Explicit GL (XGL) Programming Guide and API Reference (Draft Proposal derived from the Mantle API)

Version 0.1
API revision 0.1
July 1, 2014

Chapter I.

INTRODUCTION

MOTIVATION

While existing platform programming models – OpenGL and DirectX® – 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 API attempts to bridge PC, mobile and consoles in terms of flexibility and performance, address efficiency problems, and provide a forward-looking, system level foundation for graphics programming.

High Level Khronos Goals

The GL common working group at Khronos is driving the development of a new cross-vendor standard. The stated primary goals are:

- Produce a split level API.
    - The lower level API is close to the hardware and runs with very little validation. This proposal addresses that need.
    - The higher level API is more similar to traditional OpenGL with hazard tracking, synchronization and other high level primitives. This proposal does not attempt to address this.

- Have the ability to implement core profile OpenGL on top of the lower level API.
- Ensure consistency, cleanliness, and unambiguity in the new API. Requirements include:
    - Type safety
    - Extensibility
    - Eliminate redundancy
- It is expected that debugging and validation be possible through opt-in layers.

SOLUTION OVERVIEW

The proposed solution implements a lower system level programming model designed for high performance graphics that makes the platform 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, Explicit GL provides a reasonable level of abstraction in terms of the pipeline definition and programming model. As a part of improving the programming model, the Explicit GL API removes some legacy features found in other graphics APIs.

While the proposed programming model draws somewhat on the strengths of OpenGL and DirectX®, 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 Explicit GL.

The Explicit GL API is not for everyone, due to its lower level control of memory and synchronization features. Effectively using 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 Explicit GL can still benefit a wide range of projects as specialized higher-level middle-ware Explicit GL-based solutions and engines become available.

DEVELOPER MANIFESTO

The Explicit GL API imposes a new set of rules upon platform graphics subsystem. Because of the abstraction level in Explicit GL, which is different from other graphics API solutions in the traditional OpenGL/DirectX space, some developer expectations need to be adjusted accordingly.

Explicit GL attempts to close a gap between existing platforms and consoles in terms of flexibility and performance by implementing a lower system-level programming model. In achieving this, Explicit GL 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 Explicit GL, developers need to take responsibility for their actions with extensive validation: fixing all instances of incorrect API usage, doing things efficiently 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, these problems can no longer be efficiently worked-around in the driver. This extra responsibility is the cost developers have to pay to benefit from Explicit GL advantages.

Explicit GL is really only for those graphics developers who are willing to accept this new level of responsibility.

Chapter II.

Programming Overview

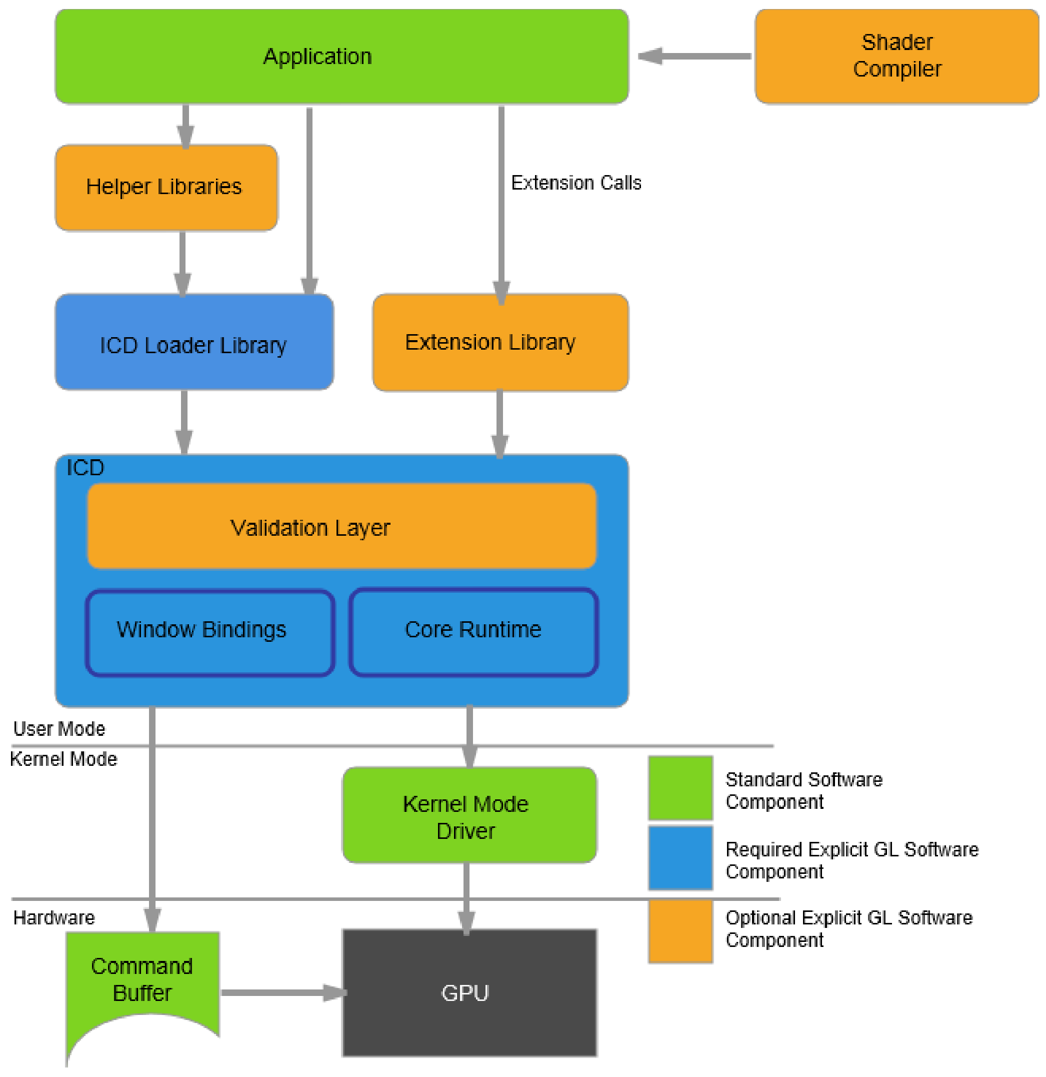
Software Infrastructure

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

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

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

Figure 1. XGL Software Infrastructure



The following static libraries are available:

Table 1. Statically Linked Explicit GL Libraries

Library file name	Description
xgl32.lib	32-bit static Explicit GL core API library
xgl64.lib	64-bit static Explicit GL core API library

The corresponding dynamic libraries for Microsoft Windows are (other Operating Systems will have similar solutions):

Table 2. Dynamically Linked Explict GL Libraries

Library file name	Description
xgl32.dll	32-bit Explicit GL loader and core API dynamic library
xgl64.dll	64-bit Explicit GL loader and core API dynamic library

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

Table 3. Explicit GL header files

Header file name	Description
xgl.h	Explicit GL core API
xglExt.h	Explicit GL extension interface
xglPlatform.h	Platform specific definitions
xglDbg.h	Explicit GL debug API
xglExtDbg.h	Debug features for Explicit GL extensions

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

Listing 1. Example of Checking presence and delay load Explicit GL library in an Operating system

```
// application is linked with /DELAYLOAD:XGL64.dll
XGL_RESULT InitXGL(
    const XGL_APPLICATION_INFO* pAppInfo,
    XGL_UINT*                   pGpuCount,
    XGL_PHYSICAL_GPU            gpus[XGL_MAX_PHYSICAL_GPUS])
{
    // Check Explicit GL library presence by trying to load it
    HMODULE hModule = LoadLibrary(TEXT("XGL64.dll"));
    if (hModule == NULL) {
        // Explicit GL library is not found
        return XGL_ERROR_UNAVAILABLE;
    } else {
        // Decrement Explicit GL library reference count and unload
        FreeLibrary(hModule);
        // Implicitly load library and initialize Explicit GL
        return xglInitAndEnumerateGpus(pAppInfo, NULL, pGpuCount, gpus);
    }
}
```

An application should avoid talking to Explicit GL 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, DMA, 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. Explicit GL provides a thin abstraction of this execution model.

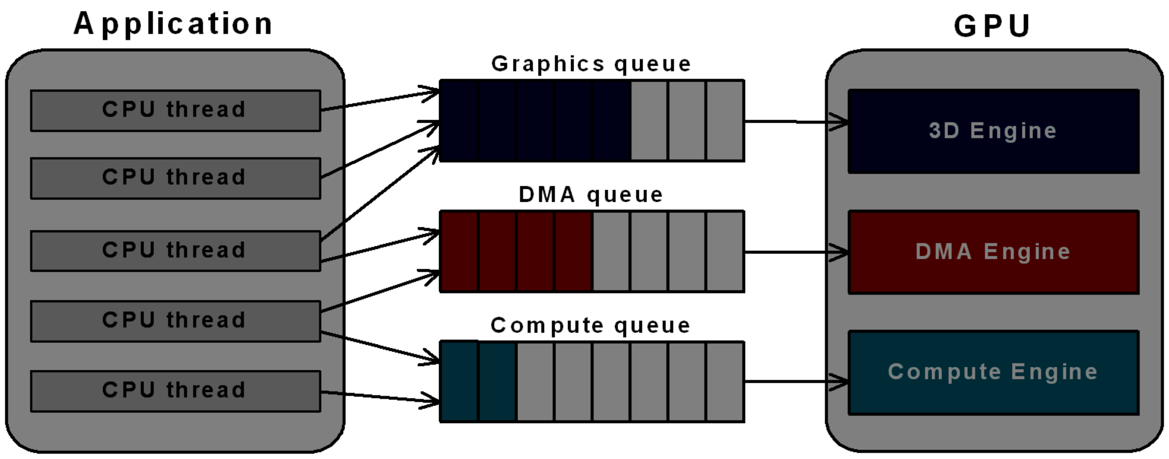


Figure 2. Queue submission model

An application in the Explicit GL programming environment controls the GPU devices by constructing command buffers containing native GPU commands through the Explicit GL 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 only minimal error checking while building command buffers in the release build of an application. Developers 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 that 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 CPU, an application can build independent command buffers on multiple CPU threads in a thread-safe manner.

After command buffers are built, they can be executed one or more times by the GPU device by submitting them to the appropriate queue. The Explicit GL 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 Explicit GL API are used to determine completion of various GPU operations and to synchronize CPU and GPU execution.

In Explicit GL, 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 Explicit GL 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 command buffer might result in resources not being paged in and a fault or incorrect rendering.

A system could include multiple Explicit GL capable GPUs, each of them exposed as a separate *physical GPU*. The Explicit GL 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 operation as required. The API provides functionality for an efficient implementation of multi-GPU rendering techniques.

MEMORY IN EXPLICIT GL

A Explicit GL *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 Explicit GL 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 Explicit GL 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 Explicit GL. In the case when a particular memory heap cannot be directly accessed by a CPU, the data is 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 EXPLICIT GL

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

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

Some of the objects might have requirements for binding GPU memory as described in section API Object Memory Binding. These memory requirements are implementation dependent.

The objects are created and destroyed through the Explicit GL API, though some of the objects are destroyed implicitly by Explicit GL. It is an application's responsibility to track 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.

Explicit GL 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 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 optional, connected in a predefined order. The capability of the graphics and compute pipelines is similar to that of DirectX® 11 and OpenGL 4.4. 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 Explicit GL 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 Explicit GL 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 XGL 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 Explicit GL with a window system is performed using a platform-specific *Window System Interface* (WSI) extension inter-operating with core Explicit GL API.

It is also possible to use Explicit GL 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 Explicit GL 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.

Explicit GL'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 Explicit GL are grouped in three categories:

- *Successful completion code* – XGL_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 (for example XGL_NOT_READY).
- *Error code* – returned when a function does not successfully complete due to error condition.

