# LIQUIDVR SDK TECHNICAL USER GUIDE

# Disclaimer

The information contained herein is for informational purposes only, and is subject to change without notice. While every precaution has been taken in the preparation of this document, it may contain technical inaccuracies, omissions and typographical errors, and AMD is under no obligation to update or otherwise correct this information.

Advanced Micro Devices, Inc. makes no representations or warranties with respect to the accuracy or completeness of the contents of this document, and assumes no liability of any kind, including the implied warranties of noninfringement, merchantability or fitness for particular purposes, with respect to the operation or use of AMD hardware, software or other products described herein. No license, including implied or arising by estoppel, to any intellectual property rights is granted by this document. Terms and limitations applicable to the purchase or use of AMD's products are as set forth in a signed agreement between the parties or in AMD's Standard Terms and Conditions of Sale.

AMD, the AMD Arrow logo, ATI Radeon™, CrossFireX™, LiquidVR™ and combinations thereof are trademarks of Advanced Micro Devices, Inc. Other product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

Windows™, Visual Studio and DirectX are trademark of Microsoft Corp.

## Copyright Notice

# Contents

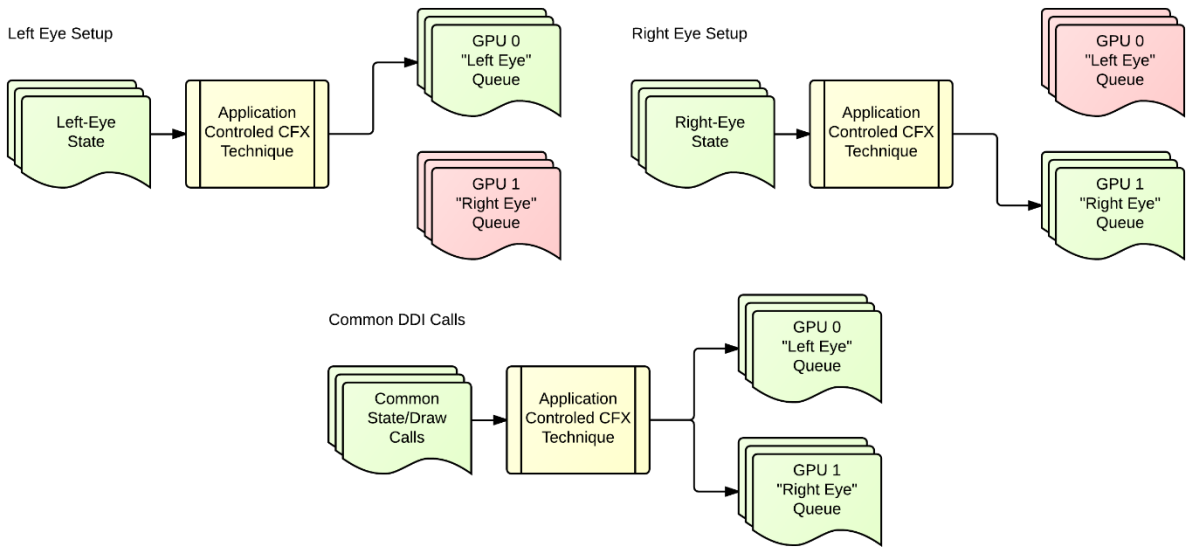
# 1 Introduction

This guide is intended for advanced users who have a good knowledge of the DirectX® API and are familiar with general graphics and GPU compute techniques.

## 1.1 Multi-GPU Affinity

The Multi-GPU Affinity feature is intended for virtual reality (VR) developers interested in maximizing performance of their applications in system configurations with multiple AMD graphics cards. The conventional alternate-frame rendering technique supported by CrossFire™ renders successive frames on different cards. Multi-GPU Affinity provides an interface to render the same frame or its portions on different GPUs by directing DirectX11® context rendering commands to a set of GPUs in a multi-GPU configuration, according to the selection mask. Additionally, it offers a feature to explicitly transfer resources between two GPUs. The application is responsible for all data marshalling and synchronization of data between the GPUs.

In case of a dual-GPU configuration, most of the rendering in a VR environment is the same for left and right eye, and a single DirectX® command stream can be broadcast to two GPUs, each generating an image for one eye. Specification of view dependent parameters is different between left and right eyes, and a mask selecting either of the GPUs is used in that case. Prior to presentation, the application uses the MGPU Affinity API to transfer a rendered image from the slave GPU to the master. A single image combining left and right eye rendering is used for presentation to HMD on the master GPU. The diagram below illustrates such operation.




A particular sequence of steps has to be taken to properly enable Multi-GPU Affinity. Once the LiquidVR factory interface is initialized, the first step is to use the *ALVRFactory::CreateGPUAffinity()* function to initialize an instance of *ALVRGPUAffinity* interface . This interface can be used to enable or disable MGPU Affinity mode on a per-device basis. The affinity mode for a device is set at creation and persists for its lifetime (cannot be changed later). To create device in MGPU Affinity mode, the affinity mode has to be enabled with *ALVRGPUAffinity::EnableGPUAffinity()* function prior to device creation. If Multi-GPU Affinity

is not available, the function fails with an appropriate error code. If the function succeeds, the Multi-GPU Affinity features are initialized and are ready for use.

Once a device in Multi-GPU Affinity mode is created, and another device without affinity is needed, the Multi-GPU Affinity can be disabled with *DisableGPUAffinity()* prior to creation of the second device. The Multi-GPU Affinity feature operates by wrapping the DirectX® device and context runtime objects in an effort to ensure communication from the application properly makes its way to the driver. These DirectX® object wrappers are created by calling *D3D11WrapDevice()*. Once wrapped, the original device and context must not be used. The DirectX® runtime filters duplicate states, which might affect consistency of state in Multi-GPU Affinity mode. The main intent of wrapped device context is to ensure that state filtering is subverted and the render mask is consistent with the application settings across all active GPUs.

## 1.2 Late Data Latch

The delay between a head movement and display of the rendered objects in VR (a.k.a. motion-to-photon latency) could cause discomfort and headaches in some individuals. The Late Data Latch is intended to increase the application responsiveness and to reduce rendering lag. For example, in a typical gaming or a VR application frames are rendered based on current user's head position and orientation. Once a render task is submitted to the GPU, it is not possible to modify its parameters. Since CPU and GPU time domains are not synchronized, there is some lag between the moment when a frame is submitted to the GPU pipeline for rendering and the moment when the rendered frame is presented on the screen. This lag depends on the amount of time needed to batch and render a scene and in certain cases can be significant, especially when the frame rates are low. At the same time the user continues to control the application unaware that the image currently being visible on the screen has been rendered according to the transformation matrix recorded at the moment when the render job was submitted to the GPU. This lag makes the application appear less responsive to user's actions; therefore it is very desirable to reduce it as much as possible.

The Late Data Latch helps applications deal with this problem by continuously storing frequently updated data, such as, real-time head position and orientation matrices, in a fixed-sized constant buffer, organized as a ring buffer. Each new snapshot of data is stored in its own consecutive data slot. The data ring buffer has to be large enough to ensure the buffer will not be overrun and latched data instance will not be overwritten during the time it could be referenced by the GPU. For example, if data is updated every 2ms, a game rendering at 100fps should have more than 50 data slots in the data ring buffer. It is advised to have at least twice the minimum number of slots to avoid data corruption. Just before the data is to be consumed by the GPU, the index to the most up-to-date snapshot of data is latched. The shader could then index into the constant buffer containing the data to find the most recent matrices for rendering.

This late latching approach allows the application to get the latest transformation matrices up to the moment when the frame is about to be rendered.

It is important to note that late latching cannot increase the application's frame rate, but it can make the application appear more responsive to user's input.

## 1.3 Asynchronous Compute

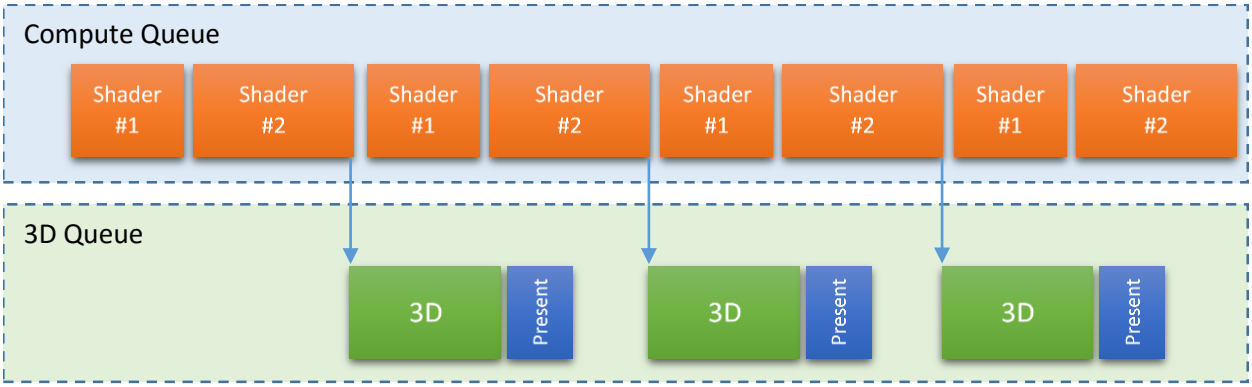
Asynchronous Compute is a performance-boosting feature, which allows to execute the compute jobs concurrently with graphics tasks. Since Microsoft DirectX 11 DirectCompute™ API does not support

submission of asynchronous compute jobs, LiquidVR™ SDK provides a private API for Asynchronous Compute.

