



ANARITM 1.0.0 – Specification

The Khronos[®] ANARI Working Group

Version 1.0.0, 2025-04-03 19:16:28Z: from git branch: Git branch not available commit: Git commit not available

Table of Contents

1. Introduction	2
1.1. Document Conventions	2
1.1.1. Informative Language	3
1.1.2. Normative Terminology	3
1.1.3. Technical Terminology	4
1.1.4. Prefixes	4
1.1.5. Normative References	4
2. API Design Choices	5
2.1. C99	5
2.2. Common Front-end Library	5
2.3. Opaque Objects, Parameterization, and Properties	5
2.4. Extensions	5
2.5. Support for Distributed Applications (Informative)	6
3. Common API Concepts	7
3.1. Thread Safety, Asynchronous Operations	7
3.2. Scene Hierarchy (Informative)	7
3.3. Object Definition	8
3.4. Object Handle Representation	8
3.5. Object Creation and Lifetime	9
3.6. Parameters, Types, and Commits	9
3.6.1. Color	17
3.7. Object Introspection	17
3.8. Object Properties	20
3.9. Arrays and Memory Ownership	21
3.10. Libraries	24
3.11. Devices	25
3.12. Error Handling	26
3.13. Attributes	27
4. Rendering Frames	29
5. Object Types and Subtypes	31
5.1. Frame	31
5.2. Camera	34
5.2.1. Perspective	36
5.2.2. Omnidirectional	37
5.2.3. Orthographic	37
5.3. Renderer	38
5.4. World	39
5.5. Instance	40

5.5.1. Transform	41
5.5.2. Motion Transform	41
5.5.3. Motion Scale Rotation Translation	42
5.6. Group	42
5.7. Light	43
5.7.1. Directional	43
5.7.2. HDRI	44
5.7.3. Point	45
5.7.4. Quad	46
5.7.5. Ring	47
5.7.6. Spot	48
5.8. Surface	49
5.9. Geometry	49
5.9.1. Cone	50
5.9.2. Curve	51
5.9.3. Cylinder	52
5.9.4. Isosurface	54
5.9.5. Quad	54
5.9.6. Sphere	56
5.9.7. Triangle	57
5.10. Sampler	59
5.10.1. Image1D	59
5.10.2. Image2D	60
5.10.3. Image3D	61
5.10.4. Primitive	63
5.10.5. Transform	63
5.11. Material	63
5.11.1. Matte	64
5.11.2. Physically Based	64
5.12. Volume	67
5.12.1. TransferFunction1D	67
5.13. Spatial Field	68
5.13.1. Structured Regular	68
5.13.2. Unstructured	69
5.14. Extension Objects	70
Appendix A: Function Index (Informative)	71
Appendix B: Extension Index (Informative)	73
Appendix C: Example Algorithm for Rendering a TransferFunction1D Volume (Informative)	75
Appendix D: Credits (Informative)	76
Working Group Contributors to ANARI	76
Other Credits	76

Copyright 2023 The Khronos Group Inc.

This Specification is protected by copyright laws and contains material proprietary to Khronos. Except as described by these terms, it or any components may not be reproduced, republished, distributed, transmitted, displayed, broadcast or otherwise exploited in any manner without the express prior written permission of Khronos.

Khronos grants a conditional copyright license to use and reproduce the unmodified Specification for any purpose, without fee or royalty, EXCEPT no licenses to any patent, trademark or other intellectual property rights are granted under these terms.

Khronos makes no, and expressly disclaims any, representations or warranties, express or implied, regarding this Specification, including, without limitation: merchantability, fitness for a particular purpose, non-infringement of any intellectual property, correctness, accuracy, completeness, timeliness, and reliability. Under no circumstances will Khronos, or any of its Promoters, Contributors or Members, or their respective partners, officers, directors, employees, agents or representatives be liable for any damages, whether direct, indirect, special or consequential damages for lost revenues, lost profits, or otherwise, arising from or in connection with these materials.

This document contains extensions which are not ratified by Khronos, and as such is not a ratified Specification.

Khronos® and Vulkan® are registered trademarks, and ANARI™, glTF™, SPIR™, SPIR-V™, and SYCL™ are trademarks of The Khronos Group Inc. OpenCL™ is a trademark of Apple Inc. used under license by Khronos. OpenGL® is a registered trademark of Hewlett Packard Enterprise used under license by Khronos. ASTC is a trademark of ARM Holdings PLC. All other product names, trademarks, and/or company names are used solely for identification and belong to their respective owners.

Chapter 1. Introduction

The fundamental problem being solved by the ANARI standard is to provide application developers with a high-level rendering API that can be used to render images of visualizations containing 3D surface geometry and volumetric data. The API will support rendering techniques such as rasterization and high-fidelity path tracing, and will do so at low application development-time cost.

Although many renderers and APIs already exist, and some of them successfully address the primary requirement above, in practice they are vendor-, hardware platform-, or rendering algorithm-specific, or they provide high-performance building blocks for rendering, but not a complete renderer implementation with a high-level API. ANARI aims to address the limitations of these existing APIs. ANARI fully abstracts vendor-, hardware platform-, and rendering algorithm-specific details behind the API. By doing so, a multiplicity of rendering back-end implementations can be used to their full capability, without the need for renderer-specific code in applications that use ANARI.

Since ANARI provides a high-level API abstraction, significant freedom is provided to back-end renderer implementations. This freedom enables implementations to use any practical rendering algorithm for image generation, although a key focus and interest for ANARI is support for high-fidelity physically based rendering methods.

ANARI applications do not specify the details of the rendering process. Using the ANARI API, applications specify object surface or volume data to be rendered, and any associated parameters that might affect appearance, such as their material properties, texturing, and color transfer functions, as appropriate. ANARI applications retain full responsibility for managing non-rendering attributes of geometry through their own means. ANARI provides rendering-focused functionality only, so higher level scene graphs and other more general functionality must be obtained through other APIs or applications which use ANARI.

Domains which leverage 3D graphics have diverse rendering needs involving tradeoffs among quality, speed, and scalability to available hardware resources. It is typical for 3D applications to use both visual and quantitative rendering techniques to satisfy user demands. Furthermore, it is common for an application to need rendering from local CPU or GPU hardware, with parallel scaling through multiple machines in a cluster to exploit additional distributed compute and memory resources. The ANARI API provides the necessary abstractions to allow these needs to be met by back-end renderers, without excessive exposure of hardware or rendering algorithm details to the application. ANARI seeks to standardize extensions for rendering domains including, but not limited to, scientific visualization, professional visualization, visual effects, and engineering CAD.

Multiple ANARI back-ends may be exposed through the API at runtime. The ANARI API provides the application with the means to enumerate available back-ends, methods for querying high-level capabilities of the available back-ends, and the ability to instantiate a back-end and at least one associated renderer, which can then be used to render images.

1.1. Document Conventions

The ANARI specification is intended for use by both implementors of the API and application

developers seeking to make use of the API, forming a contract between these parties. Specification text may address either party; typically the intended audience can be inferred from context, though some sections are defined to address only one of these parties. Any requirements, prohibitions, recommendations or options defined by [normative terminology](#) are imposed only on the audience of that text.

1.1.1. Informative Language

Some language in the specification is purely informative, intended to give background or suggestions to implementors or developers.

If an entire chapter or section contains only informative language, its title will be suffixed with "(Informative)".

All NOTES are implicitly informative.

1.1.2. Normative Terminology

Within this specification, the key words **must**, **required**, **should**, **recommended**, **may**, and **optional** are to be interpreted as described in [RFC 2119 - Key words for use in RFCs to Indicate Requirement Levels](https://www.ietf.org/rfc/rfc2119.txt) (<https://www.ietf.org/rfc/rfc2119.txt>). These key words are highlighted in the specification for clarity. In text addressing application developers, their use expresses requirements that apply to application behavior. In text addressing implementors, their use expresses requirements that apply to implementations.

In text addressing application developers, the additional key words **can** and **cannot** are to be interpreted as describing the capabilities of an application, as follows:

can

This word means that the application is able to perform the action described.

cannot

This word means that the API and/or the execution environment provide no mechanism through which the application can express or accomplish the action described.

These key words are never used in text addressing implementors.

Note



There is an important distinction between **cannot** and **must not**, as used in this Specification. **Cannot** means something the application literally is unable to express or accomplish through the API, while **must not** means something that the application is capable of expressing through the API, but that the consequences of doing so are undefined and potentially unrecoverable for the implementation (see [Error Handling](#)).

Unless otherwise noted in the section heading, all sections and appendices in this document are normative.

1.1.3. Technical Terminology

The ANARI Specification makes use of common engineering and graphics terms such as **Device**, **Sampler**, and **Frame** to identify and describe ANARI API constructs and their attributes, states, and behaviors. The Specification text provides definitions of the terms. When a term is used in normative language within the Specification, the definitions within the Specification govern and supersede any meanings the terms may have in other technical contexts (i.e., outside the Specification).

1.1.4. Prefixes

Prefixes are used in the API to denote specific semantic meaning of ANARI names, or as a label to avoid name clashes, and are explained here:

ANARI/anari

The ANARI namespace

All types, commands, enumerants and defines in this specification are prefixed with these strings.

The **ANARI_** prefix of named [data types](#) and [extensions](#) is omitted in tables and itemizations for readability and brevity.

1.1.5. Normative References

References to external documents are considered normative references if the Specification uses any of the normative terms defined in [Normative Terminology](#) to refer to them or their requirements, either as a whole or in part.

The following documents are referenced by normative sections of the specification:

IEEE. August, 2008. *IEEE Standard for Floating-Point Arithmetic*. IEEE Std 754-2008. <https://dx.doi.org/10.1109/IEEESTD.2008.4610935> .

The Khronos® Vulkan Working Group. January, 2020 *Vulkan® 1.2 – A Specification*, Section 16.8. Image Sample Operations, <https://registry.khronos.org/vulkan/> .

The Khronos® 3D Formats Working Group *glTF™ 2.0 Specification*, Appendix B: BRDF Implementation <https://registry.khronos.org/glTF/> .

Chapter 2. API Design Choices

Here we outline several fundamental ANARI API design choices that address key goals of broad application and programming language support, and ease of application development, integration, testing, and distribution.

2.1. C99

The ANARI API is specified as a C99 API in order to provide compiler-independent linkage, thereby supporting easy integration into a broad range of applications based on a variety of compiled languages, including C, C++, Fortran, and dynamic languages such as Python and Julia, among others. C99 provides improved IEEE-754 floating point rounding behavior, integer types in common with C++, and other refinements relative to ANSI C.

2.2. Common Front-end Library

ANARI is specified as a common API header and front-end library (using either static or dynamic linkage) capable of loading available ANARI back-end device implementations at runtime, known as [devices](#). ANARI back-end devices are created by standard implementors and are expected to be distributed, installed, or upgraded independently of the standard API header and front-end library.

2.3. Opaque Objects, Parameterization, and Properties

The ANARI API is designed to encapsulate scene data and rendering operations as opaque object handles using string-value pairs to parameterize them. This facilitates a dominantly unidirectional flow of scene information from the application to the instantiated ANARI device. As a result, the vast majority of ANARI API calls have a write-only behavior pattern to minimize imposed implementation requirements.

An application can create ANARI objects and set named inputs on them called [parameters](#). However, no mechanism is provided to subsequently query parameter data, since even the existence of a query mechanism would impose additional performance and storage restrictions for back-end renderer implementations (e.g. distributed rendering contexts). [Object introspection](#) can be used to query object subtypes and information about their parameters.

Additionally, ANARI objects can publish named outputs called [properties](#). Such output is specific to the type and semantics of the object, but has a generic interface function for access.

2.4. Extensions

The ANARI specification defines a set of usable extensions exposed via the C API. Extensions are loosely defined as observable behavior from a particular use of the ANARI API. This most commonly is the existence of usable object subtypes, but also can be pre/post conditions on a function call, specific behavior of an object parameter, or guaranteed availability of a property.

Applications can query extensions made available by a device implementation through queries

either on the library or the live device instance. Querying available extensions via the library represents the list of extensions implemented by the vendor at all, while querying extensions on a live device instance will also factor in the runtime environment (i.e. available underlying hardware to the device). In either case, extension queries are intended to be the way an application determines if the device is acceptable in meeting the application's requirements, or if the application wants to adapt itself to what extensions are available.

Vendor extensions are extensions that have not yet made it into the ANARI specification. These extensions are reported in the same list as core extensions, but have vendor specific names, which may exist in more than one device from that vendor. This is a good way for vendors to agree upon and get experience with a extension prior to bring it officially into the specification.

All extensions represent functionality which provides additional capabilities to the application. A reported extension should never represent a reduction in capability.

Testable extension names are inlined in the specification alongside their definition. All extensions follow the prefix convention of `ANARI_[VENDOR]_[NAME]`, where ratified extensions use `KHR` as the vendor string.

Implemented extensions may be queried via [introspection](#) functions. Currently available extensions for instantiated devices and renderers may be queried via the [extension property](#).

A full list of ANARI extensions can be found in the [Extension Index \(Informative\)](#) appendix.

2.5. Support for Distributed Applications (Informative)

ANARI is designed to be compatible with applications which are distributed across multiple processes which may reside on multiple machines. ANARI implementors provide distributed rendering capabilities using existing API semantics relevant to running in distributed environments, such as multiple processes invoking API calls in lock-step.



Note

Interoperability with underlying networking APIs, such as MPI, are expected to be documented by the vendor.

Chapter 3. Common API Concepts

This section describes concepts common to all object types for creation, parameterization, and property queries.

3.1. Thread Safety, Asynchronous Operations

The ANARI API may be called from any thread but calls that share objects must be synchronized and may not overlap. By default, this includes the device. Therefore, calls to the same device must be externally synchronized.

Expensive operations can be implemented asynchronously, which avoids blocking the calling application. Each asynchronous operation is specified as to what API calls are legal to make while the operation has not yet completed.

Note



Making some operations asynchronous complements the thread safety of ANARI, and typically is used for a different purpose. Where a multithreaded application may be trying to work on constructing a scene (which could be many smaller ANARI API calls) while another scene is being rendered in parallel, asynchronous rendering focuses on removing the need for long running operations to cause an application to use multithreading just to avoid blocking on ANARI.

Devices that implement extension `KHR_DEVICE_SYNCHRONIZATION` relax synchronization requirements by excluding the device from the external synchronization requirement if the device is passed as the first argument only. Calls that are operating on the device as the object must still be synchronized with all other calls involving the same device. Likewise `anariRenderFrame` must be synchronized with all other ANARI calls as well. This only applies to the `anariRenderFrame` call itself, not the asynchronous render operations which remain subject to asynchronous rendering restrictions.

This lowers the scope of required synchronization to individual ANARI objects allowing different threads to operate concurrently on different objects within the same device.

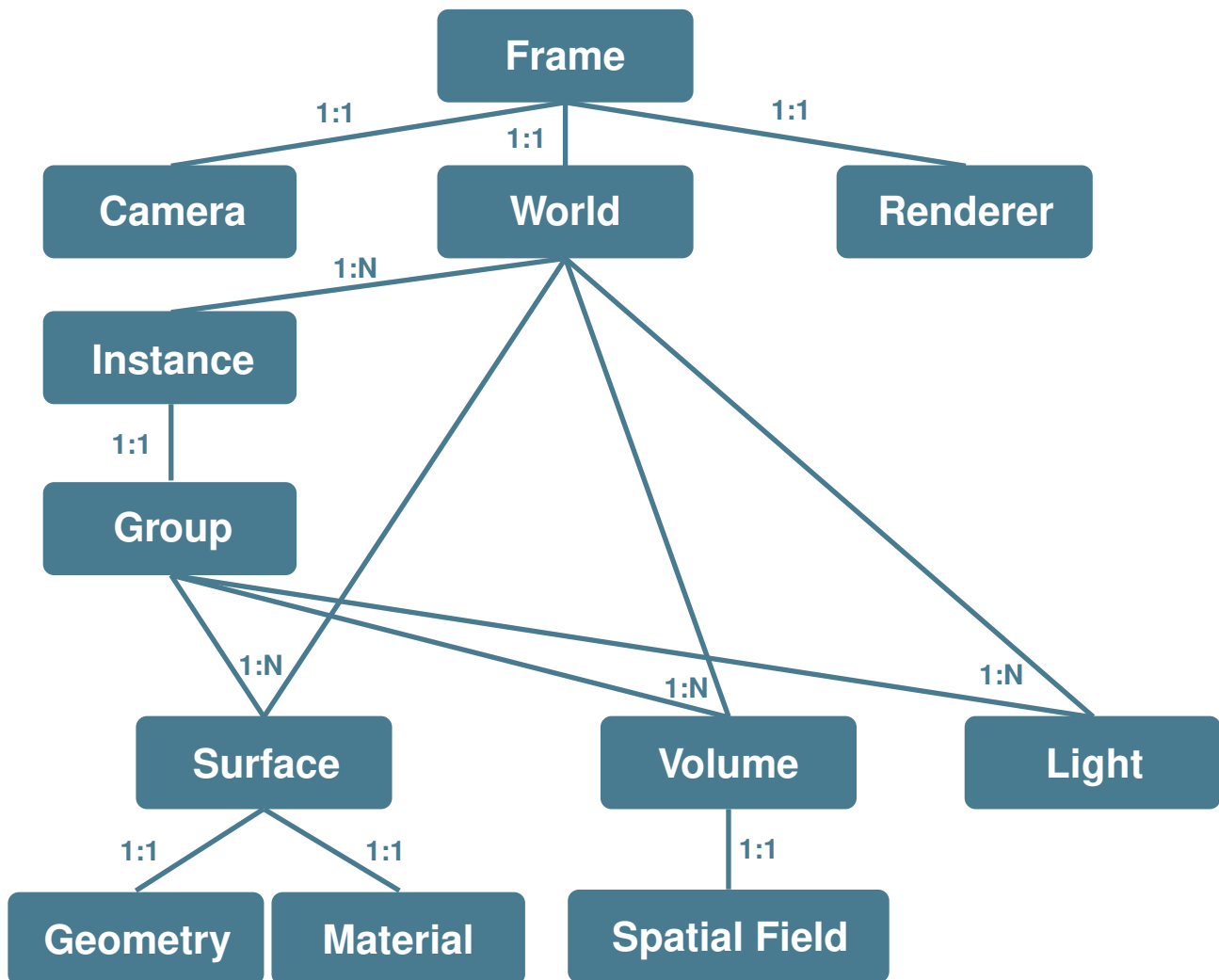
3.2. Scene Hierarchy (Informative)

ANARI scenes are represented as hierarchies of objects. This section describes their relationship to each other.

