot initialize the Mantle device or driver.

GR_ERROR_OUT_OF_MEMORY

Cannot complete the operation due to insufficient system memory.

GR_ERROR_OUT_OF_GPU_MEMORY

Cannot complete the operation due to insufficient video memory.

GR_ERROR_DEVICE_ALREADY_CREATED

Cannot create a device since there is already an active device for the same physical GPU.

GR_ERROR_DEVICE_LOST

The device was lost due to its removal or possible hang and recovery condition.

GR_ERROR_INVALID_POINTER

An invalid pointer was passed to the call.

GR_ERROR_INVALID_VALUE

An invalid value was passed to the call.

GR_ERROR_INVALID_HANDLE

An invalid API object handle was passed to the call.

GR_ERROR_INVALID_ORDINAL

An invalid ordinal value was passed to the call.

GR_ERROR_INVALID_MEMORY_SIZE

An invalid memory size was specified as an input parameter for the operation.

GR_ERROR_INVALID_EXTENSION

An invalid extension was requested during device creation.

GR_ERROR_INVALID_FLAGS

Invalid flags were passed to the call.

GR_ERROR_INVALID_ALIGNMENT

An invalid alignment was specified for the requested operation.

GR_ERROR_INVALID_FORMAT

An invalid resource format was specified.

GR_ERROR_INVALID_IMAGE

The requested operation cannot be performed on the provided image object.

GR_ERROR_INVALID_DESCRIPTOR_SET_DATA

The descriptor set data are invalid or does not match pipeline expectations.

GR_ERROR_INVALID_QUEUE_TYPE

An invalid queue type was specified for the requested operation.

GR_ERROR_INVALID_OBJECT_TYPE

An invalid object type was specified for the requested operation.

GR_ERROR_UNSUPPORTED_SHADER_IL_VERSION

Unsupported shader IL version.

GR_ERROR_BAD_SHADER_CODE

Corrupt or invalid shader code detected.

GR_ERROR_BAD_PIPELINE_DATA

Invalid pipeline data are detected.

GR_ERROR_TOO_MANY_MEMORY_REFERENCES

Too many memory references are used for this queue operation.

GR_ERROR_NOT_MAPPABLE

The memory object cannot be mapped as it does not reside in a CPU visible heap.

GR_ERROR_MEMORY_MAP_FAILED

The map operation failed due to an unknown or system reason.

GR_ERROR_MEMORY_UNMAP_FAILED

The unmap operation failed due to an unknown or system reason.

GR_ERROR_INCOMPATIBLE_DEVICE

The pipeline load operation failed due to an incompatible device.

GR_ERROR_INCOMPATIBLE_DRIVER

The pipeline load operation failed due to an incompatible driver version.

GR_ERROR_INCOMPLETE_COMMAND_BUFFER

The requested operation cannot be completed due to an incomplete command buffer construction.

GR_ERROR_BUILDING_COMMAND_BUFFER

The requested operation cannot be completed due to a failed command buffer construction.

GR_ERROR_MEMORY_NOT_BOUND

The operation cannot complete since not all objects have valid memory bound to them.

GR_ERROR_INCOMPATIBLE_QUEUE

The requested operation failed due to incompatible queue type.

GR_ERROR_NOT_SHAREABLE

The object cannot be created or opened for sharing between multiple GPU devices.

Chapter XIX.

Mantle Debug and Validation API Reference

Functions

grDbgSetValidationLevel

Sets the current validation level for the given device. The level cannot exceed the maximum validation level requested at device creation.

```
GR_RESULT GR_STDCALL grDbgSetValidationLevel(
    GR_DEVICE device,
    GR_ENUM   validationLevel);
```

Parameters

device

Device handle.

validationLevel

Requested validation level. See `GR_VALIDATION_LEVEL`.

Returns

`grDbgSetValidationLevel()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_VALUE if the validation level exceeds the maximum level requested at device creation
- GR_ERROR_INCOMPLETE_COMMAND_BUFFER if the application is in the middle of command buffer creation during this call
- GR_ERROR_UNAVAILABLE if the debug layer has not been enabled at device creation

Notes

Cannot be called while any command buffers are in the building state.

Thread safety

Not thread safe.

grDbgRegisterMsgCallback

Registers an error message callback function. Multiple callbacks can be registered simultaneously; however, the order of callback invocation is not guaranteed.

```
GR_RESULT grDbgRegisterMsgCallback(
    GR_DBG_MSG_CALLBACK_FUNCTION pfnMsgCallback,
    GR_VOID* pUserData);
```

Parameters

pfnMsgCallback

[in] User message callback function pointer. See GR_DBG_MSG_CALLBACK_FUNCTION.

pUserData

[in] Pointer to user data that needs to be passed to the callback. Can be NULL.

Returns

grDbgRegisterMsgCallback() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns the following error:

- GR_ERROR_INVALID_POINTER if the callback function pointer is invalid

Notes

It is allowed to register the same function multiple times without unregistering it first. This just replaces the old user data with a new one, keeping only one instance of the callback function registered.

This function does not generate debug message callbacks.

Thread safety

Not thread safe.

grDbgUnregisterMsgCallback

Unregisters a previously registered error message callback function.

```
GR_RESULT grDbgUnregisterMsgCallback(
    GR_DBG_MSG_CALLBACK_FUNCTION pfnMsgCallback);
```

Parameters

pfnMsgCallback

[in] User message callback function pointer.

Returns

`grDbgUnregisterMsgCallback()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns the following error:

- `GR_ERROR_INVALID_POINTER` if the callback function pointer is invalid or hasn't been previously registered

Notes

This function does not generate debug message callbacks.

Thread safety

Not thread safe.

grDbgSetMessageFilter

Enables filtering of a registered error message callback function for a specific message type. Multiple message types can be simultaneously filtered by calling this function multiple times. Debug message filtering does not affect returned error codes for any API functions.

```
GR_RESULT grDbgSetMessageFilter(
    GR_DEVICE device,
    GR_ENUM msgCode,
    GR_ENUM filter);
```

Parameters

device

Device handle.

msgCode

Message code to filter.

filter

Filter to apply to a particular message type. See `GR_DBG_MSG_FILTER`.

Returns

grDbgSetMessageFilter() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid.
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type.
- GR_ERROR_INVALID_VALUE if the message code or filter type is invalid.

Notes

Errors generated by the ICD loader cannot be filtered. The messages repetition status is kept globally per device. If multiple objects generate messages of the same type and the filter is set to GR_DBG_MSG_FILTER_REPEATED, then only the first message across these objects results in an application message callback.

Calling grDbgSetMessageFilter() with any filter type resets the message repetition state for the given message type.

This function does not generate debug message callbacks.

Thread safety

Not thread safe.

grDbgSetObjectTag

Attaches an application specific binary data object (tag) to any Mantle object, including devices, queues, and memory objects. Tags cannot be attached to physical GPU objects.

```
GR_RESULT grDbgSetObjectTag(
    GR_BASE_OBJECT object,
    GR_SIZE tagSize,
    const GR_VOID* pTag);
```

Parameters

object

Any Mantle object handle other than a physical GPU.

tagSize

Size of the binary tag to store with the object.

pTag

[in] Binary tag to attach to the object. Can be NULL.

Returns

grDbgSetObjectTag() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the object handle is invalid
- `GR_ERROR_INVALID_MEMORY_SIZE` if tag size is zero and tag pointer is not `NULL`

Notes

Object tagging is only available when the validation layer is enabled at any validation level. If the validation layer is disabled, the operation has no effect.

The driver makes an internal copy of the tag data when storing it with an object.

Specifying a `NULL` tag pointer removes the previously set tag data for the given object.

This function does not generate debug message callbacks.

Thread safety

Not thread safe for the tagged object.

grDbgSetGlobalOption

Sets global debug and validation options.

```
GR_RESULT grDbgSetGlobalOption(
    GR_DBG_GLOBAL_OPTION dbgOption,
    GR_SIZE dataSize,
    const GR_VOID* pData);
```

Parameters

dbgOption

Debug option being set. See `GR_DBG_GLOBAL_OPTION`.

dataSize

Data size being set for the debug option.

pData

[in] Data to be set for the debug option.

Returns

`grDbgSetGlobalOption()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_VALUE` if `dbgOption` is invalid
- `GR_ERROR_INVALID_VALUE` if the data are invalid for the given debug option
- `GR_ERROR_INVALID_MEMORY_SIZE` if the data size is invalid

Notes

None.

Thread safety

Not thread safe.

grDbgSetDeviceOption

Sets device-specific miscellaneous debug and validation options.

```
GR_RESULT grDbgSetDeviceOption(
    GR_DEVICE device,
    GR_DBG_DEVICE_OPTION dbgOption,
    GR_SIZE dataSize,
    const GR_VOID* pData);
```

Parameters

device

Device handle.

dbgOption

Debug option being set. See GR_DBG_DEVICE_OPTION.

dataSize

Data size being set for the debug option.

pData

[in] Data to be set for the debug option.

Returns

grDbgSetDeviceOption() returns GR_SUCCESS if the function executed successfully. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_VALUE if dbgOption is invalid
- GR_ERROR_INVALID_VALUE if the data are invalid for the given debug option
- GR_ERROR_INVALID_MEMORY_SIZE if the data size is invalid

Notes

None.

Thread safety

Not thread safe.

grCmdDbgMarkerBegin

Inserts a debug “begin” marker for command buffer debugger inspection.

```
GR_VOID GR_STDCALL grCmdDbgMarkerBegin(
    GR_CMD_BUFFER cmdBuffer,
    const GR_CHAR* pMarker);
```

Parameters

cmdBuffer

Command buffer handle.

pMarket

[in] Debug marker string.

Notes

None.

grCmdDbgMarkerEnd

Inserts a debug “end” marker for command buffer debugger inspection.

```
GR_VOID GR_STDCALL grCmdDbgMarkerEnd(
    GR_CMD_BUFFER cmdBuffer);
```

Parameters

cmdBuffer

Command buffer handle.

Notes

None.

ENUMERATIONS

GR_DBG_DATA_TYPE

Defines type of debug related data returned by validation layer.

```
typedef enum _GR_DBG_DATA_TYPE
{
    GR_DBG_DATA_OBJECT_TYPE           = 0x00020a00,
    GR_DBG_DATA_OBJECT_CREATE_INFO    = 0x00020a01,
    GR_DBG_DATA_OBJECT_TAG            = 0x00020a02,
    GR_DBG_DATA_CMD_BUFFER_API_TRACE = 0x00020b00,
    GR_DBG_DATA_MEMORY_OBJECT_LAYOUT = 0x00020c00,
    GR_DBG_DATA_MEMORY_OBJECT_STATE  = 0x00020c01,
    GR_DBG_DATA_SEMAPHORE_IS_BLOCKED = 0x00020d00,
} GR_DBG_DATA_TYPE;
```

Values

GR_DBG_DATA_OBJECT_TYPE

Retrieves object type with `grGetObjectInfo()`.

GR_DBG_DATA_OBJECT_CREATE_INFO

Retrieves object creation information with `grGetObjectInfo()`.