The following diagram demonstrates the difference between the standard DirectCompute™ and AMD Asynchronous Compute models, where compute can be run concurrently with 3D graphics:

Standard DirectX 11 Direct Compute:


AMD Asynchronous Compute:




Please note that the Asynchronous Compute scenario depicted above is only one of the countless possibilities of scheduling Compute jobs relative to 3D jobs. Compute jobs may precede, follow, or be interleaved with 3D jobs for the arrangement providing the most optimal performance. Depending on the relative length of Compute and 3D jobs and their interdependencies performance gains from using Asynchronous Compute can be significant compared to a serialized execution provided by Direct Compute, allowing programmers to take full advantage of the capabilities of AMD Radeon hardware.

Asynchronous Compute in LiquidVR™ API leverages shaders compiled for DirectCompute™.

# 2 SDK Contents

## 2.1 LiquidVR Libraries in Windows

The LiquidVR™ runtime is distributed as a part of the AMD graphics driver package. To use LiquidVR™ in the Windows® programming environment, users must have appropriate AMD graphics drivers installed. There are 32-bit and 64-bit versions of runtime libraries available. An application should not redistribute LiquidVR™ runtime libraries outside of the AMD driver installation.

The following dynamic libraries are available:

**Table 1. Dynamically linked LiquidVR libraries**

| Library file name | Description |
|---|---|
| **amdlvr32.dll** | 32-bit LiquidVR API dynamic library |
| **amdlvr64.dll** | 64-bit LiquidVR API dynamic library |

The LiquidVR API header is located at *./inc/LiquidVR.h*. It contains entry points for API and extension libraries.

**Table 2. LiquidVR header files**

| Header file name | Description |
|---|---|
| **LiquidVR.h** | LiquidVR core API |

# 3 API Reference

## 3.1 Common Enumerations

The LiquidVR™ error messages are defined as enumerated type:

```
enum ALVR_RESULT
{
    ALVR_OK                          = 0,
    ALVR_FALSE                       = 1,
    ALVR_FAIL                        = 2,
    ALVR_INVALID_ARGUMENT            = 3,
    ALVR_NOT_INITIALIZED             = 4,
    ALVR_INSUFFICIENT_BUFFER_SIZE    = 5,
    ALVR_NOT_IMPLEMENTED             = 6,
    ALVR_NULL_POINTER                = 7,
    ALVR_ALREADY_INITIALIZED         = 8,
    ALVR_UNSUPPORTED_VERSION         = 9,
    ALVR_OUT_OF_MEMORY               = 10,
    ALVR_DISPLAY_REMOVED             = 11,
    ALVR_DISPLAY_USED                = 12,
    ALVR_DISPLAY_UNAVAILABLE         = 13,
    ALVR_DISPLAY_NOT_ENABLED         = 14,
    ALVR_OUTSTANDING_PRESENT_FRAME   = 15,
    ALVR_DEVICE_LOST                 = 16,
    ALVR_UNAVAILABLE                 = 17,
    ALVR_NOT_READY                   = 18,
    ALVR_TIMEOUT                     = 19,
    ALVR_RESOURCE_IS_NOT_BOUND       = 20
};
```

## 3.2 Interface Conventions

All LiquidVR SDK interfaces inherit from *IUnknown* and follow all standard COM rules for reference counting and object life cycle.

Pointers to interfaces can be returned either via parameters, or via function return values. The following conventions are followed in the LiquidVR SDK:

- When a pointer to an interface is returned via function parameters of the pointer-to-a-pointer-to-an-interface type, the reference count on the object implementing the interface is set to 1. The caller is responsible for calling the *Release()* method when the interface is no longer needed.
- When a pointer to an interface is returned as a function return value, its reference count is unaltered. The caller is responsible for calling *AddRef()* when saving the pointer and calling *Release()* when it is no longer needed.

## 3.3 LiquidVR Library Initialization

The LiquidVR is initialized by loading the LiquidVR dynamic library and creating an instance of the *ALVRFactory* class. In the Windows environment, use standard DLL loading method for loading LiquidVR library:

```
libHandle = LoadLibraryW(ALVR_DLL_NAME);
```

Next, retrieve a function pointer to the initialization function by name using *GetProcAddress()* function:

```
fnInit = GetProcAddress(libHandle, ALVR_INIT_FUNCTION_NAME);
```

Once the initialization function pointer is available, use it to retrieve the *ALVRFactory* interface. *ALVR_DLL_NAME* and *ALVR_INIT_FUNCTION_NAME* macros are defined in the LiquidVR header. The factory is then used to access LiquidVR features.

```
ALVR_RESULT (*ALVRInit_Fn)(uint64_t version, void **ppFactory);
```

### *Parameters*

**version**
Constant *ALVR_FULL_VERSION* defined in the LiquidVR header.

**ppFactory**
The address of a pointer to *ALVRFactory* to be initialized.

**Return**
*ALVR_OK* on success.

*ALVR_UNSUPPORTED_VERSION* when an unsupported version is requested.

*ALVR_INVALID_ARGUMENT* when *ppFactory* is *nullptr*.

## 3.4 ALVRFactory

This factory creates and initializes objects implementing all LiquidVR SDK core interfaces.

### CreateGPUAffinity

Initialize an instance of the *ALVRGPUAffinity* interface.

```
ALVR_RESULT CreateGPUAffinity(ALVRGPUAffinity **ppAffinity);
```

### *Parameters*

**ppAffinity**
A pointer to a location which will receive a pointer of the *ALVRGPUAffinity* interface.

**Return**
*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when *ppAffinity* is *nullptr*.

### CreateALVRDeviceExD3D11

Create an instance of the *ALVRDeviceExD3D11* interface.

```
ALVR_RESULT CreateALVRDeviceExD3D11(ID3D11Device* pD3DDevice,
            void* pConfigDesc,
            ALVRDeviceExD3D11** ppDevice);
```

### *Parameters*

**pD3DDevice**
A pointer to a D3D11 device for which the *ALVRDeviceExD3D11* interface is to be obtained.

### pConfigDesc

A pointer to a structure with optional configuration data. Reserved for future extensions, currently ignored.

### ppDevice

A pointer to a location which will receive a pointer of the *ALVRDeviceExD3D11* interface.

### Return

*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid.

*ALVR_NOT_INITIALIZED* or *ALVR_FAIL* when a generic driver failure occurs. Check the driver installation when receiving this error.

## CreateComputeContext

Create an instance of the *ALVRComputeContext* interface for a specific device.

```
ALVR_RESULT CreateComputeContext(ALVRDeviceEx* pDevice,
            unsigned int gpuIdx,
            CreateComputeContext* pDesc,
            ALVRComputeContext** ppContext);
```

### *Parameters*

### pDevice

A pointer to an *ALVRDeviceEx* interface representing a device for which the *ALVRComputeContext* interface is to be obtained

### gpuIdx

An index of the GPU in multi-GPU configurations, which will be used for executing compute tasks within this context.

### pDesc

A pointer to a structure with configuration data, defined as follows:

```
struct ALVRComputeContextDesc
{
    unsigned int flags; // ALVR_COMPUTE_FLAGS
};
```

where the *flags* field may contain one or more of the following values:

- *ALVR_COMPUTE_NONE* – no flags, use the regular Compute queue
- *ALVR_COMPUTE_HIGH_PRIORITY* – use the high priority Compute queue

### ppContext

A pointer to a location which will receive a pointer of the *ALVRComputeContext* interface.

### Return

*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid.

*ALVR_NOT_INITIALIZED* or *ALVR_FAIL* when a generic driver failure occurs. Check the driver installation when receiving this error.

## 3.5 ALVRPropertyStorage

Property storage implements a map of name-value pairs.

### SetProperty

Assign a value to a property.

```
ALVR_RESULT SetProperty(const wchar_t* name, ALVRVariantStruct value);

template<typename _T>
ALVR_RESULT SetProperty(const wchar_t* name, const _T& value);
```

#### *Parameters*

**name**
Property name.

**value**
Property value.

**Return**
*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when the name parameter is *nullptr*.

#### *Remarks*

Assigning a value to a property will add a new property to the map if one doesn't exist; otherwise change the value of an existing property.

### GetProperty

Retrieve a value to a property.

```
ALVR_RESULT GetProperty(const wchar_t* name, ALVRVariantStruct* value) const;

template<typename _T>
ALVR_RESULT GetProperty(const wchar_t* name, _T* value) const;
```

#### *Parameters*

**name**
Property name.

**value**
Property value.

**Return**
*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when the *name* parameter is *nullptr* or when the property has not been set previously.