The `Frame` is the root object of the scene. It holds the framebuffer configuration and the `World`, `Camera`, and `Renderer` objects. The Camera configures the projection of the rendering used to view the World. The Renderer holds parameters relating to the rendering algorithm. The World holds arrays of the drawable objects of the scene such as `Surface`, `Volume`, and `Light` either directly or via an array of `Instance` containing `Group`. A Group holds arrays of Surface, Volume, and Light to be instanced together. An Instance combines a Group with a transform for placement of the same collection of objects at multiple locations within the same World. A Surface represents drawable surfaces containing a `Geometry` and a `Material`. A Geometry specifies drawable primitives and data

associated with them. A Material specifies the surface's appearance related to the data from the Geometry. A Volume represents volumetric drawable objects and may contain [Spatial Field](#) objects. A Spatial Field represents a field of values in 3D space. A Light represents sources of illumination.

The following diagram illustrates the relationships described above:



3.3. Object Definition

All objects are characterized by the following items

- Objects are represented as an [opaque handle](#)
- Objects must be able to accept [parameters](#)
- Objects must be able to post [properties](#)
- An object application reference count must only be modifiable by [anariRelease](#) and [anariRetain](#)

3.4. Object Handle Representation

Objects and devices in ANARI are represented by opaque 64-bit handles. The [NULL](#) handle never represents a valid object. Handles are only valid for the device from which they were created, and

using handles with other devices leads to undefined behavior.

3.5. Object Creation and Lifetime

Objects are created by calling an `anariNew*` factory function (for example `anariNewGeometry`). Each object type has its own corresponding factory function. All objects, including devices, are only valid in the process in which they are created.

Object lifetime is managed by opaque reference counting. Objects have a public and internal reference count. Objects are created with a public reference count of 1, which can be increased with `anariRetain` and decreased with `anariRelease`. Once the public reference count has been decreased to 0, the object becomes inaccessible to host code, where using its handle in subsequent API calls is invalid and results in undefined behavior.

The signatures to `anariRelease` and `anariRetain` are as follows

```
void anariRelease(ANARIDevice, ANARIObject);

void anariRetain(ANARIDevice, ANARIObject);
```

Calling `anariRelease` with a `NULL` object-handle for the second argument is not an error. The internal reference count is managed by the API and will keep objects alive for as long as the implementation needs them. Therefore, user code may release objects as soon as it no longer requires access to them.

3.6. Parameters, Types, and Commits

Objects are configured by parameters, which are identified by a string name and are set using `anariSetParameter`. Multiple types can be valid for a parameter, but only one type can be set for a particular parameter name at any given time. Setting a parameter with a different type overwrites its previous value and type. Attempting to set a parameter that is unknown to the implementation has no effect on object state, but may cause an ANARI device implementation to emit warnings. The same applies if an unsupported type for a parameter is used.

The signatures to `anariSetParameter` is as follows

```
void anariSetParameter(
    ANARIDevice,
    ANARIObject,
    const char *parameterName,
    ANARIDataType,
    const void *value
);
```

The `ANARI_STRING` valued parameter named `name` on all object types is reserved across all implementations and can optionally be used by applications to inject human readable identifiers into the command stream. This can be useful for debugging purposes, for example.

Implementations are allowed to ignore this parameter and must not emit warnings when it is present.

Changes to parameter values must only take effect once `anariCommitParameters` has been called on the object:

```
void anariCommitParameters(ANARIDevice, ANARIObjct);
```

Uncommitted parameters must have no effect on the behavior of the object during rendering operations.

Data types passed to and returned from ANARI are specified using the `ANARIDataType` enum. The enum values identify ANARI object data types and C data types. Types starting with `ANARI` are provided by the ANARI headers. All other types refer to C99 types. See Table 1 below for a complete list.

Values set on objects are passed as a `void *` in `anariSetParameter`, which points to the value to be read. The type is encoded by the passed in `ANARIDataType` enum. For example, this means that, given `ANARI_INT`, the implementation casts the input `void * value` to `int *` and dereferences it accordingly. There are, however, two exceptions: `ANARI_STRING` and `ANARI_VOID_POINTER` (`const char *` and `void *` respectively) are object pointer based types, so they are instead passed by value.

Object parameters can be unset using `anariUnsetParameter`, which returns the named parameter back to a state as if it had not been set. Just like with setting parameters, changes made by `anariUnsetParameter` must only be applied when the object is committed.

The signature to `anariUnsetParameter` is as follows

```
void anariUnsetParameter(  
    ANARIDevice,  
    ANARIObjct,  
    const char *parameterName  
);
```

Similarly, all parameters on an object can be simultaneously unset using `anariUnsetAllParameters`, which returns the object back to a state as if no parameters had been set at all. Just like with unsetting individual parameters, changes made by `anariUnsetAllParameters` must only be applied when the object is committed.

The signature to `anariUnsetAllParameters` is as follows

```
void anariUnsetParameters(ANARIDevice, ANARIObjct);
```

Table 1. Data types.

Name	Closest C99-Type	Description
<code>DATA_TYPE</code>	<code>ANARIDataType = int32_t</code>	describes data types in ANARI

Name	Closest C99-Type	Description
STRING	const char *	0-terminated string
DATA_TYPE_LIST	ANARIDataType*	array of ANARIDataType values terminated by UNKNOWN
STRING_LIST	const char **	array of 0-terminated strings terminated by NULL
PARAMETER_LIST	ANARIParacter*	array of ANARIParacter structs terminated by {NULL, UNKNOWN}
VOID_POINTER	void *	void pointer
BOOL	uint8_t	boolean represented as uint8_t
FUNCTION_POINTER	void(*) (void)	generic function pointer
DELETER_CALLBACK	ANARIDeleterCallback	deleter callback function pointer
STATUS_CALLBACK	ANARIStatusCallback	status callback function pointer
FRAME_COMPLETION_CALLBACK	ANARIFrameCompletionCallback	extension KHR_FRAME_COMPLETION_CALLBACK, frame completion callback function pointer
LIBRARY	ANARILibrary	library object handle
DEVICE	ANARIDevice	device object handle
OBJECT	ANARIObject	generic object handle
ARRAY	ANARIArray	generic array object handle
ARRAY1D	ANARIArray1D	1D array object handle
ARRAY2D	ANARIArray2D	2D array object handle
ARRAY3D	ANARIArray3D	3D array object handle
CAMERA	ANARICamera	camera object handle
FRAME	ANARIFrame	frame object handle
GEOMETRY	ANARIGeometry	geometry object handle
GROUP	ANARIGroup	group object handle
INSTANCE	ANARIInstance	instance object handle
LIGHT	ANARILight	light object handle
MATERIAL	ANARIMaterial	material object handle
RENDERER	ANARIRenderer	renderer object handle
SURFACE	ANARISurface	surface object handle
SAMPLER	ANARISampler	sampler object handle
SPATIAL_FIELD	ANARISpatialField	spatial field object handle
VOLUME	ANARIVolume	volume object handle
WORLD	ANARIWorld	world object handle

Name	Closest C99-Type	Description
INT8	int8_t	8 bit signed integer
INT8_VEC2	int8_t[2]	two element 8 bit signed integer vector
INT8_VEC3	int8_t[3]	three element 8 bit signed integer vector
INT8_VEC4	int8_t[4]	four element 8 bit signed integer vector
UINT8	uint8_t	8 bit unsigned integer
UINT8_VEC2	uint8_t[2]	two element 8 bit unsigned integer vector
UINT8_VEC3	uint8_t[3]	three element 8 bit unsigned integer vector
UINT8_VEC4	uint8_t[4]	four element 8 bit unsigned integer vector
INT16	int16_t	16 bit signed integer
INT16_VEC2	int16_t[2]	two element 16 bit signed integer vector
INT16_VEC3	int16_t[3]	three element 16 bit signed integer vector
INT16_VEC4	int16_t[4]	four element 16 bit signed integer vector
UINT16	uint16_t	16 bit unsigned integer
UINT16_VEC2	uint16_t[2]	two element 16 bit unsigned integer vector
UINT16_VEC3	uint16_t[3]	three element 16 bit unsigned integer vector
UINT16_VEC4	uint16_t[4]	four element 16 bit unsigned integer vector
INT32	int32_t	32 bit signed integer
INT32_VEC2	int32_t[2]	two element 32 bit signed integer vector
INT32_VEC3	int32_t[3]	three element 32 bit signed integer vector
INT32_VEC4	int32_t[4]	four element 32 bit signed integer vector
UINT32	uint32_t	32 bit unsigned integer
UINT32_VEC2	uint32_t[2]	two element 32 bit unsigned integer vector
UINT32_VEC3	uint32_t[3]	three element 32 bit unsigned integer vector

Name	Closest C99-Type	Description
UINT32_VEC4	uint32_t[4]	four element 32 bit unsigned integer vector
INT64	int64_t	64 bit signed integer
INT64_VEC2	int64_t[2]	two element 64 bit signed integer vector
INT64_VEC3	int64_t[3]	three element 64 bit signed integer vector
INT64_VEC4	int64_t[4]	four element 64 bit signed integer vector
UINT64	uint64_t	64 bit unsigned integer
UINT64_VEC2	uint64_t[2]	two element vector 64 bit unsigned integer vector
UINT64_VEC3	uint64_t[3]	three element vector 64 bit unsigned integer vector
UINT64_VEC4	uint64_t[4]	four element 64 bit unsigned integer vector vector
FIXED8	int8_t	8 bit signed normalized fixed point number
FIXED8_VEC2	int8_t[2]	two element 8 bit signed normalized fixed point vector
FIXED8_VEC3	int8_t[3]	three element 8 bit signed normalized fixed point vector
FIXED8_VEC4	int8_t[4]	four element 8 bit signed normalized fixed point vector
UFIXED8	uint8_t	8 bit unsigned normalized fixed point number
UFIXED8_VEC2	uint8_t[2]	two element 8 bit unsigned normalized fixed point vector
UFIXED8_VEC3	uint8_t[3]	three element 8 bit unsigned normalized fixed point vector
UFIXED8_VEC4	uint8_t[4]	four element 8 bit unsigned normalized fixed point vector
FIXED16	int16_t	16 bit signed normalized fixed point number
FIXED16_VEC2	int16_t[2]	two element 16 bit signed normalized fixed point vector
FIXED16_VEC3	int16_t[3]	three element 16 bit signed normalized fixed point vector

Name	Closest C99-Type	Description
<code>FIXED16_VEC4</code>	<code>int16_t[4]</code>	four element 16 bit signed normalized fixed point vector
<code>UFIXED16</code>	<code>uint16_t</code>	16 bit unsigned normalized fixed point number
<code>UFIXED16_VEC2</code>	<code>uint16_t[2]</code>	two element 16 bit unsigned normalized fixed point vector
<code>UFIXED16_VEC3</code>	<code>uint16_t[3]</code>	three element 16 bit unsigned normalized fixed point vector
<code>UFIXED16_VEC4</code>	<code>uint16_t[4]</code>	four element 16 bit unsigned normalized fixed point vector
<code>FIXED32</code>	<code>int32_t</code>	32 bit signed normalized fixed point number
<code>FIXED32_VEC2</code>	<code>int32_t[2]</code>	two element 32 bit signed normalized fixed point vector
<code>FIXED32_VEC3</code>	<code>int32_t[3]</code>	three element 32 bit signed normalized fixed point vector
<code>FIXED32_VEC4</code>	<code>int32_t[4]</code>	four element 32 bit signed normalized fixed point vector
<code>UFIXED32</code>	<code>uint32_t</code>	32 bit unsigned normalized fixed point number
<code>UFIXED32_VEC2</code>	<code>uint32_t[2]</code>	two element 32 bit unsigned normalized fixed point vector
<code>UFIXED32_VEC3</code>	<code>uint32_t[3]</code>	three element 32 bit unsigned normalized fixed point vector
<code>UFIXED32_VEC4</code>	<code>uint32_t[4]</code>	four element 32 bit unsigned normalized fixed point vector
<code>FIXED64</code>	<code>int64_t</code>	64 bit signed normalized fixed point number
<code>FIXED64_VEC2</code>	<code>int64_t[2]</code>	two element 64 bit signed normalized fixed point vector
<code>FIXED64_VEC3</code>	<code>int64_t[3]</code>	three element 64 bit signed normalized fixed point vector
<code>FIXED64_VEC4</code>	<code>int64_t[4]</code>	four element 64 bit signed normalized fixed point vector
<code>UFIXED64</code>	<code>uint64_t</code>	64 bit unsigned normalized fixed point number
<code>UFIXED64_VEC2</code>	<code>uint64_t[2]</code>	two element 64 bit unsigned normalized fixed point vector

Name	Closest C99-Type	Description
UFIXED64_VEC3	uint64_t[3]	three element 64 bit unsigned normalized fixed point vector
UFIXED64_VEC4	uint64_t[4]	four element 64 bit unsigned normalized fixed point vector
FLOAT16	uint16_t	16 bit floating point number
FLOAT16_VEC2	uint16_t[2]	two element 16 bit floating point vector
FLOAT16_VEC3	uint16_t[3]	three element vector 16 bit floating point vector
FLOAT16_VEC4	uint16_t[4]	four element vector 16 bit floating point vector
FLOAT32	float	32 bit floating point number
FLOAT32_VEC2	float[2]	two element 32 bit floating point vector
FLOAT32_VEC3	float[3]	three element vector 32 bit floating point vector
FLOAT32_VEC4	float[4]	four element vector 32 bit floating point vector
FLOAT64	double	64 bit floating point
FLOAT64_VEC2	double[2]	two element vector 64 bit floating point vector
FLOAT64_VEC3	double[3]	three element vector 64 bit floating point vector
FLOAT64_VEC4	double[4]	four element vector 64 bit floating point vector
UFIXED8_RGBA_SRGB	uint8_t[4]	three component sRGB color with linear alpha
UFIXED8_RGB_SRGB	uint8_t[3]	three component sRGB color
UFIXED8_RA_SRGB	uint8_t[2]	one component sRGB with linear alpha
UFIXED8_R_SRGB	uint8_t[1]	single component sRGB
INT32_BOX1	int32_t[2]	one dimensional 32 bit integer box (inclusive lower and upper bounds)
INT32_BOX2	int32_t[4]	two dimensional 32 bit integer box (inclusive lower and upper bound vector)
INT32_BOX3	int32_t[6]	three dimensional 32 bit integer box (inclusive lower and upper bound vector)

Name	Closest C99-Type	Description
INT32_BOX4	int32_t[8]	four dimensional 32 bit integer box (inclusive lower and upper bound vector)
UINT64_REGION1	uint64_t[2]	one dimensional 64 bit unsigned integer region (inclusive lower and exclusive upper bounds)
UINT64_REGION2	uint64_t[4]	two dimensional 64 bit unsigned integer region (inclusive lower and exclusive upper bound vector)
UINT64_REGION3	uint64_t[6]	three dimensional 64 bit unsigned integer region (inclusive lower and exclusive upper bound vector)
UINT64_REGION4	uint64_t[8]	four dimensional 64 bit unsigned integer region (inclusive lower and exclusive upper bound vector)
FLOAT32_BOX1	float[2]	one dimensional 32 bit float box (inclusive lower and upper bounds)
FLOAT32_BOX2	float[4]	two dimensional 32 bit float box (inclusive lower and upper bound vector)
FLOAT32_BOX3	float[6]	three dimensional 32 bit float box (inclusive lower and upper bound vector)
FLOAT32_BOX4	float[8]	four dimensional 32 bit float box (inclusive lower and upper bound vector)
FLOAT64_BOX1	double[2]	one dimensional 64 bit float box (inclusive lower and upper bounds)
FLOAT64_BOX2	double[4]	two dimensional 64 bit float box (inclusive lower and upper bound vector)
FLOAT64_BOX3	double[6]	three dimensional 64 bit float box (inclusive lower and upper bound vector)
FLOAT64_BOX4	double[8]	four dimensional 64 bit float box (inclusive lower and upper bound vector)
FLOAT32_MAT2	float[4]	two by two 32 bit float matrix in column-major order

Name	Closest C99-Type	Description
FLOAT32_MAT3	float[9]	three by three 32 bit float matrix in column-major order
FLOAT32_MAT4	float[16]	four by four 32 bit float matrix in column-major order
FLOAT32_QUAT_IJKW	float[4]	quaternion

Floating point number (data types with `FLOAT`) layout and behaviour are as specified in [\[ieee-754\]](#).