Because the Explicit GL 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 Explicit GL driver implementation relies on system recovery mechanisms such as *Timeout Detection and Recovery* (TDR) in the OS to detect GPU hang conditions and gracefully recover without a need to reboot the whole system.

LOST EXPLICIT GL DEVICES

An application is notified via XGL_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 xglInitAndEnumerateGpus(), and re-initializes all necessary devices 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 Explicit GL compatible platforms. Failure to address the warnings or errors could result in intermittent rendering or any other problems, even if the application might seem to perform correctly on some system configurations.

Chapter III.

Basic Explicit GL Operation

GPU Identification and Initialization

Each Explicit GL capable GPU in a system is represented by a *physical GPU* object referenced with a XGL_PHYSICAL_GPU object handle. There could be multiple physical GPUs visible to a Explicit GL 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. Explicit GL API objects cannot be shared across different devices. At any given time there can only be a single Explicit GL device per physical GPU per process.

To use Explicit GL, an application first needs to initialize and enumerate available physical GPU devices by calling xglInitAndEnumerateGpus(), which retrieves the number of physical GPUs and their object handles. If no Explicit GL capable GPUs are found in the system, xglInitAndEnumerateGpus() 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 Explicit GL. xglInitAndEnumerateGpus() can be called multiple times. Calling it more than once forces driver reinitialization.

Explicit GL 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 XGL_MAKE_VERSION macro is used to encode the API version, application, and engine versions provided on initialization in the XGL_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 Explicit GL 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 XGL_SYSTEM_ALLOC_TYPE type. When xglInitAndEnumerateGpus() is called multiple times, the same callbacks have to be provided on each invocation. Changing the callbacks on subsequent calls to xglInitAndEnumerateGpus() causes it to fail with XGL_ERROR_INVALID_POINTER error.

To make a selection of GPU devices suitable for an application's purpose, an application can retrieve GPU properties by using the xglGetGpuInfo() function. Basic physical GPU properties are retrieved with information type parameter set to XGL_INFO_TYPE_PHYSICAL_GPU_PROPERTIES, which are returned in XGL_PHYSICAL_GPU_PROPERTIES structure. GPU performance characteristics could be obtained with the information type parameter set to XGL_INFO_TYPE_PHYSICAL_GPU_PERFORMANCE, which returns performance properties in XGL_PHYSICAL_GPU_PERFORMANCE structure.

DEVICE CREATION

A device object in Explicit GL is referenced by the XGL_DEVICE handle and can be created using the xglCreateDevice() function for a given physical GPU device. Attempts to create multiple devices for the same physical GPU fail with XGL_ERROR_DEVICE_ALREADY_CREATED error code.

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 number of queues supported can be queried by using the xglGetGpuInfo() function with information type parameter set to XGL_INFO_TYPE_PHYSICAL_GPU_QUEUE_PROPERTIES.

To access advanced or platform-specific Explicit GL 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.

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 xglDestroyDevice(). 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 platform 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 xglGetMemoryHeapCount(). 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 xglGetMemoryHeapInfo() with infoType set to XGL_INFO_TYPE_MEMORY_HEAP_PROPERTIES value. The properties are returned in XGL_MEMORY_HEAP_PROPERTIES structure.

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

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

The Explicit GL 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 XGL_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 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 xglFreeMemory() 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 in when only one memory heap is available or when 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 Framebuffer Attachments, depth-stencil targets and write-access shader resources should typically use either high memory priority XGL_MEMORY_PRIORITY_HIGH or very high priority XGL_MEMORY_PRIORITY_VERY_HIGH. Most other objects should use normal priority XGL_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 XGL_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 xglSetMemoryPriority() with one of the values defined in XGL_MEMORY_PRIORITY.

CPU Access to GPU Memory Objects

Memory objects created with xglAllocMemory() represent a block of GPU virtual address space and by default are 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 xglMapMemory(). All of the memory heap choices for the mappable memory object must be CPU visible, which is indicated by XGL_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 XGL_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 is 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 xglUnmapMemory().

The xglMapMemory() and xglUnmapMemory() functions are thread safe, provided the different threads are accessing different memory objects.

Generally, it is advised to avoid keeping memory objects stored in local video memory heaps mapped when they are referenced by executing command buffers. On the Windows® platform, keeping memory objects mapped while using them for rendering results in migration of the memory objects to non-local video memory.

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 supportsPinning in XGL_PHYSICAL_GPU_MEMORY_PROPERTIES structure, which is retrieved by calling xglGetGpuInfo() function with the information type parameter set to XGL_INFO_TYPE_PHYSICAL_GPU_MEMORY_PROPERTIES.

A *pinned memory object* representing a pinned memory region is created using xglPinSystemMemory(). The pinned memory object is associated with the heap capable of holding pinned memory objects identified by the XGL_MEMORY_HEAP_HOLDS_PINNED flag, as if it were allocated from that heap. Explicit GL 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 XGL_MEMORY_HEAP_HOLDS_PINNED flag.

The memory is unpinned by destroying pinned memory object using the xglFreeMemory() 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 Explicit GL API allows reservation of GPU address space by exposing *virtual memory objects* that can be remapped later to *real memory objects*. Since Explicit GL GPUs operate in a virtual machine (VM) environment, all memory objects are part of the GPU virtual address space; however, 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 supportsVirtualMemoryRemapping in XGL_PHYSICAL_GPU_MEMORY_PROPERTIES structure, which is retrieved by calling xglGetGpuInfo() function with the information type parameter set to XGL_INFO_TYPE_PHYSICAL_GPU_MEMORY_PROPERTIES.

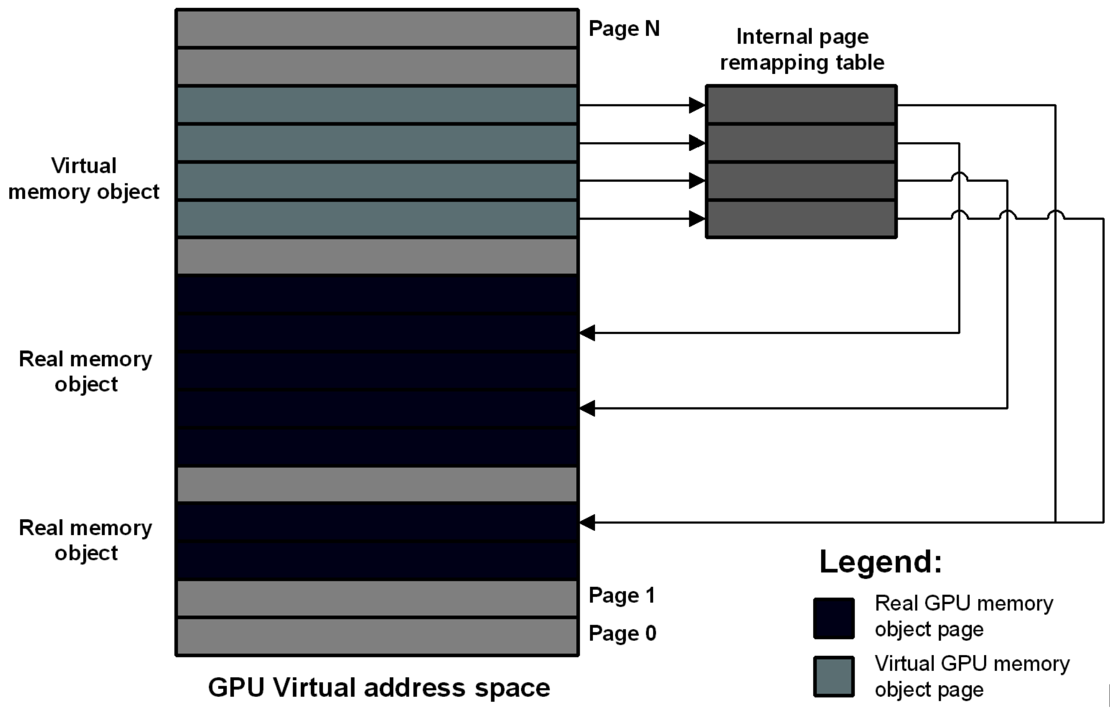

Page N
Internal page remapping table
Virtual memory object
Real memory object
Real memory object
Page 1
Page 0
GPU Virtual address space
Legend:
Real GPU memory object page
Virtual GPU memory object page

Figure 3.

Conceptual view of virtual memory remapping

Virtual memory objects are created by calling xglAllocMemory() with the XGL_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 xglFreeMemory() 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 a 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 xglRemapVirtualMemoryPages() function for managing virtual memory page remapping. The remapping functionality is only valid for virtual allocations and calls to xglRemapVirtualMemoryPages() with a real allocation or pinned memory object fail.

xglRemapVirtualMemoryPages() 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 XGL_NULL_HANDLE value for the target memory object they are remapped to.

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 application can guarantee proper execution order of operations using other methods. Before remapping, xglRemapVirtualMemoryPages() function waits on semaphores to be signaled; and after remapping it signals another set of semaphores, indicating completion of remapping. Multiple invocations of xglRemapVirtualMemoryPages() 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. Explicit GL 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 vs. 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 GPU memory manager ability to migrate memory objects by examining supportsMigration in XGL_PHYSICAL_GPU_MEMORY_PROPERTIES structure, which is retrieved by calling xglGetGpuInfo() function with the information type parameter set to XGL_INFO_TYPE_PHYSICAL_GPU_MEMORY_PROPERTIES.

In general, an application should avoid over-subscription of GPU memory to provide ideal memory object placement, which ensures high performance. In the operating sytems, where memory management is not completely under 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 operating systems video memory manager and parts of it rest on 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 XGL_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 (for example, 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 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 application worrying too much about their migration.

On the systems with relatively small visible local memory heap, 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 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 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 Windows® video memory manager.

GENERIC EXPLICIT GL API OBJECTS

The Explicit GL 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 xglDestroyObject() 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

Explicit GL 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 xglGetObjectInfo().

API OBJECT MEMORY BINDING

In Explicit GL, some API objects require video memory storage for their data. Developers are responsible for explicitly managing video memory allocations for these objects based on memory requirements reported at run-time. These API objects must be assigned 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 Explicit GL 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 Explicit GL driver. An application queries object memory requirements by calling xglGetObjectInfo() with a handle of the object of interest and the XGL_INFO_TYPE_MEM_REQUIREMENTS information type. The returned memory requirements include memory size, alignment and a list of compatible memory heaps.

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 the memory, an application should call xglBindObjectMemory() with the desired memory object handle and an offset within the memory object.

The memory alignment for some objects might be larger than 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.

The memory heap requirements for 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. Only the heaps on that list can be used for object memory placement. An application could filter the heaps according to its requirements; for example it could remove CPU invisible heaps to ensure 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.

> 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 to object binding if the memory heaps used for memory object creation do not match memory heap requirements of the particular API object, or if the memory placement requirements make the GPU object data extend past the memory object, or if the required memory alignment does not match the provided offset. The object is unbound from memory by specifying the XGL_NULL_HANDLE value for the memory object when calling xglBindObjectMemory() function.

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

> If pipeline objects have memory requirements, binding their memory automatically initializes the GPU memory by locking it and updating it with the CPU. If 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 might be migrated to non-local video memory in Windows®, resulting in degraded performance.

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. 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 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 is not initialized on memory binding and previous memory contents is preserved. The non-target images are assumed to be in the XGL_IMAGE_STATE_DATA_TRANSFER state upon memory binding. Images used as color targets or depth-stencil implicitly start in the XGL_IMAGE_STATE_UNINITIALIZED_TARGET 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 snapshot of image memory binding at the time of building a command buffer or descriptor set data is taken and recorded in command buffer or descriptor set on binding image to state or referencing image otherwise. 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 XGL_IMAGE_STATE_DATA_TRANSFER state before memory unbinding, while images used as color targets or depth-stencil must be transitioned to XGL_IMAGE_STATE_UNINITIALIZED_TARGET state. See Memory and Image States for more information about image states.