GR_DBG_DATA_OBJECT_TAG

Retrieves object debug tag with `grGetObjectInfo()`.

GR_DBG_DATA_CMD_BUFFER_API_TRACE

Retrieves recorded command buffer API trace with `grGetObjectInfo()`. Valid only for command buffer objects.

GR_DBG_DATA_MEMORY_OBJECT_LAYOUT

Retrieves ranges of memory object bindings with `grGetObjectInfo()`. Valid only for memory objects.

GR_DBG_DATA_MEMORY_OBJECT_STATE

Retrieves ranges of memory object state with `grGetObjectInfo()`. Valid only for memory objects.

GR_DBG_DATA_SEMAPHORE_IS_BLOCKED

Retrieves internal status of a semaphore with `grGetObjectInfo()`. Valid only for semaphore objects.

GR_DBG_DEVICE_OPTION

Defines per-device debug options available with validation layer.

```
typedef enum _GR_DBG_DEVICE_OPTION
{
    GR_DBG_OPTION_DISABLE_PIPELINE_LOADS          = 0x00020400,
    GR_DBG_OPTION_FORCE_OBJECT_MEMORY_REQS        = 0x00020401,
    GR_DBG_OPTION_FORCE_LARGE_IMAGE_ALIGNMENT     = 0x00020402,
    GR_DBG_OPTION_SKIP_EXECUTION_ON_ERROR         = 0x00020403,
} GR_DBG_DEVICE_OPTION;
```

Values

GR_DBG_OPTION_DISABLE_PIPELINE_LOADS

Disables pipeline loads by making any call to `grLoadPipeline()` fail with an error message if the value for this option is set to `GR_TRUE`.

GR_DBG_OPTION_FORCE_OBJECT_MEMORY_REQS

Forces all applicable API objects to have GPU memory requirements if the value for this option is set to `GR_TRUE`.

GR_DBG_OPTION_FORCE_LARGE_IMAGE_ALIGNMENT

Forces all images that are larger that a GPU memory page size to have memory requirements a multiple of the page size if the value for this option is set to `GR_TRUE`.

GR_DBG_OPTION_SKIP_EXECUTION_ON_ERROR

Controls validation layer behavior in case of an error. The core execution can be skipped if the value for this option is set to `GR_TRUE`.

GR_DBG_GLOBAL_OPTION

Defines global debug options that apply to all Mantle devices.

```
typedef enum _GR_DBG_GLOBAL_OPTION
{
    GR_DBG_OPTION_DEBUG_ECHO_ENABLE = 0x00020100,
    GR_DBG_OPTION_BREAK_ON_ERROR    = 0x00020101,
    GR_DBG_OPTION_BREAK_ON_WARNING  = 0x00020102,
} GR_DBG_GLOBAL_OPTION;
```

Values

GR_DBG_OPTION_DEBUG_ECHO_ENABLE

Enables/disables echoing debug message output. When application registers its message callback function, it might want to disable debug output to reduce the CPU overhead. By default the debug messages are logged to a debug output.

GR_DBG_OPTION_BREAK_ON_ERROR

Enables breaking into debugger on generation of an error message.

GR_DBG_OPTION_BREAK_ON_WARNING

Enables breaking into debugger on generation of a warning message.

GR_DBG_MSG_FILTER

Defines debug message filtering options.

```
typedef enum _GR_DBG_MSG_FILTER
{
    GR_DBG_MSG_FILTER_NONE     = 0x00020800,
    GR_DBG_MSG_FILTER_REPEATED = 0x00020801,
    GR_DBG_MSG_FILTER_ALL      = 0x00020802,
} GR_DBG_MSG_FILTER;
```

Values

GR_DBG_MSG_FILTER_NONE

The message is not filtered.

GR_DBG_MSG_FILTER_REPEATED

The repeated message is filtered, any message is reported only once until filtering is reset.

GR_DBG_MSG_FILTER_ALL

All instances of the message are filtered.

GR_DBG_MSG_TYPE

Defines debug message type.

```
typedef enum _GR_DBG_MSG_TYPE
{
    GR_DBG_MSG_UNKNOWN      = 0x00020000,
    GR_DBG_MSG_ERROR        = 0x00020001,
    GR_DBG_MSG_WARNING      = 0x00020002,
    GR_DBG_MSG_PERF_WARNING = 0x00020003,
} GR_DBG_MSG_TYPE;
```

Values

GR_DBG_MSG_UNKNOWN

Not a recognized message type.

GR_DBG_MSG_ERROR

Error message.

GR_DBG_MSG_WARNING

Warning message.

GR_DBG_MSG_PERF_WARNING

Performance warning message.

GR_DBG_OBJECT_TYPE

Object type returned by the validation layer for API objects.

```
typedef enum _GR_DBG_OBJECT_TYPE
{
    GR_DBG_OBJECT_UNKNOWN                = 0x00020900,
    GR_DBG_OBJECT_DEVICE                 = 0x00020901,
    GR_DBG_OBJECT_QUEUE                  = 0x00020902,
    GR_DBG_OBJECT_GPU_MEMORY             = 0x00020903,
    GR_DBG_OBJECT_IMAGE                  = 0x00020904,
    GR_DBG_OBJECT_IMAGE_VIEW             = 0x00020905,
    GR_DBG_OBJECT_COLOR_TARGET_VIEW      = 0x00020906,
    GR_DBG_OBJECT_DEPTH_STENCIL_VIEW     = 0x00020907,
    GR_DBG_OBJECT_SHADER                 = 0x00020908,
    GR_DBG_OBJECT_GRAPHICS_PIPELINE      = 0x00020909,
    GR_DBG_OBJECT_COMPUTE_PIPELINE       = 0x0002090a,
    GR_DBG_OBJECT_SAMPLER                = 0x0002090b,
    GR_DBG_OBJECT_DESCRIPTOR_SET         = 0x0002090c,
    GR_DBG_OBJECT_VIEWPORT_STATE         = 0x0002090d,
    GR_DBG_OBJECT_RASTER_STATE           = 0x0002090e,
    GR_DBG_OBJECT_MSAA_STATE             = 0x0002090f,
    GR_DBG_OBJECT_COLOR_BLEND_STATE      = 0x00020910,
    GR_DBG_OBJECT_DEPTH_STENCIL_STATE    = 0x00020911,
    GR_DBG_OBJECT_CMD_BUFFER             = 0x00020912,
    GR_DBG_OBJECT_FENCE                  = 0x00020913,
    GR_DBG_OBJECT_QUEUE_SEMAPHORE        = 0x00020914,
    GR_DBG_OBJECT_EVENT                  = 0x00020915,
    GR_DBG_OBJECT_QUERY_POOL             = 0x00020916,
    GR_DBG_OBJECT_SHARED_GPU_MEMORY      = 0x00020917,
    GR_DBG_OBJECT_SHARED_QUEUE_SEMAPHORE = 0x00020918,
    GR_DBG_OBJECT_PEER_GPU_MEMORY        = 0x00020919,
    GR_DBG_OBJECT_PEER_IMAGE             = 0x0002091a,
    GR_DBG_OBJECT_PINNED_GPU_MEMORY      = 0x0002091b,
    GR_DBG_OBJECT_INTERNAL_GPU_MEMORY    = 0x0002091c,
} GR_DBG_OBJECT_TYPE;
```

Values

GR_DBG_OBJECT_UNKNOWN

Object type is unknown.

GR_DBG_OBJECT_DEVICE

Object is a device.

GR_DBG_OBJECT_QUEUE

Object is a queue.

GR_DBG_OBJECT_GPU_MEMORY

Object is a regular GPU memory.

GR_DBG_OBJECT_IMAGE

Object is a regular image.

GR_DBG_OBJECT_IMAGE_VIEW

Object is an image view.

GR_DBG_OBJECT_COLOR_TARGET_VIEW

Object is a color target view.

GR_DBG_OBJECT_DEPTH_STENCIL_VIEW

Object is a depth-stencil view.

GR_DBG_OBJECT_SHADER

Object is a shader.

GR_DBG_OBJECT_GRAPHICS_PIPELINE

Object is a graphics pipeline.

GR_DBG_OBJECT_COMPUTE_PIPELINE

Object is a compute pipeline.

GR_DBG_OBJECT_SAMPLER

Object is a sampler.

GR_DBG_OBJECT_DESCRIPTOR_SET

Object is a descriptor set.

GR_DBG_OBJECT_VIEWPORT_STATE

Object is a viewport state.

GR_DBG_OBJECT_RASTER_STATE

Object is a rasterizer state.

GR_DBG_OBJECT_MSAA_STATE

Object is a multisampling state.

GR_DBG_OBJECT_COLOR_BLEND_STATE

Object is a color blending state.

GR_DBG_OBJECT_DEPTH_STENCIL_STATE

Object is a depth-stencil state.

GR_DBG_OBJECT_CMD_BUFFER

Object is a command buffer.

GR_DBG_OBJECT_FENCE

Object is a fence.

GR_DBG_OBJECT_QUEUE_SEMAPHORE

Object is a regular queue semaphore.

GR_DBG_OBJECT_EVENT

Object is an event.

GR_DBG_OBJECT_QUERY_POOL

Object is a query pool.

GR_DBG_OBJECT_SHARED_GPU_MEMORY

Object is a shared GPU memory.

GR_DBG_OBJECT_SHARED_QUEUE_SEMAPHORE

Object is an opened queue semaphore.

GR_DBG_OBJECT_PEER_GPU_MEMORY

Object is an opened peer GPU memory.

GR_DBG_OBJECT_PEER_IMAGE

Object is an opened peer image.

GR_DBG_OBJECT_PINNED_GPU_MEMORY

Object is pinned memory.

GR_DBG_OBJECT_INTERNAL_GPU_MEMORY

Object is an internal GPU memory.

CALLBACKS

GR_DBG_MSG_CALLBACK_FUNCTION

Application callback to allocate a block of system memory.

```
typedef GR_VOID (GR_STDCALL *GR_DBG_MSG_CALLBACK_FUNCTION)(
    GR_ENUM msgType,
    GR_ENUM validationLevel,
    GR_BASE_OBJECT srcObject,
    GR_SIZE location,
    GR_ENUM msgCode,
    const GR_CHAR* pMsg,
    GR_VOID* pUserData);
```

Parameters

msgType

Debug message type. See GR_DBG_MSG_TYPE.

validationLevel

Validation level at which the debug message was generated. See GR_VALIDATION_LEVEL.

srcObject

API handle for the object that generated the debug message.

location

Optional location or array element that is responsible for the debug message.

msgCode

Debug message code. See GR_DBG_MSG_CODE.

pMsg

Debug message text.

pUserData

User data passed to the driver when registering the debug message callback.

Chapter XX.

Window System Interface (WSI) for Windows

Functions

grWsiWinGetDisplays

Retrieves a list of displays attached to the device.

```
GR_RESULT grWsiWinGetDisplays(
    GR_DEVICE device,
    GR_UINT* pDisplayCount,
    GR_WSI_WIN_DISPLAY* pDisplayList);
```

Parameters

device

Device handle.

pDisplayCount

[in/out] The maximum number of displays to enumerate, and the output value specifies the total number of displays that were enumerated in `pDisplayList`.

pDisplayList

[out] Array of returned display handles. Can be `NULL`.