3.6.1. Color

The following data types must be accepted by geometry's `color` attribute parameter and by [samplers](#), which are grouped here for brevity:

- `UFIXED8`
- `UFIXED8_VEC2`
- `UFIXED8_VEC3`
- `UFIXED8_VEC4`
- `UFIXED16`
- `UFIXED16_VEC2`
- `UFIXED16_VEC3`
- `UFIXED16_VEC4`
- `UFIXED32`
- `UFIXED32_VEC2`
- `UFIXED32_VEC3`
- `UFIXED32_VEC4`
- `FLOAT32`
- `FLOAT32_VEC2`
- `FLOAT32_VEC3`
- `FLOAT32_VEC4`
- `UFIXED8_RGBA_SRGB`
- `UFIXED8_RGB_SRGB`
- `UFIXED8_RA_SRGB`
- `UFIXED8_R_SRGB`

3.7. Object Introspection

ANARI supports introspection of objects, i.e., querying supported extensions, object subtypes and information about their parameters, which is particularly useful to enumerate and inspect

implementation-specific object extensions. The intended use is to allow for building a graphical user interface for, e.g., specific renderers or materials, and to provide hints for the debug layer to aid in validating extensions.

The information queried with the following functions is static and will reflect what the implementation is capable of supporting. If an extension is unavailable for runtime dependent reasons, for example due to missing hardware support, the device may still advertise the associated parameters and object subtypes. The dynamic runtime extension availability can be queried using [properties](#).

Device subtypes and their implemented extensions can be queried from a library before instantiating a device. Object and parameter specific information can be queried from an instantiated device object.

```
const char **anariGetDeviceSubtypes(ANARILibrary);
```

A list of device subtypes implemented in a library is retrieved by calling `anariGetDeviceSubtypes`. It returns `NULL` if there are no devices, or a `NULL`-terminated list of `0`-terminated C-strings with the names of the devices. The first (if any) device is the default device.

```
const char **anariGetDeviceExtensions(ANARILibrary, const char *deviceSubtype);
```

The list extensions implemented by a device is retrieved by calling `anariGetDeviceExtensions` with a device subtype returned by `anariGetDeviceSubtypes`. It returns a `NULL`-terminated list of `0`-terminated C-strings with the names of the implemented extensions.

```
const char **anariGetObjectSubtypes(ANARIDevice, ANARIDataType objectType);
```

To enumerate the subtypes of type `objectType` supported by device `deviceSubtype` call function `anariGetObjectSubtypes`. It returns `NULL` if there are no subtypes, or a `NULL`-terminated list of `0`-terminated C-strings with the names of the subtypes. The following object types are expected to have subtypes:

- `CAMERA`
- `RENDERER`
- `INSTANCE`
- `LIGHT`
- `GEOMETRY`
- `SAMPLER`
- `MATERIAL`
- `VOLUME`
- `SPATIAL_FIELD`

```
typedef struct
{
    const char *name;
    ANARIDataType type;
} ANARIPParameter;

const void *anariGetObjectInfo(ANARIDevice,
    ANARIDataType objectType,
    const char *objectSubtype
    const char *infoName,
    ANARIDataType infoType);
```

An object (sub)type `objectType/objectSubtype` can be queried for information with `anariGetObjectInfo` (passing `NULL` or an empty string for `objectSubtype` to query an object type directly that does not have subtypes). The function returns the result for `infoName` of type `infoType`, or `NULL` if the info cannot be retrieved. The following infos of objects can be queried:

Table 2. Info of objects for introspection.

Name	Type	Required	Description
description	STRING	No	explanation of the object, e.g., for a tooltip
parameter	PARAMETER_LIST	Yes	list of supported parameters (array of <code>ANARIPParameter</code>)
channel	STRING_LIST	for FRAME	list of supported channels
extension	STRING_LIST	for RENDERERS	list of supported extensions

Parameters that can be of multiple types are reported multiple times (once for each supported variant: with the same `name`, but different `type`).

```
const void *anariGetParameterInfo(ANARIDevice,
    ANARIDataType objectType,
    const char *objectSubtype,
    const char *parameterName,
    ANARIDataType parameterType,
    const char *infoName,
    ANARIDataType infoType);
```

Finally, a parameter can be inspected with `anariGetParameterInfo`, returning the result for `infoName` of type `infoType`, or `NULL` if the info cannot be retrieved. The following infos of parameters can be queried:

Table 3. Info of parameters for introspection.

Name	Type	Required	Description
description	STRING	No	explanation of the parameter, e.g., for a tooltip
minimum	type	No	set values will be clamped to this minimum
maximum	type	No	set values will be clamped to this maximum
default	type	No	default value, must be in <code>minimum..maximum</code> if present
elementType	DATA_TYPE_LIST	for <code>ARRAYs</code>	array of supported element types of the <code>ANARIArray</code> parameter, with last element <code>UNKNOWN</code>
required	BOOL	Yes	whether the parameter must be set for an object to be valid, must be <code>FALSE</code> if a <code>default</code> is present
value	STRING_LIST or DATA_TYPE_LIST	No	list of accepted strings or data types for parameters that only recognize specific values

3.8. Object Properties

Implementations may expose object properties through the object query interface.

```
int anariGetProperty(ANARIDevice,
    ANARIObject,
    const char *propertyName,
    ANARIDataType propertyType,
    void *memory,
    uint64_t size,
    uint32_t waitMask);
```

Properties are identified by a string name and type. The `waitMask` indicates whether the property query will wait for the value to become available or should return instantly.

If the property is available, the value will be written to `memory`, and the function returns `1`. Otherwise, the value is not written to `memory`, and the return value is `0`.

The length of a string property can be queried as an `ANARI_ULONG` by appending `.size` to the property name.

At most `size` bytes will be written to `memory`.



Note

Return value does not replace error handling via the callbacks. A property may be unavailable for various reasons including not being supported by the device, the value not being computed yet, or any other device-specific reason. Errors related to property queries are still returned via the error callback.

3.9. Arrays and Memory Ownership

There are two methods of expressing array data on objects: directly mapping array elements on an object parameter in order to write data elements, or creating array objects represented by a handle.

The first method lets applications directly map a device's internally-managed one, two, or three dimensional array data on any given object using

```
void *anariMapParameterArray1D(ANARIDevice,
    ANARIObject,
    const char *parameterName
    ANARIDataType elementType,
    uint64_t numElements,
    uint64_t *elementStride);

void *anariMapParameterArray2D(ANARIDevice,
    ANARIObject,
    const char *parameterName
    ANARIDataType elementType,
    uint64_t numElements1,
    uint64_t numElements2,
    uint64_t *elementStride);

void *anariMapParameterArray3D(ANARIDevice,
    ANARIObject,
    const char *parameterName
    ANARIDataType elementType,
    uint64_t numElements1,
    uint64_t numElements2,
    uint64_t numElements3,
    uint64_t *elementStride);
```

Each of these functions return a write-only array for the application to fill, where the number of elements `numElementsN` must be positive (there cannot be an empty mapped array). Whenever an array is directly mapped, any previous array configuration (size, dimensionality, etc.) is discarded in favor of the currently mapped array configuration.

Devices are only required to allocate directly mapped arrays for parameters corresponding to extensions that the device implements. In cases where the parameter is unknown by the device, the device is permitted to return `NULL`.

Devices are permitted to have a non-dense element stride, which is returned by writing to `elementStride`. The `elementStride` argument must not be `NULL`.

Once the array has been filled, applications signal that the device is free to consume the data with


```
void anariUnmapParameterArray(ANARIDevice, ANARIObjct, const char *parameterName);
```

Directly mapped arrays behave like normal parameters in that the underlying array data will not be used in the [next rendered frame](#) until the object itself is committed with [anariCommitParameters](#). Furthermore, using [anariUnsetParameter](#) will subsequently reset the parameter to an unset state, effectively clearing the previously mapped array on the object. Using [anariSetParameter](#), [anariMapParameterArray](#), or [anariUnsetParameter](#) on an array that is still mapped is an error – applications must first unmap the array before altering the parameter through other means. Writing to the mapped pointer after calling [anariUnmapParameterArray](#) is undefined behavior and should be avoided.

The second method of expressing array data on objects uses handles. [ANARIArray](#) handles represent data arrays in memory, which are shared with device implementations. The memory shared with ANARI can be either owned by the application, have ownership transferred to the device via a deleter callback, or be managed by the device.

The signature of the deleter is provided in [anari.h](#) as

```
typedef void (*ANARIDeleterCallback)(const void *userPtr, const void *appMemory);
```

To create a data array, use any of the following API calls based on the desired dimension:

```
ANARIArray1D anariNewArray1D(ANARIDevice,  
    const void *appMemory,  
    ANARIDeleterCallback,  
    const void *userPtr,  
    ANARIDataType elementType,  
    uint64_t numElements);
```

```
ANARIArray2D anariNewArray2D(ANARIDevice,  
    const void *appMemory,  
    ANARIDeleterCallback,  
    const void *userPtr,  
    ANARIDataType elementType,  
    uint64_t numElements1,  
    uint64_t numElements2);
```

```
ANARIArray3D anariNewArray3D(ANARIDevice,  
    const void *appMemory,  
    ANARIDeleterCallback,  
    const void *userPtr,  
    ANARIDataType elementType,  
    uint64_t numElements1,  
    uint64_t numElements2,  
    uint64_t numElements3);
```

The number of elements [numElementsN](#) must be positive (there cannot be an empty array object).

In each creation function a deleter can be passed in (with an associated pointer to any needed application data or state), which ANARI will use to free the original pointer passed during construction. If the application passes `NULL`, ANARI will fully rely on the application to free the memory.

When `appMemory` is not `NULL`, then the array is considered to be a *shared* array, where both the application and device observe the same memory. Passing `NULL` in `appMemory` creates a *managed* array. The backing memory of the array is managed by the device and is only writable via mapping.

Applications are permitted to release `ANARIArray` objects even if the device still contains internal references to it. When releasing a shared array object ANARI relinquishes shared ownership of the memory, which may result in the creation of internal copies. When a shared array is created with a deleter callback, it is implementation defined when an implementation frees host memory after the array object has been released. Deleter callbacks are never called for managed arrays.

Applications are only permitted to write to the memory visible to `ANARIArray` objects if all array objects involved are mapped. Mapping a shared array object indicates that the device should not execute any rendering operations (or internal state updates, such as building acceleration structures) of any parent objects to the mapped array object which is directly referencing mapped memory.

Array objects are mapped using

```
void *anariMapArray(ANARIDevice, ANARIArray);
```

Mapped array objects are unmapped using

```
void anariUnmapArray(ANARIDevice, ANARIArray);
```

Mapping a shared array will always result in the same address originally used when constructing the array object. Mapping a managed array may return a different pointer each time it is mapped. The contents of memory mapped from managed arrays is undefined until written to and the entire mapped range must be specified before unmapping to avoid populating the array with undefined values.

`ANARIArray` objects containing object handles increase the ref count of all objects in the array. Reference counts are updated on creation of the array object and when unmapping the array.

Use of the arrays inside ANARI may cause accesses at indices derived from parameters or other arrays (for example when drawing an indexed geometry). Out of bounds accesses caused this way have undefined behavior.

Note



Array parameters of objects that are accessed by the same index (because they represent an "array of structures") have per naming convention a common prefix followed by a period. An example are the vertex attributes of the `triangle geometry`, which all start with `vertex.` followed by the attribute name.

Arrays define a valid region of elements, which by default is always the full capacity of the array. Implementations can allow applications to keep a singular allocation of data, but use a parameter to tell the implementation only to use a subset of elements. This gives applications the ability to more efficiently change the number of elements used by parent objects at the cost of memory size efficiency.

Implementations provide this functionality by implementing extension `KHR_ARRAY1D_REGION`, which adds the following parameter to `ANARIArray1D`:

Table 4. *Parameters understood by the 1D array.*

Name	Type	Description
region	<code>UINT64_REGION1</code>	extension <code>KHR_ARRAY1D_REGION</code> , region of elements currently in use

If `region` is not set, all elements are used as specified when the array was constructed. When `region` is set, it is clamped to the range of the array's constructed size respectively and warnings should be emitted by the debug layer if `region.upper` is larger than capacity.

The values specified by `region` are clamped such that the array will always be a valid range containing at least one element. Thus `region` is clamped according to the following:

- `region.upper` is clamped from below by `region.lower + 1`
- `region.upper` is clamped from above by the array's capacity
- `region.lower` is clamped from above by `region.upper - 1`

Mapping an array always returns the full capacity, regardless of the elements specified by `region`. Object handles within `region` must always be valid. Object arrays can be constructed with invalid handles, as long as the array `region` contains only valid handles by the time the array is used in a render operation.



Note

As with all `REGION` types, the element from `region.lower` is *inclusive*, while `region.upper` is *exclusive*. This is intended to mimic C++ `begin/end` iterators on containers.



Note

Directly mapped arrays from object parameters maximizes the device's opportunity to be efficient in its underlying implementation. This is preferred method applications should use to set array data on objects. However, if the array data needs to be set on more than one object, then applications should prefer using array objects also detailed in this section so that a single array handle can be set on more than one parent object.

3.10. Libraries

ANARI libraries are the mechanism that applications use to manage API device implementations. Libraries are solely responsible for creating instances of `devices`. Implementors may use a library to

cache data or objects which are truly global to their device implementations, such as contexts from underlying APIs. Libraries are generally the first thing loaded by an application and the last thing cleaned up. While libraries are represented by an opaque handle, they are not considered an object per the given [definition](#) of an object and are thus only usable in API calls which explicitly take `ANARILibrary` handles.

To load a library, use

```
ANARILibrary anariLoadLibrary(const char *name,  
    ANARIStatusCallback defaultStatusCallback,  
    const void *defaultStatusCallbackUserData);
```

This will look for a shared library named `anari_library_[name]`, open it, and look for entry points for (implementation defined) initialization. The status callback passed is used as the default value for the `statusCallback` parameter on devices created from the returned library object. Similarly, the user pointer passed is used as the default value for the `statusCallbackUserData` device parameter.

To unload a library (where library resource cleanup occurs), use

```
void anariUnloadLibrary(ANARILibrary);
```

It is undefined behavior to unload a library while instances of devices from that library have not been released.



Note

Vendors are also permitted to implement direct device creation functions to allow an application to directly link their ANARI library at compile time. Please reference your vendor's documentation for whether direct linking is supported by their ANARI library.

3.11. Devices

ANARI coordinates the use of one or more *devices*. A device is an object which provides the implementation of all ANARI API calls outside of libraries. Devices represent the global state which an implementor may reuse between different objects to render images. It is common for applications to only use one device at a time, but the API permits concurrent use of multiple devices to independently render from each other.



Note

ANARI devices are a *software* construct. Because ANARI abstracts away the details of an entire rendering system, the underlying hardware which a device may use is entirely up to the implementation. Please read your vendor's device documentation to see what parameters are available to configure and what underlying hardware is both available and used to render frames.

The ANARI device is responsible for coordinating the sharing of execution resources with the calling application, such as a CPU thread pool or GPU kernel queues.

ANARI devices are represented by an **ANARIDevice** handle and created using

```
ANARIDevice anariNewDevice(ANARILibrary, const char *subtype);
```

Version information can be queried as properties on the **ANARIDevice** using **anariGetProperty**.