QUEUES AND COMMAND BUFFERS

In Explicit GL all commands are sent to 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

Explicit GL 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 xglGetGpuInfo() function with the information type parameter set to XGL_INFO_TYPE_PHYSICAL_GPU_QUEUE_PROPERTIES, which returns an array of XGL_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 xglGetGpuInfo() with a NULL data pointer. The expected data size for all queue property structures is returned in pDataSize.

Explicit GL API defines two queue types: a universal queue (XGL_QUEUE_UNIVERSAL) and an asynchronous compute queue (XGL_QUEUE_COMPUTE_ONLY). Other queue types, such as DMA and so on can be exposed through extensions. There is at least one universal queue available for the Explicit GL 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 Explicit GL are referenced using XGL_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 xglGetDeviceQueue() 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.

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 Explicit GL are referenced using XGL_CMD_BUFFER object handles. A command buffer can be executed by the GPU multiple times and recycled, provided that 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 xglCreateCommandBuffer(). 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 xglDestroyObject().

Command Buffer Building

The Explicit GL 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 xglBeginCommandBuffer() 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 xglEndCommandBuffer(). 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 xglBeginCommandBuffer() and xglEndCommandBuffer() 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 running with validation layer enabled.

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 xglBeginCommandBuffer() implicitly recycles the command buffer before starting a new recording session. An application could explicitly recycle the command buffer by calling xglResetCommandBuffer(). 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.

It is allowed to record and submit empty command buffers with no actual commands between xglBeginCommandBuffer() and xglEndCommandBuffer() calls.

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

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 xglEndCommandBuffer(). The error is not returned during the command buffer construction, and command buffer building function silently fail unless running with validation layer enabled. An application must be able to gracefully handle a case when termination of a command buffer fails.

COMMAND BUFFER OPTIMIZATIONS

At command buffer building time an application specifies optional optimization hints that could help the Explicit GL driver to tailor command buffer contents for different performance scenarios. Specifying the XGL_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 XGL_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 XGL_CMD_BUFFER_OPTIMIZE_PIPELINE_SWITCH hint optimizes command buffer for cases when application frequently changes pipelines between draw and dispatch operations. Similarly, XGL_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 vs. the GPU performance.

An application could detect at run time if it is CPU or GPU bound and in which parts of the frame 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 xglQueueSubmit(). 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 xglQueueSubmit() 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.

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

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 XGL_MEMORY_REF structure. The supplied memory object handle cannot be XGL_NULL_HANDLE. It is an application's responsibility to guarantee completeness of the memory references list. This includes all memory used by all Explicit GL 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 command buffer execution time.

There are two complimentary methods for supplying memory references. First, a list of memory references is specified at command buffer submission time. Second, a set of *global memory references* is made available on a per-queue basis using xglQueueSetGlobalMemReferences() 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 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 global memory references is taken at submission time and applied to submitted command buffers. Changing global memory references does not apply to already submitted command buffers.

The xglQueueSetGlobalMemReferences() 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 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 XGL_MEMORY_REF_READ_ONLY flag to indicate that memory object is used for read-only GPU access and its contents will not change during 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
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

Operation	Access Type
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

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 Explicit GL API supports dispatching compute operations using a compute pipeline and currently bound command buffer compute state. The compute is dispatched with explicit work dimensions by calling xglCmdDispatch(), 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 xglCmdDispatchIndirect() function. The dispatch argument data must be 4-byte aligned and the memory range containing the indirect data must be in the XGL_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	XGL_UINT32	Number of thread groups in X direction
0x04	XGL_UINT32	Number of thread groups in Y direction
0x08	XGL_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 currently bound command buffer graphics state. All parts of the state must be properly set for 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 xglCmdDraw() function for rendering both instanced and non-instanced objects. Indexed geometry can be rendered with xglCmdDrawIndexed(). Indexed geometry can only be rendered when valid index data is bound to command buffer state with xglCmdBindIndexData(). If objects are not instanced, the firstInstance should be set to zero and 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, Explicit GL supports indirect draw functions whose execution is driven by data stored in GPU memory objects. Indirect rendering is performed by either calling xglCmdDrawIndirect() or xglCmdDrawIndexedIndirect() function, depending on 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 XGL_MEMORY_STATE_INDIRECT_ARG state. The layout of the indirect draw argument data is shown in Table 6 and Table 7.

Multiple draws can be launched from a single call to xglCmdDrawIndirect() or xglCmdDrawIndexedIndirect(). They each have count and stride arguments that specify how many draws to launch, and the stride in memory for each draw's argument data.

Table 6. Argument data layout for indirect draw

Offset	Data type	Description
0x00	XGL_UINT32	Number of vertices per instance
0x04	XGL_UINT32	Number of instances
0x08	XGL_INT32	Vertex offset
0x0C	XGL_UINT32	Instance offset

Table 7. Argument data layout for indexed indirect draw

Offset	Data type	Description
0x00	XGL_UINT32	Number of indices per instance
0x04	XGL_UINT32	Number of instances
0x08	XGL_UINT32	Index offset
0x0C	XGL_INT32	Vertex offset
0x10	XGL_UINT32	Instance offset

PRIMITIVE TOPOLOGY

Explicit GL 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

Explicit GL supports occlusion and pipeline statistics queries. Occlusion queries are only available on universal queues while pipeline statistic queries are available on universal and compute queues.

Queries in the Explicit GL API are managed using query pools – homogeneous collections of queries of a certain type. Query pools are represented by XGL_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 xglCreateQueryPool().

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 xglCmdResetQueryPool(). 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 xglCmdBeginQuery() and xglCmdEndQuery() 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 xglCmdBeginQuery() function call with xglCmdEndQuery(), results in an undetermined query result value, invalid query completion status, and could produce an undetermined rendering result. For example, calling xglCmdBeginQuery() twice in a row matched by a single xglCmdEndQuery() call, or matching a single xglCmdBeginQuery() call with multiple xglCmdEndQuery() is not allowed.

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

An application retrieves results of any query in a pool by calling xglGetQueryPoolResults().

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 XGL_NOT_READY return code, the returned data is undefined for all requested query slots. An application must ensure there is enough space provided to store results for all requested query slots. Calling xglGetQueryPoolResults() 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 application needs to perform a lot of frequent query checks, and memory assignment for query pool objects allow it, the query pool objects can be bound to pinned memory. This ensures expensive memory map operations are not performed.

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

TIMESTAMPS

For timing the execution of operations in command buffers, Explicit GL provides ability to write GPU *timestamps* to memory from command buffers using xglCmdWriteTimestamp() 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 a difference of 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 pipe timestamp is written out when 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 XGL_MEMORY_STATE_WRITE_TIMESTAMP state using an appropriate preparation operation.

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

SYNCHRONIZATION

The Explicit GL API provides a comprehensive set of synchronization primitives to synchronize between CPU and 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.

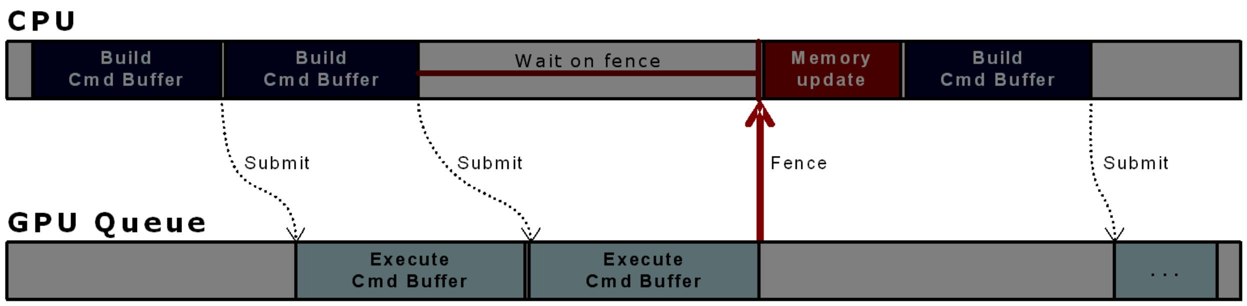


Figure 4. Synchronization with fences

A fence object, represented by XGL_FENCE object handle, can be created by calling xglCreateFence() 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 xglGetFenceStatus() function. If the fence has not been reached, the XGL_NOT_READY code is returned to the application. An attempt to check fence status before it is submitted returns XGL_ERROR_UNAVAILABLE error code.

An application can also sleep one of its threads while waiting for a fence or a group of fences to come back by calling xglWaitForFences(). If multiple fences are specified and the xglWaitForFences() 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.

EVENTS

Events in Explicit GL can be used for more fine-grain synchronization between a GPU and a CPU than fences, as 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 CPU. The events in Explicit GL are represented by XGL_EVENT object handle.

Event objects are created by calling xglCreateEvent() function, and are set and reset by the CPU by using xglSetEvent() and xglResetEvent() functions. From command buffers the events are similarly manipulated using xglCmdSetEvent() and xglCmdResetEvent() functions. Event operations are supported by both universal and compute queues.

An application checks the event's state using the CPU by calling xglGetEventStatus(). When created, the event starts in undefined state and it should be explicitly set or reset before it can be queried.

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 allow 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 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

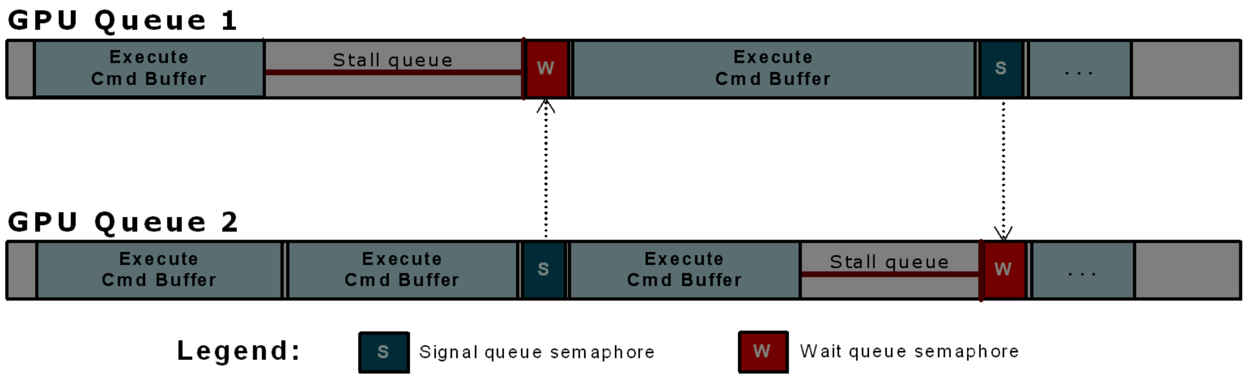


Figure 5. Queue synchronization with semaphores

required order of execution.

Queue semaphore objects are represented by XGL_QUEUE_SEMAPHORE object handles and are created by calling xglCreateQueueSemaphore(). At creation time an application can specify an initial semaphore count.

An application issues *signal* and *wait* operations on the queues by calling xglSignalQueueSemaphore() and xglWaitQueueSemaphore() 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 the 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 Explicit GL API provides functions xglQueueWaitIdle() and xglDeviceWaitIdle() 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. xglDeviceWaitIdle() 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 Explicit GL 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 to be in the XGL_MEMORY_STATE_QUEUE_ATOMIC state.

An atomic operation can be recorded in a command buffer using xglCmdMemoryAtomic(). The memory offset for 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 Explicit GL 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 Explicit GL 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 xglCmdInitAtomicCounters() or by copying the data from a memory object using xglCmdLoadAtomicCounters(). An atomic counter value could also be saved into a memory location using xglCmdSaveAtomicCounters().

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 has to be in the XGL_MEMORY_STATE_DATA_TRANSFER state before issuing the load or save operation.