Returns

If successful, `grWsiWinGetDisplays()` returns `GR_SUCCESS` and the handles of attached displays are written to `pDisplayList`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pDisplayCount` is `NULL`
- `GR_ERROR_INVALID_MEMORY_SIZE` if `pDisplayList` is not `NULL` and the `pDisplayCount` input value is smaller than the number of attached displays

Notes

If `pDisplayList` is `NULL`, the input `pDisplayCount` value does not matter and the function returns the number of displays in `pDisplaysCount`.

Thread safety

Not thread safe.

grWsiWinGetDisplayModeList

Retrieves a list of supported display modes for the display object.

```
GR_RESULT grWsiWinGetDisplayModeList(
    GR_WSI_WIN_DISPLAY display,
    GR_UINT* pDisplayModeCount,
    GR_WSI_WIN_DISPLAY_MODE* pDisplayModeList);
```

Parameters

display

Display object handle.

pDisplayModeCount

[in/out] The maximum number of display modes to enumerate, and the output value specifies the total number of display modes that were enumerated in `pDisplayModeList`.

pDisplayModeList

[out] Array of returned display modes. See `GR_WSI_WIN_DISPLAY_MODE`. Can be `NULL`.

Returns

If successful, `grWsiWinGetDisplayModeList()` returns `GR_SUCCESS` and the display mode information written to `pDisplayModeList`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pDisplayModeCount` is `NULL`

- `GR_ERROR_INVALID_MEMORY_SIZE` if `pDisplayModeList` is not `NULL` and the `pDisplayModeCount` input value is smaller than the number of attached displays

Notes

If `pDisplayModeList` is `NULL`, the input `pDisplayModeCount` value does not matter and the function returns the number of displays in `pDisplayModeCount`.

Thread safety:

Not thread safe.

grWsiWinTakeFullscreenOwnership

Application enters fullscreen mode.

```
GR_RESULT grWsiWinTakeFullscreenOwnership(
    GR_WSI_WIN_DISPLAY display,
    GR_IMAGE image);
```

Parameters

display

Display object handle.

image

Presentable image object handle.

Returns

`grWsiWinTakeFullscreenOwnership()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the display or image handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the display handle references an invalid object type
- `GR_ERROR_UNAVAILABLE` if display is in fullscreen mode
- `GR_WSI_WIN_ERROR_INVALID_RESOLUTION` if presentable image resolution does not match the fullscreen mode

Notes

The presentable image should specify `GR_WSI_WIN_IMAGE_CREATE_FULLSCREEN_PRESENT` flag on creation and must be associated with this display.

Thread safety

Not thread safe.

grWsiWinReleaseFullscreenOwnership

Application exits fullscreen mode after it was entered with `grWsiWinTakeFullscreenOwnership()`.

```
GR_RESULT grWsiWinReleaseFullscreenOwnership(
    GR_WSI_WIN_DISPLAY display);
```

Parameters

display

Display object handle.

Returns

`grWsiWinReleaseFullscreenOwnership()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the display handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the display handle references an invalid object type
- `GR_ERROR_UNAVAILABLE` if display is not in fullscreen mode

Notes

Applications must release fullscreen ownership before destroying an associated device. Furthermore, the application must respond to losing focus (i.e., `WM_KILLFOCUS` events) by releasing fullscreen ownership and retaking fullscreen ownership when appropriate (i.e., a subsequent `WM_SETFOCUS` event).

Thread safety

Not thread safe.

grWsiWinSetGammaRamp

Sets custom gamma ramp in fullscreen mode.

```
GR_RESULT grWsiWinSetGammaRamp(
    GR_WSI_WIN_DISPLAY display,
    const GR_WSI_WIN_GAMMA_RAMP* pGammaRamp);
```

Parameters

display

Display object handle.

pGammaRamp

[in] Gamma ramp parameters. See `GR_WSI_WIN_GAMMA_RAMP`.

Returns

`grWsiWinSetGammaRamp()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the display handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the display handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pGammaRamp` is `NULL`
- `GR_ERROR_INVALID_VALUE` if any of the gamma ramp parameters are not in a valid range
- `GR_ERROR_UNAVAILABLE` if display is not in fullscreen mode

Notes

The gamma ramp is reset when exiting fullscreen exclusive mode. The application should restore custom gamma ramp when returning to fullscreen exclusive mode.

Thread safety

Not thread safe.

grWsiWinWaitForVerticalBlank

Waits for vertical blanking interval on display.

```
GR_RESULT grWsiWinWaitForVerticalBlank(
    GR_WSI_WIN_DISPLAY display);
```

Parameters

display

Display object handle.

Returns

`grWsiWinWaitForVerticalBlank()` returns `GR_SUCCESS` if the function successfully waited for vertical blanking interval. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the display handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the display handle references an invalid object type
- `GR_ERROR_UNAVAILABLE` if display is not in fullscreen mode and functionality is unavailable in windowed mode

Notes

None.

Thread safety

Not thread safe.

grWsiWinGetScanLine

Returns current scan line for the display.

```
GR_RESULT grWsiWinGetScanLine(
    GR_WSI_WIN_DISPLAY display,
    GR_INT* pScanLine);
```

Parameters

display

Display object handle.

pScanLine

[out] Current scan line.

Returns

If successful, `grWsiWinGetScanLine()` returns `GR_SUCCESS` and the current scan line is written to `pScanLine`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the display handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the display handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pScanLine` is `NULL`
- `GR_ERROR_UNAVAILABLE` if display is not in fullscreen mode and functionality is unavailable in windowed mode

Notes

A value of -1 indicates the display is currently in its vertical blanking period.

Thread safety

Not thread safe.

grWsiWinCreatePresentableImage

Creates an image that can be used as a source for presentation.

```
GR_RESULT grWsiWinCreatePresentableImage(
    GR_DEVICE device,
    const GR_WSI_WIN_PRESENTABLE_IMAGE_CREATE_INFO* pCreateInfo,
    GR_IMAGE* pImage,
    GR_GPU_MEMORY* pMem);
```

Parameters

device

Device handle.

pCreateInfo

[in] Presentable image creation info. See GR_WSI_WIN_PRESENTABLE_IMAGE_CREATE_INFO.

pImage

[out] Presentable image object handle.

pMem

[out] Memory handle for presentable image.

Returns

If successful, grWsiWinCreatePresentableImage() returns GR_SUCCESS and the created image object and its internal memory object is written to the location specified by pImage and pMem. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_VALUE if the image dimensions are invalid
- GR_ERROR_INVALID_VALUE if the refresh rate is invalid
- GR_ERROR_INVALID_POINTER if pCreateInfo or pImage or pMem is NULL
- GR_ERROR_INVALID_FORMAT if the format cannot be used for presentable image
- GR_ERROR_INVALID_FLAGS if invalid presentable image creation flags or image usage flags are specified

Notes

By definition, presentable images have a 2D image type, optimal tiling, a depth of 1, 1 mipmap level, and are single sampled. Fullscreen stereo images have an implicit array size of 2; all other presentable images have an implicit array size of 1.

The internal memory object for presentable image returned in pMem cannot be freed, mapped or used for object binding.

Thread safety

Not thread safe.

grWsiWinQueuePresent

Displays a presentable image.

```
GR_RESULT grWsiWinQueuePresent(
    GR_QUEUE queue,
    const GR_WSI_WIN_PRESENT_INFO* pPresentInfo);
```

Parameters

device

Device handle.

pPresentInfo

[in] Presentation parameters. See `GR_WSI_WIN_PRESENT_INFO`.

Returns

`grWsiWinQueuePresent()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the queue handle or presentable image handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the queue handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pPresentInfo` is `NULL`
- `GR_ERROR_INVALID_VALUE` if the presentation mode or presentation interval is invalid
- `GR_ERROR_INVALID_FLAGS` if presentation flags are invalid or do not match the present mode
- `GR_ERROR_INVALID_IMAGE` if image is not presentable or if it does not match the present mode

Notes

The presentable image has to be in the appropriate state for the used presentation method. Image has to be in the `GR_WSI_WIN_IMAGE_STATE_PRESENT_WINDOWED` state for windowed presentation and the `GR_WSI_WIN_IMAGE_STATE_PRESENT_FULLSCREEN` state for fullscreen presentation.

Thread safety

Not thread safe.

grWsiWinSetMaxQueuedFrames

Specifies how many frames can be placed in the presentation queue.

```
GR_RESULT grWsiWinSetMaxQueuedFrames(
    GR_DEVICE device,
    GR_UINT maxFrames);
```

Parameters

device

Device handle.

maxFrames

Maximum number of frames that can be batched. Specifying a value of zero resets the queue limit to a default system value (3 frames).

Returns

`grWsiWinSetMaxQueuedFrames()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if specified queue limit is invalid

Notes

When specifying the presentation queue limit for multiple GPUs used in multi-device configurations (e.g., for alternate frame rendering), the same value has to be set on all GPUs.

Thread safety

Not thread safe.

ENUMERATIONS

GR_WSI_WIN_IMAGE_STATE

Image states used for presenting images.

```
typedef enum _GR_WSI_WIN_IMAGE_STATE
{
    GR_WSI_WIN_IMAGE_STATE_PRESENT_WINDOWED   = 0x00200000,
    GR_WSI_WIN_IMAGE_STATE_PRESENT_FULLSCREEN = 0x00200001,
} GR_WSI_WIN_IMAGE_STATE;
```

Values

GR_WSI_WIN_IMAGE_STATE_PRESENT_WINDOWED

Image is used as a source for windowed presentation operations.

GR_WSI_WIN_IMAGE_STATE_PRESENT_FULLSCREEN

Image is used as a source for fullscreen presentation operations.

GR_WSI_WIN_INFO_TYPE

Defines types of information related to WSI functionality that can be retrieved from different objects.

```
typedef enum _GR_WSI_WIN_INFO_TYPE
{
    GR_WSI_WIN_INFO_TYPE_QUEUE_PROPERTIES              = 0x00206800,
    GR_WSI_WIN_INFO_TYPE_DISPLAY_PROPERTIES            = 0x00206801,
    GR_WSI_WIN_INFO_TYPE_GAMMA_RAMP_CAPABILITIES       = 0x00206802,
    GR_WSI_WIN_INFO_TYPE_DISPLAY_FREESYNC_SUPPORT      = 0x00206803,
    GR_WSI_WIN_INFO_TYPE_PRESENTABLE_IMAGE_PROPERTIES = 0x00206804,
    GR_WSI_WIN_INFO_TYPE_EXTENDED_DISPLAY_PROPERTIES  = 0x00206805,
} GR_WSI_WIN_INFO_TYPE;
```

Values

GR_WSI_WIN_INFO_TYPE_QUEUE_PROPERTIES

Retrieve WSI related queue properties with `grGetObjectInfo()`. Valid for `GR_QUEUE` objects.