## HasProperty

Check if a property exists in the storage.

```
bool HasProperty(const wchar_t* name) const;
```

### *Parameters*

**name**

Property name

**Return**

*true* when the property exists in the storage.

*false* when the property does not exist in the storage.

## GetPropertyCount

Get the number of properties in a property storage.

```
size_t GetPropertyCount() const;
```

### *Parameters*

None

**Return**

The number of properties in the storage.

## GetPropertyAt

Retrieve a value to a property at a specific location.

```
ALVR_RESULT GetPropertyAt(size_t index,
                          wchar_t* name,
                          size_t nameSize,
                          ALVRVariantStruct* value) const;
```

### *Parameters*

**index**

Property index. Index must be in the range greater than or equal to 0 and less than the value returned by *GetPropertyCount().*

**name**

A pointer to a buffer to store a property name. The size of the buffer is passed through the *nameSize* parameter.

**nameSize**

The size of the buffer to receive the property name in characters.

**value**
Property value.

**Return**
*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when the *name* or *value* parameter is *nullptr* or when the property index is out of range.

## 3.6 ALVRDeviceEx

*ALVRDeviceEx* inherits from *ALVRPropertyStorage*. This is a base class for other interfaces and currently does not define any methods.

## 3.7 ALVRDeviceExD3D11

*ALVRDeviceExD3D11* inherits from *ALVRDeviceEx*.

### CreateFence

Create a GPU fence. GPU fence is a special synchronization object which allows the CPU to wait for an event triggered by the GPU.

```
ALVR_RESULT CreateFence(ALVRFence** ppFence);
```

#### *Parameters*

**ppFence**
A pointer to a location which will receive a pointer to the *ALVRFence* interface.

**Return**
*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when *ppFence* is *nullptr*

*ALVR_FAIL* on any other error

### SubmitFence

Submit a GPU fence in the immediate context of D3D11 device.

```
ALVR_RESULT SubmitFence(unsigned int gpuIdx, ALVRFence* pFence);
```

#### *Parameters*

**gpuIdx**
An index of a GPU to submit a fence to.

**pFence**
A pointer to the *ALVRFence* interface to be submitted to the GPU.

**Return**
*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when *pFence* is *nullptr* or *gpuIdx* is out of range

*ALVR_FAIL* on any other error

## CreateGpuSemaphore

Create a GPU semaphore. GPU semaphores are synchronization objects that allow one GPU to wait for events triggered by another GPU in multi-GPU configurations, or to synchronize between the queues in the same GPU.

```
ALVR_RESULT CreateGpuSemaphore(ALVRGpuSemaphore** ppSemaphore);
```

### *Parameters*

**ppSemaphore**

A pointer to a location which will receive a pointer to the *ALVRGpuSemaphore* interface.

**Return**

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when *ppSemaphore* is *nullptr*

*ALVR_FAIL* on any other error

## QueueSemaphoreSignal

Queue a sempahore signal to a specific GPU queue in the immediate context of D3D11 device.

```
ALVR_RESULT QueueSemaphoreSignal(ALVR_GPU_ENGINE gpuEngine,
                                 unsigned int gpuIdx,
                                 ALVRGpuSemaphore* pSemaphore);
```

### *Parameters*

**gpuEngine**

The type of a GPU engine (*ALVR_GPU_ENGINE_3D* or *ALVR_GPU_ENGINE_DMA*).

**gpuIdx**

GPU index in multi-GPU configurations.

**pSemaphore**

A pointer to the semaphore interface to signal.

**Return**

*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

## QueueSemaphoreWait

Queue a semaphore wait to a specific GPU queue in the immediate context of D3D11 device.

```
ALVR_RESULT QueueSemaphoreWait(ALVR_GPU_ENGINE gpuEngine,
                               unsigned int gpuIdx,
                               ALVRGpuSemaphore* pSemaphore);
```

### *Parameters*

**gpuEngine**
The type of a GPU engine (*ALVR_GPU_ENGINE_3D* or *ALVR_GPU_ENGINE_DMA*).

**gpuIdx**
GPU index.

**pSemaphore**
A pointer to the semaphore interface to wait on.

**Return**
*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

## CreateGpuTimeline

Create an instance of the *ALVRGpuTimeline* interface for a specific GPU.

```
ALVR_RESULT CreateGpuTimeline(unsigned int gpuIdx,
                              ALVRGpuTimeline** ppGpuTimeline);
```

### *Parameters*

**gpuIdx**
GPU index

**pSemaphore**
A pointer to the location to receive a pointer to the *ALVRGpuTimeline* interface.

**Return**
*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

### CreateLateLatchConstantBufferD3D11

Create an interface encapsulating a constant buffer for Late Latch. This buffer may contain various parameters such as world, view or projection matrices used by shaders.

```
ALVR_RESULT CreateLateLatchConstantBufferD3D11(
            size_t updateSize,
            unsigned int numberOfUpdates,
            unsigned int bufferFlags,
            ALVRLateLatchConstantBufferDX11** ppBuffer);
```

#### *Parameters*

**updateSize**
The size of the data structure used per update,

**numberOfUpdates**
The number of update slots in the data buffer to be allocated; it should be greater or equal to the value statically defined in the shader.

**bufferFlags**
Can be either *ALVR_LATE_LATCH_NONE* or one or more of the following values:

*ALVR_LATE_LATCH_SHARED_BUFFER* – must be set when the buffer will be accessible via multiple APIs, for example, Direct3D and Asynchronous Compute.

**ppBuffer**
A pointer to the location to receive a pointer to *ALVRLateLatchConstantBufferDX11* interface.

**Return**
*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when any of the arguments are invalid.

## 3.8 ALVRFence

### Wait

Sleep CPU on a fence until a GPU fence is reached.

```
ALVR_RESULT Wait(unsigned int timeout);
```

#### *Parameters*

**timeout**
Wait timeout in milliseconds.

**Return**
*ALVR_OK* on success.

*ALVR_TIMEOUT* when the fence has not been signaled before the *timeout* elapses.

*ALVR_FAIL* on failure.

## 3.9 ALVRGpuAffinity

*ALVRGPUAffinity* inherits from the *ALVRPropertyStorage* interface.

### EnableGpuAffinity

Enable GPU Affinity mode for D3D11 device creation.

```
ALVR_RESULT EnableGpuAffinity(ALVR_GPU_AFFINITY_FLAGS flags);
```

#### *Parameters*

**flags**

Select options when enabling GPU Affinity by setting an enumeration defined as:

```
enum  ALVR_GPU_AFFINITY_FLAGS
{
    ALVR_GPU_AFFINITY_FLAGS_NONE = 0,
};
```

**Return**

*ALVR_OK* on success.

*ALVR_FAIL* on failure.

### DisableGpuAffinity

Disable  GPU affinity.

```
ALVR_RESULT DisableGpuAffinity();
```

#### *Parameters*

None

**Return**

*ALVR_OK* on success.

*ALVR_FAIL* on failure.

### WrapDeviceD3D11

Intercept standard DX11 interfaces for *D3D11Device* and *D3D11DeviceContext* for use with LiquidVR and obtain a pointer to the *ALVRMultiGpuDeviceContext* interface.

```
ALVR_RESULT WrapDeviceD3D11(ID3D11Device* pDevice,
                    ID3D11Device** ppWrappedDevice,
                    ID3D11DeviceContext** ppWrappedContext,
                    ALVRMultiGpuDeviceContext** ppMultiGpuDeviceContext);
```

#### *Parameters*

**pDevice**

A pointer to an original *ID3D11Device* to be wrapped by this function.

**ppWrappedDevice**

The address of a pointer to a wrapped *ID3D11Device*.

**ppWrappedContext**

The address of a pointer to a wrapped *ID3D11DeviceContext*.

**ppMultiGpuDeviceContext**

The address of a pointer to a wrapped *ALVRMultiGpuDeviceContext*.

**Return**

*ALVR_OK* on success.

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid.

## 3.10 ALVRMultiGpuDeviceContext

Defines an AMD LiquidVR custom multi-GPU device context that wraps an immediate *ID3D11Device* context.

### GetGpuControlInfo

Query Multi-GPU information for the wrapped *ID3D11Device*.

```
void GetGpuControlInfo(AmdAppControlInfo* pInfo);
```

#### *Parameters*

**pInfo**

A pointer to the *ALVRGpuControlInfo* structure to receive the number of active GPUs in the system, a mask of active GPUs and a mask of GPUs with attached displays available for presentation.

```
struct ALVRGpuControlInfo
{
    unsigned int     numGpus; ///< Number of GPUs available for control
    unsigned int     maskAllGpus; ///< GPU Mask representing all active GPUs
    unsigned int     maskDisplayGpu; ///< GPU Mask representing the display GPU
};
```

**Return**

Void.

### SetGpuRenderAffinity

Set the GPU affinity to send immediate context rendering commands to a one or more GPUs (e.g., for either left-eye only, right-eye only or both eyes).

```
void SetGpuRenderAffinity(UINT affinityMask);
```

#### *Parameters*

**affinityMask**

A mask for setting active GPUs.