Chapter IV.

RESOURCE OBJECTS AND VIEWS

The Explicit GL GPU operates on data stored in memory objects. There are several ways the data can be accessed depending on its intended use. Texture and Framebuffer Attachment data is represented by *image* objects and is 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 Explicit GL API representing them due to 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 shader sees and interprets the raw data in memory: a format and element stride could be specified. The memory view is defined by XGL_DYNAMIC_MEMORY_VIEW_SLOT_INFO structure.

Interpretation of memory view data depends on 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. 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 Explicit GL are containers used to store texture data. They are also used for color Attachments 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 is to be interpreted by the GPU, Explicit GL 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 Explicit GL 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 later be overwritten with a compatible format at view creation time. The image dimensions are specified in texels for the topmost mip 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 xglCreateImage(). 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, for example, data transfer, it is allowed to omit all usage flags.

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 Explicit GL:

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

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

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

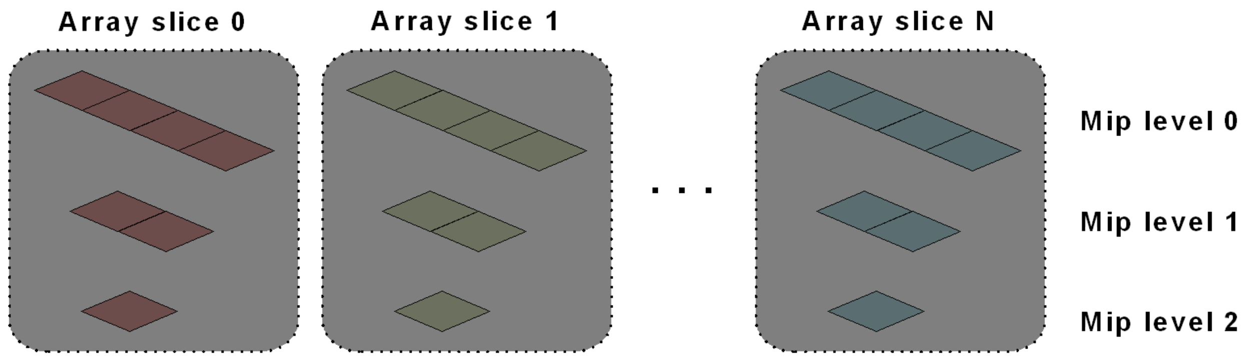


Figure 6. 1D image organization

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

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.

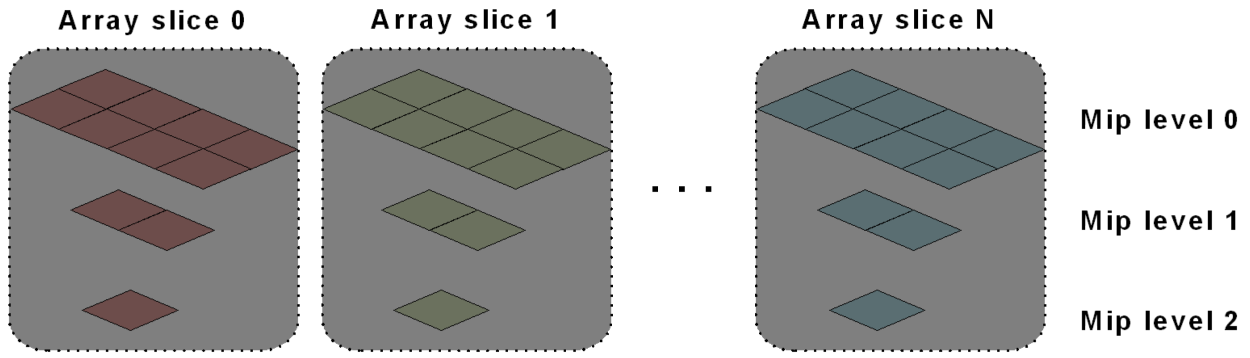


Figure 7. 2D image organization

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.

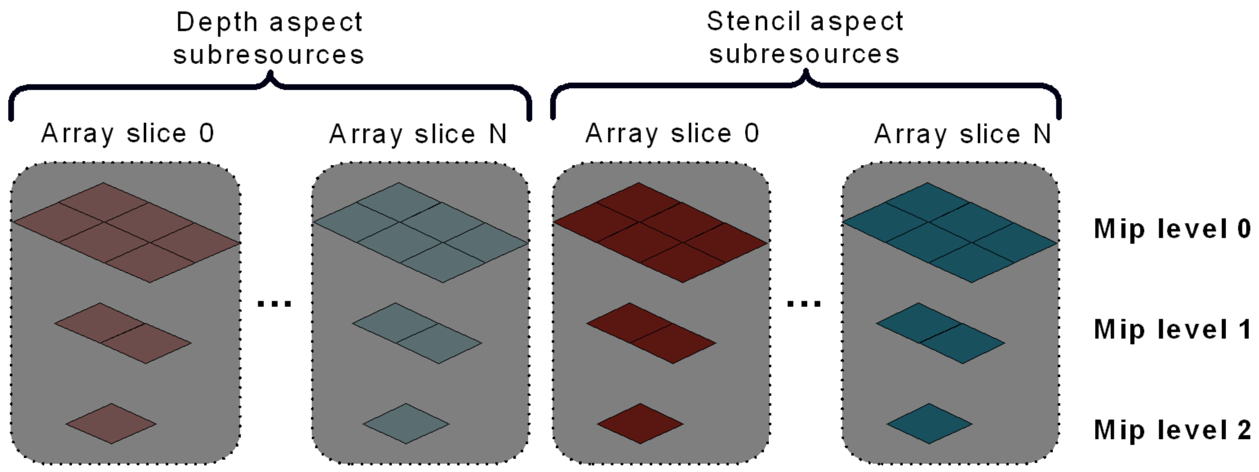


Figure 8. Depth-stencil image organization

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 mip-mapped volume starting with the

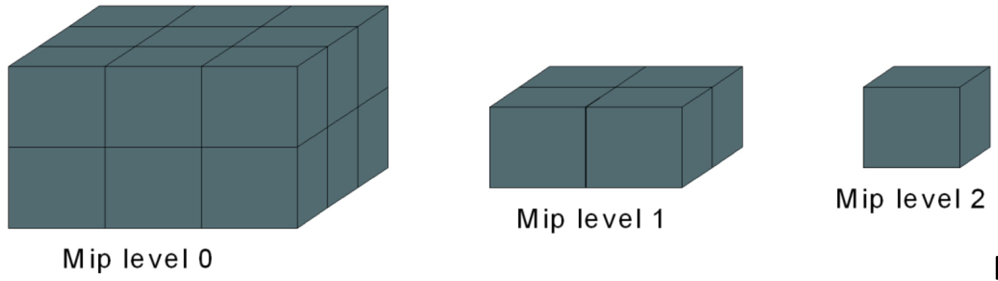


Figure 9. 3D image organization

topmost mip-level. An example 3D image organization is show in the Figure 9.

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 image width is padded to a required stride. While simple and efficient for CPU access, the linear tiling does not play well with 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 XGL_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 Explicit GL, 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 (color targets, depth-stencil targets and multisampled images). The primary use for the transparent images is data transfer to and from the GPU.

RESOURCE FORMATS AND CAPABILITIES

The *resource format* is used for specifying image element type and memory view element type for shader access. It is specified using a XGL_FORMAT format descriptor that contains information about the *numeric format* and the *channel format*. The numeric format describes how the data is to be interpreted while the channel specification describes the number of channels and their bit depth. The XGL_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 different encoding scheme per block, and formats with alternative channel ordering which are used to handle certain OS-specific interoperability issues, such as XGL_CH_FMT_B5G6R5 and XGL_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 xglGetFormatInfo(). A separate set of capabilities is reported for linear and optimal tiling modes in XGL_FORMAT_PROPERTIES structure.

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 (XGL_CH_FMT_BC1 through XGL_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. 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 on images of compatible types and should represent a valid subset of image subresources. The resource usage flags should have XGL_IMAGE_USAGE_SHADER_ACCESS_READ and/or XGL_IMAGE_USAGE_SHADER_ACCESS_WRITE set for successful creation of image views of all types.

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 xglCreateImageView(). 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.

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 mip-map levels and array slices has to be a subset of the subresources in the parent image. If application wants to use all mip-levels or slices in an image, the number of mip-levels or slices can be set to a special value of XGL_LAST_MIP_OR_SLICE without knowing the exact number of mip-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 XGL_CH_FMT_R4G4 or compressed format from a shader results in undefined behavior. Similarly, attempting to write to an image that did not have XGL_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 XGL_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.

Framebuffer Attachments

In Explicit GL there are two different types of Framebuffer Attachments:

- Color Attachments
- Depth-stencil Framebuffer Attachments

Color Attachments

Color Attachments are 2D or 3D image objects created with the XGL_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 xglCreateColorTargetView(). A color target view can represent a contiguous range of image array slices at any particular mip level. A color target view cannot reference multiple mip levels.

A variety of different formats is supported for color Attachments. A valid image format must be specified for the color target view. It can be different from image format, provided the view format is compatible with the format of the parent image.

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

Depth-Stencil Framebuffer Attachments

The *depth-stencil targets* are represented by depth-stencil views created from 2D image marked with XGL_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 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 xglCreateDepthStencilView().

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. The Table 10 list 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)
XGL_CH_FMT_R8 / XGL_NUM_FMT_DS	N/A	XGL_CH_FMT_R8 / XGL_NUM_FMT_UINT
XGL_CH_FMT_R16 / XGL_NUM_FMT_DS	XGL_CH_FMT_R16 / XGL_NUM_FMT_UINT	N/A
XGL_CH_FMT_R32 / XGL_NUM_FMT_DS	XGL_CH_FMT_R32 / XGL_NUM_FMT_FLOAT	N/A
XGL_CH_FMT_R16G8 / XGL_NUM_FMT_DS	XGL_CH_FMT_R16 / XGL_NUM_FMT_UINT	XGL_CH_FMT_R8 / XGL_NUM_FMT_UINT
XGL_CH_FMT_R32G8 / XGL_NUM_FMT_DS	XGL_CH_FMT_R32 / XGL_NUM_FMT_FLOAT	XGL_CH_FMT_R8 / XGL_NUM_FMT_UINT

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 target are simultaneously bound to command buffer state with xglCmdBindTargets(). 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 xglCmdBindTargets(). 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 per target image state that represents the expected state for all subresources in the view at the draw time. For depth-stencil view a separate state is specified for depth and stencil aspects. The depth and stencil states could be different, for example, in case of read-only depth or stencil. For unused color targets, as well as for unused depth-stencil aspects an application should specify

XGL_IMAGE_STATE_UNINITIALIZED_TARGET state.

READ-ONLY DEPTH-STENCIL VIEWS

Read-only depth-stencil view allows rendering with read-only access to 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 XGL_DEPTH_STENCIL_VIEW_CREATE_READ_ONLY_DEPTH flag and read-only stencil by XGL_DEPTH_STENCIL_VIEW_CREATE_READ_ONLY_STENCIL flag at depth-stencil creation time.

If depth or stencil aspect is used for simultaneous read access as depth-stencil target and as an image view from the graphic shaders, it has to be in XGL_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 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. Generally, 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 Framebuffer Attachment 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 Explicit GL, 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 means of sharing data, saving memory and reducing memory copy operations. The subresources of transparent images (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.