Table 5. *Properties* queryable on a device.

Name	Type	Required	Description
version	INT32	Yes	unique device version number guaranteed to increase between versions
version.major	INT32	No	semantic device version major value (major.minor.patch)
version.minor	INT32	No	semantic device version minor value (major.minor.patch)
version.patch	INT32	No	semantic device version patch value (major.minor.patch)
version.name	STRING	No	human readable device name/title
anariVersion.major	INT32	Yes	targeted ANARI specification version major value (major.minor)
anariVersion.minor	INT32	Yes	targeted ANARI specification version minor value (major.minor)
geometryMaxIndex	UINT64	Yes	largest supported index into vertex arrays in geometries
extension	STRING_LIST	Yes	list of supported extensions

3.12. Error Handling

Errors and other messages (warnings, validation messages, debug information etc.) from the device are reported via a status callback. The status callback function can be set using

```
typedef void (*ANARIStatusCallback)(const void *userPtr,  
    ANARIDevice,  
    ANARIObject source,  
    ANARIDataType sourceType,  
    ANARIStatusSeverity,  
    ANARIStatusCode,  
    const char *message);
```

The following values for **ANARIStatusCode** are defined:

- STATUS_NO_ERROR
- STATUS_UNKNOWN_ERROR
- STATUS_INVALID_ARGUMENT
- STATUS_INVALID_OPERATION
- STATUS_OUT_OF_MEMORY
- STATUS_UNSUPPORTED_DEVICE
- STATUS_VERSION_MISMATCH

The following values for `ANARISStatusSeverity` are defined:

- SEVERITY_FATAL_ERROR
- SEVERITY_ERROR
- SEVERITY_WARNING
- SEVERITY_PERFORMANCE_WARNING
- SEVERITY_INFO
- SEVERITY_DEBUG

Table 6. *Parameters* understood by all devices.

Name	Type	Description
statusCallback	STATUS_CALLBACK	callback used to report information to the application
statusCallbackUserData	VOID_POINTER	optional pointer passed as the first argument of the status callback

Statuses may be reported at an undefined time after the API call causing them is made. Furthermore, these callbacks must themselves be thread safe as they can be called on any thread.

3.13. Attributes

Attributes are quantities that are passed between objects during rendering. Attributes are identified by strings.

Surface attributes are passed from [geometries](#) and [instances](#) to [materials](#) and [samplers](#). Attributes are either set explicitly by user-provided data as array parameters (and may be interpolated in a geometry-specific way) or implicitly by the surface. Unspecified (components of) attributes default to zero for the first three components and to one for the fourth component.

Table 7. *Attributes*

Identifier	Internal Type	Description
color	FLOAT32_VEC4	color
worldPosition	FLOAT32_VEC4	world space position

Identifier	Internal Type	Description
worldNormal	FLOAT32_VEC4	world space shading normal
objectPosition	FLOAT32_VEC4	object space position
objectNormal	FLOAT32_VEC4	object space shading normal
attribute0	FLOAT32_VEC4	generic attribute 0
attribute1	FLOAT32_VEC4	generic attribute 1
attribute2	FLOAT32_VEC4	generic attribute 2
attribute3	FLOAT32_VEC4	generic attribute 3
primitiveId	UINT32 / UINT64	geometry specific primitive identifier, at most device limit geometryMaxIndex large

Chapter 4. Rendering Frames

Rendering is asynchronous (non-blocking), and is done by combining a framebuffer, renderer, camera, and world. The process of rendering a frame is known as a *frame operation*. Frame operations are invoked with

```
void anariRenderFrame(ANARIDevice, ANARIFrame);
```

This call may not block, and the ANARIFrame itself can be used to synchronize with the application, cancel, or query for progress of the running task. When `anariRenderFrame` is called, there is no guarantee when the associated task will begin execution.

Applications can query for the status of or wait on a running frame with

```
int anariFrameReady(ANARIDevice, ANARIFrame, ANARIWaitMask);
```

If `ANARI_NO_WAIT` is passed as the wait mask, then the function returns true if the frame has completed. Alternatively, passing `ANARI_WAIT` will block the calling thread until the frame has completed and will always return true.

Applications can query how long an async task ran with the `duration` property on the `ANARIFrame`. If available, this returns the wall clock execution time of the task in seconds. This is useful for applications to query exactly how long an asynchronous task executed without the overhead of measuring both task execution & synchronization by the calling application.



Note

The use of `anariRenderFrame` requires that all objects in the scene being rendered are valid before rendering occurs.

Applications can signal that an in-flight frame should be cancelled if possible using

```
void anariDiscardFrame(ANARIDevice, ANARIFrame);
```

This call is not required to block until the frame completes, rather it only signals to the implementation to attempt cancelling the currently rendered frame instead of going all the way to completion. The contents of a mapped frame which has been discarded is undefined.

The application can map the given channel of a frame – and thus access the stored pixel information – via

```
const void *anariMapFrame(ANARIDevice,  
                           ANARIFrame,  
                           const char *channel,  
                           uint32_t *width,  
                           uint32_t *height,
```



```
ANARIDataType *pixelType);
```

Only channels that have been set as parameters to the frame can be mapped, the type of the pixels matches the corresponding parameter value. The arguments `width`, `height`, and `pixelType` are output parameters for the application to validate the exact dimensions and per-pixel data type in the mapped image.



Note

ANARI makes a clear distinction between the *external* format of channels of the frame and the internal one: The external format is the format the user specifies as `DATA_TYPE` parameter for channels, which corresponds to the element type of the returned buffer when calling `anariMapFrame`. Implementations may do significant amounts of reformatting, compression/decompression, ..., in-between the generation of the *internal* frame and the mapping of the externally visible one.

The origin of the screen coordinate system in ANARI is the lower left corner (as in OpenGL), thus the first pixel addressed by the returned pointer is the lower left pixel of the image.

A previously mapped channel of a frame can be unmapped calling

```
void anariUnmapFrame(ANARIDevice, ANARIFrame, const char *channel);
```

Chapter 5. Object Types and Subtypes

This section describes the object types (and subtypes where available) which are used to compose a scene in ANARI and which are involved in rendering it.

5.1. Frame

The frame contains all the objects necessary to render and holds the resulting rendered 2D image (and optionally auxiliary information associated with pixels). To create a new frame, use

```
ANARIFrame anariNewFrame(ANARIDevice);
```

The frame uses parameters to encode size, color format, and which channels to use. Channels are identified by string channel names beginning with `channel.`. The frame object has an identically named parameter for each reported channel. Setting these parameters to one of the allowed data types enables the channel and determines the data type it can be mapped as. The same channel names are used to map the channel contents with `anariMapFrame`. Each channel has its own associated extension name.

Table 8. *Parameters understood by the frame.*

Name	Type	Description
world	WORLD	required world to be rendered
camera	CAMERA	required camera used to render the world
renderer	RENDERER	required renderer which renders the frame
size	UINT32_VEC2	required size of the frame in pixels (width × height)
accumulation	BOOL	extension KHR_FRAME_ACCUMULATION, whether additional internal buffers are created to potentially improve the image quality when multiple subsequent calls to <code>anariRenderFrame</code> are made, default FALSE
frameCompletionCallback	FRAME_COMPLETION_CALLBACK	extension KHR_FRAME_COMPLETION_CALLBACK, callback to invoke as continuation when the rendered frame is complete
frameCompletionCallbackUserData	VOID_POINTER	extension KHR_FRAME_COMPLETION_CALLBACK, optional user pointer passed as the first argument of the frame completion callback

The `world`, `camera`, `renderer`, and `size` parameters are required, `size` must be positive.

The extension `KHR_FRAME_ACCUMULATION` indicates that implementations use progressive rendering techniques like Monte Carlo integration. If enabled via parameter `accumulation`, multiple subsequent calls to `anariRenderFrame` without changes to `Frame` improve the image quality (e.g., reduce noise levels).

Devices which implement extension `KHR_FRAME_COMPLETION_CALLBACK` add two parameters to `Frame`: a callback invoked as a continuation after the frame completes, and an associated pointer to application state to be passed to the invoked continuation. This continuation must be complete before returning from `anariFrameReady()` when called with `ANARI_WAIT`.

The signature of the continuation is provided in `anari.h` as

```
typedef void (*ANARIFrameCompletionCallback)(const void *userPtr, ANARIDevice,
ANARIFrame);
```



Note

Implementations are strongly encouraged to invoke the continuation on a background thread. This helps physically maintain asynchronous behavior with calling application API threads. ANARI API calls are legal within the continuation, except for calling `anariFrameReady()` with `ANARI_WAIT`, as this will incur a deadlock.

Table 9. Frame channel enabling parameters and their associated extension names.

Name	Type	Description
channel.color	DATA_TYPE	enable mapping the <code>color</code> channel and specify its observable type; RGB color including alpha; possible values: <code>UFIXED8_VEC4</code> , <code>UFIXED8_RGBA_SRGB</code> , <code>FLOAT32_VEC4</code>
channel.depth	DATA_TYPE	enable mapping the <code>depth</code> channel and specify its observable type; euclidean distance to the camera (<i>not</i> to the image plane), for multiple samples per pixel their minimum is taken; possible values: <code>FLOAT32</code>
channel.normal	DATA_TYPE	extension <code>KHR_FRAME_CHANNEL_NORMAL</code> , enable mapping the <code>normal</code> channel and specify its observable type; average world-space normal of the first hit; possible values: <code>FIXED16_VEC3</code> , <code>FLOAT32_VEC3</code>
channel.albedo	DATA_TYPE	extension <code>KHR_FRAME_CHANNEL_ALBEDO</code> , enable mapping the <code>albedo</code> channel and specify its observable type; average material albedo (color without illumination) at the first hit; possible values: <code>UFIXED8_VEC3</code> , <code>UFIXED8_RGB_SRGB</code> , <code>FLOAT32_VEC3</code>
channel.primitiveId	DATA_TYPE	extension <code>KHR_FRAME_CHANNEL_PRIMITIVE_ID</code> , enable mapping the <code>primitiveId</code> channel and specify its observable type; <code>primitiveId</code> attribute of the first hit; possible values: <code>UINT32</code>
channel.objectId	DATA_TYPE	extension <code>KHR_FRAME_CHANNEL_OBJECT_ID</code> , enable mapping the <code>objectId</code> channel and specify its observable type; user defined <code>Surface</code> / <code>Volume id</code> , possible values: <code>UINT32</code>
channel.instanceId	DATA_TYPE	extension <code>KHR_FRAME_CHANNEL_INSTANCE_ID</code> , enable mapping the <code>instanceId</code> channel and specify its observable type; user defined <code>Instance id</code> , possible values: <code>UINT32</code>

The following information can be queried as properties on the `ANARIFrame` using `anariGetProperty`.

Table 10. *Properties* queryable on a frame.

Name	Type	Required	Description
duration	FLOAT32	Yes	time (in seconds) between start and completion of the frame
renderProgress	FLOAT32	No	progress of the current frame task since the last call to <code>anar iRenderFrame</code> , in [0..1]
refinementProgress	FLOAT32	No	extension <code>KHR_FRAME_ACCUMULATION</code> , progress of frame refinement, in [0..1]

5.2. Camera

Cameras express viewpoint and viewport projection information for rendering a scene. To create a new camera of given type `subtype` use

```
ANARICamera anar iNewCamera(ANARIDevice, const char *subtype);
```

All cameras accept the following parameters:

Table 11. *Parameters understood by cameras.*

Name	Type	Default	Description
position	FLOAT32_VEC3	(0, 0, 0)	position of the camera in world-space
direction	FLOAT32_VEC3	(0, 0, -1)	main viewing direction of the camera
up	FLOAT32_VEC3	(0, 1, 0)	up direction of the camera
imageRegion	FLOAT32_BOX2	((0, 0), (1, 1))	region of the sensor in normalized screen-space coordinates
apertureRadius	FLOAT32	0	extension <code>KHR_CAMERA_DEPTH_OF_FIELD</code> , size of the aperture, controls the depth of field
focusDistance	FLOAT32	1	extension <code>KHR_CAMERA_DEPTH_OF_FIELD</code> , distance at where the image is sharpest when depth of field is enabled
stereoMode	STRING	none	extension <code>KHR_CAMERA_STEREO</code> , possible values: <code>none</code> , <code>left</code> , <code>right</code> , <code>sideBySide</code> , <code>topBottom</code> (left eye at top half)
interpupillaryDistance	FLOAT32	0.0635	extension <code>KHR_CAMERA_STEREO</code> , distance between left and right eye when stereo is enabled
shutter	FLOAT32_BOX1	[0.5, 0.5]	extension <code>KHR_CAMERA_SHUTTER</code> , start and end of shutter time, clamped to [0, 1]
rollingShutterDirection	STRING	none	extension <code>KHR_CAMERA_ROLLING_SHUTTER</code> , rolling direction of the shutter, possible values: <code>none</code> , <code>left</code> , <code>right</code> , <code>down</code> , <code>up</code>
rollingShutterDuration	FLOAT32	0	extension <code>KHR_CAMERA_ROLLING_SHUTTER</code> , the “open” time per line, clamped to [0, shutter.upper-shutter.lower]

The camera is placed and oriented in the world with `position`, `direction` and `up`. ANARI uses a right-handed coordinate system.

The region of the camera sensor that is rendered to the image can be specified in normalized screen-space coordinates with `imageRegion`. This can be used, for example, to crop the image, to achieve asymmetrical view frusta, or to horizontally flip the image to view scenes which are specified in a left-handed coordinate system. Note that values outside the default range of [0–1] are valid, which is useful to easily realize overscan or film gate, or to emulate a shifted sensor.

The extension `KHR_CAMERA_DEPTH_OF_FIELD` indicates that cameras support depth of field via the `apertureRadius` and `focusDistance` parameters.

The extension `KHR_CAMERA_STEREO` indicates that a renderer permits 3D stereo rendering, which is enabled by setting `stereoMode` and `interpupillaryDistance`.

Table 12. Additional parameters understood by all cameras with extension `KHR_CAMERA_MOTION_TRANSFORMATION`.

Name	Type	Default	Description
<code>motion.transform</code>	ARRAY1D of <code>FLOAT32_MAT4</code>		uniformly distributed world-space transformations
<code>motion.scale</code>	ARRAY1D of <code>FLOAT32_VEC3</code>		uniformly distributed scale, overridden by <code>motion.transform</code>
<code>motion.rotation</code>	ARRAY1D of <code>FLOAT32_QUAT_IJKW</code>		uniformly distributed quaternion rotation, overridden by <code>motion.transform</code>
<code>motion.translation</code>	ARRAY1D of <code>FLOAT32_VEC3</code>		uniformly distributed transformation, overridden by <code>motion.transform</code>
<code>time</code>	<code>FLOAT32_BOX1</code>	[0, 1]	time associated with first and last key in <code>motion.*</code> arrays

Extension `KHR_CAMERA_MOTION_TRANSFORMATION`: Uniformly (in `time`) distributed transformation keys can be set with `motion.transform` to achieve camera motion blur (in combination with extension `KHR_CAMERA_SHUTTER`). Alternatively, the transformation keys can also be given as decomposed `motion.scale`, `motion.rotation` and `motion.translation`.

The extension `KHR_CAMERA_ROLLING_SHUTTER` provides more control over the type of shutter implemented. It depends on extension `KHR_CAMERA_SHUTTER`.

5.2.1. Perspective

Extension `KHR_CAMERA_PERSPECTIVE`

The perspective camera represents a simple thin lens camera for perspective rendering. It is created by passing the type string `perspective` to `anariNewCamera`. In addition to the [general parameters](#) understood by all cameras the perspective camera supports the special parameters listed in the table below.

Table 13. Additional [parameters](#) understood by the perspective [camera](#).

Name	Type	Default	Description
<code>fovy</code>	<code>FLOAT32</code>	$\pi/3$	the field of view (angle in radians) of the frame's height
<code>aspect</code>	<code>FLOAT32</code>	1	ratio of width by height of the frame (and image region)
<code>near</code>	<code>FLOAT32</code>		near clip plane distance
<code>far</code>	<code>FLOAT32</code>		far clip plane distance

Note that when computing the `aspect` ratio a potentially set image region (using `imageRegion`) needs to be regarded as well.

If `near` and/or `far` are explicitly set they need to fulfill $0 < \text{near} < \text{far}$. If they are not set they are determined by the renderer.



Note

Some rasterization based rendering algorithms intrinsically require near and far planes. The choice of these may also affect depth precision. By default ANARI devices using such algorithms will try to determine appropriate values from the scene contents and internal heuristics. When the algorithm does not require clip planes and `near/far` are not set it may opt to perform no clipping (effectively setting them to 0 and infinity).