**Return**

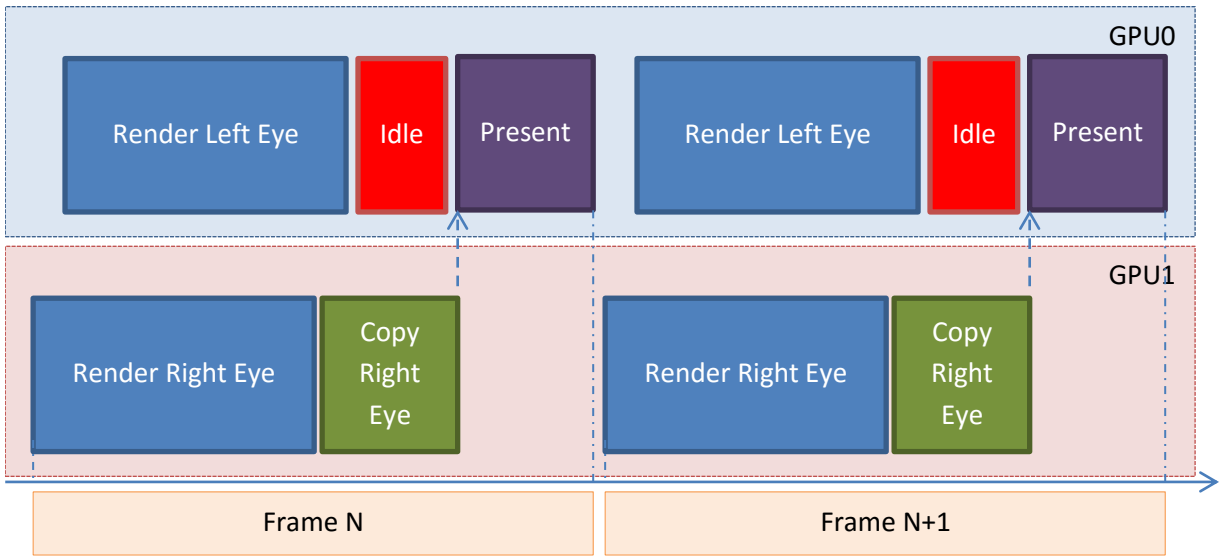
Void.

## *Remarks*

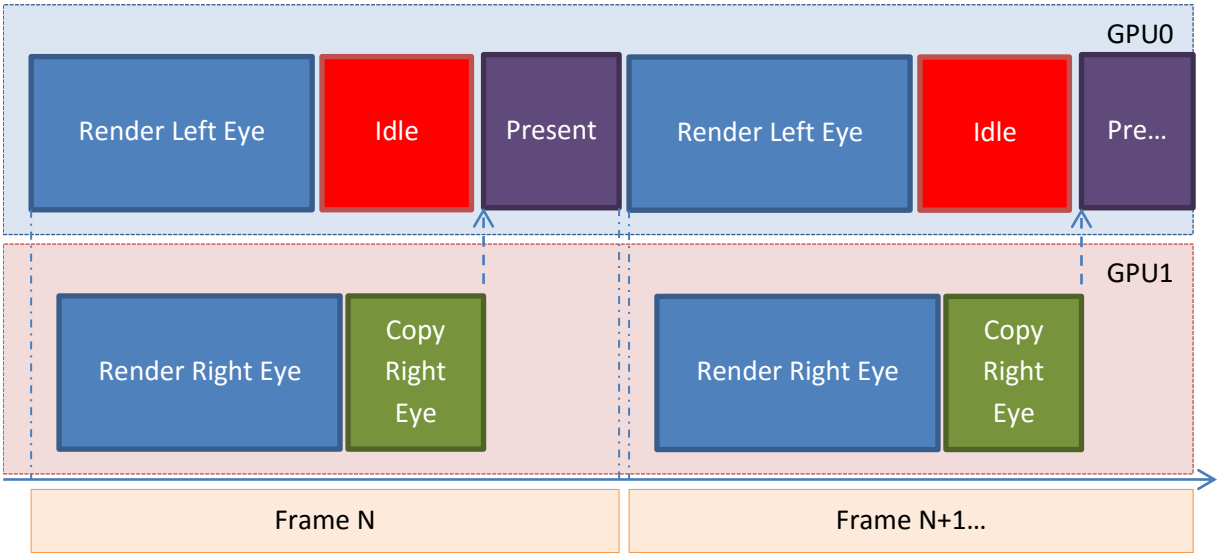
*SetGpuRenderAffinity* can be used to implement two different rendering approaches: single submission and multiple submissions. The choice between the two approaches is driven by various performance considerations depending on the goal of reducing CPU load vs. maximizing rendering speed on the GPU.

In a typical case of using multiple GPUs for rendering a stereoscopic scene, the image for one eye is rendered by one GPU, while the image for the other eye is rendered by another GPU. Both GPUs are running in the CrossFire™ mode with the display connected to the primary GPU. The image rendered on the secondary GPU eventually needs to be transferred to the primary GPU to be presented. Presentation cannot occur before both rendering tasks are completed and the image for the right eye is transferred from the secondary GPU to the primary GPU, resulting in some idle time on the primary GPU.

In the absolute majority of cases the amounts of time needed to render a frame for the left and the right eye are very similar. Since presentation on the primary GPU cannot start until the image rendered on the secondary GPU is transferred over to the primary GPU, and the image transfer is performed on the secondary (source) GPU, the secondary GPU has more work to do before a frame can be presented. Thus, it is recommended to submit rendering tasks to the secondary GPU first, so that the idle time on the primary GPU could be minimized. In the multiple submissions mode the application can control which GPU a job is submitted to first. The following diagrams illustrate the difference in performance between scenarios of submitting a job to the secondary GPU vs the primary GPU first:




When the rendering task one eye is submitted to the primary GPU first, before the rendering task for the other eye is submitted to the secondary GPU, the idle time between rendering and presentation would increase, resulting in a lower frame rate:




When GPU tasks are submitted to both GPUs at the same, the software running on the CPU has no control over which GPU would start executing the submitted rendering task first, which may result in lower overall frame rate. The ability to control the order of execution on both GPUs can be gained by explicitly setting the GPU render affinity before submitting render tasks to each GPU, as shown below:

| Single Submission Mode | Multiple Submissions Mode |
|---|---|
| `SetGpuRenderAffinity(0x02);`<br>`PassParametersToShader(RightEyeParams);`<br>`SetGpuRenderAffinity(0x01);`<br>`PassParametersToShader(LeftEyeParams);`<br>`SetGpuRenderAffinity(0xFFFFFFFF);`<br>`DrawFrame();` | `SetGpuRenderAffinity(0x02);`<br>`PassParametersToShader(RightEyeParams);`<br>`DrawFrame();`<br>`SetGpuRenderAffinity(0x01);`<br>`PassParametersToShader(LefttEyeParams);`<br>`DrawFrame();` |

However, in the single submission mode the number of calls *to DrawFrame()*, which sets up the GPU to perform a rendering operation, is less by one for every frame compared to the multiple submissions mode. The amount of CPU time needed to set up the GPU is not negligible, therefore the multiple submissions mode would result in a higher CPU usage.

The choice between the two drawing modes is determined by whether the graphics performance is limited by CPU or GPU. When CPU is the bottleneck, the single submission mode would be preferable, while the multiple submissions mode would result in better frame rates when performance is GPU-bound.

## MarkResourceAsInstanced

GPU resources can be either “mirrored” or “instanced”. The views of mirrored resources by different GPUs are synchronized, an update of a resource on one GPU alters the view of this resource by other GPUs.

Instanced resources, on the other hand, are viewed differently by different GPUs. An update to an instanced resource on one GPU will change this resource for this GPU only. To synchronize the views of the same resource by different GPUs, the most recent content of the resource must be copied to other GPUs.

By default resources are created in the mirrored mode. Individual resources can be marked as instanced by using the *MarkResourceAsInstanced()* method.

```
void MarkResourceAsInstanced(ID3D11Resource* pResource);
```

### *Parameters*

**pResource**
A pointer to a D3D11 resource.

**Return**
Void.

### *Remarks*

This is only needed on resources where the application needs to send different data to each GPU using the *ID3D11DeviceContext::Map()* and *ID3D11DeviceContext::Unmap()* interfaces. It should be noted that once a resource is marked as "instanced", calling *ID3D11DeviceContext::Map()* while the GPU Affinity Mask is set to more than one GPU will result in undefined behavior.

## TransferResource

Copy a resource from one GPU to another.

```
ALVR_RESULT TransferResource(ID3D11Resource*     pSrcResource,
                      ID3D11Resource*     pDstResource,
                      unsigned int        srcGpuIdx,
                      unsigned int        dstGpuIdx,
                      unsigned int        srcSubResourceIndex,
                      unsigned int        dstSubResourceIndex,
                      const D3D11_RECT*   pSrcRegion,
                      const D3D11_RECT*   pDstRegion);
```

### *Parameters*

**pSrcResource**
A pointer to a D3D11 resource stored on the source GPU.

**pDstResource**
A pointer to a D3D11 resource stored on the destination GPU.

**srcGpuIdx**
Source GPU index. The master GPU is the one with a display attached (can be queried), and others are slaves.