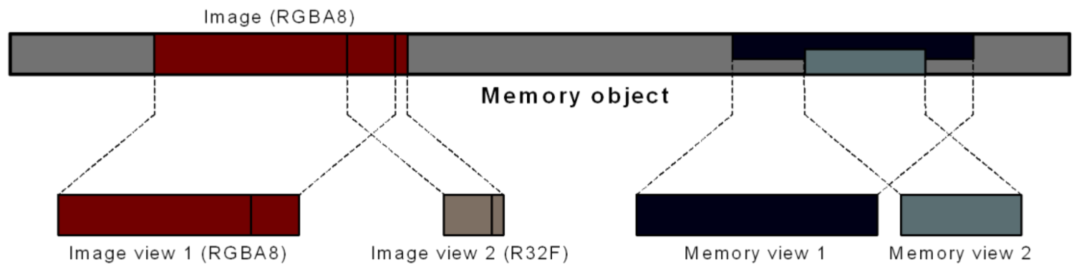


Figure 10.

Examples of data aliasing in Explicit GL

No assumption about preserving memory contents should be made when reusing memory between multiple target images (for example, depth-stencil targets, color Attachments, 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 image is in XGL_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 XGL_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 XGL_IMAGE_FLAG_INVARIANT_DATA flag is specified at image creation time for both old and new image object.

RESOURCE STATES AND PREPARATION

When the GPU accesses a memory or an image, the memory range or image is assumed to be in a particular *state* that matches the GPU expectations for its behavior with respect to cache residency, state of the meta-data, and so on. There has to be consistency between the memory state or the image state and its current GPU resource usage to produce correct results. The Explicit GL driver does not keep track of the persistent memory or image state, nor does it track hazard conditions for performance reasons. In Explicit GL, it becomes an application's responsibility to track memory and image state states and ensure their consistency with operations performed by the GPU. For some operations, an application also must communicate to the driver the current state at the time of performing the operation.

In Explicit GL, 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 XGL_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 XGL_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 range state needs to be tracked for 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 variety of states depending on the pipeline and resource access type, while memory containing draw index data has to be only in XGL_MEMORY_STATE_INDEX_DATA state. The color targets or depth-stencil images could be in either XGL_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL or XGL_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL state, which is communicated to Explicit GL at the target bind time. Naturally, the XGL_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 XGL_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL state for rendering. A list of allowed states modes for various operations in Explicit GL is presented in table below.

Table 11. Allowed resource states for various operations

Operation or usage	Allowed resource states
CPU resource access	XGL_MEMORY_STATE_DATA_TRANSFER XGL_IMAGE_STATE_DATA_TRANSFER
GPU resource copy	XGL_MEMORY_STATE_DATA_TRANSFER XGL_IMAGE_STATE_DATA_TRANSFER
Immediate memory update	XGL_MEMORY_STATE_DATA_TRANSFER
Load/save atomic counter	XGL_MEMORY_STATE_DATA_TRANSFER
Copy occlusion data	XGL_MEMORY_STATE_DATA_TRANSFER
Queue atomics	XGL_MEMORY_STATE_QUEUE_ATOMIC
Write timestamp	XGL_MEMORY_STATE_WRITE_TIMESTAMP
Resource cloning	Any image state except XGL_IMAGE_STATE_UNINITIALIZED_TARGET
Indirect draw/dispatch argument data	XGL_MEMORY_STATE_INDIRECT_ARG
Index data	XGL_MEMORY_STATE_INDEX_DATA
Graphics shader access	XGL_MEMORY_STATE_GRAPHICS_SHADER_READ_ONLY XGL_MEMORY_STATE_GRAPHICS_SHADER_WRITE_ONLY XGL_MEMORY_STATE_GRAPHICS_SHADER_READ_WRITE XGL_MEMORY_STATE_MULTI_SHADER_READ_ONLY XGL_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY XGL_IMAGE_STATE_GRAPHICS_SHADER_WRITE_ONLY XGL_IMAGE_STATE_GRAPHICS_SHADER_READ_WRITE XGL_IMAGE_STATE_TARGET_AND_SHADER_READ_ONLY
Compute shader access	XGL_MEMORY_STATE_COMPUTE_SHADER_READ_ONLY XGL_MEMORY_STATE_COMPUTE_SHADER_WRITE_ONLY XGL_MEMORY_STATE_COMPUTE_SHADER_READ_WRITE XGL_MEMORY_STATE_MULTI_SHADER_READ_ONLY XGL_IMAGE_STATE_COMPUTE_SHADER_READ_ONLY XGL_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY XGL_IMAGE_STATE_COMPUTE_SHADER_READ_WRITE XGL_IMAGE_STATE_MULTI_SHADER_READ_ONLY

Operation or usage	Allowed resource states
Color targets	XGL_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL XGL_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL
Depth-stencil targets	XGL_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL XGL_IMAGE_STATE_TARGET_SHADER_ACCESS_OPTIMAL XGL_IMAGE_STATE_TARGET_AND_SHADER_READ_ONLY
Image clear	XGL_IMAGE_STATE_CLEAR
Resolve source	XGL_IMAGE_STATE_RESOLVE_SOURCE
Resolve destination	XGL_IMAGE_STATE_RESOLVE_DESTINATION

When memory objects are created or non-target images are bound to memory, they are assumed to be in XGL_MEMORY_STATE_DATA_TRANSFER or XGL_IMAGE_STATE_DATA_TRANSFER state. Images that could be used as color Attachments or depth-stencil buffers are assumed to be in XGL_IMAGE_STATE_UNINITIALIZED_TARGET state when bound to memory and have to be transitioned to an appropriate state on a graphics capable queue.

Before unbinding GPU updated images from memory, an application transitions target images to XGL_IMAGE_STATE_UNINITIALIZED_TARGET state and non-target images to XGL_IMAGE_STATE_DATA_TRANSFER state. This ensures the GPU caches are properly flushed and avoids a possibility of data corruption.

STATE PREPARATIONS

An application indicates a memory range or an image state transition by adding special preparation commands into the GPU command buffer before 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 xglCmdPrepareMemoryRegions(). Images are similarly prepared by using xglCmdPrepareImages().

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.

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 executed, the Framebuffer Attachment and depth-stencil view of that image cannot be bound in a command buffer, 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.

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 XGL_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL to XGL_IMAGE_STATE_COMPUTE_SHADER_READ_ONLY transition. The only exceptions to this are that transitions from any of the XGL_MEMORY_STATE_DATA_TRANSFER, XGL_IMAGE_STATE_DATA_TRANSFER and XGL_IMAGE_STATE_UNINITIALIZED_TARGET states should be performed on the queue that will use resources next.

Failing to prepare memory range or image on the queue that was last to update or otherwise use 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 state transition.

Multisampled Image Preparation

Preparation of multisampled images requires a correct 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 xglCmdPrepareImages() function.

HAZARDS

The Explicit GL 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 Explicit GL 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 Attachment writes – the Fragments of the second draw are not written until all of the Fragments from the first draw are written to the targets. Explicit GL also guarantees ordering of copy operations for memory ranges in XGL_MEMORY_STATE_DATA_TRANSFER state and images in XGL_IMAGE_STATE_DATA_TRANSFER state. 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 to 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 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 to the same state the image or memory is in. 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 the XGL_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY to XGL_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY. While there is not an actual transition of state, such preparation would be an indication to the Explicit GL driver of a write-after-write hazard condition. Inserting hazard processing ensures non-overlapping nature of the copy operations.

There is never a write-after-write hazard when performing operations on memory in XGL_MEMORY_STATE_DATA_TRANSFER state and on images in XGL_IMAGE_STATE_DATA_TRANSFER state. When performing back-to-back copies of data, the Explicit GL 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 – for example, change from shader readable state to Framebuffer Attachment use.

Table 12. Hazard and state transition examples

Usage scenario	Hazard	Transition
Read the Framebuffer Attachment previously in render-optimal state	RAW	XGL_IMAGE_STATE_TARGET_RENDER_ACCESS_OPTIMAL to XGL_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY
Write to image from compute shader after it was read by graphics pipeline	WAR	XGL_IMAGE_STATE_GRAPHICS_SHADER_READ_ONLY to XGL_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY
Write to image from compute shader on consecutive dispatches	WAW	XGL_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY to XGL_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY
Write to image from Fragment shader (non-target write) on consecutive draws	WAW	XGL_IMAGE_STATE_GRAPHICS_SHADER_WRITE_ONLY to XGL_IMAGE_STATE_GRAPHICS_SHADER_WRITE_ONLY
Draw indirect with data generated by the compute shader	RAW	XGL_IMAGE_STATE_COMPUTE_SHADER_WRITE_ONLY to XGL_MEMORY_STATE_INDIRECT_ARG
Draw indirect with data loaded by the CPU	N/A	XGL_MEMORY_STATE_DATA_TRANSFER to XGL_MEMORY_STATE_INDIRECT_ARG
Draw with indices output by the compute shader	RAW	XGL_MEMORY_STATE_COMPUTE_SHADER_WRITE_ONLY to XGL_MEMORY_STATE_INDEX_DATA
Back-to-back image clears	WAW	XGL_IMAGE_STATE_CLEAR to XGL_IMAGE_STATE_CLEAR
Reading the GPU timestamp data by the CPU	N/A	XGL_MEMORY_STATE_WRITE_TIMESTAMP to XGL_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 Explicit GL 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 xglCmdCopyMemory() and a portion of an image could be copied to another image with xglCmdCopyImage(). The image data can also be copied to and from memory using xglCmdCopyImageToMemory() and xglCmdCopyMemoryToImage(). 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 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, but they cannot be used as copy destination. An attempt to copy to a depth image produces undefined behavior.

> If 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 Fragment 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 source and destination image formats support conversion, which is indicated by XGL_FORMAT_CONVERSION format capability flag. In that case the Fragment size (or compression block size for compressed images) has to match, and raw image data is 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 XGL_MEMORY_STATE_DATA_TRANSFER and images must be transitioned to the XGL_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 XGL_MEMORY_STATE_DATA_TRANSFER or XGL_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. 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.

RESOURCE CLONING

The image copy operations described in Resource Copies, while flexible, require images to be put into the XGL_IMAGE_STATE_DATA_TRANSFER 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 xglCmdCloneImageData().

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 XGL_IMAGE_CREATE_CLONEABLE flag. 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, for example to quickly feed new parameter values into shaders. In the Explicit GL API this can be accomplished by using xglCmdUpdateMemory(). 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 XGL_MEMORY_STATE_DATA_TRANSFER 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 can be queried from the physical GPU properties (see GPU Identification and Initialization); however it is guaranteed to be at least 256 DWORDs.

RESOURCE UPLOAD STRATEGIES

Explicit GL 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 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 is loaded into that *staging area* using the CPU memory copy. Second, the memory or image data is copied to the final destination using the GPU. If a DMA queue 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 xglCmdFillMemory(). The destination and fill size have to be 4-byte aligned. The memory range needs to be in the XGL_MEMORY_STATE_DATA_TRANSFER 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 xglCmdClearColorImage() or xglCmdClearColorImageRaw(). Depth-stencil targets can be cleared by calling xglCmdClearDepthStencil(). 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 XGL_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 number of unique clear colors per image.

If multiColorTargetClears in GPU properties reports XGL_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 image to multiple values on GPUs that do not support that capability produces undefined results.

When only a subset of a resource needs to be cleared is smaller than 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 xglCmdClearColorImage() 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 XGL_FORMAT_CONVERSION capability flag exposed. For the sRGB formats the clear color is specified in a linear space, which is appropriately converted by the Explicit GL driver to sRGB color space. Raw clears perform no format conversion and are available for all image formats. The provided clear data is directly stored regardless of the format's numeric type (including sRGB formats). xglCmdClearColorImageRaw() takes a number of least significant bits from per-channel UINT 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 xglCmdClearDepthStencil() 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 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 Explicit GL. The built-in implementation understands all sample counts and is guaranteed to work on a variety of formats. The resolve operation could be recorded into a command buffer using xglCmdResolveImage(). 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 an appropriate preparation commands, designating them as resolve source and destination using the XGL_IMAGE_STATE_RESOLVE_SOURCE and XGL_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 Explicit GL by XGL_SAMPLER handle, describe how images are processed (for example filtered, converted and so on) on a texture fetch operation. A sampler object is created by calling xglCreateSampler().