GR_WSI_WIN_INFO_TYPE_DISPLAY_PROPERTIES

Retrieve display properties with `grGetObjectInfo()`. Valid for `GR_WSI_WIN_DISPLAY` objects.

GR_WSI_WIN_INFO_TYPE_GAMMA_RAMP_CAPABILITIES

Retrieve display gamma ramp capabilities with `grGetObjectInfo()`. Valid for `GR_WSI_WIN_DISPLAY` objects.

GR_WSI_WIN_INFO_TYPE_DISPLAY_FREESYNC_SUPPORT

Retrieve FreeSync display capabilities. Reserved.

GR_WSI_WIN_INFO_TYPE_PRESENTABLE_IMAGE_PROPERTIES

Retrieve presentable image properties with `grGetObjectInfo()`. Valid for presentable images only.

GR_WSI_WIN_INFO_TYPE_EXTENDED_DISPLAY_PROPERTIES

Retrieve extended display properties with `grGetObjectInfo()`. Valid for `GR_WSI_WIN_DISPLAY` objects.

GR_WSI_WIN_PRESENT_MODE

Presentation mode.

```
typedef enum _GR_WSI_WIN_PRESENT_MODE
{
    GR_WSI_WIN_PRESENT_MODE_WINDOWED   = 0x00200200,
    GR_WSI_WIN_PRESENT_MODE_FULLSCREEN = 0x00200201,
} GR_WSI_WIN_PRESENT_MODE;
```

Values

GR_WSI_WIN_PRESENT_MODE_WINDOWED

Windowed mode presentation.

GR_WSI_WIN_PRESENT_MODE_FULLSCREEN

Fullscreen mode presentation.

GR_WSI_WIN_ROTATION_ANGLE

Display rotation angle.

```
typedef enum _GR_WSI_WIN_ROTATION_ANGLE
{
    GR_WSI_WIN_ROTATION_ANGLE_0   = 0x00200100,
    GR_WSI_WIN_ROTATION_ANGLE_90  = 0x00200101,
    GR_WSI_WIN_ROTATION_ANGLE_180 = 0x00200102,
    GR_WSI_WIN_ROTATION_ANGLE_270 = 0x00200103,
} GR_WSI_WIN_ROTATION_ANGLE;
```

Values

GR_WSI_WIN_ROTATION_ANGLE_0

Display is not rotated.

GR_WSI_WIN_ROTATION_ANGLE_90

Display is rotated 90 degrees clockwise.

GR_WSI_WIN_ROTATION_ANGLE_180

Display is rotated 180 degrees clockwise.

GR_WSI_WIN_ROTATION_ANGLE_270

Display is rotated 270 degrees clockwise.

Flags

GR_WSI_WIN_EXTENDED_DISPLAY_FLAGS

Extended display property flags.

```
typedef enum _GR_WSI_WIN_EXTENDED_DISPLAY_FLAGS
{
    GR_WSI_WIN_WINDOWED_VBLANK_WAIT  = 0x00000001,
    GR_WSI_WIN_WINDOWED_GET_SCANLINE = 0x00000002,
} GR_WSI_WIN_EXTENDED_DISPLAY_FLAGS;
```

Values

GR_WSI_WIN_WINDOWED_VBLANK_WAIT

Wait on V-blank period with the `grWsiWinWaitForVerticalBlank()` function is supported in windowed mode.

GR_WSI_WIN_WINDOWED_GET_SCANLINE

Current display scanline can be retrieved with the `grWsiWinGetScanLine()` function in windowed mode.

GR_WSI_WIN_IMAGE_CREATE_FLAGS

WSI creation flags for presentable image.

```
typedef enum _GR_WSI_WIN_IMAGE_CREATE_FLAGS
{
    GR_WSI_WIN_IMAGE_CREATE_FULLSCREEN_PRESENT = 0x00000001,
    GR_WSI_WIN_IMAGE_CREATE_STEREO             = 0x00000002,
} GR_WSI_WIN_IMAGE_CREATE_FLAGS;
```

Values

GR_WSI_WIN_IMAGE_CREATE_FULLSCREEN_PRESENT

Create presentable image for fullscreen presentation.

GR_WSI_WIN_IMAGE_CREATE_STEREO

Create image for stereoscopic rendering and display.

GR_WSI_WIN_PRESENT_FLAGS

Presentation flags.

```
typedef enum _GR_WSI_WIN_PRESENT_FLAGS
{
    GR_WSI_WIN_PRESENT_FULLSCREEN_DONOTWAIT = 0x00000001,
    GR_WSI_WIN_PRESENT_FULLSCREEN_STEREO    = 0x00000002,
} GR_WSI_WIN_PRESENT_FLAGS;
```

Values

GR_WSI_WIN_PRESENT_FULLSCREEN_DONOTWAIT

Fail present call if present queue is full. Application could use this mode in conjunction with command buffer control features to reduce frame latency. Only valid if `presentMode` is `GR_WSI_WIN_PRESENT_MODE_FULLSCREEN`.

GR_WSI_WIN_PRESENT_FULLSCREEN_STEREO

Present should present both right and left images of a stereo allocation. Only valid if `presentMode` is `GR_WSI_WIN_PRESENT_MODE_FULLSCREEN`.

GR_WSI_WIN_PRESENT_SUPPORT_FLAGS

Flags describing types of present operation supported by the queue.

```
typedef enum _GR_WSI_WIN_PRESENT_SUPPORT_FLAGS
{
    GR_WSI_WIN_FULLSCREEN_PRESENT_SUPPORTED = 0x00000001,
    GR_WSI_WIN_WINDOWED_PRESENT_SUPPORTED   = 0x00000002,
} GR_WSI_WIN_PRESENT_SUPPORT_FLAGS;
```

Values

GR_WSI_WIN_FULLSCREEN_PRESENT_SUPPORTED

Queue supports fullscreen mode presentation.

GR_WSI_WIN_WINDOWED_PRESENT_SUPPORTED

Queue supports windowed mode presentation.

Data Structures

GR_RGB_FLOAT

Color in RGB format.

```
typedef struct _GR_RGB_FLOAT
{
    GR_FLOAT red;
    GR_FLOAT green;
    GR_FLOAT blue;
} GR_RGB_FLOAT;
```

Members

red

Red channel value.

green

Green channel value.

blue

Blue channel value.

GR_WSI_WIN_DISPLAY_MODE

Display mode description.

```
typedef struct _GR_WSI_WIN_DISPLAY_MODE
{
    GR_EXTENT2D extent;
    GR_FORMAT   format;
    GR_UINT     refreshRate;
    GR_BOOL     stereo;
    GR_BOOL     crossDisplayPresent;
} GR_WSI_WIN_DISPLAY_MODE;
```

Members

extent

Display mode dimensions. See `GR_EXTENT2D`.

format

The pixel format of the display mode. See `GR_FORMAT`.

refreshRate

Refresh rate in Hz.

stereo

The display mode supports stereoscopic rendering and display, if `GR_TRUE`.

crossDisplayPresent

The display mode supports cross-display presentation to the display (present through hardware compositor in multi-device configurations), if `GR_TRUE`.

GR_WSI_WIN_DISPLAY_PROPERTIES

Display properties.

```
typedef struct _GR_WSI_WIN_DISPLAY_PROPERTIES
{
    HMONITOR hMonitor;
    GR_CHAR  displayName[GR_MAX_DEVICE_NAME_LEN];
    GR_RECT  desktopCoordinates;
    GR_ENUM  rotation;
} GR_WSI_WIN_DISPLAY_PROPERTIES;
```

Members

hMonitor

Monitor handle for physical display in Windows®.

displayName

String specifying the device name of the display.

desktopCoordinates

Specifies the display rectangle, expressed in virtual screen coordinates. Note that if the display is not the desktop’s primary display, some of the rectangle’s coordinates may be negative values. See `GR_RECT`.

rotation

Display rotation angle. See `GR_WSI_WIN_ROTATION_ANGLE`.

GR_WSI_WIN_EXTENDED_DISPLAY_PROPERTIES

Extended display properties.

```
typedef struct _GR_WSI_WIN_EXTENDED_DISPLAY_PROPERTIES
{
    GR_FLAGS extendedProperties;
} GR_WSI_WIN_EXTENDED_DISPLAY_PROPERTIES;
```

Members

extendedProperties

Extended display property flags. See `GR_WSI_WIN_EXTENDED_DISPLAY_FLAGS`.

GR_WSI_WIN_GAMMA_RAMP

Definition of custom gamma ramp.

```
typedef struct _GR_WSI_WIN_GAMMA_RAMP
{
    GR_RGB_FLOAT scale;
    GR_RGB_FLOAT offset;
    GR_RGB_FLOAT gammaCurve[GR_MAX_GAMMA_RAMP_CONTROL_POINTS];
} GR_WSI_WIN_GAMMA_RAMP;
```

Members

scale

RGB float scale value. Scaling is performed after gamma curve conversion, but before the offset is added.

offset

RGB float offset value. Offset is added after scaling.

gammaCurve

RGB float values corresponding to output value per control point. Gamma curve conversion is performed before any scale or offset are applied. Gamma curve defined by approximation across control points, including the end points. The actual number of curve control point used is retrieved in gamma ramp capabilities. See GR_WSI_WIN_GAMMA_RAMP_CAPABILITIES.

GR_WSI_WIN_GAMMA_RAMP_CAPABILITIES

Custom gamma ramp capabilities.

```
typedef struct _GR_WSI_WIN_GAMMA_RAMP_CAPABILITIES
{
    GR_BOOL  supportsScaleAndOffset;
    GR_FLOAT minConvertedValue;
    GR_FLOAT maxConvertedValue;
    GR_UINT  controlPointCount;
    GR_FLOAT controlPointPositions[GR_MAX_GAMMA_RAMP_CONTROL_POINTS];
} GR_WSI_WIN_GAMMA_RAMP_CAPABILITIES;
```

Members

supportsScaleAndOffset

The display supports post-conversion scale and offset support, if GR_TRUE.

minConvertedValue

Minimum supported output value.

maxConvertedValue

Maximum supported output value.

controlPointCount

Number of valid entries in the `controlPointPositions` array.

controlPointPositions

Array of floating point values describing the position of each control point.

GR_WSI_WIN_PRESENT_INFO

Presentation information.

```
typedef struct _GR_WSI_WIN_PRESENT_INFO
{
    HWND     hWndDest;
    GR_IMAGE srcImage;
    GR_ENUM  presentMode;
    GR_UINT  presentInterval;
    GR_FLAGS flags;
} GR_WSI_WIN_PRESENT_INFO;
```

Members

hWndDest

Windows® handle of the destination window. Must be `NULL` if `presentMode` is `GR_WSI_WIN_PRESENT_MODE_FULLSCREEN`.

srcImage

Source image for the present.

presentMode

Type of present. See `GR_WSI_WIN_PRESENT_MODE`.

presentInterval

Integer from 0 to 4. Indicates if the fullscreen mode presentation should occur immediately (0) or after 1-4 vertical syncs. For windowed mode only, immediate presentation is valid.

flags

Presentation flags. See `GR_WSI_WIN_PRESENT_FLAGS`.