5.2.2. Omnidirectional

Extension `KHR_CAMERA_OMNIDIRECTIONAL`

The omnidirectional camera captures the complete surrounding. It is created by passing the type string `omnidirectional` to `anariNewCamera`. It is placed and oriented in the scene by using the [general parameters](#) understood by all cameras.

Table 14. Additional [parameters](#) understood by the omnidirectional camera.

Name	Type	Default	Description
layout	<code>STRING</code>	<code>equirectangular</code>	pixel layout, possible values: <code>equirectangular</code>

The image content outside of [0–1] of `imageRegion` is undefined for the omnidirectional camera.

5.2.3. Orthographic

Extension `KHR_CAMERA_ORTHOGRAPHIC`

The orthographic camera represents a simple camera with orthographic projection. It is created by passing the type string `orthographic` to `anariNewCamera`. In addition to the [general parameters](#) understood by all cameras the orthographic camera supports the following special parameters:

Table 15. Additional [parameters](#) understood by the orthographic camera.

Name	Type	Default	Description
aspect	<code>FLOAT32</code>	1	ratio of width by height of the frame (and image region)
height	<code>FLOAT32</code>	1	height of the image plane in world units
near	<code>FLOAT32</code>		near clip plane distance
far	<code>FLOAT32</code>		far clip plane distance

Note that when computing the `aspect` ratio a potentially set image region (using `imageRegion`) needs to be regarded as well.

If `near` and/or `far` are explicitly set they need to fulfill $0 \leq \text{near} < \text{far}$. If they are not set they are determined by the renderer.



Note

Some rasterization based rendering algorithms intrinsically require near and far planes. The choice of these may also affect depth precision. By default ANARI

devices using such algorithms will try to determine appropriate values from the scene contents and internal heuristics. When the algorithm does not require clip planes and `near/far` are not set it may opt to perform no clipping (effectively setting them to 0 and infinity).

5.3. Renderer

A renderer is the central object for rendering in ANARI. Different renderers implement different extensions, rendering algorithms, and support different materials. To create a new renderer of given subtype `subtype` use

```
ANARIRenderer anariNewRenderer(ANARIDevice, const char *subtype);
```

Every ANARI device offers a default renderer, which works without setting any parameters (i.e., all parameters, if any, have meaningful defaults). Thus, passing `default` as subtype string to `anariNewRenderer` will result in a usable renderer. Further renderers and their parameters can be enumerated with the [Object Introspection](#) API. The default renderer is an alias to an existing renderer that is returned first in the list by `anariGetObjectSubtypes` when queried for `RENDERER`. Also refer to the documentation of ANARI implementations.

Extension support information can be queried as properties on the `ANARIRenderer` using `anariGetProperty`.

Table 16. [Properties](#) queryable on a renderer.

Name	Type	Required	Description
extension	<code>STRING_LIST</code>	Yes	list of supported extensions

Even though renderers (and their parameters) are highly implementation specific, some typical parameters are specified by the following extensions.

Table 17. *Parameters understood by some renderers.*

Name	Type	Default	Description
background	FLOAT32_VEC4	(0, 0, 0, 1)	extension <code>KHR_RENDERER_BACKGROUND_COLOR</code> , the background color
background	ARRAY2D of Color		extension <code>KHR_RENDERER_BACKGROUND_IMAGE</code> , the background image, rescaled to the size of the <code>Frame</code> by linear filtering
ambientColor	FLOAT32_VEC3	(1, 1, 1)	extension <code>KHR_RENDERER_AMBIENT_LIGHT</code> , ambient light color
ambientRadiance	FLOAT32	0	extension <code>KHR_RENDERER_AMBIENT_LIGHT</code> , the amount of light emitted by a point on the ambient light source in a direction, in W/sr/m ²
denoise	BOOL	FALSE	extension <code>KHR_RENDERER_DENOISE</code> , whether the rendered image should be denoised

If both `KHR_RENDERER_BACKGROUND_COLOR` and `KHR_RENDERER_BACKGROUND_IMAGE` are supported the `background` parameter can accept either type.

The ambient light is a light with an invisible source which surrounds the scene and illuminates it from infinity.

5.4. World

Worlds are a container of scene data represented by `instances`. Worlds are created with

```
ANARIWorld anariNewWorld(ANARIDevice);
```

Objects are placed in the world through an array of instances, geometries, volumes, or lights. Similar to `instances`, each array of objects is optional; there is no need to create empty arrays if there are no instances (though there might be nothing to render).

Table 18. *Parameters understood by the world.*

Name	Type	Description
instance	ARRAY1D of <code>INSTANCE</code>	optional array with handles of <code>instances</code>
surface	ARRAY1D of <code>SURFACE</code>	optional array with handles of <code>surfaces</code>
volume	ARRAY1D of <code>VOLUME</code>	optional array with handles of <code>volumes</code>
light	ARRAY1D of <code>LIGHT</code>	optional array with handles of <code>lights</code>

Table 19. *Properties queryable on a world.*

Name	Type	Description
bounds	FLOAT32_BOX3	axis-aligned bounding box in world-space (excluding the lights)

5.5. Instance

Instances apply transforms to groups for placement in the World. Instances are created with

```
ANARIInstance anariNewInstance(ANARIDevice, const char *subtype);
```

All instances take a [Group](#) representing all the objects which share the instances world-space transform. They also always take a generic [FLOAT32_MAT4](#) transform which serves as a fall back but may be overridden by subtype specific transform definitions.

Table 20. [Parameters](#) understood by instances.

Name	Type	Default	Description
group	GROUP		Group to be instanced, required
transform	FLOAT32_MAT4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	world-space transformation matrix for all attached objects
id	UINT32	-1u	extension KHR_FRAME_CHANNEL_INSTANCE_ID , optional user Id, for frame channel instanceId
color	FLOAT32_VEC4		uniform color attribute
attribute0	FLOAT32_VEC4		uniform attribute0
attribute1	FLOAT32_VEC4		uniform attribute1
attribute2	FLOAT32_VEC4		uniform attribute2
attribute3	FLOAT32_VEC4		uniform attribute3

The matrix [transform](#) represents an affine transformation which is applied to homogeneous coordinates.

The uniform attribute parameters apply to all objects contained in the instance and have a lower precedence than attribute values found in objects within the instance when both are present.



Note

Since the [FLOAT32_MAT4](#) matrix [transform](#) is in column-major layout the translational part of the affine transformation is the 13th to 15th float in memory. Implementations are free to always treat the last row (4th, 8th, 12th and 16th float) as being (0, 0, 0, 1).

Table 21. [Properties](#) queryable on an instance.

Name	Type	Description
bounds	FLOAT32_BOX3	axis-aligned bounding box in world-space (excluding the lights)



Note

Since the parameters of all other instance subtypes are supersets of the basic parameters listed above, it is recommend that the `transform` parameter should always be set in addition to any subtype specific transform parameters.

5.5.1. Transform

Extension `KHR_INSTANCE_TRANSFORM`

The `transform` instance subtype is a basic instance that implements the parameters and properties listed above.

The `KHR_INSTANCE_TRANSFORM_ARRAY` extension adds to the `transform` subtype additional types for the `transform` (and related) parameters by allowing arrays to do multi-instantiation. This gives applications a more efficient means to transform a single group many times by avoiding the need to create an entire object per-transform.

Table 22. Additional *parameters* introduced by `KHR_INSTANCE_TRANSFORM_ARRAY` *instance*.

Name	Type	Description
<code>transform</code>	<code>ARRAY1D</code> of <code>FLOAT32_MAT4</code>	array of matrices, instantiating <code>group</code> once per-element matrix
<code>id</code>	<code>ARRAY1D</code> of <code>UINT32</code>	<code>extension</code> <code>KHR_FRAME_CHANNEL_INSTANCE_ID</code> , optional user Id, for frame <code>channel</code> <code>instanceId</code>

The additional `id` array applies an instance identifier value to each element instance of the `transform` array. The size of the `id` array must be large enough to cover each element of the `transform` array. If the `id` array is not present when `transform` is set as an array, the single value version of `id` is used.

Note



Since setting transforms using an array of matrices does not have a sensible fallback when a device does not implement it, a unique extension `KHR_INSTANCE_TRANSFORM_ARRAY` is provided so this can be detected as early as possible for applications to decide whether or not the device is considered usable.

5.5.2. Motion Transform

Extension `KHR_INSTANCE_MOTION_TRANSFORM`

The `motionTransform` instance subtype represents uniformly (in `time`) distributed transformation keys as defined by `motion.transform` to achieve transformation motion blur in combination with `time` and extension `KHR_CAMERA_SHUTTER`.

Table 23. Additional *parameters* understood by the motion transform *instance*.

Name	Type	Default	Description
motion.transform	ARRAY1D of FLOAT32_MAT4		uniformly distributed world-space transformations
time	FLOAT32_BOX1	[0, 1]	time associated with first and last key in <code>motion.transform</code> array

The `motion.transform` parameter takes precedence over the generic `transform` parameter.

5.5.3. Motion Scale Rotation Translation

Extension `KHR_INSTANCE_MOTION_SCALE_ROTATION_TRANSLATION`

The `motionScaleRotationTranslation` instance subtype represents uniformly (in `time`) distributed transformation keys as defined by decomposed `motion.scale`, `motion.rotation` and `motion.translation` transform components (applied in this order) to achieve transformation motion blur in combination with `time` and extension `KHR_CAMERA_SHUTTER`.

Table 24. Additional *parameters* understood by the motion scale rotation translation *instance*.

Name	Type	Default	Description
motion.scale	ARRAY1D of FLOAT32_VEC3		uniformly distributed scale
motion.rotation	ARRAY1D of FLOAT32_QUAT_IJKW		uniformly distributed quaternion rotation
motion.translation	ARRAY1D of FLOAT32_VEC3		uniformly distributed translation
time	FLOAT32_BOX1	[0, 1]	time associated with first and last key in <code>motion.*</code> arrays

The `motion.*` parameters takes precedence over the generic `transform` parameter.

5.6. Group

Groups in ANARI represent collections of [surfaces](#), [volumes](#), and [lights](#) which share a common local-space coordinate system. Groups are created with

```
ANARIGroup anariNewGroup(ANARIDevice);
```

Each array on a group is optional; there is no need to create empty arrays if there are no surfaces, no volumes, or no lights instanced.

Table 25. *Parameters* understood by groups.

Name	Type	Description
surface	ARRAY1D of SURFACE	optional array with handles of surfaces
volume	ARRAY1D of VOLUME	optional array with handles of volumes
light	ARRAY1D of LIGHT	optional array with handles of lights

Table 26. *Properties* queryable on a group.

Name	Type	Description
bounds	FLOAT32_BOX3	axis-aligned bounding box (excluding the lights)

5.7. Light

Lights in ANARI are virtual objects that emit light into the world and thus illuminate objects. To create a new light object of given subtype `subtype` use

```
ANARILight anariNewLight(ANARIDevice, const char *subtype);
```

All light sources accept the following parameters:

Table 27. *Parameters* understood by all lights.

Name	Type	Default	Description
color	FLOAT32_VEC3	(1, 1, 1)	color of the light, in 0..1
visible	BOOL	TRUE	extension KHR_LIGHT_PRIMARY_VISIBILITY, whether the light can be directly seen (only meaningful for area lights)

The `color` parameter is a unitless factor and acts as a filter of the emitted light.

The transformation of an `instance` apply only to vectors like `position` or `direction` of a light, but *not* to affect sizes like `radius`.

5.7.1. Directional

Extension `KHR_LIGHT_DIRECTIONAL`

The directional light is thought to be far away (outside of the scene), thus its light arrives (mostly) as parallel rays. It is created by passing the subtype string `directional` to `anariNewLight`. In addition to the *general parameters* understood by all lights, the directional light supports the following special parameters:

Table 28. Additional *parameters* understood by the directional *light*.

Name	Type	Default	Description
direction	<code>FLOAT32_VEC3</code>	(0, 0, -1)	main emission direction of the directional light
angularDiameter	<code>FLOAT32</code>	0	apparent size (angle in radians) of the light
irradiance	<code>FLOAT32</code>	1	the amount of light arriving at a surface point, assuming the light is oriented towards to the surface, in W/m^2
radiance	<code>FLOAT32</code>		alternative specification of the brightness (if <code>irradiance</code> is not explicitly set): the amount of light emitted in a direction, in W/sr/m^2

If `irradiance` and `radiance` are both explicitly set, then `irradiance` takes precedence.

Note

The main way to specify the brightness of the directional light via `irradiance` and its alternative specification via `radiance` differs in behavior when the `angularDiameter` is changing: With `irradiance`, increasing the angular diameter will result in softer shadows (when soft shadows are supported by the implementation), but the overall brightness in the scene stays roughly the same. With `radiance` however, increasing the angular diameter will also result in a brighter illumination in the scene. Using `radiance` is only meaningful if `angularDiameter` is larger than zero (otherwise the directional light is not emitting any light).

Using a direction light with an angular diameter is a good approximation for sunlight: the apparent size of the sun is about 0.53° or 0.00925 rad.

5.7.2. HDRI

Extension `KHR_LIGHT_HDRI`

The HDRI light is a textured light source surrounding the scene and illuminating it from infinity. It is created by passing the subtype string `hdri` to `anariNewLight`. In addition to the *general parameters* the HDRI light supports the following special parameters:

Table 29. Additional *parameters* understood by the HDRI *light*.

Name	Type	Default	Description
up	FLOAT32_VEC3	(0, 0, 1)	up direction of the light in world-space
direction	FLOAT32_VEC3	(1, 0, 0)	direction to which the center of the texture will be mapped to
radiance	ARRAY2D of FLOAT32_VEC3		environment map, typically HDR with values >1, the amount of light emitted by a point on the light source in a direction, in W/sr/m ²
layout	STRING	equiangular	possible values: equiangular
scale	FLOAT32	1	scale factor for <i>radiance</i>

5.7.3. Point

Extension KHR_LIGHT_POINT

The point light (or with *radius* > 0 the sphere light) is a light emitting uniformly in all directions (from the surface toward the outside). It is created by passing the subtype string *point* to *anariNewLight*. In addition to the *general parameters* understood by all lights, the point light supports the following special parameters:

Table 30. Additional *parameters* understood by the point *light*.

Name	Type	Default	Description
position	FLOAT32_VEC3	(0, 0, 0)	the position of the point light
radius	FLOAT32	0	the size of the point light (becoming a sphere)
intensity	FLOAT32	1	the overall amount of light emitted by the light in a direction, in W/sr
power	FLOAT32		alternative specification of the brightness (if <i>intensity</i> is not explicitly set): the overall amount of light energy emitted, in W
radiance	FLOAT32		alternative specification of the brightness (if neither <i>intensity</i> nor <i>power</i> is explicitly set): the amount of light emitted by a point on the light source in a direction, in W/sr/m ²

The precedence order is *intensity*, *power*, *radiance*: the first (in this ordering) explicitly set parameter is used.



Note

The main way to specify the brightness of the point light via *intensity* and its alternative specifications via *power* or *radiance* differs in behavior when the *radius* is changing: With *intensity* or *power*, increasing the radius will result in softer shadows (when soft shadows are supported by the implementation), but the overall brightness in the scene stays roughly the same. With *radiance* however,

increasing the radius will also result in a brighter illumination in the scene. Using **radiance** is only meaningful if **radius** is larger than zero (otherwise the point light is not emitting any light).

Since power is the integral of intensity over the sphere of directions, using the same value for **power** and **intensity** will differ by a constant factor of 4π in brightness.

5.7.4. Quad

Extension **KHR_LIGHT_QUAD**

The quad light is a planar, procedural area light source, a parallelogram, emitting uniformly on one side into the half-space. It is created by passing the subtype string **quad** to **anariNewLight**. In addition to the **general parameters** understood by all lights, the quad light supports the following special parameters:

Table 31. Additional **parameters** understood by the quad **light**.

Name	Type	Default	Description
position	FLOAT32_VEC3	(0, 0, 0)	position of one vertex of the quad light
edge1	FLOAT32_VEC3	(1, 0, 0)	vector to one adjacent vertex
edge2	FLOAT32_VEC3	(0, 1, 0)	vector to the other adjacent vertex
intensity	FLOAT32	1	the overall amount of light emitted by the light in a direction, in W/sr
power	FLOAT32	1	the overall amount of light energy emitted, in W; intensity takes precedence if also specified
radiance	FLOAT32	1	the amount of light emitted by a point on the light source in a direction, in W/sr/m ² ; intensity (or power) takes precedence if also specified
side	STRING	front	side into which light is emitted; possible values: front , back , both
intensityDistribution	ARRAY1D / ARRAY2D of FLOAT32		luminous intensity distribution for photometric lights; can be 2D for asymmetric illumination; values are assumed to be uniformly distributed

Measured light sources (IES, EULUMDAT, ...) are supported by providing an **intensityDistribution array** to modulate the intensity per direction. The mapping is using the C- γ coordinate system: the values of the first (or only) dimension of **intensityDistribution** are uniformly mapped to γ in $[0-\pi]$; the first intensity value to 0, the last value to π , thus at least two values need to be present. If the array has a second dimension then the intensities are not rotational symmetric around **direction**, but are accordingly mapped to the C-halfplanes in $[0-2\pi]$; the first **row** of values to 0 and 2π , the other rows such that they have uniform distance to its neighbors. The orientation of the C0-plane is aligned with **edge1**.