**dstGpuIdx**
Destination GPU index. The master GPU is the one with a display attached (can be queried), and other are slaves.

**srcSubResourceIndex**
Source subresource index.

**dstSubResourceIndex**
Destination subresource index.

### pSrcRegion

Define the region to read from the source resource.

### pDstRegion

Define the region to write to in the destination resource.

### Return

*ALVR_OK* on success

*ALVR_INVALID_PARAMETER* when any of the parameters are invalid

*ALVR_FAIL* on any other error

## TransferResourceEx

Copy a resource from one GPU to another using a specific GPU engine.

```
ALVR_RESULT TransferResourceEx(ID3D11Resource*     pSrcResource,
                        ID3D11Resource*     pDstResource,
                        unsigned int        srcGpuIdx,
                        unsigned int        dstGpuIdx,
                        unsigned int        srcSubResourceIndex,
                        unsigned int        dstSubResourceIndex,
                        const D3D11_RECT*   pSrcRegion,
                        const D3D11_RECT*   pDstRegion,
                        ALVR_GPU_ENGINE     transferEngine,
                        bool                performSync);
```

### *Parameters*

### pSrcResource

A pointer to a D3D11 resource stored on the source GPU.

### pDstResource

A pointer to a D3D11 resource stored on the destination GPU.

### srcGpuIdx

Source GPU index. The master GPU is the one with a display attached (can be queried), and others are slaves.

### dstGpuIdx

Destination GPU index. The master GPU is the one with a display attached (can be queried), and others are slaves.

### srcSubResourceIndex

Source subresource index.

### dstSubResourceIndex

Destination subresource index.

### pSrcRegion

Define the region to read from the source resource. Supported only for transfers by the D3D engine. Must be set to *nullptr* when using the DMA engine for the transfer.

**pDstRegion**

Define the region to write to in the destination resource. Supported only for transfers by the D3D engine. Must be set to *nullptr* when using the DMA engine for the transfer.

**transferEngine**

Specify which engine to use to perform the transfer (*ALVR_GPU_ENGINE_3D* or *ALVR_GPU_ENGINE_DMA*).

**performSync**

Perform internal synchronization of 3D and DMA jobs across GPUs.

**Return**

*ALVR_OK* on success

*ALVR_INVALID_PARAMETER* when any of the parameters are invalid

*ALVR_FAIL* on any other error

### *Remarks*

When the *performSync* parameter is set to *true*, internal engine synchronization is performed. The transfer is executed only when all previously queued operations on both the source and the destination GPUs have completed. This mode is recommended when the order of submission of GPU tasks is unknown. When the *performSync* parameter is set to *false*, no internal engine synchronization is performed.

When the order of GPU task submission is known, it is recommended to perform synchronization manually using GPU semaphores and set the *performSync* parameter to *false*.

Resource transfers with specified source or destination sub-regions (partial transfers) are only supported by the D3D engine. When using the DMA engine to perform the transfer between GPUs, the *pSrcRegion* and *pDstRegion* parameters must be set to *nullptr*. When an engine that does not support partial transfers is used and either of these parameters is not *nullptr,* *ALVR_INVALID_PARAMETER* error is returned.

## 3.11 ALVRLateLatchConstantBufferDX11

Defines interface for Late Data Latch using D3D11 API.

### Update

Update data in a new slot in the data constant buffer and move the current slot index to the next available slot. Typically the data for VR rendering contains world, view and projection matrices calculated based on the most recent user head positions and orientations.

```
ALVR_RESULT Update(const void* pData, size_t offset, size_t size);
```

### *Parameters*

**pData**

A pointer to a memory buffer which contains information such as view, projection and world matrices and other parameters. The format of the buffer is defined in the vertex shader.

**offset**

An offset in bytes in the current slot of the Late Latch constant buffer where the data will be copied to. Note that the offset applies to destination only.

**size**

The size of data in bytes to be copied to the Late Latch constant buffer.

**Return**

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* when the update fails

### *Remarks*

It is important to note that a single call to *Update* updates a single Late Data Latch buffer slot. Once *Update* returns, the current slot index is incremented and previous slots can no longer be updated. Specifying a non-zero offset leaves the memory at the beginning of the slot uninitialized, therefore caution should be exercised every time *Update* is called with a non-zero offset. A non-zero offset should only be used in cases when a portion of a Late Data Latch buffer slot needs to be filled by GPU (to provide GPU generated constants to the shader).

## QueueLatch

Queue latch of the latest data slot index in the D3D11 immediate context. During rendering the shader would use this index to retrieve the corresponding data from the data constant buffer.

```
ALVR_RESULT QueueLatch(void);
```

### *Parameters*

**Return**

*ALVR_OK* on success

*ALVR_FAIL* when the update fails

## GetIndexD3D11

Retrieve a pointer to a D3D11 buffer which contains a latched index (stored as a 32-bit unsigned integer). Using the index, the shader knows which slot of the data constant buffer contains the latest latched data.

```
ID3D11Buffer* GetIndexD3D11(void);
```

### *Parameters*

None

**Return**

A pointer to an *ID3D11Buffer* that contains a 32-bit index of the current data slot in the buffer returned by *GetDataD3D11()*.

### *Remarks*

Note that the reference counter of the buffer object returned is unaffected by this call. *AddRef()* should be called explicitly when this pointer is saved and a matching call to *Release()* should be performed when the saved pointer is no longer needed.

## GetDataD3D11

Retrieve a pointer to a D3D11 data constant buffer which contains an array of data slots updated with Update(). The latched slot is referenced by an index stored in the constant buffer returned by GetIndexD3D11().

```
ID3D11Buffer* GetDataD3D11(void);
```

### *Parameters*

None

#### Return

A pointer to an *ID3D11Buffer* containing an array of updated data slots.

### *Remarks*

Note that the reference counter of the buffer object returned is unaffected by this call. *AddRef()* should be called explicitly when this pointer is saved and a matching call to *Release()* should be performed when the saved pointer is no longer needed.

## GetIndex

Retrieve a pointer to a LiquidVR buffer object which contains a latched index (stored as a 32-bit unsigned integer). Using the index, the shader knows which slot of the data constant buffer contains the latest latched data.

```
ALVRBuffer* GetIndex(void);
```

### *Parameters*

None

#### Return

A pointer to an *ALVRBuffer* that contains a 32-bit index of the current data slot in the buffer returned by *GetData()*.

### *Remarks*

Note that the reference counter of the buffer object returned is unaffected by this call. *AddRef()* should be called explicitly when this pointer is saved and a matching call to *Release()* should be performed when the saved pointer is no longer needed.

### GetData

Retrieve a pointer to a LiquidVR buffer object which contains an array of data slots updated with Update(). The latched slot is referenced by an index stored in the constant buffer returned by GetIndex().

```
ALVRBuffer* GetData(void);
```

#### *Parameters*

None

#### Return

A pointer to an *ALVRBuffer* containing an array of updated data slots.

#### *Remarks*

Note that the reference counter of the buffer object returned is unaffected by this call. *AddRef()* should be called explicitly when this pointer is saved and a matching call to *Release()* should be performed when the saved pointer is no longer needed.

## 3.12 ALVRComputeContext

The *ALVRComputeContext* interface is responsible for providing access to the Asynchronous Compute functionality in LiquidVR. The *ALVRComputeContext* interface inherits from the *ALVRPropertyStorage* interface.

### CreateComputeTask

Create a new Asynchronous Compute task

```
ALVR_RESULT CreateComputeTask(ALVR_SHADER_MODEL shaderModel,
                              unsigned int flags,
                              const void* pCode,
                              size_t codeSize,
                              ALVRComputeTask** ppTask);
```

#### *Parameters*

#### shaderModel

Shader model (currently *ALVR_SHADER_MODEL_D3D11* only)

#### flags

Compute task flags, reserved for future use, currently ignored.

#### pCode

A pointer to a compiled shader bytecode. For *ALVR_SHADER_MODEL_D3D11* shader model this is the data returned by the *D3DCompileFromFile* function (it is retrieved with *ID3DBlob::GetBufferPointer()* method on a compiled blob returned by the *D3DCompileFromFile* function).

#### codeSize

The size of the shader bytecode passed in *pCode*.

### ppTask

A pointer to the *ALVRComputeTask* interface representing the newly created task.

### Return

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

## CreateBuffer

Create a buffer resource for Asynchronous Compute.

```
ALVR_RESULT CreateBuffer(const ALVRBufferDesc* pDesc,
                                ALVRBuffer** ppBuffer);
```

### *Parameters*

### pDesc

A pointer to the *ALVRBufferDesc* structure containing buffer creation parameters.

### ppBuffer

A pointer to the *ALVRBuffer* interface representing the newly created buffer.