GR_WSI_WIN_PRESENTABLE_IMAGE_CREATE_INFO

Presentable image creation information.

```
typedef struct _GR_WSI_WIN_PRESENTABLE_IMAGE_CREATE_INFO
{
    GR_FORMAT          format;
    GR_FLAGS           usage;
    GR_EXTENT2D        extent;
    GR_WSI_WIN_DISPLAY display;
    GR_FLAGS           flags;
} GR_WSI_WIN_PRESENTABLE_IMAGE_CREATE_INFO;
```

Members

format

Presentable image pixel format. See GR_FORMAT.

usage

Image usage flags. See GR_IMAGE_USAGE_FLAGS.

extent

Width and height of the image in pixels. See GR_EXTENT2D.

display

Mantle display object corresponding to this image. Only valid for fullscreen presentable images.

flags

WSI specific presentable image flags. See GR_WSI_WIN_IMAGE_CREATE_FLAGS.

GR_WSI_WIN_PRESENTABLE_IMAGE_PROPERTIES

Information about presentable image object.

```
typedef struct _GR_WSI_WIN_PRESENTABLE_IMAGE_PROPERTIES
{
    GR_WSI_WIN_PRESENTABLE_IMAGE_CREATE_INFO createInfo;
    GR_GPU_MEMORY                            mem;
} GR_WSI_WIN_PRESENTABLE_IMAGE_PROPERTIES;
```

Members

createInfo

Presentable image creation information. See GR_WSI_WIN_PRESENTABLE_IMAGE_CREATE_INFO.

mem

Handle of GPU memory object that is bound to presentable image.

GR_WSI_WIN_QUEUE_PROPERTIES

WSI related queue properties.

```
typedef struct _GR_WSI_WIN_QUEUE_PROPERTIES
{
    GR_FLAGS presentSupport;
} GR_WSI_WIN_QUEUE_PROPERTIES;
```

Members

presentSupport

Flags indicating type of presentation mode (windowed or fullscreen) supported by the queue. See `GR_WSI_WIN_PRESENT_SUPPORT_FLAGS`.

GR_WSI_WIN_PRESENT_OCCLUDED

The window for presentation is completely occluded.

GR_WSI_WIN_ERROR_FULLSCREEN_UNAVAILABLE

The fullscreen exclusive mode cannot be entered due to incompatible display mode or because the display is already in fullscreen mode.

GR_WSI_WIN_ERROR_DISPLAY_REMOVED

The requested operation cannot be completed due to removal of the display device.

GR_WSI_WIN_ERROR_INCOMPATIBLE_DISPLAY_MODE

Fullscreen display mode is incompatible with presentable image.

GR_WSI_WIN_ERROR_MULTI_DEVICE_PRESENT_FAILED

Cross-device presentation failed.

GR_WSI_WIN_ERROR_WINDOWED_PRESENT_UNAVAILABLE

Windowed presentation is unavailable on the used queue.

GR_WSI_WIN_ERROR_INVALID_RESOLUTION

The presentable image resolution is invalid for the given operation.

Chapter XXI.

Mantle Extension Reference

Library Versioning

Functions

grGetExtensionLibraryVersion

Retrieves version of the AMD extension library interface.

```
GR_UINT32 grGetExtensionLibraryVersion();
```

Parameters

None.

Returns

`grGetExtensionLibraryVersion()` returns an AMD extension library interface encoded using the `GR_MAKE_VERSION` macro.

Notes

An application should not use the extension library if returned interface version is smaller than the `GR_AXL_VERSION` value from the headers used by the application.

Thread safety

Thread safe.

Enumerations

GR_EXT_INFO_TYPE

Defines an extension library version information that can be retrieved for physical GPU.

```
typedef enum _GR_EXT_INFO_TYPE
{
    GR_EXT_INFO_TYPE_PHYSICAL_GPU_SUPPORTED_AXL_VERSION = 0x00306100,
} GR_EXT_INFO_TYPE;
```

Values

GR_EXT_INFO_TYPE_PHYSICAL_GPU_SUPPORTED_AXL_VERSION

Retrieves a range of AMD extension library versions supported by Mantle ICD for a physical GPU with `grGetGpuInfo()`.

DATA STRUCTURES

GR_PHYSICAL_GPU_SUPPORTED_AXL_VERSION

Range of supported extension library versions for the physical GPU object.

```
typedef struct _GR_PHYSICAL_GPU_SUPPORTED_AXL_VERSION
{
    GR_UINT32 minVersion;
    GR_UINT32 maxVersion;
} GR_PHYSICAL_GPU_SUPPORTED_AXL_VERSION;
```

Members

minVersion

Minimum AMD extension library version supported by the Mantle ICD for the physical GPU. Encoded using the `GR_MAKE_VERSION` macro.

maxVersion

Maximum AMD extension library version supported by the Mantle ICD for the physical GPU. Encoded using the `GR_MAKE_VERSION` macro.

Border Color Palette Extension

Functions

grCreateBorderColorPalette

Creates a border color palette object.

```
GR_RESULT grCreateBorderColorPalette(
    GR_DEVICE device,
    const GR_BORDER_COLOR_PALETTE_CREATE_INFO* pCreateInfo,
    GR_BORDER_COLOR_PALETTE* pPalette);
```

Parameters

device

Device handle.

pCreateInfo

[in] Border color palette creation info. See `GR_BORDER_COLOR_PALETTE_CREATE_INFO`.

pPalette

[out] Border color palette object handle.

Returns

If successful, `grCreateBorderColorPalette()` returns `GR_SUCCESS` and the handle of the created border color palette object is written to the location specified by `pPalette`. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the device handle references an invalid object type
- `GR_ERROR_INVALID_VALUE` if the palette size is zero or larger than supported by device
- `GR_ERROR_INVALID_POINTER` if `pCreateInfo` or `pPalette` is `NULL`

Notes

None.

Thread safety

Thread safe.

grUpdateBorderColorPalette

Updates border color palette.

```
GR_RESULT grUpdateBorderColorPalette(
    GR_BORDER_COLOR_PALETTE palette,
    GR_UINT firstEntry,
    GR_UINT entryCount,
    const GR_FLOAT* pEntries);
```

Parameters

palette

Border color palette object handle.

firstEntry

First entry in a palette to update.

entryCount

Number of palette entries to update.

pEntries

[in] Border color data for update.

Returns

`grUpdateBorderColorPalette()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the palette handle is invalid
- `GR_ERROR_INVALID_OBJECT_TYPE` if the palette handle references an invalid object type
- `GR_ERROR_INVALID_POINTER` if `pEntries` is `NULL`
- `GR_ERROR_INVALID_VALUE` if palette entries are not in a valid range
- `GR_ERROR_MEMORY_NOT_BOUND` if border color palette requires GPU memory, but it wasn't bound

Notes

The color entries are specified as four consecutive floats per entry in R, G, B, A order.

Thread safety

Not thread safe for calls referencing the same border color palette object.

grCmdBindBorderColorPalette

Binds a border color palette to the current command buffer state.

```
GR_VOID grCmdBindBorderColorPalette(
    GR_CMD_BUFFER cmdBuffer,
    GR_ENUM pipelineBindPoint,
    GR_BORDER_COLOR_PALETTE palette);
```

Parameters

cmdBuffer

Command buffer handle.

pipelineBindPoint

Pipeline binding point (graphics or compute). See GR_PIPELINE_BIND_POINT.

palette

Border color palette object handle.

Notes

None.

ENUMERATIONS

GR_EXT_BORDER_COLOR_TYPE

Defines values for referencing border color palette entries.

```
typedef enum _GR_EXT_BORDER_COLOR_TYPE
{
    GR_EXT_BORDER_COLOR_TYPE_PALETTE_ENTRY_0 = 0x0030a000,
} GR_EXT_BORDER_COLOR_TYPE;
```

Values

GR_EXT_BORDER_COLOR_TYPE_PALETTE_ENTRY_0

The value for referencing the first palette entry in sampler creation information. Subsequent palette entries can be referenced with an offset relative to this value using `GR_EXT_BORDER_COLOR_TYPE_PALETTE_ENTRY` macro.

GR_EXT_INFO_TYPE

Defines a type of information that can be retrieved for border color palette objects.

```
typedef enum _GR_EXT_INFO_TYPE
{
    GR_EXT_INFO_TYPE_QUEUE_BORDER_COLOR_PALETTE_PROPERTIES = 0x00306800,
} GR_EXT_INFO_TYPE;
```

Values

GR_EXT_INFO_TYPE_QUEUE_BORDER_COLOR_PALETTE_PROPERTIES

Retrieves border color palette properties with `grGetObjectInfo()`. Valid for `GR_BORDER_COLOR_PALETTE` objects.

DATA STRUCTURES

GR_BORDER_COLOR_PALETTE_PROPERTIES

Border color palette properties for the queue.

```
typedef struct _GR_BORDER_COLOR_PALETTE_PROPERTIES
{
    GR_UINT maxPaletteSize;
} GR_BORDER_COLOR_PALETTE_PROPERTIES;
```

Members

maxPaletteSize

Maximum size of the border color palette supported by the queue. Border color palette is not supported by the queue if reported value is zero.

GR_BORDER_COLOR_PALETTE_CREATE_INFO

Border color palette creation information.

```
typedef struct _GR_BORDER_COLOR_PALETTE_CREATE_INFO
{
    GR_UINT paletteSize;
} GR_BORDER_COLOR_PALETTE_CREATE_INFO;
```

Members

paletteSize

Size of the border color palette to create. Must be smaller than the maximum supported palette size.

Advanced Multisampling Extension

Functions

grCreateAdvancedMsaaState

Creates an MSAA dynamic state object with advanced capabilities.

```
GR_RESULT grCreateAdvancedMsaaState(
    GR_DEVICE device,
    const GR_ADVANCED_MSAA_STATE_CREATE_INFO* pCreateInfo,
    GR_MSAA_STATE_OBJECT* pState);
```

Parameters

device

Device handle.

pCreateInfo

[in] Advanced MSAA state object creation data. See GR_ADVANCED_MSAA_STATE_CREATE_INFO.

pState

[out] Advanced MSAA state object handle.

Returns

If successful, grCreateAdvancedMsaaState() returns GR_SUCCESS and the handle of the created advanced MSAA state object is written to the location specified by pState. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_POINTER if pCreateInfo or pState is NULL
- GR_ERROR_INVALID_VALUE if any number of samples is specified incorrectly or is unsupported
- GR_ERROR_INVALID_VALUE if custom sample pattern is enabled and sample positions are out of range

Notes

None.

Thread safety

Thread safe.

grCreateFmaskImageView

Creates an FMask image view for multisampled color targets that can be bound to the graphics or compute pipeline for shader read-only access.

```
GR_RESULT grCreateFmaskImageView(
    GR_DEVICE device,
    const GR_FMASK_IMAGE_VIEW_CREATE_INFO* pCreateInfo,
    GR_IMAGE_VIEW* pView);
```

Parameters

device

Device handle.

pCreateInfo

[in] FMask image view creation data. See GR_FMASK_IMAGE_VIEW_CREATE_INFO.

pView

[out] FMask image view handle.