The front side is determined by the cross product of **edge2** \times **edge1**.



Note

Some renderers will compute soft shadows from the quad light. Other implementations may just sample the center of the quad light (resulting in hard shadows), or may not compute shadows at all.

5.7.5. Ring

Extension **KHR_LIGHT_RING**

The ring light is a light emitting into a cone of directions. It is created by passing the subtype string **ring** to **anariNewLight**. In addition to the **general parameters** understood by all lights, the ring light supports the special parameters listed in the table. The ring light is an area light and thus has a cosine falloff.

Table 32. Additional **parameters** understood by the ring **light**.

Name	Type	Default	Description
position	FLOAT32_VEC3	(0, 0, 0)	the center of the ring light
direction	FLOAT32_VEC3	(0, 0, -1)	main emission direction, the center axis of the ring
openingAngle	FLOAT32	π	full opening angle (in radians) of the cone of directions; outside of this cone is no illumination
falloffAngle	FLOAT32	0.1	size (angle in radians) of the region between the rim (of the illumination cone) and full intensity; should be smaller than half of openingAngle
intensity	FLOAT32	1	the overall amount of light emitted by the light in a direction, in W/sr
power	FLOAT32	1	the overall amount of light energy emitted, in W; intensity takes precedence if also specified
radius	FLOAT32	0	the (outer) size of the ring, the radius of a disk with normal direction
innerRadius	FLOAT32	0	in combination with radius turns the disk into a ring; must be smaller than radius
radiance	FLOAT32	1	the amount of light emitted by a point on the light source in a direction, in W/sr/m ² ; intensity (or power) takes precedence if also specified
intensityDistribution	ARRAY1D / ARRAY2D of FLOAT32		luminous intensity distribution for photometric lights; can be 2D for asymmetric illumination; values are assumed to be uniformly distributed
c0	FLOAT32_VEC3	(1, 0, 0)	orientation, i.e., direction of the C0-(half)plane (only needed if illumination via intensityDistribution is asymmetric)

Setting the radius to a value greater than zero will result in soft shadows when the renderer uses stochastic sampling. Additionally setting the inner radius will result in a ring instead of a disk

emitting the light.

Measured light sources (IES, EULUMDAT, ...) are supported by providing an **intensityDistribution** array to modulate the intensity per direction. The mapping is using the C- γ coordinate system: the values of the first (or only) dimension of **intensityDistribution** are uniformly mapped to γ in $[0-\pi]$; the first intensity value to 0, the last value to π , thus at least two values need to be present. If the array has a second dimension then the intensities are not rotational symmetric around **direction**, but are accordingly mapped to the C-halfplanes in $[0-2\pi]$; the first row of values to 0 and 2π , the other rows such that they have uniform distance to its neighbors. The orientation of the C0-plane is specified via **c0**.

5.7.6. Spot

Extension **KHR_LIGHT_SPOT**

The spotlight is a light emitting into a cone of directions. It is created by passing the subtype string **spot** to **anariNewLight**. In addition to the **general parameters** understood by all lights, the spotlight supports the special parameters listed in the table.

Table 33. Additional **parameters** understood by the **spotlight**.

Name	Type	Default	Description
position	FLOAT32_VEC3	(0, 0, 0)	the center of the spotlight
direction	FLOAT32_VEC3	(0, 0, -1)	main emission direction, the axis of the spot
openingAngle	FLOAT32	π	full opening angle (in radians) of the spot; outside of this cone is no illumination
falloffAngle	FLOAT32	0.1	size (angle in radians) of the region between the rim (of the illumination cone) and full intensity of the spot; should be smaller than half of openingAngle
intensity	FLOAT32	1	the overall amount of light emitted by the light in a direction, in W/sr
power	FLOAT32		alternative specification of the brightness (if intensity is not explicitly set): the overall amount of light energy emitted, in W

The intensity distribution is constant within the core cone (**openingAngle** - $2 \times$ **falloffAngle**) and then falls off to zero (smoothly interpolated with the smoothstep function).

If **intensity** and **power** are both explicitly set, then **intensity** takes precedence.

Note



The main way to specify the brightness of the spotlight via **intensity** and its alternative specification via **power** differs in behavior when the **openingAngle** is changing: Increasing the opening angle with **power** will result in roughly the same overall brightness in the scene, but already lit areas will become darker, because the same energy is spread over a larger cone of illumination. With **intensity** the behavior is the opposite.

Power is the cosine-weighted integral of intensity over the cone of directions (within the opening angle). It can be approximated (ignoring the falloff) by $\pi(1 - \cos^2 \frac{1}{2} \text{openingAngle})$.

5.8. Surface

Geometries are matched with appearance information through Surfaces. These take a geometry, which defines the spatial representation, and applies either full-object or per-primitive color and material information. Surfaces are created with

```
ANARISurface anariNewSurface(ANARIDevice);
```

Table 34. *Parameters understood by Surface.*

Name	Type	Default	Description
geometry	GEOMETRY		Geometry object used by this surface
material	MATERIAL		Material applied to the geometry
id	UINT32	-1u	extension KHR_FRAME_CHANNEL_OBJECT_ID, optional user Id, for frame channel objectId

Surfaces require a valid [Geometry](#) to be set as the `geometry` parameter and a valid [Material](#) to be set as the `material` parameter.

5.9. Geometry

Geometries in ANARI are objects that describe the spatial representation of a surface. To create a new geometry object of given subtype `subtype` use

```
ANARIGeometry anariNewGeometry(ANARIDevice, const char *subtype);
```

All geometries support the following attribute setting parameters.

Table 35. *Parameters* understood by all geometries.

Name	Type	Description
color	FLOAT32_VEC4	uniform color attribute
attribute0	FLOAT32_VEC4	uniform attribute0
attribute1	FLOAT32_VEC4	uniform attribute1
attribute2	FLOAT32_VEC4	uniform attribute2
attribute3	FLOAT32_VEC4	uniform attribute3
primitive.color	ARRAY1D of Color	per-primitive color attribute
primitive.attribute0	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4	per-primitive attribute0
primitive.attribute1	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4	per-primitive attribute1
primitive.attribute2	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4	per-primitive attribute2
primitive.attribute3	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4	per-primitive attribute3

When both uniform and per-primitive attribute parameters are present, then the per-primitive parameter takes precedence. If geometries have additional per-vertex parameters specifying the same attribute, then the geometry specific *vertex.** parameters take precedence.



Note

Geometries are typically limited to a maximum of 2^{32} primitives.

5.9.1. Cone

Extension *KHR_GEOMETRY_CONE*

A geometry consisting of individual cones, is created by calling *anariNewGeometry* with subtype string *cone*.



Note

Implementations are allowed to internally tessellate the cone, or to render them procedurally such that the cones are perfectly round.

Table 36. Additional *parameters* understood by the cone *geometry*.

Name	Type	Default	Description
vertex.position	ARRAY1D of FLOAT32_VEC3		required center positions of the cones
vertex.radius	ARRAY1D of FLOAT32		radius at each vertex
vertex.cap	ARRAY1D of UINT8		per-vertex end cap flags (0 means no caps, 1 means flat-capped)
vertex.color	ARRAY1D of Color		vertex colors
vertex.attribute0	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4		vertex attribute0
vertex.attribute1	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4		vertex attribute1
vertex.attribute2	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4		vertex attribute2
vertex.attribute3	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4		vertex attribute3
primitive.index	ARRAY1D of UINT32_VEC2 / UINT64_VEC2	[(0, 1), (2, 3), ...]	optional indices into the <i>vertex.*</i> arrays, each pair defines one cone
caps	STRING	none	default vertex caps for all cones if <i>vertex.cap</i> is not set, possible values: <i>none</i> , <i>first</i> , <i>second</i> , <i>both</i>

Parameter *vertex.position* must be set and contain at least two elements to yield a valid cone geometry.

If no *primitive.index* is given, then a "cone soup" is assumed, i.e., each two consecutive vertices form one cone (if the size of the *vertex.position* array is not a multiple of two the remainder vertex is ignored).

All *primitive.** arrays must be at least as large as (explicitly set or implicitly derived) *primitive.index*. All *vertex.** arrays must be large enough to be indexed by the indices in (explicitly set or implicitly derived) *primitive.index*, which must be at most *device limit geometryMaxIndex*.

5.9.2. Curve

Extension *KHR_GEOMETRY_CURVE*

A geometry consisting of multiple curves is created by calling `anariNewGeometry` with type string `curve`. The vertices of the curve(s) are connected by linear, round segments (i.e., cones or clinders). The parameters defining this geometry are listed in the table below.



Note

Implementations are allowed to internally tessellate the curve, or to render them procedurally such that the curve segments are perfectly round.

Table 37. Additional *parameters* understood by the curve *geometry*.

Name	Type	Default	Description
vertex.position	ARRAY1D of <code>FLOAT32_VEC3</code>		required vertex positions
vertex.radius	ARRAY1D of <code>FLOAT32</code>		per-vertex radius
vertex.color	ARRAY1D of <code>Color</code>		vertex colors
vertex.attribute0	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute0
vertex.attribute1	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute1
vertex.attribute2	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute2
vertex.attribute3	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute3
primitive.index	ARRAY1D of <code>UINT32</code> / <code>UINT64</code>	[0, 1, 2, ...]	optional indices into <code>vertex.*</code> arrays, each index defines the start of one segment of a curve (the end is implicitly given with index+1)
radius	<code>FLOAT32</code>	1	default radius for all curve vertices (if <code>vertex.radius</code> is not set)

Parameter `vertex.position` must be set and contain at least two elements to yield a valid curve geometry. If no `primitive.index` is given, then a single curve is assumed, connecting all vertices.

All `primitive.*` arrays must be at least as large as (explicitly set or implicitly derived) `primitive.index`. All `vertex.*` arrays must be large enough to be indexed by the indices in (explicitly set or implicitly derived) `primitive.index`, which must be at most `device limit geometryMaxIndex` minus 1.

5.9.3. Cylinder

Extension `KHR_GEOMETRY_CYLINDER`

A geometry consisting of individual cylinders, each of which can have an own radius, is created by calling `anariNewGeometry` with subtype string `cylinder`.



Note

Implementations are allowed to internally tessellate the cylinder, or to render them procedurally such that the cylinders are perfectly round.

Table 38. Additional *parameters* understood by the cylinder *geometry*.

Name	Type	Default	Description
vertex.position	ARRAY1D of <code>FLOAT32_VEC3</code>		required center positions of the cylinders
vertex.cap	ARRAY1D of <code>UINT8</code>		per-vertex end cap flags (0 means no caps, 1 means flat-capped)
vertex.color	ARRAY1D of <code>Color</code>		vertex colors
vertex.attribute0	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute0
vertex.attribute1	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute1
vertex.attribute2	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute2
vertex.attribute3	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute3
primitive.index	ARRAY1D of <code>UINT32_VEC2</code> / <code>UINT64_VEC2</code>	[(0, 1), (2, 3), ...]	optional indices into <code>vertex.*</code> arrays, each pair defines one cylinder
primitive.radius	ARRAY1D of <code>FLOAT32</code>		per-cylinder radius
radius	<code>FLOAT32</code>	1	default radius for all cylinders (if <code>primitive.radius</code> is not set)
caps	<code>STRING</code>	<code>none</code>	default vertex caps for all cylinders if <code>vertex.cap</code> is not set, possible values: <code>none</code> , <code>first</code> , <code>second</code> , <code>both</code>

Parameter `vertex.position` must be set and contain at least two elements to yield a valid cylinder geometry.

If no `primitive.index` is given, then a "cylinder soup" is assumed, i.e., each two consecutive vertices form one cylinder (if the size of the `vertex.position` array is not a multiple of two the remainder vertex is ignored).

All `primitive.*` arrays must be at least as large as (explicitly set or implicitly derived) `primitive.index`. All `vertex.*` arrays must be large enough to be indexed by the indices in (explicitly set or implicitly derived) `primitive.index`, which must be at most `device limit geometryMaxIndex`.

5.9.4. Isosurface

Extension `KHR_GEOMETRY_ISOSURFACE`

A geometry defined implicitly as the surface(s) of constant value of a spatial field is created by calling `anariNewGeometry` with subtype string `isosurface`. A isosurface geometry recognizes the following parameters:

Table 39. Additional *parameters* understood by the isosurface *geometry*.

Name	Type	Description
isovalue	<code>FLOAT32</code> / <code>ARRAY1D</code> of <code>FLOAT32</code>	isovalue(s) defining the isosurface(s)
field	<code>SPATIAL_FIELD</code>	<code>Spatial Field</code> to be isosurfaced

Both parameters, `isovalue` and `field`, must be set to yield a valid isosurface geometry. If multiple isovalues are given (as array) then multiple surfaces are created, which are the primitives of the isosurface geometry.

5.9.5. Quad

Extension `KHR_GEOMETRY_QUAD`

A geometry consisting of quads is created by calling `anariNewGeometry` with subtype string `quad`. A quad geometry recognizes the following parameters:

Table 40. Additional *parameters* understood by the quad *geometry*.

Name	Type	Default	Description
vertex.position	ARRAY1D of <code>FLOAT32_VEC3</code>		required vertex positions
vertex.normal	ARRAY1D of <code>FIXED16_VEC3</code> / <code>FLOAT32_VEC3</code>		vertex normals
vertex.tangent	ARRAY1D of <code>FIXED16_VEC3</code> / <code>FIXED16_VEC4</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex tangents
vertex.color	ARRAY1D of <code>Color</code>		vertex colors
vertex.attribute0	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute0
vertex.attribute1	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute1
vertex.attribute2	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute2
vertex.attribute3	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute3
faceVarying.color	ARRAY1D of <code>Color</code>		face-varying colors
faceVarying.attribute0	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		face-varying attribute0
faceVarying.attribute1	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		face-varying attribute1
faceVarying.attribute2	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		face-varying attribute2
faceVarying.attribute3	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		face-varying attribute3
primitive.index	ARRAY1D of <code>UINT32_VEC4</code> / <code>UINT64_VEC4</code>	[(0, 1, 2, 4), (5, 6, 7, 8), ...]	optional indices (into the vertex array(s)), each 4-tuple defines one quad

Parameter `vertex.position` must be set and contain at least four elements to yield a valid quad geometry.

The fourth component of an element of `vertex.tangent` (if present) indicates the handedness of the tangent space coordinate system and thus must be 1 or -1.

Each element of `primitive.index` defines one quad, its four components index into the `vertex.*` arrays. If no `primitive.index` is given, then a "quad soup" is assumed, i.e., each four consecutive vertices form one quad (if the size of the `vertex.position` array is not a multiple of four the remainder one, two, or three vertices are ignored).

All `primitive.*` arrays must be at least as large as (explicitly set or implicitly derived) `primitive.index`. All `vertex.*` arrays must be large enough to be indexed by the indices in (explicitly

set or implicitly derived) `primitive.index`, which must be at most `device limit geometryMaxIndex`.

The `faceVarying.*` attributes map unique values to each (implicitly or explicitly) indexed primitive vertices, where each vertex takes a unique value per-primitive that uses it. Each primitive associates vertex attributes values from the attribute array using the formula $4 * \text{primID} + \{0, 1, 2, 3\}$.

Face-varying attributes take precedence over vertex attributes when both arrays of the same attribute are present.

Table 41. Additional parameters understood by a quad geometry with extension `KHR_GEOMETRY_QUAD_MOTION_DEFORMATION`.

Name	Type	Default	Description
<code>motion.vertex.position</code>	ARRAY1D of ARRAY1D of <code>FLOAT32_VEC3</code>		uniformly distributed vertex position arrays
<code>motion.vertex.normal</code>	ARRAY1D of ARRAY1D of <code>FIXED16_VEC3</code> / <code>FLOAT32_VEC3</code>		uniformly distributed vertex normal arrays
<code>motion.vertex.tangent</code>	ARRAY1D of ARRAY1D of <code>FIXED16_VEC3</code> / <code>FIXED16_VEC4</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		uniformly distributed vertex tangents arrays
<code>time</code>	<code>FLOAT32_BOX1</code>	[0, 1]	time associated with first and last key in <code>motion.*</code> arrays

The `motion.*` arrays represents uniformly (in `time`) distributed vertex data keys to achieve deformation motion blur in combination with extension `KHR_CAMERA_SHUTTER`.