### Return

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

### *Remarks*

The *ALVRBufferDesc* structure describes the buffer being created and is defined as follows:

```
struct ALVRBufferDesc
{
    unsigned int        bufferFlags;
    unsigned int        cpuAccessFlags;
    unsigned int        apiSupport;
    size_t              size;
    size_t              structureStride;
    ALVR_FORMAT         format;
};
```

### bufferFlags

A combination of the following values describing how the buffer is used by the shader:

- *ALVR_BUFFER_SHADER_INPUT* – the buffer is used as input by the Asynchronous Compute shader
- *ALVR_BUFFER_SHADER_OUTPUT* – the buffer is used as output by the Asynchronous Compute shader

- *ALVR_BUFFER_CONSTANT* – the buffer data is constant and is not modifiable by the Asynchronous Compute shader

**cpuAccessFlags**

A combination of the following values describing how the buffer is used by the CPU:

- *ALVR_CPU_ACCESS_NONE* – the buffer is not accessible to the CPU
- *ALVR_CPU_ACCESS_READ* – the buffer can be accessed by the CPU for reading
- *ALVR_CPU_ACCESS_WRITE* – the buffer can be accessed by the CPU for writing

**apiSupport**

A combination of the following values describing how the buffer is accessed through different APIs:

- *ALVR_RESOURCE_API_ASYNC_COMPUTE* – the buffer is used by an Asynchronous Compute shader
- *ALVR_RESOURCE_API_D3D11* – the buffer is used for rendering through the Direct3D 11 API

**size**

Buffer size in bytes.

**structureStride**

Size of the data element contained in the buffer. A buffer is organized as an array of equal size structures with the number of elements being equal to *size/structureStride*.

**format**

The format of data contained in the buffer. When the buffer is untyped, the *format* field should be set to *ALVR_FORMAT_UNKNOWN*.

**Return**

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

## CreateSampler

Create a sampler object for Asynchronous Compute.

```
ALVR_RESULT CreateSampler(const ALVRSamplerDesc* pDesc,
                          ALVRSampler** ppSampler);
```

### *Parameters*

**spDesc**

Sampler descriptor. A sample descriptor is represented by a structure of the *ALVRSamplerDesc* type:

```
struct ALVRSamplerDesc
{
   ALVR_FILTER_MODE    filterMode;
   ALVR_ADDRESS_MODE   addressU;
   ALVR_ADDRESS_MODE   addressV;
   ALVR_ADDRESS_MODE   addressW;
};
```

The *filterMode* field can be set to either:

- *ALVR_FILTER_POINT* for point sampling for minification, magnification and mip-level sampling (similar to *D3D11_FILTER_MIN_MAG_MIP_POINT* in Direct3D 11)
- *ALVR_FILTER_LINEAR* for linear interpolation for minification, magnification and mip-level sampling (similar to *D3D11_FILTER_MIN_MAG_MIP_LINEAR* in Direct3D 11)

The *addressU*, *addressV* and *addressW* members specify a method to resolve a U, V or W texture coordinate outside the 0 to 1 range and can be set to one of the following values:

- *ALVR_ADDRESS_WRAP* – repeat the texture at every integer texture coordinate value
- *ALVR_ADDRESS_MIRROR* – flip and repeat the texture at every integer texture coordinate value
- *ALVR_ADDRESS_CLAMP* – texture coordinates below 0 are set to the texture color at 0 and coordinates above 1 are set to the texture color at 1

### ppSampler

A pointer to the *ALVRSampler* interface representing the newly created sampler.

### Return

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

## CreateSurface

Create a surface object (a.k.a. texture or image) for Asynchronous Compute.

```
ALVR_RESULT CreateSurface(const ALVRSurfaceDesc* pDesc,
                          ALVRSurface** ppSurface);
```

### *Parameters*

### spDesc

Surface descriptor. A surface descriptor is represented by a structure of the *ALVRSurfaceDesc* type:

```
struct ALVRSurfaceDesc
{
    ALVR_SURFACE_TYPE   type;
    ALVR_FORMAT         format;
    unsigned int        surfaceFlags;
    unsigned int        apiSupport;
    unsigned int        width;
    unsigned int        height;
    unsigned int        depth;
    ALVR_FORMAT         shaderInputFormat;
    ALVR_FORMAT         shaderOutputFormat;
};
```

*type* can be set to either *ALVR_SURFACE_1D*, *ALVR_SURFACE_2D* or *ALVR_SURFACE_3D* for single dimension, two-dimensional and three-dimensional surfaces respectively

*format* specifies the pixel format of the surface being created.

*surfaceFlags* is a combination of the following values:

- *ALVR_SURFACE_SHADER_INPUT* – indicates that the surface is used as Asynchronous Compute shader input
- *ALVR_SURFACE_SHADER_OUTPUT* – indicates that the surface is used as Asynchronous Compute shader input
- *ALVR_SURFACE_RENDER_TARGET* – indicates that the surface is used for 3D rendering outside of Asynchronous Compute API (can be combined with either *ALVR_SURFACE_SHADER_INPUT,* or *ALVR_SURFACE_SHADER_OUTPUT*, or both)

*apiSupport* indicates which APIs a surface can be used with and can be any combination of the following values:

- *ALVR_SURFACE_API_ASYNC_COMPUTE* –the surface is used with Asynchronous Compute shaders
- *ALVR_SURFACE_API_D3D11* –the surface is used with Direct3D 11 API

The *width*, *height* and *depth* members specify the size of the surface for each dimension

*shaderInputFormat* and *shaderOutputFormat* optionally define how the surface data should be interpreted by a shader when the surface is bound to a shader as input or output. These shader format overrides must be the same bit depth as the *format*. *shaderOutputFormat* must be a format that the shader can write. When these field are not in use, they should be set to *ALVR_FORMAT_UNKNOWN*.

### ppSurface

A pointer to the *ALVRSurface* interface representing the newly created surface.

### Return

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

## CreateTimestamp

Create the *ALVRComputeTimestamp* object

```
ALVR_RESULT CreateTimestamp(ALVRComputeTimestamp** ppTimestamp);
```

### *Parameters*

#### ppTimestamp

A pointer to a location to receive a pointer to the *ALVRComputeTimestamp* interface.

#### Return

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when *ppTimestamp* is *nullptr*

*ALVR_FAIL* on any other error

## QueueTask

Queue an Asynchronous Compute task to the Asynchronous Compute context.

```
ALVR_RESULT QueueTask(ALVRComputeTask* pTask,
                      const ALVRPoint3D* pOffset,
                      const ALVRSize3D* pSize);
```

### *Parameters*

#### pTask

A pointer to an *ALVRComputeTask* object.

#### pOffset

Offsets used to calculate threadgroup indices in the shader. The *ALVRPoint3D* structure is defined as follows:

```
struct ALVRPoint3D
{
    unsigned int x;
    unsigned int y;
    unsigned int z;
};
```

For 2D tasks only *x* and *y* fields are used, for 1D tasks only the *x* field is used.

When *pOffset* is *nullptr*, it will be assumed that the threadgroup indices starts at zero value.

#### pSize

Dimensions of the task in threadgroups. The *ALVRSize3D* structure is defined as follows:

```
struct ALVRSize3D
{
    unsigned int width;
    unsigned int height;
    unsigned int depth;
};
```

**Return**

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when *ppTimestamp* is *nullptr*

*ALVR_FAIL* on any other error

### *Remarks*

Note that queuing a task does not start it, but rather just adds it to the task queue. To execute all queued tasks call the *ALVRComputeContext::Flush* method.

## QueueTimestamp

Queue a GPU timestamp write request to the Asynchronous Compute context.

```
ALVR_RESULT QueueTimestamp(ALVRComputeTimestamp* pTimestamp);
```

### *Parameters*

**pTimestamp**

A pointer to an *ALVRComputeTimestamp* object.

**Return**

*ALVR_OK* on success

*ALVR_FAIL* on any error

## QueueSemaphoreSignal

Queue a GPU semaphore signal request to the Asynchronous Compute context.

```
ALVR_RESULT QueueSemaphoreSignal(ALVRGpuSemaphore* pSemaphore);
```

### *Parameters*

**pTimestamp**

A pointer to an *ALVRGpuSemaphore* object.