Returns

If successful, grCreateFmaskImageView() returns GR_SUCCESS and the handle of the created FMask image view is written to the location specified by pView. Otherwise, it returns one of the following errors:

- GR_ERROR_INVALID_HANDLE if the device handle is invalid
- GR_ERROR_INVALID_OBJECT_TYPE if the device handle references an invalid object type
- GR_ERROR_INVALID_POINTER if pCreateInfo or pView is NULL
- GR_ERROR_INVALID_HANDLE if the pCreateInfo->image handle is invalid
- GR_ERROR_INVALID_VALUE if the base slice is invalid for the given image object
- GR_ERROR_INVALID_VALUE if the number of array slices is zero or the range of slices is greater than what is available in the image object
- GR_ERROR_INVALID_IMAGE if the image object does not support the FMask

Notes

None.

Thread safety

Thread safe.

ENUMERATIONS

GR_EXT_IMAGE_STATE

Image states for FMask shader access.

```
typedef enum _GR_EXT_IMAGE_STATE
{
    GR_EXT_IMAGE_STATE_GRAPHICS_SHADER_FMASK_LOOKUP = 0x00300100,
    GR_EXT_IMAGE_STATE_COMPUTE_SHADER_FMASK_LOOKUP  = 0x00300101,
} GR_EXT_IMAGE_STATE;
```

Values

GR_EXT_IMAGE_STATE_GRAPHICS_SHADER_FMASK_LOOKUP

Range of image subresources can be used as a read-only image view with FMask lookup in the graphics pipeline.

GR_EXT_IMAGE_STATE_COMPUTE_SHADER_FMASK_LOOKUP

Range of image subresources can be used as a read-only image view with FMask lookup in the compute pipeline.

DATA STRUCTURES

GR_ADVANCED_MSAA_STATE_CREATE_INFO

Advanced dynamic MSAA state creation information.

```
typedef struct _GR_ADVANCED_MSAA_STATE_CREATE_INFO
{
    GR_UINT                     coverageSamples;
    GR_UINT                     pixelShaderSamples;
    GR_UINT                     depthStencilSamples;
    GR_UINT                     colorTargetSamples;
    GR_SAMPLE_MASK              sampleMask;
    GR_UINT                     sampleClusters;
    GR_UINT                     alphaToCoverageSamples;
    GR_BOOL                     disableAlphaToCoverageDither;
    GR_BOOL                     customSamplePatternEnable;
    GR_MSAA_QUAD_SAMPLE_PATTERN customSamplePattern;
} GR_ADVANCED_MSAA_STATE_CREATE_INFO;
```

Members

coverageSamples

Controls the sample rate of the rasterizer. This may be set to 1, 2, 4, 8, or 16.

The rasterizer sample rate must be greater than or equal to all other sample rates (`pixelShaderSamples`, `depthTargetSamples`, `colorTargetSamples`, `colorTargetFragments`, `sampleClusters`, and `alphaToCoverageSamples`).

pixelShaderSamples

Controls the pixel shader execution rate for pixel shaders that use inputs that are evaluated per sample (i.e., an HLSL SV_SampleIndex input or an input using the sample interpolation modifier). This may be set to 1, 2, 4, or 8.

The pixel shader sample rate must be less than or equal to `coverageSamples`, `depthTargetSamples`, and `colorTargetSamples`. When rendering with a pipelines that does not require per-sample PS inputs, the value has no effect.

depthStencilSamples

Specifies the number of samples in the bound depth target. This may be set to 1, 2, 4, or 8. The value is ignored if no depth target is bound.

The depth target sample rate must be less than or equal to `coverageSamples`.

colorTargetSamples

Specifies the maximal number of coverage samples stored in each any color target’s FMask. This may be set to 1, 2, 4, 8, or 16.

The color target samples must be greater than or equal to `colorTargetFragments` and less than or equal to `coverageSamples`.

sampleMask

Sample bit-mask. Determines which samples in color targets are updated. Lower bit represents sample zero.

sampleClusters

Specifies number of sample clusters for controlling over-rasterization. If any sample in a cluster is covered, then all samples in a cluster are marked as covered as well. For example, specifying a single sample cluster makes all samples appear covered if any of them are covered. This may be set to 1, 2, 4, 8, or 16.

The number of sample clusters has to be less than or equal to `coverageSamples`.

alphaToCoverageSamples

Specifies how many samples of quality are generated when alpha-to-coverage is enabled. If `alphaToCoverageSamples` is less than `depthTargetSamples` or `colorTargetSamples`, the additional sample coverage values are extrapolated.

The alpha-to-coverage samples must be less than or equal to `coverageSamples`.

disableAlphaToCoverageDither

By default, the alpha-to-coverage implementation dithers the generated coverage over a 2x2 quad in order to more closely approximate the specified alpha coverage. Setting `disableAlphaToCoverageDither` to `GR_TRUE` disables that dithering.

customSamplePatternEnable

Enables custom sampler pattern specified in `customSamplePattern`. Setting `customSamplePatternEnable` to `GR_FALSE` uses default sample pattern.

customSamplePattern

Pixel quad sample positions for the custom sample pattern. See `GR_MSAA_QUAD_SAMPLE_PATTERN`.

GR_FMASK_IMAGE_VIEW_CREATE_INFO

FMask image view creation information.

```
typedef struct _GR_FMASK_IMAGE_VIEW_CREATE_INFO
{
    GR_IMAGE image;
    GR_UINT  baseArraySlice;
    GR_UINT  arraySize;
} GR_FMASK_IMAGE_VIEW_CREATE_INFO;
```

Members

image

Image for the view.

baseArraySlice

First array slice for the view in 2D image array.

arraySize

Number of array slice for the view in 2D image array.

GR_MSAA_QUAD_SAMPLE_PATTERN

Custom multisampling pattern for a pixel quad.

```
typedef struct _GR_MSAA_QUAD_SAMPLE_PATTERN
{
    GR_OFFSET2D topLeft[GR_MAX_MSAA_RASTERIZER_SAMPLES];
    GR_OFFSET2D topRight[GR_MAX_MSAA_RASTERIZER_SAMPLES];
    GR_OFFSET2D bottomLeft[GR_MAX_MSAA_RASTERIZER_SAMPLES];
    GR_OFFSET2D bottomRight[GR_MAX_MSAA_RASTERIZER_SAMPLES];
} GR_MSAA_QUAD_SAMPLE_PATTERN;
```

Members

topLeft

Sample locations for top-left pixel of the quad. See GR_OFFSET2D.

topRight

Sample locations for top-right pixel of the quad. See GR_OFFSET2D.

bottomLeft

Sample locations for bottom-left pixel of the quad. See GR_OFFSET2D.

bottomRight

Sample locations for bottom-right pixel of the quad. See GR_OFFSET2D.

Copy Occlusion Query Data Extension

Functions

grCmdCopyOcclusionData

Copies occlusion query results to memory location.

```
GR_RESULT grCmdCopyOcclusionData(
    GR_CMD_BUFFER cmdBuffer,
    GR_QUERY_POOL queryPool,
    GR_UINT startQuery,
    GR_UINT queryCount,
    GR_GPU_MEMORY destMem,
    GR_GPU_SIZE destOffset,
    GR_BOOL accumulateData);
```

Parameters

cmdBuffer

Command buffer handle.

queryPool

Query pool handle.

startQuery

First query pool slot from which to copy occlusion data.

queryCount

Number of query pool slots to copy.

destMem

Destination memory object.

destOffset

Byte offset in the memory object to the beginning of the copied data.

accumulateData

If `GR_TRUE`, occlusion data are added to the value at destination memory location; otherwise, occlusion data are stored at the provided location, overwriting the previous value.

Notes

Only occlusion query pools can be used with the `grCmdCopyOcclusionData()` function. Using any other query type results in undefined behavior.

Destination offset must be 4 byte aligned.

ENUMERATIONS

GR_EXT_MEMORY_STATE

Memory state for copying occlusion query data.

```
typedef enum _GR_EXT_MEMORY_STATE
{
    GR_EXT_MEMORY_STATE_COPY_OCCLUSION_DATA = 0x00300000,
} GR_EXT_MEMORY_STATE;
```

Values

GR_EXT_MEMORY_STATE_COPY_OCCLUSION_DATA

Memory state for copying occlusion query data.

Command Buffer Control Flow Extension

Functions

grCmdSetOcclusionPredication

Sets occlusion-based predication in the command buffer.

```
GR_VOID grCmdSetOcclusionPredication(
    GR_CMD_BUFFER cmdBuffer,
    GR_QUERY_POOL queryPool,
    GR_UINT slot,
    GR_ENUM condition,
    GR_BOOL waitResults,
    GR_BOOL accumulateData);
```

Parameters

cmdBuffer

Command buffer handle.

queryPool

Query pool handle.

slot

Query pool slot to use for setting occlusion predication.

condition

Occlusion condition for setting predication. See `GR_EXT_OCCLUSION_CONDITION`.

waitResults

accumulateData

Notes

Only occlusion query pools can be used with the `grCmdSetOcclusionPredication()` function. Using any other query type results in undefined behavior.

grCmdResetOcclusionPredication

Resets occlusion-based predication in command buffer.

```
GR_VOID grCmdResetOcclusionPredication(
    GR_CMD_BUFFER cmdBuffer);
```

Parameters

cmdBuffer

Command buffer handle.

Notes

None.

grCmdSetMemoryPredication

Sets memory value based predication in command buffer.

```
GR_VOID grCmdSetMemoryPredication(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY mem,
    GR_GPU_SIZE offset);
```

Parameters

cmdBuffer

Command buffer handle.

mem

Memory object containing value for setting memory-based predication.

offset

Byte offset within memory object to the predication value.

Notes

Behavior is undefined if predication value is not zero or one.

grCmdResetMemoryPredication

Sets memory value based predication in command buffer.

```
GR_VOID grCmdResetMemoryPredication(
    GR_CMD_BUFFER cmdBuffer);
```

Parameters

cmdBuffer

Command buffer handle.

Notes

None.

grCmdIf

Starts a conditional block in the command buffer.

```
GR_VOID grCmdIf(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY srcMem,
    GR_GPU_SIZE srcOffset,
    GR_UINT64 data,
    GR_UINT64 mask,
    GR_ENUM func);
```

Parameters

cmdBuffer

Command buffer handle.

srcMem

Memory object containing value for conditional execution.

srcOffset

Byte offset within memory object to the value for conditional execution.

data

Literal value to be used for condition evaluation. Should have bit mask already applied (AND'ed with the mask) by the application.

mask

Bit mask to be applied to the value in memory before comparing it with the literal value.

func

Comparison function for the condition evaluation. See `GR_COMPARE_FUNC`.

Notes

None.

grCmdElse

Terminates a conditional block in the command buffer and starts a block for an opposite condition.

```
GR_VOID grCmdElse(
    GR_CMD_BUFFER cmdBuffer);
```

Parameters

cmdBuffer

Command buffer handle.