The core Explicit GL specification defines a limited support for border colors available to application when XGL_TEX_ADDRESS_CLAMP_BORDER addressing mode is used. Available border colors are specified using XGL_BORDER_COLOR_TYPE. More options for border colors may be exposed through an 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 command buffer state to be accessed by the shaders of the currently active pipeline. A descriptor set is created by calling xglCreateDescriptorSet().

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 is 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 descriptor set and its bindings is shown in Figure 11.

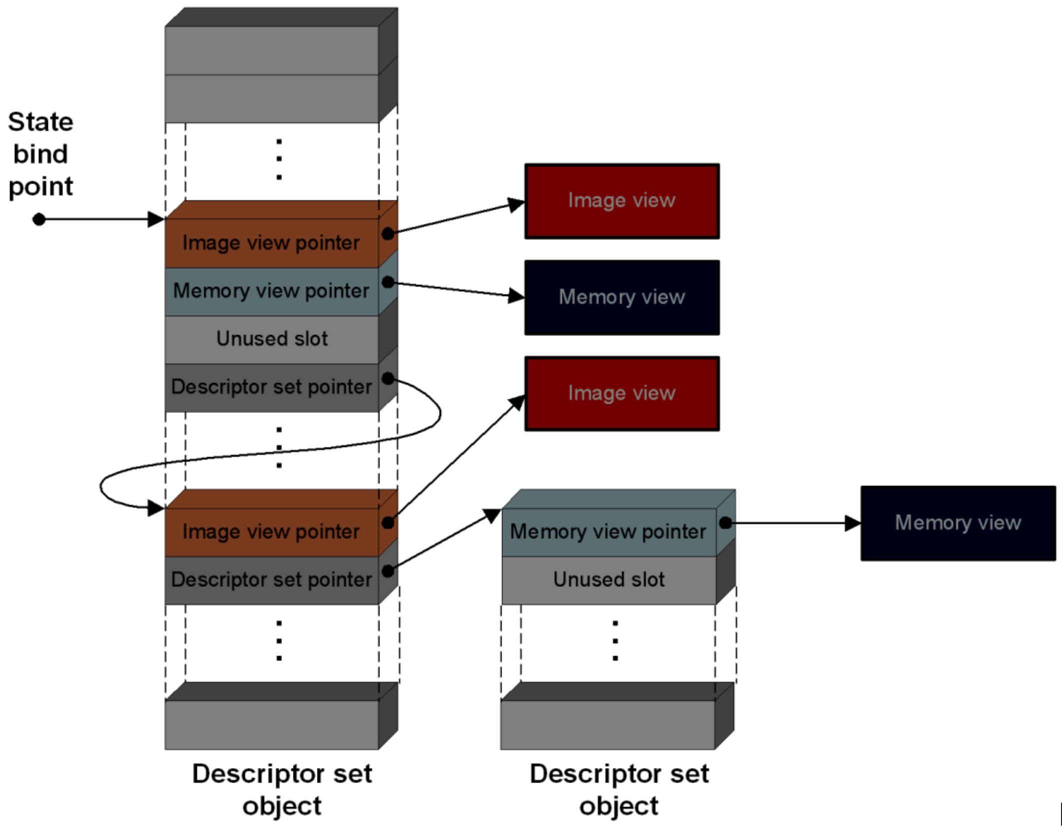


Figure 11.

Descriptor set binding example

Explicit GL 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. 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 provides a wide range of CPU and GPU performance trade-offs. One example of such strategy is to divide sampler and resource objects into separate descriptor sets: one dedicated to resources and another for samplers – 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 xglBeginDescriptorSetUpdate(), 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 – xglAttachMemoryViewDescriptors() for memory views, xglAttachImageViewDescriptors() for image views, xglAttachSamplerDescriptors() for samplers and xglAttachNestedDescriptors() for building hierarchical descriptor sets. After an update is complete, an application calls xglEndDescriptorSetUpdate(). Failure to match xglBeginDescriptorSetUpdate() with a call to xglEndDescriptorSetUpdate(), or performing an update without beginning the update, results in undefined behavior.

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 images memory binding are not reflected in previously built descriptor sets. The memory for shader access is bound as described in Memory Views.

For performance reasons it is recommended to avoid calling xglBeginDescriptorSetUpdate() and updating descriptor set while the memory object associated with descriptor set is used for rendering.

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 xglClearDescriptorSetSlots(). 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 possibility of a memory mapping operation on a call to xglBeginDescriptorSetUpdate() and memory unmapping on a call to xglEndDescriptorSetUpdate(). In 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 case of a large descriptor set used as a pool, only a single set of xglBeginDescriptorSetUpdate() and xglEndDescriptorSetUpdate() 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

Explicit GL 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 Explicit GL – 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
Framebuffer Attachments	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 Explicit GL :

- Rasterizer state (XGL_RASTER_STATE_OBJECT)
- Viewport and scissor state (XGL_VIEWPORT_STATE_OBJECT)
- Color blender state (XGL_COLOR_BLEND_STATE_OBJECT)
- Depth-stencil state (XGL_DEPTH_STENCIL_STATE_OBJECT)
- Multisampling state (XGL_MSAA_STATE_OBJECT)

Explicit GL API specifies a set of matching bind points that state objects blocks can be attached to using xglCmdBindStateObject(). 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 XGL_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 xglCreateRasterState(). The rasterizer state is bound to XGL_STATE_BIND_RASTER binding point.

Viewport and Scissor State

The viewport state object is represented by XGL_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 xglCreateViewportState(). The viewport state is bound to XGL_STATE_BIND_VIEWPORT binding point.

Color Blender State

The color blender state object is represented by XGL_COLOR_BLEND_STATE_OBJECT handle. It describes color blending state for the pipelines that enable blending operations. The color blender state is created by calling xglCreateColorBlendState(). The color blender state is bound to XGL_STATE_BIND_COLOR_BLEND binding point.

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

- XGL_BLEND_SRC1_COLOR
- XGL_BLEND_ONE_MINUS_SRC1_COLOR
- XGL_BLEND_SRC1_ALPHA
- XGL_BLEND_ONE_MINUS_SRC1_ALPHA

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

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

Depth-stencil State

The depth-stencil state object is represented by XGL_DEPTH_STENCIL_STATE_OBJECT handle. It describes depth-stencil test operations in the graphics pipeline. The depth-stencil state is created by calling xglCreateDepthStencilState(). The depth-stencil state is bound to XGL_STATE_BIND_DEPTH_STENCIL binding point.

Multisampling State

The multisampling state object is represented by XGL_MSAA_STATE_OBJECT handle. It describes the multisampling anti-aliasing (MSAA) options for the graphics rendering. The multisampling state is created by calling xglCreateMsaaState(). The multisampling state is bound to XGL_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.

Using multisampling state objects that have 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

In Explicit GL 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 Explicit GL for 2-sample, 4-samples and 8-samples.

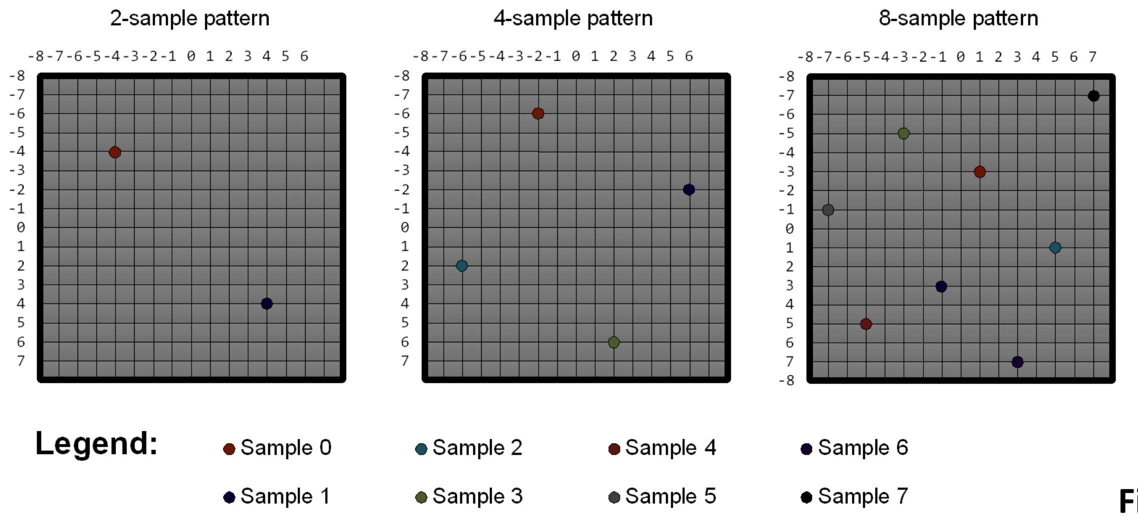


Figure 12. Default sample patterns

SHADERS

Shader objects are used to represent code executing on programmable pipeline stages. The input shaders in Explicit GL are specified in binary *intermediate language* (IL) format. The currently supported intermediate language is a subset of *XGL IL*. The shaders can be developed in IL assembly or high-level languages and compiled off-line to a binary IL. The Explicit GL 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 Explicit GL.

Shader objects, represented by XGL_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 Explicit GL driver only once on 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 xglCreateShader().

PIPELINES

The Explicit GL 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 Explicit GL, regardless of their type, are represented by XGL_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, Explicit GL'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-shader* 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 xglCreateComputePipeline() with a compute shader object handle in the pipeline creation parameters. It is invalid to specify XGL_NULL_HANDLE for the compute shader.

GRAPHICS PIPELINES

The graphics pipeline is created by calling xglCreateGraphicsPipeline() 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.