**Return**

*ALVR_OK* on success

*ALVR_FAIL* on any error

## QueueSemaphoreWait

Queue a GPU semaphore wait request to the Asynchronous Compute context.

```
ALVR_RESULT QueueSemaphoreWait(ALVRGpuSemaphore* pSemaphore);
```

### *Parameters*

**pTimestamp**
A pointer to an *ALVRGpuSemaphore* object.

**Return**
*ALVR_OK* on success

*ALVR_FAIL* on any error

## QueueCopyBufferToBuffer

Queue a request to copy a region in a buffer to another buffer using the Asynchronous Compute context.

```
ALVR_RESULT QueueCopyBufferToBuffer(ALVRBuffer* pSrc,
                                    size_t srcOffset,
                                    ALVRBuffer* pDst,
                                    size_t dstOffset,
                                    size_t size);
```

### *Parameters*

**pSrc**
A pointer to the source *ALVRBuffer* object.

**srcOffset**
The offset in bytes in the source buffer to copy data from.

**pDst**
A pointer to the destination *ALVRBuffer* object.

**dstOffset**
The offset in bytes in the destination buffer to copy data to.

**size**
The size in bytes of the buffer region to be copied.

**Return**
*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when *pSrc* or *pDst* are *nullptr* or when *size* is 0

*ALVR_FAIL* on any other error

## QueueCopyBufferToSurface

Queue a request to copy a buffer region to a surface using the Asynchronous Compute context.

```
ALVR_RESULT QueueCopyBufferToSurface(ALVRBuffer* pSrc,
                                     size_t srcOffset,
                                     ALVRSurface* pDst,
                                     const ALVRBox* pDstBox);
```

### *Parameters*

#### pSrc

A pointer to the source *ALVRBuffer* object.

#### srcOffset

The offset in bytes in the source buffer to copy surface data from.

#### pDst

A pointer to the destination *ALVRSurface* object.

#### pDstBox

The destination surface region to copy data to. The *ALVRBox* structure is defined as follows:

```
struct ALVRBox
{
    unsigned int left;
    unsigned int top;
    unsigned int front;
    unsigned int right;
    unsigned int bottom;
    unsigned int back;
};
```

#### Return

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when *pSrc*, *pDst* or *pDstBox* are *nullptr* or when *size* is 0

*ALVR_FAIL* on any error

## QueueCopySurfaceToSurface

Queue a request to copy a surface region to another surface using the Asynchronous Compute context.

```
ALVR_RESULT QueueCopySurfaceToSurface(ALVRSurface* pSrc,
                                      const ALVRBox* pSrcBox,
                                      ALVRSurface* pDst,
                                      const ALVRPoint3D* pDstOffset);
```

### *Parameters*

#### pSrc

A pointer to the source *ALVRSurface* object.

**pSrcBox**

The source surface region to copy data from. When *pSrcBox* is *nullptr*, it is assumed that the entire source surface is being copied.

**pDst**

A pointer to the destination *ALVRSurface* object.

**pDstOffset**

The destination surface offset to the region where data is copied. The *ALVRPoint3D* structure is defined as follows:

```
struct ALVRPoint3D
{
    unsigned int x;
    unsigned int y;
    unsigned int z;
};
```

**Return**

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when *pSrc* or *pDst* are *nullptr* or when *size* is 0

*ALVR_FAIL* on any error

## QueueCopySurfaceToBuffer

Queue a request to copy a surface region to a buffer using the Asynchronous Compute context.

```
ALVR_RESULT QueueCopySurfaceToBuffer(ALVRSurface* pSrc,
                                     const ALVRBox* pSrcBox,
                                     ALVRBuffer* pDst,
                                     size_t dstOffset);
```

### *Parameters*

**pSrc**

A pointer to the source *ALVRSurface* object.

**pSrcBox**

The source surface region to copy data from. When *pSrcBox* is *nullptr*, it is assumed that the entire source surface is being copied.

**pDst**

A pointer to the destination *ALVRBuffer* object.

**dstOffset**

An offset in bytes in the destination buffer to copy data to.

When *pDstOffset* is *nullptr*, it is assumed that copied region starts at the beginning of the destination buffer.

**Return**

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when *pSrc* or *pDst* are *nullptr* or when *size* is 0

*ALVR_FAIL* on any error

## Flush

Execute all operations previously queued to the Asynchronous Compute context and optionally trigger a fence when the last operation is completed.

```
ALVR_RESULT Flush(ALVRFence* pFence);
```

### *Parameters*

**pFence**

An optional fence to be triggered when the last operation queued to the Asynchronous Compute prior to flush is completed.

**Return**

*ALVR_OK* on success

*ALVR_FAIL* on any error

## OpenSharedBuffer

Open a buffer created outside of the Asynchronous Compute context using a shared handle. The buffer can be shared between different processes, devices and different APIs, such as Direct3D 11 and LiquidVR.

```
ALVR_RESULT OpenSharedBuffer(const ALVROpenBufferDesc* pDesc,
                             ALVRBuffer** ppBuffer);
```

### *Parameters*

**pDesc**

A pointer to the *ALVROpenBufferDesc* structure containing parameters for opening a shared buffer.

**ppBuffer**

A pointer to the *ALVRBuffer* interface representing the newly opened buffer.

**Return**

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

### *Remarks*

The *OpenSharedBuffer* method allows sharing a buffer between Direct3D and LiquidVR, within a process or between multiple processes. For more information on shared resources please refer to the following MSDN article: https://msdn.microsoft.com/en-us/library/windows/desktop/bb219800(v=vs.85).aspx#Sharing_Resources

The *ALVROpenBufferDesc* structure describes the opened buffer and is defined as follows:

```
struct ALVROpenBufferDesc
{
    HANDLE              sharedHandle;
    unsigned int        bufferFlags;
    size_t              structureStride;
    ALVR_FORMAT         format;
    unsigned int        openFlags;
};
```

### sharedHandle

A shared resource handle obtained from D3D11 API.

### bufferFlags

A combination of the following values describing how the buffer is used by the shader:

*ALVR_BUFFER_SHADER_INPUT* – the buffer is used as input by the Asynchronous Compute shader

*ALVR_BUFFER_SHADER_OUTPUT* – the buffer is used as output by the Asynchronous Compute shader

*ALVR_BUFFER_CONSTANT* – the buffer is constant and is not modifiable by the Asynchronous Compute shader

### structureStride

Size of the data element contained in the buffer. A buffer is organized as an array of equal size structures with the number of elements being equal to *size/structureStride*.

### format

The format of data contained in the buffer. When the buffer is untyped, the *format* field should be set to *ALVR_FORMAT_UNKNOWN*.

### openFlags

Reserved. Must be set to 0.

### Return

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

## OpenSharedSurface

Open a surface object created outside of the Asynchronous Compute context using a shared resource handle. The surface can be shared between different processes, devices and different APIs, such as Direct3D 11 and LiquidVR.

```
ALVR_RESULT OpenSharedSurface(const ALVROpenSurfaceDesc* pDesc,
                              ALVRSurface** ppSurface);
```

### *Parameters*

#### spDesc

Surface descriptor. A sample descriptor is represented by a structure of the *ALVROpenSurfaceDesc* type.

#### ppSurface

A pointer to the *ALVRSurface* interface representing the newly opened surface.

#### Return

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* on any other error

### *Remarks*

The *OpenSharedSurface* method allows sharing a surface between Direct3D and LiquidVR, within a process or between multiple processes. For more information on shared resources please refer to the following MSDN article: https://msdn.microsoft.com/en-us/library/windows/desktop/bb219800(v=vs.85).aspx#Sharing_Resources

The *ALVROpenSurfaceDesc* structure describes the opened surface and is defined as follows:

```
struct ALVROpenSurfaceDesc
{
    HANDLE              sharedHandle;
    ALVR_FORMAT         format;
    unsigned int        surfaceFlags;
    unsigned int        openFlags;
    ALVR_FORMAT         shaderInputFormat;
    ALVR_FORMAT         shaderOutputFormat;
};
```

#### sharedHandle

A texture handle obtained from APIs like *Direct3DDevice9::CreateTexture* when the *pSharedHandle* parameter is not *nullptr*.

#### format

Surface format of the opened surface, which must be the same bit-depth as the format of the original shared surface.

#### surfaceFlags

is a combination of the following values:

*ALVR_SURFACE_SHADER_INPUT* – indicates that the surface is used as Asynchronous Compute shader input

*ALVR_SURFACE_SHADER_OUTPUT* – indicates that the surface is used as Asynchronous Compute shader input

*ALVR_SURFACE_RENDER_TARGET* – indicates that the surface is used for 3D rendering outside of Asynchronous Compute API (can be combined with either *ALVR_SURFACE_SHADER_INPUT,* or *ALVR_SURFACE_SHADER_OUTPUT*, or both)

**openFlags**

Reserved. Must be set to 0.

**shaderInputFormat**

**shaderOutputFormat**

*shaderInputFormat* and *shaderOutputFormat* optionally define how the surface data should be interpreted by a shader when the surface is bound to a shader as input or output. These shader format overrides must be the same bit depth as the *format*. When these field are not in use, they should be set to *ALVR_FORMAT_UNKNOWN*.

## 3.13 ALVRComputeTask

### BindConstantBuffer

Bind a constant buffer to an Asynchronous Compute task.

```
ALVR_RESULT BindConstantBuffer(unsigned int slot, ALVRBuffer* pBuffer);
```

#### *Parameters*

**slot**

A zero-based index of a constant buffer slot.

**pBuffer**

A pointer to an *ALVRBuffer* object.

**Return**

*ALVR_OK* on success

*ALVR_FAIL* on any error

### BindSampler

Bind a sampler to an Asynchronous Compute task.

```
ALVR_RESULT BindSampler(unsigned int slot, ALVRSampler* pBuffer);
```

#### *Parameters*

**slot**

A zero-based index of a sampler slot.