Notes

None.

grCmdEndIf

Terminates a conditional block in the command buffer.

```
GR_VOID grCmdEndIf(
    GR_CMD_BUFFER cmdBuffer);
```

Parameters

cmdBuffer

Command buffer handle.

Notes

None.

grCmdWhile

Starts a loop in the command buffer.

```
GR_VOID grCmdWhile(
    GR_CMD_BUFFER cmdBuffer,
    GR_GPU_MEMORY srcMem,
    GR_GPU_SIZE srcOffset,
    GR_UINT64 data,
    GR_UINT64 mask,
    GR_ENUM func);
```

Parameters

cmdBuffer

Command buffer handle.

srcMem

Memory object containing value for conditional execution.

srcOffset

Byte offset within memory object to the value for conditional execution.

data

Literal value to be used for condition evaluation. Should have bit mask already applied (AND'ed with the mask) by the application.

mask

Bit mask to be applied to the value in memory before comparing it with the literal value.

func

Comparison function for the condition evaluation. See `GR_COMPARE_FUNC`.

Notes

None.

grCmdEndWhile

Terminates a loop in the command buffer.

```
GR_VOID grCmdEndWhile(
    GR_CMD_BUFFER cmdBuffer);
```

Parameters

cmdBuffer

Command buffer handle.

Notes

None.

ENUMERATIONS

GR_EXT_INFO_TYPE

Defines a type of information that can be retrieved for border color palette objects.

```
typedef enum _GR_EXT_INFO_TYPE
{
    GR_EXT_INFO_TYPE_QUEUE_CONTROL_FLOW_PROPERTIES = 0x00306801,
} GR_EXT_INFO_TYPE;
```

Values

GR_EXT_INFO_TYPE_QUEUE_CONTROL_FLOW_PROPERTIES

Retrieves control flow properties for a queue with `grGetObjectInfo()`. Valid for `GR_QUEUE` objects.

GR_EXT_MEMORY_STATE

Memory state for command buffer control data.

```
typedef enum _GR_EXT_MEMORY_STATE
{
    GR_EXT_MEMORY_STATE_CMD_CONTROL = 0x00300001,
} GR_EXT_MEMORY_STATE;
```

Values

GR_EXT_MEMORY_STATE_CMD_CONTROL

Memory state for command buffer control data.

GR_EXT_OCCLUSION_CONDITION

Condition for setting occlusion-based predication.

```
typedef enum _GR_EXT_OCCLUSION_CONDITION
{
    GR_EXT_OCCLUSION_CONDITION_VISIBLE      = 0x00300300,
    GR_EXT_OCCLUSION_CONDITION_INVISIBLE    = 0x00300301,
} GR_EXT_OCCLUSION_CONDITION;
```

Values

GR_EXT_OCCLUSION_CONDITION_VISIBLE

Set predication if occluded object is visible.

GR_EXT_OCCLUSION_CONDITION_INVISIBLE

Set predication if occluded object is invisible.

FLAGS

GR_EXT_CONTROL_FLOW_FEATURE_FLAGS

Queue capability flags for command buffer control flow.

```
typedef enum _GR_EXT_CONTROL_FLOW_FEATURE_FLAGS
{
    GR_EXT_CONTROL_FLOW_OCCLUSION_PREDICATION = 0x00000001,
    GR_EXT_CONTROL_FLOW_MEMORY_PREDICATION    = 0x00000002,
    GR_EXT_CONTROL_FLOW_CONDITIONAL_EXECUTION = 0x00000004,
    GR_EXT_CONTROL_FLOW_LOOP_EXECUTION        = 0x00000008,
} GR_EXT_CONTROL_FLOW_FEATURE_FLAGS;
```

Values

GR_EXT_CONTROL_FLOW_OCCLUSION_PREDICATION

Queue supports occlusion-based predication.

GR_EXT_CONTROL_FLOW_MEMORY_PREDICATION

Queue supports memory-based predication.

GR_EXT_CONTROL_FLOW_CONDITIONAL_EXECUTION

Queue supports conditional command buffer execution.

GR_EXT_CONTROL_FLOW_LOOP_EXECUTION

Queue supports loops in command buffers.

Data Structures

GR_QUEUE_CONTROL_FLOW_PROPERTIES

Queue capabilities for command buffer control flow.

```
typedef struct _GR_QUEUE_CONTROL_FLOW_PROPERTIES
{
    GR_UINT  maxNestingLimit;
    GR_FLAGS controlFlowOperations;
} GR_QUEUE_CONTROL_FLOW_PROPERTIES;
```

Members

maxNestingLimit

Maximum level of nested control flow allowed in command buffer.

controlFlowOperations

Capability flags. See `GR_EXT_CONTROL_FLOW_FEATURE_FLAGS`.

DMA Queue Extension

Enumerations

GR_EXT_IMAGE_STATE

Image states for DMA data transfer.

```
typedef enum _GR_EXT_IMAGE_STATE
{
    GR_EXT_IMAGE_STATE_DATA_TRANSFER_DMA_QUEUE      = 0x00300102,
} GR_EXT_IMAGE_STATE;
```

Values

GR_EXT_IMAGE_STATE_DATA_TRANSFER_DMA_QUEUE

Range of image subresources can be used for data transfer on compute queue.

GR_EXT_QUEUE_TYPE

Queue type for DMA extension.

```
typedef enum _GR_EXT_QUEUE_TYPE
{
    GR_EXT_QUEUE_DMA   = 0x00300200,
} GR_EXT_QUEUE_TYPE;
```

Values

GR_EXT_QUEUE_DMA

DMA queue type.

Timer Queue Extension

Functions

grQueueDelay

Inserts delay into a timer queue.

```
GR_RESULT grQueueDelay(
    GR_QUEUE queue,
    GR_FLOAT delay);
```

Parameters

queue

Queue handle.

delay

Queued delay in seconds.

Returns

`grQueueDelay()` returns `GR_SUCCESS` if the function executed successfully. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the queue handle is invalid
- `GR_ERROR_INVALID_QUEUE_TYPE` if the queue is not a timer queue
- `GR_ERROR_INVALID_VALUE` if delay is less than zero

Notes

The function is only valid for the queues of the `GR_EXT_QUEUE_TIMER` type.

Thread safety

Not thread safe for calls referencing the same queue object.

ENUMERATIONS

GR_EXT_QUEUE_TYPE

Queue type for timer queue extension.

```
typedef enum _GR_EXT_QUEUE_TYPE
{
    GR_EXT_QUEUE_TIMER = 0x00300201,
} GR_EXT_QUEUE_TYPE;
```

Values

GR_EXT_QUEUE_TIMER

Timer queue type.

GPU Timestamp Calibration Extension

Functions

grCalibrateGpuTimestamp

Correlates the current GPU timestamp to the CPU clock.

```
GR_RESULT grCalibrateGpuTimestamp(
    GR_DEVICE device,
    GR_GPU_TIMESTAMP_CALIBRATION* pCalibrationData);
```

Parameters

device

Device handle.

pCalibrationData

[out] Returned GPU timestamp calibration data. See `GR_GPU_TIMESTAMP_CALIBRATION`.

Returns

If successful, `grCalibrateGpuTimestamp()` returns `GR_SUCCESS` and the GPU timestamps synchronization data. Otherwise, it returns one of the following errors:

- `GR_ERROR_INVALID_HANDLE` if the device handle is invalid
- `GR_ERROR_INVALID_POINTER` if `pCalibrationData` is `NULL`

Notes

None.

Thread safety

Not thread safe for calls referencing the same device object.

DATA STRUCTURES

GR_GPU_TIMESTAMP_CALIBRATION

GPU timestamp calibration data. Correlates a current GPU timestamp with a current CPU clock value.

```
typedef struct _GR_GPU_TIMESTAMP_CALIBRATION
{
    GR_UINT64 gpuTimestamp;
    union
    {
        GR_UINT64 cpuWinPerfCounter;
        GR_BYTE   _padding[16];
    };
} GR_GPU_TIMESTAMP_CALIBRATION;
```

Members

gpuTimestamp

Current GPU timestamp value compatible with timestamps written to memory by `grCmdWriteTimestamp()`.

cpuWinPerfCounter

Current CPU performance counter value at the time of the corresponding GPU timestamp. Compatible with values returned by the `QueryPerformanceCounter()` function in the Windows® OS.

Resource State Access Extension

Flags

GR_EXT_ACCESS_CLIENT

Resource access flags describing where memory or image is going to be accessed.

```
typedef enum _GR_EXT_ACCESS_CLIENT
{
    GR_EXT_ACCESS_DEFAULT         = 0x00000000,
    GR_EXT_ACCESS_CPU             = 0x01000000,
    GR_EXT_ACCESS_UNIVERSAL_QUEUE = 0x02000000,
    GR_EXT_ACCESS_COMPUTE_QUEUE   = 0x04000000,
    GR_EXT_ACCESS_DMA_QUEUE       = 0x08000000,
} GR_EXT_ACCESS_CLIENT;
```

Values

GR_EXT_ACCESS_DEFAULT

Memory or image can be accessed by any applicable GPU queues or CPU.

GR_EXT_ACCESS_CPU

Memory or image is going to only be accessed by the CPU.

GR_EXT_ACCESS_UNIVERSAL_QUEUE

Memory or image is going to only be accessed by the universal GPU queue.

GR_EXT_ACCESS_COMPUTE_QUEUE

Memory or image is going to only be accessed by the compute GPU queue.