5.9.6. Sphere

Extension `KHR_GEOMETRY_SPHERE`

A geometry consisting of individual spheres, each of which can have an own radius, is created by calling `anariNewGeometry` with subtype string `sphere`.



Note

Implementations are allowed to internally tessellate the sphere, or to render them procedurally such that the spheres are perfectly round.

Table 42. Additional *parameters* understood by the sphere *geometry*.

Name	Type	Default	Description
vertex.position	ARRAY1D of FLOAT32_VEC3		required center positions of the spheres
vertex.radius	ARRAY1D of FLOAT32		per-sphere radius
vertex.color	ARRAY1D of Color		vertex colors
vertex.attribute0	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4		vertex attribute0
vertex.attribute1	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4		vertex attribute1
vertex.attribute2	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4		vertex attribute2
vertex.attribute3	ARRAY1D of FLOAT32 / FLOAT32_VEC2 / FLOAT32_VEC3 / FLOAT32_VEC4		vertex attribute3
primitive.index	ARRAY1D of UINT32 / UINT64	[0, 1, 2, ...]	optional indices (into the vertex array(s))
radius	FLOAT32	1	default radius for all spheres (if <i>vertex.radius</i> is not set)

Parameter *vertex.position* must be set to yield a valid sphere geometry.

Each element of *primitive.index* defines one sphere, indexing into the *vertex.** arrays. If no *primitive.index* is given, then a "sphere soup" is assumed, using all spheres at *vertex.position*.

All *primitive.** arrays must be at least as large as (explicitly set or implicitly derived) *primitive.index*. All *vertex.** arrays must be large enough to be indexed by the indices in (explicitly set or implicitly derived) *primitive.index*, which must be at most *device limit geometryMaxIndex*.

5.9.7. Triangle

Extension *KHR_GEOMETRY_TRIANGLE*

A geometry consisting of triangles is created by calling *anariNewGeometry* with subtype string *triangle*. A triangle geometry recognizes the following parameters:

Table 43. Additional *parameters* understood by the triangle *geometry*.

Name	Type	Default	Description
vertex.position	ARRAY1D of <code>FLOAT32_VEC3</code>		required vertex positions
vertex.normal	ARRAY1D of <code>FIXED16_VEC3</code> / <code>FLOAT32_VEC3</code>		vertex normals
vertex.tangent	ARRAY1D of <code>FIXED16_VEC3</code> / <code>FIXED16_VEC4</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex tangents
vertex.color	ARRAY1D of <code>Color</code>		vertex colors
vertex.attribute0	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute0
vertex.attribute1	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute1
vertex.attribute2	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute2
vertex.attribute3	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		vertex attribute3
faceVarying.color	ARRAY1D of <code>Color</code>		face-varying colors
faceVarying.attribute0	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		face-varying attribute0
faceVarying.attribute1	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		face-varying attribute1
faceVarying.attribute2	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		face-varying attribute2
faceVarying.attribute3	ARRAY1D of <code>FLOAT32</code> / <code>FLOAT32_VEC2</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		face-varying attribute3
primitive.index	ARRAY1D of <code>UINT32_VEC3</code> / <code>UINT64_VEC3</code>	[(0, 1, 2), (4, 5, 6), ...]	optional indices (into the vertex array(s)), each 3-tuple defines one triangle

Parameter `vertex.position` must be set and contain at least three elements to yield a valid triangle geometry.

The fourth component of an element of `vertex.tangent` (if present) indicates the handedness of the tangent space coordinate system and thus must be 1 or -1.

Each element of `primitive.index` defines one triangle, its three components index into the `vertex.*` arrays. If no `primitive.index` is given, then a "triangle soup" is assumed, i.e., each three consecutive vertices form one triangle (if the size of the `vertex.position` array is not a multiple of three the remainder one or two vertices are ignored).

All `primitive.*` arrays must be at least as large as (explicitly set or implicitly derived) `primitive.index`. All `vertex.*` arrays must be large enough to be indexed by the indices in (explicitly

set or implicitly derived) `primitive.index`, which must be at most `device limit geometryMaxIndex`.

The `faceVarying.*` attributes map unique values to each (implicitly or explicitly) indexed primitive vertices, where each vertex takes a unique value per-primitive that uses it. Each primitive associates vertex attributes values from the attribute array using the formula $3 * \text{primID} + \{0, 1, 2\}$.

Face-varying attributes take precedence over vertex attributes when both arrays of the same attribute are present.

Table 44. Additional parameters understood by a triangle geometry with extension `KHR_GEOMETRY_TRIANGLE_MOTION_DEFORMATION`.

Name	Type	Default	Description
<code>motion.vertex.position</code>	<code>ARRAY1D</code> of <code>ARRAY1D</code> of <code>FLOAT32_VEC3</code>		uniformly distributed vertex position arrays
<code>motion.vertex.normal</code>	<code>ARRAY1D</code> of <code>ARRAY1D</code> of <code>FIXED16_VEC3</code> / <code>FLOAT32_VEC3</code>		uniformly distributed vertex normal arrays
<code>motion.vertex.tangent</code>	<code>ARRAY1D</code> of <code>ARRAY1D</code> of <code>FIXED16_VEC3</code> / <code>FIXED16_VEC4</code> / <code>FLOAT32_VEC3</code> / <code>FLOAT32_VEC4</code>		uniformly distributed vertex tangents arrays
<code>time</code>	<code>FLOAT32_BOX1</code>	<code>[0, 1]</code>	time associated with first and last key in <code>motion.*</code> arrays

The `motion.*` arrays represents uniformly (in `time`) distributed vertex data keys to achieve deformation motion blur in combination with extension `KHR_CAMERA_SHUTTER`.

5.10. Sampler

ANARI sampler objects map attribute data into other object inputs such materials. To create a new sampler use

```
ANARISampler anariNewSampler(ANARIDevice, const char *subtype);
```

5.10.1. Image1D

Extension `KHR_SAMPLER_IMAGE1D`

A one dimensional image sampler is created by calling `anariNewSampler` with subtype string `image1D`. It accepts the following parameters.

Table 45. *Parameters understood by the 1D image sampler.*

Name	Type	Default	Description
inAttribute	STRING	attribute0	surface attribute used as texture coordinate, possible values: float Attributes
inTransform	FLOAT32_MAT4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the input transform before sampling
inOffset	FLOAT32_VEC4	(0, 0, 0, 0)	offset added to input transform result
image	ARRAY1D of Color		array backing the sampler
filter	STRING	linear	filter of the sampler, possible values: nearest, linear
wrapMode	STRING	clampToEdge	wrap mode of the sampler, possible values: clampToEdge, repeat, mirrorRepeat
outTransform	FLOAT32_MAT4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the sampled values
outOffset	FLOAT32_VEC4	(0, 0, 0, 0)	offset added to output transform result

The attribute values indicated by `inAttribute` are first multiplied by the `inTransform` and added to the `inOffset`, then its first component is used as normalized coordinate to sample `image`, taking `filter` and `wrapMode` into account. Refer to [\[vulkan-samplers\]](#) for the exact definitions of the sampling and filtering operation. The sampled value is completed to four components (if needed: unspecified components default to zero for the first three components and to one for the fourth component), multiplied by `outTransform` and added to `outOffset` to yield the final result of the sampler.

Applications which set unknown values for `filter` will result in the default being used.



Note

Like all `FLOAT32_MAT4` matrices, `inTransform` and `outTransform` are in column-major memory layout and are applied to column vectors (multiplied from the left `transform * x + offset`).

5.10.2. Image2D

Extension `KHR_SAMPLER_IMAGE2D`

A two dimensional image sampler is created by calling `anarNewSampler` with subtype string `image2D`. It accepts the following parameters.

Table 46. *Parameters* understood by the 2D image sampler.

Name	Type	Default	Description
inAttribute	STRING	attribute0	surface attribute used as texture coordinate, possible values: float Attributes
inTransform	FLOAT32_MAT4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the input transform before sampling
inOffset	FLOAT32_VEC4	(0, 0, 0, 0)	offset added to input transform result
image	ARRAY2D of Color / FIXED8_VEC3 / FIXED8_VEC4		array backing the sampler
filter	STRING	linear	filter of the sampler, possible values: nearest, linear
wrapMode1	STRING	clampToEdge	wrap mode of the sampler for the 1st dimension, possible values: clampToEdge, repeat, mirrorRepeat
wrapMode2	STRING	clampToEdge	wrap mode of the sampler for the 2nd dimension, possible values: clampToEdge, repeat, mirrorRepeat
outTransform	FLOAT32_MAT4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the sampled values
outOffset	FLOAT32_VEC4	(0, 0, 0, 0)	offset added to output transform result

The attribute values indicated by `inAttribute` are first multiplied by the `inTransform` and added to the `inOffset`, then its first component is used as normalized coordinate to sample `image`, taking `filter` and `wrapMode` into account. Refer to [\[vulkan-samplers\]](#) for the exact definitions of the sampling and filtering operation. The sampled value is completed to four components (if needed: unspecified components default to zero for the first three components and to one for the fourth component), multiplied by `outTransform` and added to `outOffset` to yield the final result of the sampler.

Applications which set unknown values for `filter` will result in the default being used.



Note

Like all `FLOAT32_MAT4` matrices, `inTransform` and `outTransform` are in column-major memory layout and are applied to column vectors (multiplied from the left `transform * x + offset`).

5.10.3. Image3D

Extension `KHR_SAMPLER_IMAGE3D`

A three dimensional image sampler is created by calling `anariNewSampler` with subtype string

image3D. It accepts the following parameters.

Table 47. *Parameters understood by the 3D image sampler.*

Name	Type	Default	Description
inAttribute	STRING	attribute0	surface attribute used as texture coordinate, possible values: float Attributes
inTransform	FLOAT32_MAT4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the input transform before sampling
inOffset	FLOAT32_VEC4	(0, 0, 0, 0)	offset added to input transform result
image	ARRAY3D of Color		array backing the sampler
filter	STRING	linear	filter of the sampler, possible values: nearest , linear
wrapMode1	STRING	clampToEdge	wrap mode of the sampler for the 1st dimension, possible values: clampToEdge , repeat , mirrorRepeat
wrapMode2	STRING	clampToEdge	wrap mode of the sampler for the 2nd dimension, possible values: clampToEdge , repeat , mirrorRepeat
wrapMode3	STRING	clampToEdge	wrap mode of the sampler for the 3rd dimension, possible values: clampToEdge , repeat , mirrorRepeat
outTransform	FLOAT32_MAT4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the sampled values
outOffset	FLOAT32_VEC4	(0, 0, 0, 0)	offset added to output transform result

The attribute values indicated by **inAttribute** are first multiplied by the **inTransform** and added to the **inOffset**, then its first component is used as normalized coordinate to sample **image**, taking **filter** and **wrapMode** into account. Refer to [\[vulkan-samplers\]](#) for the exact definitions of the sampling and filtering operation. The sampled value is completed to four components (if needed: unspecified components default to zero for the first three components and to one for the fourth component), multiplied by **outTransform** and added to **outOffset** to yield the final result of the sampler.

Applications which set unknown values for **filter** will result in the default being used.



Note

Like all **FLOAT32_MAT4** matrices, **inTransform** and **outTransform** are in column-major memory layout and are applied to column vectors (multiplied from the left $\text{transform} * x + \text{offset}$).

5.10.4. Primitive

Extension `KHR_SAMPLER_PRIMITIVE`

The primitive sampler samples an `Array1D` at integer coordinates based on the `primitiveId` surface attribute. It is created by calling `anariNewSampler` with subtype string `primitive`. The `inOffset` parameter value is added to `primitiveID` before sampling.

Table 48. *Parameters understood by the primitive sampler.*

Name	Type	Default	Description
array	<code>ARRAY1D</code> of <code>Color</code>		backing array of the sampler
inOffset	<code>UINT64</code>	0	offset into the array



Note

Primitive samplers serve a similar role as per primitive attributes on geometries but are attached to the material instead.

5.10.5. Transform

Extension `KHR_SAMPLER_TRANSFORM`

The transform sampler multiplies by the input attribute by `outTransform` and adds it `outOffset` returns the result as the sampled value. It is created by calling `anariNewSampler` with subtype string `transform`.

Table 49. *Parameters understood by the transform sampler.*

Name	Type	Default	Description
inAttribute	<code>STRING</code>	<code>attribute0</code>	surface attribute used as input
outTransform	<code>FLOAT32_MAT4</code>	$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$	transform applied to input
outOffset	<code>FLOAT32_VEC4</code>	$(0, 0, 0, 0)$	offset added to output transform result



Note

The transform sampler does not actually sample any array or image but merely applies a transformation to an attribute for purposes such as swizzling or scaling the components.

Like all `FLOAT32_MAT4` matrices, `outTransform` is in column-major memory layout and is applied to column vectors (multiplied from the left $\text{transform} * \mathbf{x} + \text{offset}$).

5.11. Material

Materials describe how light interacts with surfaces to give objects their appearance. Materials are

created with

```
ANARIMaterial anariNewMaterial(ANARIDevice, const char *subtype);
```

Most material parameters can be set to a constant, **ANARISampler** or string value. Unless otherwise noted, the string selects the surface **attribute** that will be used to source the parameter during rendering.

5.11.1. Matte

Extension **KHR_MATERIAL_MATTE**

The **matte** material reflects light uniformly into the hemisphere, i.e., it exhibits Lambertian reflectance and thus its apparent brightness is independent of the viewing direction. The matte material supports (partial) cut-out transparency depending on **alphaMode**.

Table 50. *Parameters of the matte material.*

Name	Type	Default	Description
color	FLOAT32_VEC3 / SAMPLER / STRING	(0.8, 0.8, 0.8)	diffuse color
opacity	FLOAT32 / SAMPLER / STRING	1.0	opacity
alphaMode	STRING	opaque	control cut-out transparency, possible values: opaque , blend , mask
alphaCutoff	FLOAT32	0.5	threshold when alphaMode is mask

If **color** is of type **ANARI_SAMPLER** or **ANARI_STRING**, the fourth component of the value fetched from a sampler or attribute is multiplied with the opacity value. The final opacity is determined by **alphaMode**:

- **opaque**: fully opaque (opacity treated as 1)
- **blend**: partial opacity
- **mask**: fully opaque (opacity treated as 1) if the computed opacity value is greater than or equal to **alphaCutoff**, otherwise fully transparent (opacity treated 0)

5.11.2. Physically Based

Extension **KHR_MATERIAL_PHYSICALLY_BASED**

The **physicallyBased** material offers a wealth of extensions while being user friendly. It aims to be compatible to glTF's pbrMetallicRoughness material and ratified Khronos material extensions, thus refer to [\[gltf-brdf\]](#) for the exact definitions of the BRDF and also non-normative implementation hints, as well as to glTF extensions [KHR_materials_specular](#), [KHR_materials_clearcoat](#), [KHR_materials_emissive_strength](#), [KHR_materials_ior](#), [KHR_materials_transmission](#), [KHR_materials_volume](#), [KHR_materials_sheen](#), and [KHR_materials_iridescence](#).

Table 51. *Parameters of the physicallyBased material.*

Name	Type	Default	Description
baseColor	FLOAT32_VEC3 / SAMPLER / STRING	(1.0, 1.0, 1.0)	base color
opacity	FLOAT32 / SAMPLER / STRING	1.0	opacity
metallic	FLOAT32 / SAMPLER / STRING	1.0	metalness
roughness	FLOAT32 / SAMPLER / STRING	1.0	roughness
normal	SAMPLER		normal map for the base layer
emissive	FLOAT32_VEC3 / SAMPLER / STRING	(0.0, 0.0, 0.0)	emissive
occlusion	SAMPLER		occlusion map
alphaMode	STRING	opaque	control cut-out transparency, possible values: opaque , blend , mask
alphaCutoff	FLOAT32	0.5	threshold when alphaMode is mask
specular	FLOAT32 / SAMPLER / STRING	0.0	strength of the specular reflection
specularColor	FLOAT32_VEC3 / SAMPLER / STRING	(1.0, 1.0, 1.0)	color of the specular reflection at normal incidence
clearcoat	FLOAT32 / SAMPLER / STRING	0.0	strength of the clearcoat layer
clearcoatRoughness	FLOAT32 / SAMPLER / STRING	0.0	roughness of the clearcoat layer
clearcoatNormal	SAMPLER		normal map for the clearcoat layer
transmission	FLOAT32 / SAMPLER / STRING	0.0	strength of the transmission
ior	FLOAT32	1.5	index of refraction
thickness	FLOAT32 / SAMPLER / STRING	0.0	thickness of the volume beneath the surface (with 0 the material is thin-walled)
attenuationDistance	FLOAT32	∞	average distance that light travels in the medium before interacting with a particle
attenuationColor	FLOAT32_VEC3	(1.0, 1.0, 1.0)	color that white light turns into due to absorption when reaching the attenuation distance
sheenColor	FLOAT32_VEC3 / SAMPLER / STRING	(0.0, 0.0, 0.0)	sheen color
sheenRoughness	FLOAT32 / SAMPLER / STRING	0.0	sheen roughness

Name	Type	Default	Description
iridescence	FLOAT32 / SAMPLER / STRING	0.0	strength of the thin-film layer
iridescenceIOR	FLOAT32	1.3	index of refraction of the thin-film layer
iridescenceThickness	FLOAT32 / SAMPLER / STRING	0.0	thickness of the thin-film layer

If `baseColor` is of type `ANARI_SAMPLER` or `ANARI_STRING` and `alphaMode` is not `opaque`, the fourth component of the value fetched from a sampler or attribute is multiplied with the `opacity` value. The final opacity is determined by `alphaMode`:

- `opaque`: fully opaque (opacity treated as 1)
- `blend`: partial opacity
- `mask`: fully opaque (opacity treated as 1) if the computed opacity value is greater than or equal to `alphaCutoff`, otherwise fully transparent (opacity treated 0)

The values sampled from samplers `normal` and `clearcoatNormal` represent tangent space normals (the first two components are expected to be in range `[-1, 1]` and the third component is expected to be in range `[0, 1]`).