**pSampler**
A pointer to an *ALVRSampler* object.

**Return**
*ALVR_OK* on success

*ALVR_FAIL* on any error

## BindInput

Bind a resource as input to an Asynchronous Compute task.

```
ALVR_RESULT BindInput(unsigned int slot, ALVRResource* pResource);
```

### *Parameters*

**slot**
A zero-based index of an input resource slot.

**pResource**
A pointer to an *ALVRResource* object to be used as input by the shader.

**Return**
*ALVR_OK* on success

*ALVR_FAIL* on any error

## BindOutput

Bind a resource as output to an Asynchronous Compute task.

```
ALVR_RESULT BindOutput(unsigned int slot, ALVRResource* pResource);
```

### *Parameters*

**slot**
A zero-based index of an output resource slot.

**pResource**
A pointer to an *ALVRResource* object to be used as output by the shader.

**Return**
*ALVR_OK* on success

*ALVR_FAIL* on any error

## 3.14 ALVRComputeTimestamp

### GetValue

Get the value of an Asynchronous Compute timestamp once it is written by an operation queued to the Asynchronous Compute context with *QueueTimestamp*. Use a fence to ensure timestamps are available before retrieving them.

```
ALVR_RESULT GetValue(uint64_t* pValue);
```

#### *Parameters*

**pValue**
A pointer to receive a 64-bit GPU timestamp value.

**Return**
*ALVR_OK* on success

*ALVR_FAIL* on any error

## 3.15 ALVRBuffer

### GetSize

Get size of a buffer.

```
size_t GetSize(void) const;
```

#### *Parameters*

None.

**Return**
Size of the buffer in bytes.

### Map

Map a GPU buffer into a CPU address space and return a pointer to the buffer in system memory for CPU access.

```
ALVR_RESULT Map(void** pData);
```

#### *Parameters*

**pData**
Returned CPU pointer to mapped GPU buffer memory.

**Return**
*ALVR_OK* on success

*ALVR_FAIL* on any error

### *Remarks*

A call to *ALVRBuffer::Map()* maps GPU memory to the CPU address space. The pointer to a buffer in system memory remains valid until *ALVRBuffer::Unmap()* is called. Calls to *ALVRBuffer::Map()* and *ALVRBuffer::Unmap()* are not reference-counted, i.e. subsequent calls to *ALVRBuffer::Map()* on an already mapped buffer will return the same pointer and a single call to *ALVRBuffer::Unmap()* would unmap the buffer regardless of how many times *ALVRBuffer::Map()* has been called.

## Unmap

Unmap a GPU buffer previously mapped with *ALVRBuffer::Map()*.

```
ALVR_RESULT Unmap(void);
```

### *Parameters*

None.

#### Return

*ALVR_OK* on success

*ALVR_FAIL* on any error

## 3.16 ALVRResourceD3D11

An encapsulation type for a Direct3D 11 resource, defined as follows:

```
struct ALVRResourceD3D11
{
    ID3D11Resource* pResource;
};
```

## 3.17 ALVRResource

The *ALVRResource* interface is a common interface to access ALVR resources.

## GetApiResource

Retrieve native API-specific resource object.

```
ALVR_RESULT GetApiResource(ALVR_RENDER_API renderApi, void* pResource);
```

### *Parameters*

#### renderApi

An ID of the rendering API to retrieve native resource object for. Can one of the following values:

- *ALVR_RENDER_API_D3D11* for Direct3D 11
- *ALVR_RENDER_API_ASYNC_COMPUTE* for Asynchronous Compute

#### pResource

A pointer to a location to receive the requested resource. The type of this object is API dependent (passed in the *renderApi* parameter). For Direct3D 11 it is a structure of the *ALVRResourceD3D11* type.

**Return**

*ALVR_OK* on success

*ALVR_INVALID_ARGUMENT* when any of the parameters are invalid

*ALVR_FAIL* when the update fails

## 3.18 ALVRSurface

The *ALVRSurface* interface provides access to a surface object. *ALVRSurface* inherits from *ALVRResource*. Currently the *ALVRSurface* interface does not add any additional methods to the *ALVRResource* interface.

To create a LiquidVR surface, call the *ALVRComputeContext::CreateSurface* method.

LiquidVR surface, specified at creation, can be 1D, 2D or 3D, as defined by the *ALVR_SURFACE_TYPE* enumeration as follows:

```
enum ALVR_SURFACE_TYPE
{
    ALVR_SURFACE_1D,
    ALVR_SURFACE_2D,
    ALVR_SURFACE_3D,
};
```

A surface's pixel format is defined by the *ALVR_FORMAT* enumeration defined as follows:

```
enum ALVR_FORMAT
{
    ALVR_FORMAT_UNKNOWN = 0,
    ALVR_FORMAT_R32G32B32A32_FLOAT = 2,
    ALVR_FORMAT_R32G32B32A32_UINT = 3,
    ALVR_FORMAT_R32G32B32A32_SINT = 4,
    ALVR_FORMAT_R16G16B16A16_FLOAT = 10,
    ALVR_FORMAT_R16G16B16A16_UNORM = 11,
    ALVR_FORMAT_R16G16B16A16_UINT = 12,
    ALVR_FORMAT_R16G16B16A16_SNORM = 13,
    ALVR_FORMAT_R16G16B16A16_SINT = 14,
    ALVR_FORMAT_R32G32_FLOAT = 16,
    ALVR_FORMAT_R32G32_UINT = 17,
    ALVR_FORMAT_R32G32_SINT = 18,
    ALVR_FORMAT_R10G10B10A2_UNORM = 24,
    ALVR_FORMAT_R10G10B10A2_UINT = 25,
    ALVR_FORMAT_R11G11B10_FLOAT = 26,
    ALVR_FORMAT_R8G8B8A8_UNORM = 28,
    ALVR_FORMAT_R8G8B8A8_UNORM_SRGB = 29,
    ALVR_FORMAT_R8G8B8A8_UINT = 30,
    ALVR_FORMAT_R8G8B8A8_SNORM = 31,
    ALVR_FORMAT_R8G8B8A8_SINT = 32,
    ALVR_FORMAT_R16G16_FLOAT = 34,
    ALVR_FORMAT_R16G16_UNORM = 35,
    ALVR_FORMAT_R16G16_UINT = 36,
    ALVR_FORMAT_R16G16_SNORM = 37,
    ALVR_FORMAT_R16G16_SINT = 38,
    ALVR_FORMAT_R32_FLOAT = 41,
    ALVR_FORMAT_R32_UINT = 42,
    ALVR_FORMAT_R32_SINT = 43,
    ALVR_FORMAT_R8G8_UNORM = 49,
    ALVR_FORMAT_R8G8_UINT = 50,
    ALVR_FORMAT_R8G8_SNORM = 51,
    ALVR_FORMAT_R8G8_SINT = 52,
    ALVR_FORMAT_R16_FLOAT = 54,
    ALVR_FORMAT_R16_UNORM = 56,
    ALVR_FORMAT_R16_UINT = 57,
    ALVR_FORMAT_R16_SNORM = 58,
    ALVR_FORMAT_R16_SINT = 59,
    ALVR_FORMAT_R8_UNORM = 61,
    ALVR_FORMAT_R8_UINT = 62,
    ALVR_FORMAT_R8_SNORM = 63,
    ALVR_FORMAT_R8_SINT = 64,
    ALVR_FORMAT_R9G9B9E5_SHAREDEXP = 67,
    ALVR_FORMAT_BC1_UNORM = 71,
    ALVR_FORMAT_BC1_UNORM_SRGB = 72,
    ALVR_FORMAT_BC2_UNORM = 74,
    ALVR_FORMAT_BC2_UNORM_SRGB = 75,
    ALVR_FORMAT_BC3_UNORM = 77,
    ALVR_FORMAT_BC3_UNORM_SRGB = 78,
    ALVR_FORMAT_BC4_UNORM = 80,
    ALVR_FORMAT_BC4_SNORM = 81,
    ALVR_FORMAT_BC5_UNORM = 83,
    ALVR_FORMAT_BC5_SNORM = 84,
```

```
    ALVR_FORMAT_B5G6R5_UNORM = 85,
    ALVR_FORMAT_B5G5R5A1_UNORM = 86,
    ALVR_FORMAT_B8G8R8A8_UNORM = 87,
    ALVR_FORMAT_B8G8R8X8_UNORM = 88,
    ALVR_FORMAT_B8G8R8A8_UNORM_SRGB = 91,
    ALVR_FORMAT_B8G8R8X8_UNORM_SRGB = 93,
    ALVR_FORMAT_BC6H_UF16 = 95,
    ALVR_FORMAT_BC6H_SF16 = 96,
    ALVR_FORMAT_BC7_UNORM = 98,
    ALVR_FORMAT_BC7_UNORM_SRGB = 99,
    ALVR_FORMAT_FORCE_UINT = 0xffffffff
};
```