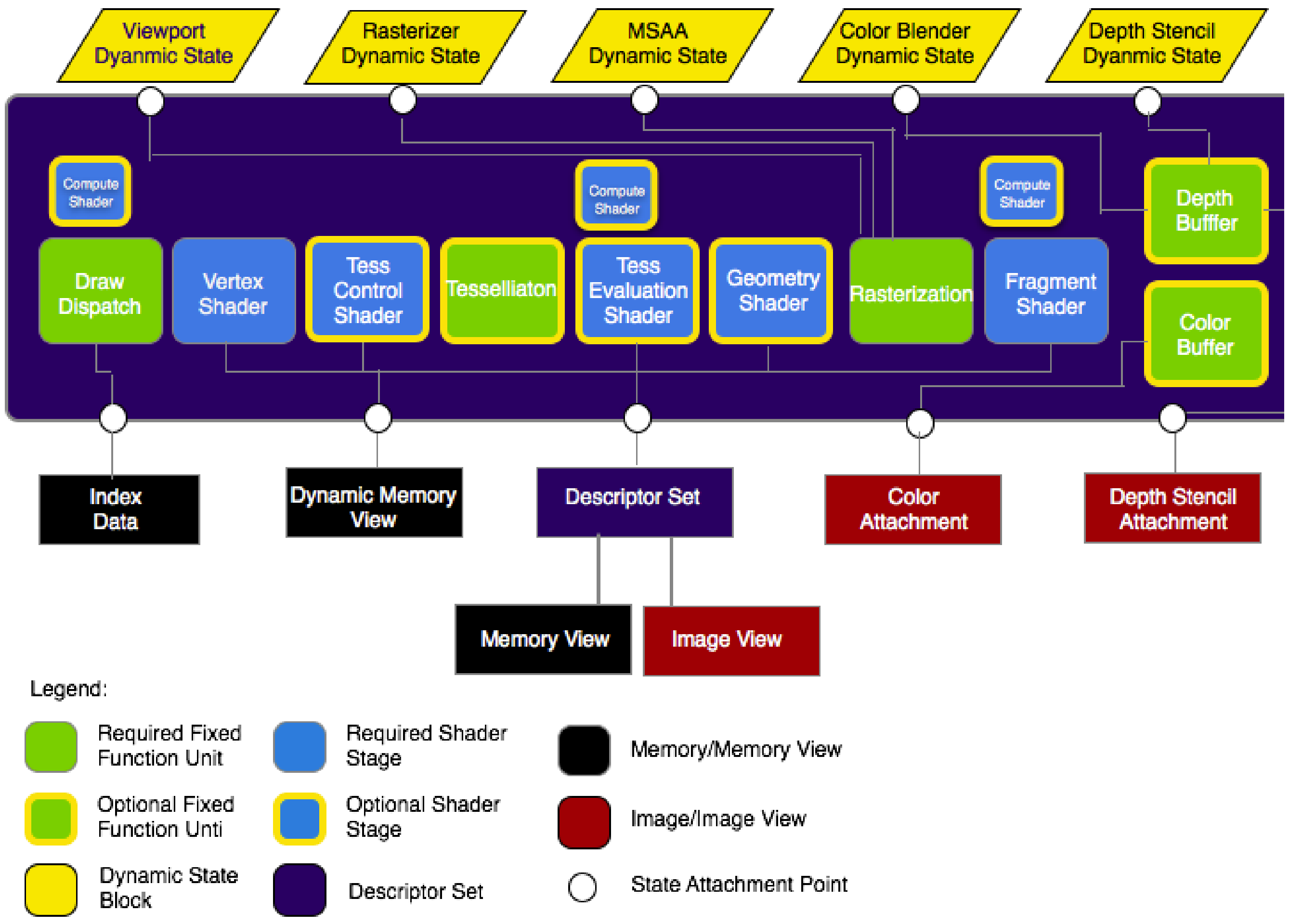


Figure 13. Graphics pipeline and its state

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	Tessellation control shader
TESS	Fixed function	Tessellator
DS	Shader	Tessellation evaluation shader
GS	Shader	Geometry shader
RS	Fixed function	Rasterizer
PS	Shader	Fragment 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 fragment shader is always required for color output and blending, but is optional for depth-only rendering;
- ▼ both tessellation control and tessellation evaluation shaders must be present at the same time to enable tessellation.

Presence of the shader stage in a pipeline is indicated by specifying a valid shader object. The application uses XGL_NULL_HANDLE value to indicate the shader stage is not needed. 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 Tessellation Control and Tessellation Evaluation 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 Fragment shader (for example, using Fragment 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 Explicit GL 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 Fragment shader and optionally forwarded to the color and depth fixed function units for processing and output.

VERTEX FETCH IN GRAPHICS PIPELINE

Unlike other APIs, Explicit GL 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 instance offset, in 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 XGL_PIPELINE_IA_STATE structure. The state includes information about primitive topology and vertex reuse.

The XGL_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 vertex, or geometry shader, or any of the tessellation shaders write data out to memory or images.

enablePrimitiveRestart and primitiveRestartIndex allow a particular index value to be specified that will trigger a new primitive to be started with the following index. This is useful to drawing multiple triangle strips from a single draw command. Enabling this feature may have a negative performance impact on some implications, and should be left disabled unless the feature is required.

provokingVertex chooses whether the first or last vertex in a primitive supplies the values for flat shaded attributes, among other things. The default, should the value be left set to 0, is XGL_PROVOKING_VERTEX_LAST.

Tessellator Static State

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

The tessellator static state is only used when valid Tessellation Control and Tessellation Evaluation 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 ability to specify the optimization hint that indicates to the Explicit GL driver what target tessellation factor to optimize the pipeline for. 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 application is unsure of 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 XGL_PIPELINE_RS_STATE structure. The state includes information about the depth clip mode and rasterization discard. Rasterization discard is a feature that allows rasterization to be disabled even when a fragment shader is otherwise bound and enabled. It is useful when the front-end of the pipeline has visible side effects such as writing to images or modifying atomic counters, but fragment shader execution is not required. It is also used in some performance analysis tools to override application state and disable rasterization during bottleneck analysis.

Depth-stencil Static State

The depth-stencil static state for the graphics pipeline is specified using XGL_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 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 XGL_CH_FMT_UNDEFINED channel format and XGL_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 XGL_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 XGL_CH_FMT_UNDEFINED as the channel format and XGL_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.

XGL_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 Attachments. The logic operation may only be non-default on targets of XGL_NUM_FMT_UINT and XGL_NUM_FMT_SINT numeric formats, other formats fail pipeline creation.

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, Explicit GL 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 xglStorePipeline(). Before calling xglStorePipeline(), 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 xglStorePipeline() function with NULL value in pData. xglStorePipeline() fails if insufficient data buffer space is specified.

A pipeline object is loaded from memory with xglLoadPipeline(). 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 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 Explicit GL reduces the need for off-line creation of 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. Explicit GL uses Uniform Buffer facilities available in shader IL to designate shader data that would be specified at pipeline linkage. The IL Uniform Buffers in Explicit GL shaders can only be used for this purpose; an application must use conventional memory views for passing run-time data to shaders. Multiple link Uniform Buffers per shader can be used, and each shader in a pipeline could have its own set of link time Uniform 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 XGL_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 xglCmdBindPipeline() function. The pipeline bind point is specified in 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 XGL_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 Deltas

Graphics pipeline objects represent the entire graphics pipeline as a large, monolithic object. Binding a new graphics pipeline may require a large amount of state commands to be sent to the GPU by the driver, consuming driver CPU overhead and front-end overhead in the GPU. Explicit GL offers a way to reduce these overheads with pipeline delta objects.

A pipeline delta object is created by specifying two existing XGL_PIPELINE objects, p1 and p2. The driver will examine these two objects and compute the least amount of state possible to transition from p1 to p2. While building a command buffer, if p1 is the current pipeline state, the application can choose apply this pipeline delta in lieu of binding p2.

For implementations where this up front optimization is not helpful, applying the delta may simply bind p2. If the application applies a delta whose p1 state does not match the currently bound pipeline, results are undefined.

Pipeline Resource Access

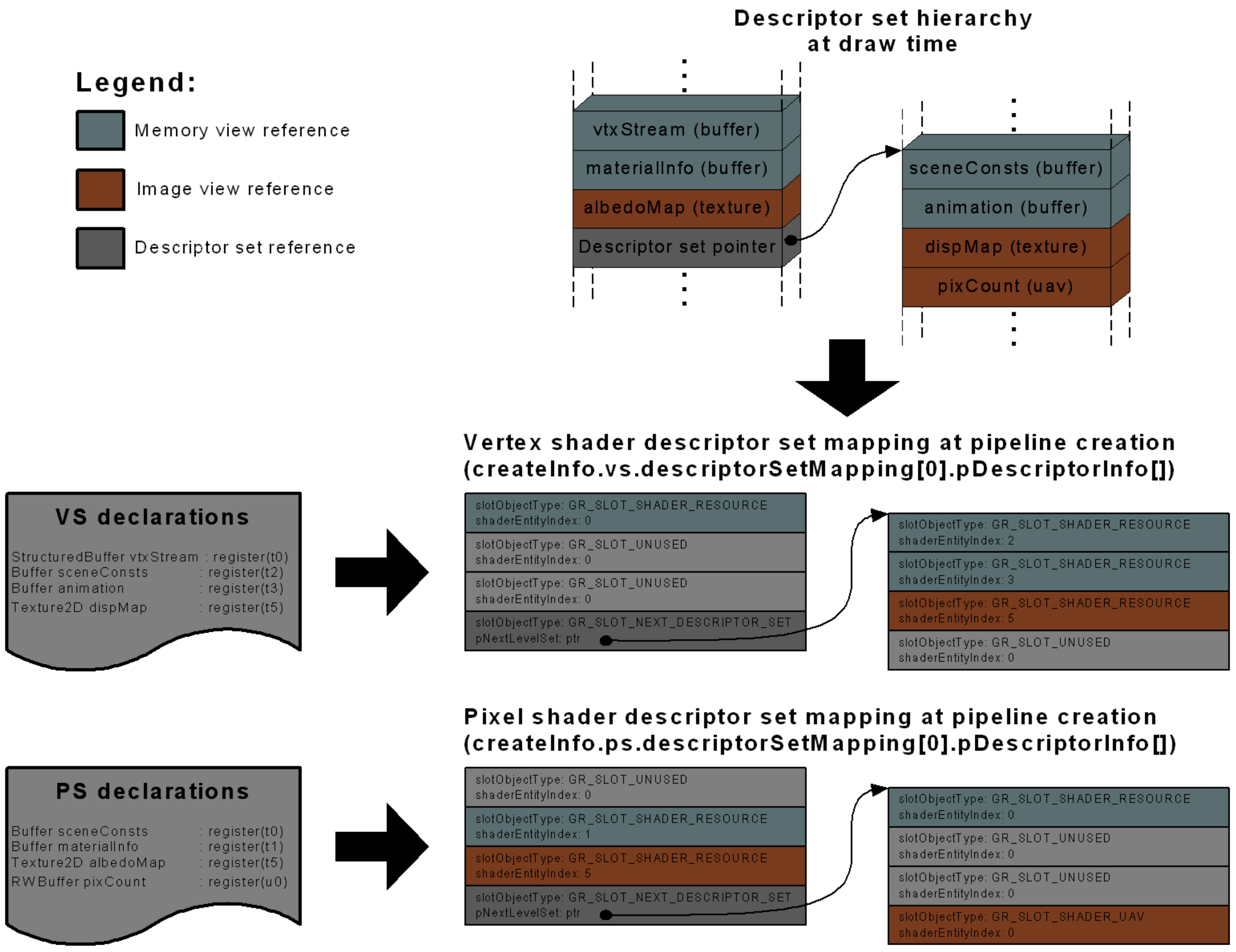
Pipeline shaders access shader resources specified in descriptor sets bound to the command buffer state at the time of executing a draw or dispatch command. Additionally, a dynamic memory view can be used for buffer-like access to memory. The expected descriptor set layout and its mapping to shader resources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

Pipeline Resource Mapping

On the one hand, hierarchical descriptor set structures bound to the pipeline are shared by all shaders forming the pipeline. On the other hand, the shaders themselves use flat resource addressing scheme with different resource namespaces for distinct resource usages (read-only textures, UAVs, Uniform Buffers and etc.), 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 used by a given IL shader for all active shader stages. Even if it is known that resource access is optimized out by the Explicit GL 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 Fragment 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 it maps to. If a descriptor set element does not map to any IL shader resource, it must have the XGL_SLOT_UNUSED object type specified. An indirection to the next level of descriptor set hierarchy is specified by using the XGL_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.



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
StructuredBuffer vtxStream : register(t0)
Buffer sceneConsts : register(t2)
Buffer animation : register(t3)
Texture2D dispMap : register(t5)
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
Buffer sceneConsts : register(t0)
Buffer materialInfo : register(t1)
Texture2D albedoMap : register(t5)
RWBuffer pixCount : register(u0)
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

Figure 14. Descriptor set mapping to pipeline shaders

Figure 14 shows an example of the descriptor set remapping structures for a simple pipeline consisting of vertex and Fragment 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.

DESCRIPTOR SET BINDING

Descriptor sets are bound to command buffer state using xglCmdBindDescriptorSet(). There are separate descriptor sets for each pipeline type – graphics and compute. The pipeline bind point specified in the xglCmdBindDescriptorSet() 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 XGL_PIPELINE_BIND_POINT_COMPUTE.

Specifying XGL_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 XGL_MEMORY_VIEW_ATTACH_INFO structure passed to xglCmdBindDynamicMemoryView() 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 XGL_SLOT_SHADER_RESOURCE or XGL_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 XGL_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 withing 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

Explicit GL 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 Explicit GL capable GPU in a system is presented as an independent device that is managed by an application.