GR_EXT_ACCESS_DMA_QUEUE

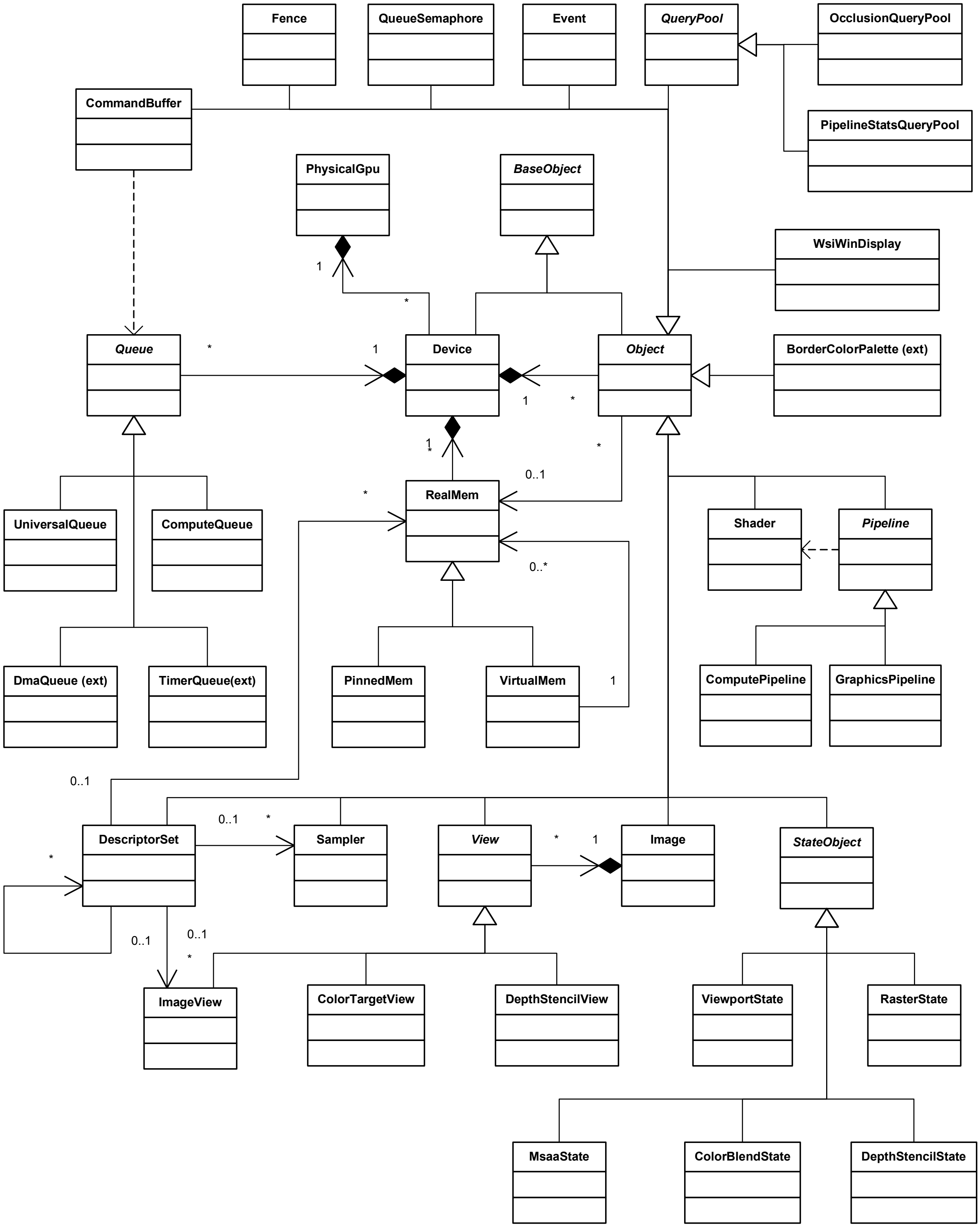
Memory or image is going to only be accessed by the DMA GPU queue.

Appendix A.

Mantle Class Diagram

The following Mantle class diagram provides a conceptual view of Mantle API object relationship.

Figure 19. Conceptual Mantle object diagram



Appendix B.

Feature Mapping to Other APIs

Not all features available in other graphics APIs such as OpenGL and DirectX® 11 are supported in Mantle. They are removed either because of the limited utility and performance tax one has to pay for their implementation, or because there is a better and more forward-looking way of implementing a similar functionality available in Mantle. This section lists explains how the absence of particular features from other APIs can be worked around in Mantle.

Index reset

Index reset is a rarely used DirectX® 11 feature that does not provide significant performance benefits. Applications should use indexed primitive lists to emulate index strips with reset.

Shader subroutines

Shader subroutines is a rarely used feature that could provide small benefits over uber-shaders in terms of reduced register pressure, and in some cases could actually somewhat hurt performance due to extra overhead of indexing resources. Shader subroutines can be substituted with a balanced mix of uber-shaders and pipeline link time constants.

Geometry stream-out

The geometry stream-out offers limited functionality while introducing unnecessary complexity to the graphics pipeline management in the driver. Using compute shaders or even graphics pipeline shaders with outputs to memory and images can easily supersede stream-out functionality and offer increased performance and flexibility. Ordered output from stream-out, matching the geometry processing order, is something that is not directly available with write-access memory, images, and unordered appends. However, the ordering can be artificially enforced in the shaders, if at all necessary.

Specialized vertex and constant buffers

In the Mantle API, specialized vertex and constant buffers are removed in favor of more general buffer support. An application can use regular buffers to store vertex data, shader constants, or any other data.

Per-resource LOD clamping

Mantle allows specifying LOD clamping in samplers, making per-resource LOD clamping mostly a redundant feature. Additionally, Mantle exposes advanced memory and resource management capabilities to applications that could be more efficiently used to manage actual memory footprint for the resources.

Sample pattern access in shaders

The shader IL exposed in Mantle does not support instructions for querying sample patterns from shaders. If shaders need to rely on sample position information, an application has to pass that information to shaders using the constant data.

Mipmap generation

Mantle is a fairly minimalistic API targeting the core hardware functionality necessary for writing applications. Mipmap generation can be easily implemented on top of Mantle with a utility library using pixel or compute shaders.

Line AA

Line anti-aliasing (AA) is a largely antiquated feature, rarely used outside of workstation applications, that adds unnecessary complexity to the driver. If necessary, a limited form of line AA support can be provided through a special extension.

Line Stipple

Similarly to line AA feature, implementing the line stipple would add unnecessary complexity to the driver. If necessary, a limited form of line stipple support can be provided through a special extension.

Appendix C.

Format Capabilities

Some formats expose slightly different sets of capabilities for linear and optimal tiling modes. The following table provide a minimal expected set of capabilities for supported formats. Additional capabilities can be queried as described in Table 23.

Table 23. Format capabilities

Channel format	UNORM	SNORM	UINT	SINT	FLOAT	SRGB	DS
R4G4	Lr Or						
R4G4B4A4	Lrw Orw C B X						
R5G6B5	Lrw Orw C B X						
B5G6R5	C B X					C B	
R5G5B5A1	Lrw Orw C B X						
R8	Lrw Orw Trw C B M X	Lrw Orw Trw C B M X	Lrw Orw Trw C M X	Lrw Orw Trw C M X		Lr Or C B M	Lr Or S M
R8G8	Lrw Orw Trw C B M X	Lrw Orw Trw C B M X	Lrw Orw Trw C M X	Lrw Orw Trw C M X			

Channel format	UNORM	SNORM	UINT	SINT	FLOAT	SRGB	DS
R8G8B8A8	Lrw Orw Trw C B M X	Lrw Orw Trw C B M X	Lrw Orw Trw C M X	Lrw Orw Trw C M X		Lr Or C B M	
B8G8R8A8	C B X					C B	
R10G11B11					Lrw Orw Trw C B M X		
R11G11B10					Lrw Orw C B M X		
R10G10B10A2	Lrw Orw Trw C B M X		Lrw Orw Trw C M X				
R16	Lrw Orw Trw C B M X	Lrw Orw Trw C B M X	Lrw Orw Trw C M X	Lrw Orw Trw C M X	Lrw Orw Trw C B M X		Orw D M
R16G16	Lrw Orw Trw C B M X	Lrw Orw Trw C B M X	Lrw Orw Trw C M X	Lrw Orw Trw C M X	Lrw Orw Trw C B M X		
R16G16B16A16	Lrw Orw Trw C B M X	Lrw Orw Trw C B M X	Lrw Orw Trw C M X	Lrw Orw Trw C M X	Lrw Orw Trw C B M X		
R32			Lrw Orw Trw C M X	Lrw Orw Trw C M X	Lrw Orw Trw C B M X		Orw D M
R32G32			Lrw Orw Trw C M X	Lrw Orw Trw C M X	Lrw Orw Trw C B M X		
R32G32B32			Trw	Trw	Trw		
R32G32B32A32			Lrw Orw Trw C M X	Lrw Orw Trw C M X	Lrw Orw Trw C B M X		
R16G8							Orw D S M
R32G8							Orw D S M

Channel format	UNORM	SNORM	UINT	SINT	FLOAT	SRGB	DS
R9G9B9E5					Lrw Orw		
BC1	Or					Or	
BC2	Or					Or	
BC3	Or					Or	
BC4	Or	Or					
BC5	Or	Or					
BC6U					Or		
BC6S					Or		
BC7	Or					Or	

Legend: Lr – read-only access of linearly tiled images
Lrw – read/write access of linearly tiled images
Or – read-only access of optimally tiled images
Orw – read/write access of optimally tiled images
Trw – read/write access of typed memory views
C – color target output (no blending)
B – color target blending
D – depth target output
S – stencil target output
M – multisampled color or depth-stencil target
X – format conversion on copy

Multisampled and depth-stencil images, as well as images of compressed formats (BC*n*) are only compatible with optimal tiling.

Appendix D.

Command Buffer Building Function Summary

Not all command buffer building functions are supported across all queue types. The following tables define function compatibility with queue types.

Table 24. Mantle core command building function summary

Function	Universal queue	Compute queue	DMA queue (extension)
grCmdBindPipeline	X	X	
grCmdBindStateObject	X		
grCmdBindDescriptorSet	X	X	
grCmdBindDynamicMemoryView	X	X	
grCmdBindIndexData	X		
grCmdBindTargets	X		
grCmdPrepareMemoryRegions	X	X	X
grCmdPrepareImages	X	X	X

Function	Universal queue	Compute queue	DMA queue (extension)
grCmdDraw	X		
grCmdDrawIndexed	X		
grCmdDrawIndirect	X		
grCmdDrawIndexedIndirect	X		
grCmdDispatch	X	X	
grCmdDispatchIndirect	X	X	
grCmdCopyMemory	X	X	X
grCmdCopyImage	X	X	X
grCmdCopyMemoryToImage	X	X	X
grCmdCopyImageToMemory	X	X	X
grCmdResolveImage	X		
grCmdCloneImageData	X		
grCmdUpdateMemory	X	X	X
grCmdFillMemory	X	X	X
grCmdClearColorImage	X	X	
grCmdClearColorImageRaw	X	X	
grCmdClearDepthStencil	X		
grCmdSetEvent	X	X	X
grCmdResetEvent	X	X	X
grCmdMemoryAtomic	X	X	
grCmdBeginQuery	X	X	
grCmdEndQuery	X	X	
grCmdResetQueryPool	X	X	X
grCmdWriteTimestamp	X	X	X
grCmdInitAtomicCounters	X	X	

Function	Universal queue	Compute queue	DMA queue (extension)
grCmdLoadAtomicCounters	X	X	
grCmdSaveAtomicCounters	X	X	

The grCmdClearColorImage and grCmdClearColorImageRaw functions support only non-target images on compute queue.

Table 25. Mantle command building debug function summary

Function	Universal queue	Compute queue	DMA queue (extension)
grCmdDbgMarkerBegin	X	X	X
grCmdDbgMarkerEnd	X	X	X

Table 26. Mantle command building extension function summary

Function	Universal queue	Compute queue	DMA queue (extension)
grCmdBindBorderColorPalette	X	X	
grCmdCopyOcclusionData	X		
grCmdSetOcclusionPredication	X		
grCmdResetOcclusionPredication	X		
grCmdSetMemoryPredication	X	X	X
grCmdResetMemoryPredication	X	X	X
grCmdIf	X	X	
grCmdElse	X	X	
grCmdEndIf	X	X	
grCmdWhile	X	X	
grCmdEndWhile	X	X	

Support for some of the functions is subject to feature capabilities reported by the GPU properties.