The following features are exposed by the Explicit GL 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

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 Explicit GL GPUs. First, if multiple Explicit GL 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. Some platforms may provide additional multi-device functionality depending on a system's GPU topology, so the application must also query and consider the availability of these features as well.

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 the GPU
- Performance
- Multi-device capabilities

The discovery of display connectivity must be provided via an OS-specified Window System Interface (WSI) extension. In addition to display connectivity, a WSI extension must report what displays can be used for *cross-device presentation*.

The general GPU capabilities and performance are reported by the Explicit GL core API using xglGetGpuInfo() 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 a best possible way.

There are two aspects to device compatibility. The first aspect is matching GPU features and image quality. Second is the ability to use advanced multi-device functionality:the ability to share memory and synchronization objects and to composite displayable output. Not all GPUs or GPU combinations could expose these extra features. Multi-device compatibility can be queried with xglGetMultiGpuCompatibility(). The compatibility information is returned in the XGL_GPU_COMPATIBILITY_INFO structure containing various compatibility flags.

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 alternate frame rendering mode it is important to produce images of similar quality on the alternating frames.

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

SHARING MEMORY BETWEEN GPUS

Memory objects residing in some non-local memory heaps can be made shareable across devices if the GPUs have the XGL_GPU_COMPAT_SHARED_MEMORY flag 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 Explicit GL 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 XGL_MEMORY_HEAP_FLAG_SHAREABLE flag reported in heap properties. See GPU Memory Heaps for information on heap properties. If no heaps expose XGL_MEMORY_HEAP_FLAG_SHAREABLE, 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 xglAllocMemory() function. A shared memory object is marked by the XGL_MEMORY_ALLOC_SHAREABLE flag in its creation information and can only be created in heaps marked by the XGL_MEMORY_HEAP_FLAG_SHAREABLE heap property flag.

A shared memory object created on one device can be opened on another compatible device using xglOpenSharedMemory(). 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 objects. 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 xglFreeMemory(). 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 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 XGL_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 XGL_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 XGL_GPU_COMPAT_SHARED_SYNC flag 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 Explicit GL devices:

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

SHARED SEMAPHORE CREATION

Shared queue semaphores are created just like any other regular semaphores using the xglCreateQueueSemaphore() function. The shared queue semaphore object is marked by

XGL_SEMAPHORE_CREATE_SHAREABLE in creation info. 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 xglOpenSharedQueueSemaphore().

The shared semaphore cannot be opened on the device on which it was created. Just like with any other Explicit GL 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 xglDestroyObject(). 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

It is not possible to transfer data between the memory and image objects from different GPUs by directly referencing their handles, since only objects local to device can be used for the copy operations. For optimal copying of image and other data between GPUs, an application uses *peer-to-peer transfers*. These allow direct device-to-device communication over a system bus without intermediate storage of data in system memory. Explicit GL supports peer-to-peer transfers between GPUs if the XGL_GPU_COMPAT_FLAG_PEER_TRANSFER flag is reported in the GPU compatibility information.

There are several parts to enabling peer-to-peer transfers across multiple Explicit GL 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 xglOpenPeerMemory(). 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 xglFreeMemory(). 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 xglOpenPeerImage().

The xglOpenPeerImage() returns a peer image and a peer memory object associated with the peer image at the time of opening it. These are associated with the original image and memory bound to it at the time of opening peer image. A valid memory object has to be bound to an original image before opening 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 the XGL_GPU_COMPAT_ASIC_FEATURES compatibility flag set, the peer transfer destination image can use XGL_OPTIMAL_TILING tiling, otherwise only XGL_LINEAR_TILING must be used for the destination image.

Peer memory objects returned by xglOpenPeerImage() 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 xglDestroyObject(). 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 and so on.

An application should be careful with selection of the GPU used for execution of peer-to-peer transfer of data. The peer opened memory and image objects should only be used as a destination for writing data. Reading of peer memory or image may result in very slow read transactions across a system bus and should be avoided for performance reasons.

Before a peer transfer can take place, the source and destination memory or images have to be transferred to XGL_MEMORY_STATE_DATA_TRANSFER and XGL_IMAGE_STATE_DATA_TRANSFER states. 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 Explicit GL configurations might include the display compositing capabilities for automatically transferring and displaying images between the GPUs using dedicated hardware. The hardware compositor in Explicit GL 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 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 Explicit GL display.

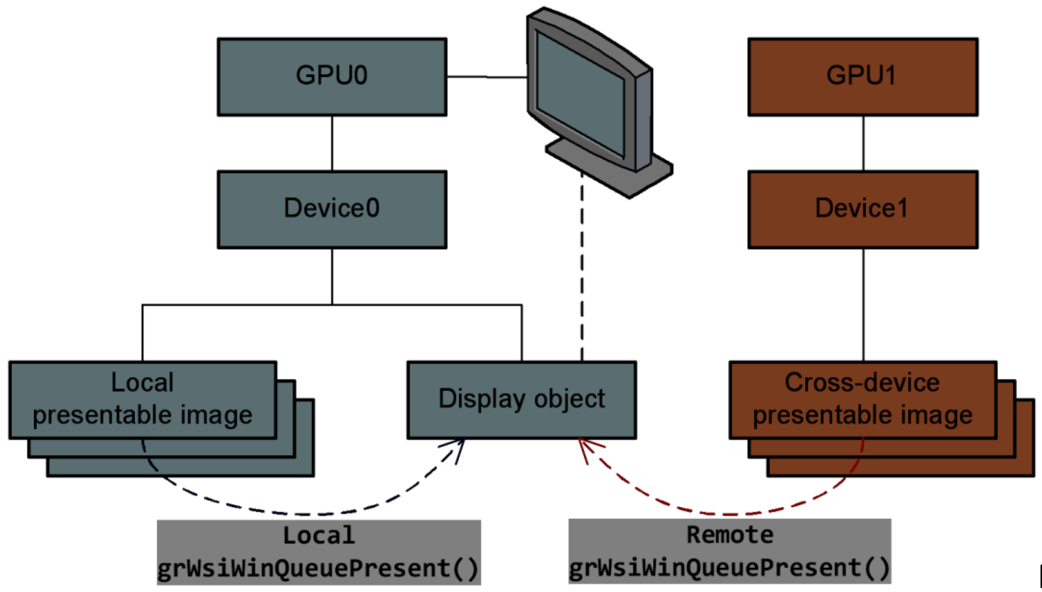

GPU0
Device0
GPU1
Device1
Local presentable image
Display object
Cross-device presentable image
Local grWsiWinQueuePresent()
Remote grWsiWinQueuePresent()

Figure 16.

Conceptual view of cross-device presentation

Discovering Cross-device Display Capabilities

An application detects if a GPU can present to a display from another GPU by examining the XGL_GPU_COMPAT_FLAG_SHARED_GPU0_DISPLAY and XGL_GPU_COMPAT_FLAG_SHARED_GPU1_DISPLAY compatibility flags. If neither flag is present, software compositing should be used. If cross-device presentation is supported, an application should further check if it is available for a particular display mode through the appropriate WSI extension.

Without cross-device presentation support, 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.

Cross-device Presentable Images

Cross-device presentable images created through a WSI extension 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 the necessary devices is not available for the requested resolution. In case a presentable image cannot be created, an application must fall back to a software compositing.

If multiple Explicit GL 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.

Chapter VII.

DEBUGGING AND VALIDATION LAYER

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

The Explicit GL debug infrastructure is layered on top of the core Explicit GL 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 Explicit GL 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 XGL_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 XGL_DEVICE_CREATE_VALIDATION flag the maximum debug level has to be set to XGL_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, Explicit GL 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 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 xglDbgSetValidationLevel().

Setting the validation level is not a thread safe operation. Additionally, when changing validation level an application should ensure it is not in the middle of building any command buffers. Switching 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 XGL_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 Explicit GL 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 xglDbgRegisterMsgCallback(). The callback function is an application's function defined of XGL_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 debug message callback before Explicit GL is initialized.

When it no longer needs to receive debug messages, an application unregisters the callback with xglDbgUnregisterMsgCallback(). 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 xglDbgRegisterMsgCallback() or xglDbgUnregisterMsgCallback() 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 xglDbgSetMessageFilter(). Previously disabled messages are re-enabled at any time by specifying XGL_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 Explicit GL object other than the XGL_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 xglDbgSetObjectTag(). 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 is copied by the Explicit GL driver when xglDbgSetObjectTag() is called.

To retrieve a previously set object tag an application calls xglGetObjectInfo() with the XGL_DBG_DATA_OBJECT_TAG debug data type.

INTERNAL DEBUG AND VALIDATION INFORMATION

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 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 xglCmdDbgMarkerBegin() and xglCmdDbgMarkerEnd() functions.

DEBUG INFRASTRUCTURE SETTINGS

The debug infrastructure has various settings that can be used during debugging to force specific Explicit GL 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 xglDbgSetGlobalOption() and per-device settings are configured with xglDbgSetDeviceOption(). 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. Feature Additions to Match OpenGL Needs

In order to be able to implement the current core profile of OpenGL on top of Explicit GL, and to not propose feature loss moving from OpenGL to Explicit GL, several feature additions to the proposed spec should be addressed.

Features Supported by Current OpenGL

The following features are present in existing core profile OpenGL and should be considered for support in Explicit GL to allow an OpenGL driver to be implemented on top of it.

Point Sprites

Point sprites allow hardware generation of a *UV* coordinate that varies across a point primitive. The Explicit GL proposal does not expose point sprites, but they are a mandatory feature of core profile OpenGL and OpenGL ES, and will be required.

Fortunately, point sprite coordinate generation does not require any API changes as it is possible to enable it solely in the fragment shader. In OpenGL, the built-in variable “gl_PointCoord” contains the generated hardware point sprite UV. Generation of this coordinate is implicitly enabled by its access in a shader. Though the Explicit GL proposal does not have a high level shading language, it should be possible to reference the point coordinate system value in a shader to enable the feature. We simply need to document an affirmation that this works in Explicit GL.

Transform Feedback

Transform feedback is a pipeline stage that appears logically after all vertex and geometry processing but before primitive assembly and rasterization. This feature is known as "stream-out" in DirectX and is sometimes shortened to XFB in OpenGL literature. This feature is not supported in the Explicit GL proposal, with a suggestion that UAV writes from early pipeline stages covers most of the offered functionality. This may not be sufficient for Explicit GL as the stream compaction offered by geometry shader decimation is not possible.

A detailed proposal for XFB support in Explicit GL is not available at this time. Most likely, it would involve creating XFB memory views including vertex stride, size, etc. and attaching them to descriptor sets. Analysis would need to be done to determine which states are needed in the graphics pipeline state to enable proper pipeline creation; it may be that all necessary state can be inferred by the shaders themselves. Additional work will need to be done to support XFB PRIMITIVES_WRITTEN and a new draw interface would be needed to mimic glDrawTransformFeedback.

UNSIGNED BYTE INDICES

All versions of OpenGL and OpenGL ES support 8-bit unsigned index data whereas the Explicit GL proposal currently does not. Some hardware may not have native support for this index format. Emulating that support in the driver is possible in OpenGL, but not with the proposed Explicit GL model. This support should be added to the API as an optional feature.

FEATURE MODIFICATIONS

The following suggestions are on the removal, modification or replacement of existing Mantle features.

FLEXIBLE QUEUE TYPES

The current Explicit GL core proposal only includes support for two types of queues, universal and compute, and at least one universal queue must be exposed. This is not flexible enough to match the restrinctions and capabilities of a large range of hardware. For example, an implementation may need to expose a graphics-only queue, or a compute-only device might need to expose only a compute queue.

Instead:

- The XGL device should expose the ability to query how many queues are supported.
- The XGL queue should expose the ability to query extensible caps flags reporting support for at least:
    - Graphics
    - Compute
    - DMA
- An implementation is only required to export support for at least 1 queue.