Note

To use the glTF `metallicRoughnessTexture` create two samplers with the same `image` array and a "swizzle" `outTransform` for `roughness`.

The normal maps `normal` and `clearcoatNormal` typically use an `Image2D` sampler with `image` array element type `UFIXED8_VEC3` with a transformation from range `[0, 1]` to range `[-1, 1]` via `outTransform` and `outOffset`, computing $2 * texel - 1$ per channel. The glTF `scale` parameter of normal maps should be factored into the `outTransform` and `outOffset` of the corresponding `sampler`. Putting it all together, set `outTransform` to $((2 * scale, 0, 0, 0), (0, 2 * scale, 0, 0), (0, 0, 2, 0), (0, 0, 0, 1))$ and `outOffset` to $(-scale, -scale, -1, 0)$.

Similarly, glTF parameters `emissiveStrength` and `emissiveFactor` should be factored into `emissive` (or `outTransform` if `emissive` is an `ANARI_SAMPLER`).

Also, the glTF parameters `iridescenceThicknessMinimum` and `iridescenceThicknessMaximum` should be mapped to `outTransform` and `outOffset` of the `iridescenceThickness` sampler.

Note

Implementations should apply the `occlusion` attenuation only for otherwise unoccluded ambient lighting. The glTF parameter `strength` should be factored into the diagonal of `outTransform` (as scale) and $1 - strength$ added to `outOffset` of its sampler.

5.12. Volume

Volumes in ANARI represent volumetric objects (complementing [surfaces](#)), encapsulating spatial data as well as appearance information. To create a new volume object of given subtype [subtype](#) use

```
ANARIVolume anariNewVolume(ANARIDevice, const char *subtype);
```

Table 52. [Parameters](#) understood by volumes.

Name	Type	Default	Description
id	UINT32	-1u	extension KHR_FRAME_CHANNEL_OBJECT_ID , optional user Id, for frame channel objectId

5.12.1. TransferFunction1D

Extension [KHR_VOLUME_TRANSFER_FUNCTION1D](#)

The 1D transfer function volume is created by passing the subtype string [transferFunction1D](#) to [anariNewVolume](#). It supports the following parameters:

Table 53. [Parameters](#) understood by the [transferFunction1D](#) volume.

Name	Type	Default	Description
value	SPATIAL_FIELD		Spatial field used for the scalar values of the volume
valueRange	FLOAT32_BOX1 / FLOAT64_BOX1	[0, 1]	sampled values of value are clamped to this range
color	FLOAT32_VEC4 / FLOAT32_VEC3 / ARRAY1D of Color	(1.0, 1.0, 1.0, 1.0)	color to which the sampled values are mapped to
opacity	FLOAT32 / ARRAY1D of FLOAT32	1.0	opacity to which the sampled values are mapped to
unitDistance	FLOAT32	1.0	distance after which a 'opacity' fraction of light traveling through the volume is absorbed



Note

The [color](#) and [opacity](#) parameters together represent a transfer function, which is used to visually emphasize the structure or certain features in the [value](#) data.

The parameters [color](#) and [opacity](#) represent a look-up table to map the sampled values of the spatial field [value](#) (after clamping to [valueRange](#)) to a color and a opacity: The first array element representing the lowest value in [valueRange](#) and the last element representing the the last element in [valueRange](#) and elements are linearly interpolated in between. A [color](#) array can have a different size than a [opacity](#) array.

The fourth component of the resulting color is multiplied with the resulting opacity to form the final opacity.

The 1D transfer function volume represents (monochromatic) absorbing and (colored) light emitting particles whose local density is defined at each point of the spatial field by the final opacity and `unitDistance`: the fraction of light which is absorbed while traveling a `unitDistance` length through the volume will be 'opacity'.



Note

An opacity of 1.0 results in an infinitely high density of particles, a solid surface. Implementations may limit opacity to a value slightly smaller than one.



Note

A possible implementation of a rendering algorithm for this volume type is given in [Example Algorithm for Rendering a TransferFunction1D Volume \(Informative\)](#).

5.13. Spatial Field

ANARI spatial field objects define collections of data values spread throughout a local coordinate system in order to be sampled in space. Spatial fields are created with

```
ANARISpatialField anariNewSpatialField(ANARIDevice, const char *subtype);
```

5.13.1. Structured Regular

Extension `KHR_SPATIAL_FIELD_STRUCTURED_REGULAR`

Structured regular spatial fields are created by passing the subtype string `structuredRegular` to `anariNewSpatialField`. The parameters understood by structured regular spatial fields are summarized in the table below.

Table 54. Configuration parameters for structured regular spatial fields.

Name	Type	Default	Description
data	<code>ARRAY3D</code> of <code>UFIXED8</code> / <code>FIXED16</code> / <code>UFIXED16</code> / <code>FLOAT32</code> / <code>FLOAT64</code>		the actual field values for the 3D grid, i.e., the scalars are vertex-centered; the size of the spatial field is inferred from <code>data</code>
origin	<code>FLOAT32_VEC3</code>	(0, 0, 0)	origin of the grid in object-space
spacing	<code>FLOAT32_VEC3</code>	(1, 1, 1)	size of the grid cells in object-space
filter	<code>STRING</code>	<code>linear</code>	filter used for reconstructing the field, possible values: <code>nearest</code> , <code>linear</code> , <code>cubic</code> (extension <code>KHR_SPATIAL_FIELD_FILTER_CUBIC</code>)

The spatial field grid can be moved with `origin` and scaled with `spacing`, in local object coordinates. The spatial field data is interpreted to be vertex-centered, which means at least two data values

need to be specified in each dimension to avoid a degenerated spatial field. Its local bounds are `[origin, origin + (data.size - 1) × spacing]`.

Applications which set unknown values for `filter` will result in the default being used.



Note

Structured regular fields only need to store the values of the samples, because their addresses in memory can be easily computed from a 3D position. A common type of structured spatial fields are regular grids.

5.13.2. Unstructured

Extension `KHR_SPATIAL_FIELD_UNSTRUCTURED`

Unstructured spatial fields are created by passing the subtype string `unstructured` to `anariNewSpatialField`. The parameters understood by unstructured spatial fields are summarized in the table below.

Table 55. Configuration parameters for unstructured spatial fields.

Name	Type	Description
<code>vertex.position</code>	ARRAY1D of <code>FLOAT32_VEC3</code>	array of vertex positions
<code>vertex.data</code>	ARRAY1D of <code>UFIXED8</code> / <code>FIXED16</code> / <code>UFIXED16</code> / <code>FLOAT32</code> / <code>FLOAT64</code>	array of values at vertices
<code>index</code>	ARRAY1D of <code>UINT32</code> / <code>UINT64</code>	array of indices into the <code>vertex.*</code> arrays that form cells
<code>cell.data</code>	ARRAY1D of <code>UFIXED8</code> / <code>FIXED16</code> / <code>UFIXED16</code> / <code>FLOAT32</code> / <code>FLOAT64</code>	array of values in cells
<code>cell.type</code>	ARRAY1D of <code>UINT8</code>	array of cell types, where 10 encodes tetrahedral, 12 hexahedral, 13 wedge, and 14 pyramidal cells
<code>cell.index</code>	ARRAY1D of <code>UINT32</code> / <code>UINT64</code>	array of indices into the <code>index</code> array, specifying the first (index of the) vertex of each cell

Sampled cell values can be specified either per-vertex (via `vertex.data`) or per-cell (via `cell.data`). If both arrays are set, `vertex.data` takes precedence.

Each cell is formed by a consecutive group of indices from `index` into the vertices, the first index of the group indicated by `cell.index`.

The index order for a tetrahedron (cell type 10) is bottom triangle counterclockwise, then the top vertex.

For hexahedral cells (type 12), each hexahedron is formed by a group of eight indices into the vertices and data values. Vertex ordering is four bottom vertices counterclockwise, then top four counterclockwise.

For wedge cells (type 13), each wedge is formed by a group of six indices into the vertices and data values. Vertex ordering is three bottom vertices counterclockwise, then top three counterclockwise.

For pyramid cells (type 14), each cell is formed by a group of five indices into the vertices and data values. Vertex ordering is four bottom vertices counterclockwise, then the top vertex.



Note

The unstructured spatial field is modelled after and thus compatible to `vtkUnstructuredGrid`: `cell.type` values are the same as VTK cell types, and the index order of the cells is the same as their VTK counterparts (`VTK_TETRA`, `VTK_HEXAHEDRON`, `VTK_WEDGE`, and `VTK_PYRAMID`). The arrays `cell.index` corresponds to the Offsets array and `index` to the Connectivity array of (current, not legacy) `vtkCellArray`.

5.14. Extension Objects

Extensions may need to introduce custom object types. To create such an object use

```
ANRIOObject anariNewObject(ANARIDevice, const char *type, const char *subtype);
```

Consult the extension documentation for supported `type` and `subtype` values.

Appendix A: Function Index (Informative)

- `anariCommitParameters`
- `anariDiscardFrame`
- `ANARIFrameCompletionCallback`
- `anariFrameReady`
- `anariGetDeviceExtensions`
- `anariGetDeviceSubtypes`
- `anariGetObjectInfo`
- `anariGetObjectSubtypes`
- `anariGetParameterInfo`
- `anariGetProperty`
- `anariLoadLibrary`
- `anariMapArray`
- `anariMapFrame`
- `anariMapParameterArray1D`
- `anariMapParameterArray2D`
- `anariMapParameterArray3D`
- `ANARIMemoryDeleter`
- `anariNewArray1D`
- `anariNewArray2D`
- `anariNewArray3D`
- `anariNewCamera`
- `anariNewDevice`
- `anariNewFrame`
- `anariNewGeometry`
- `anariNewGroup`
- `anariNewInstance`
- `anariNewLight`
- `anariNewMaterial`
- `anariNewObject`
- `anariNewRenderer`
- `anariNewSampler`
- `anariNewSpatialField`
- `anariNewSurface`

- `anariNewVolume`
- `anariNewWorld`
- `anariRelease`
- `anariRenderFrame`
- `anariRetain`
- `anariSetParameter`
- `ANARIStatusCallback`
- `anariUnloadLibrary`
- `anariUnmapArray`
- `anariUnmapFrame`
- `anariUnmapParameterArray`
- `anariUnsetAllParameters`
- `anariUnsetParameter`

Appendix B: Extension Index (Informative)

- `KHR_ARRAY1D_REGION`: Permit runtime changeable valid array region
- `KHR_CAMERA_DEPTH_OF_FIELD`: depth of field is supported
- `KHR_CAMERA_MOTION_TRANSFORMATION`: camera transformation motion blur is supported
- `KHR_CAMERA_OMNIDIRECTIONAL`: `omnidirectional` camera subtype is supported
- `KHR_CAMERA_ORTHOGRAPHIC`: `orthographic` camera subtype is supported
- `KHR_CAMERA_PERSPECTIVE`: `perspective` camera subtype is supported
- `KHR_CAMERA_ROLLING_SHUTTER`: camera rolling shutter is supported
- `KHR_CAMERA_SHUTTER`: camera global shutter and motion blur in combination with other extensions is supported
- `KHR_CAMERA_STEREO`: 3D stereo rendering is supported
- `KHR_DEVICE_SYNCHRONIZATION`: Relax API function synchronization requirement
- `KHR_FRAME_ACCUMULATION`: Frames are progressively rendered
- `KHR_FRAME_CHANNEL_NORMAL`: `channel.normal` is supported on `ANARIFrame`
- `KHR_FRAME_CHANNEL_ALBEDO`: `channel.albedo` is supported on `ANARIFrame`
- `KHR_FRAME_CHANNEL_PRIMITIVE_ID`: `channel.primitiveId` is supported on `ANARIFrame`
- `KHR_FRAME_CHANNEL_OBJECT_ID`: `channel.objectId` is supported on `ANARIFrame`
- `KHR_FRAME_CHANNEL_INSTANCE_ID`: `channel.instanceId` is supported on `ANARIFrame`
- `KHR_FRAME_COMPLETION_CALLBACK`: Frame completion callback function parameters on `ANARIFrame` are supported
- `KHR_GEOMETRY_CONE`: `cone` geometry subtype is supported
- `KHR_GEOMETRY_CURVE`: `curve` geometry subtype is supported
- `KHR_GEOMETRY_CYLINDER`: `cylinder` geometry subtype is supported
- `KHR_GEOMETRY_ISOSURFACE`: `isosurface` geometry subtype is supported
- `KHR_GEOMETRY_QUAD`: `quad` geometry subtype is supported
- `KHR_GEOMETRY_QUAD_MOTION_DEFORMATION`: `quad` geometry subtype supports deformation motion blur
- `KHR_GEOMETRY_SPHERE`: `sphere` geometry subtype is supported
- `KHR_GEOMETRY_TRIANGLE`: `triangle` geometry subtype is supported
- `KHR_GEOMETRY_TRIANGLE_MOTION_DEFORMATION`: `triangle` geometry subtype supports deformation motion blur
- `KHR_INSTANCE_TRANSFORM`: basic `transform` instance subtype is supported
- `KHR_INSTANCE_TRANSFORM_ARRAY`: the basic `transform` instance subtype supports matrix arrays
- `KHR_INSTANCE_MOTION_TRANSFORM`: `motionTransform` instance subtype and instance transformation motion blur is supported

- `KHR_INSTANCE_MOTION_SCALE_ROTATION_TRANSLATION`: `motionScaleRotationTranslation` instance subtype and instance transformation motion blur is supported
- `KHR_LIGHT_DIRECTIONAL`: `directional` light subtype is supported
- `KHR_LIGHT_HDRI`: `hdri` light subtype is supported
- `KHR_LIGHT_POINT`: `point` light subtype is supported
- `KHR_LIGHT_PRIMARY_VISIBILITY`: whether primary visibility of area lights can be controlled
- `KHR_LIGHT_QUAD`: `quad` light subtype is supported
- `KHR_LIGHT_RING`: `ring` light subtype is supported
- `KHR_LIGHT_SPOT`: `spot` light subtype is supported
- `KHR_MATERIAL_MATTE`: `matte` material subtype is supported
- `KHR_MATERIAL_PHYSICALLY_BASED`: `physicallyBased` material subtype is supported
- `KHR_RENDERER_AMBIENT_LIGHT`: the renderer supports ambient lighting
- `KHR_RENDERER_BACKGROUND_COLOR`: the renderer supports a background color
- `KHR_RENDERER_BACKGROUND_IMAGE`: the renderer supports a background image
- `KHR_RENDERER_DENOISE`: the renderer supports denoising
- `KHR_SAMPLER_IMAGE1D`: `image1D` sampler subtype is supported
- `KHR_SAMPLER_IMAGE2D`: `image2D` sampler subtype is supported
- `KHR_SAMPLER_IMAGE3D`: `image3D` sampler subtype is supported
- `KHR_SAMPLER_PRIMITIVE`: `primitive` sampler subtype is supported
- `KHR_SAMPLER_TRANSFORM`: `transform` sampler subtype is supported
- `KHR_SPATIAL_FIELD_FILTER_CUBIC`: spatial fields with parameter `filter` support `cubic` filtering
- `KHR_SPATIAL_FIELD_STRUCTURED_REGULAR`: `structuredRegular` spatial field subtype is supported
- `KHR_SPATIAL_FIELD_UNSTRUCTURED`: `unstructured` spatial field subtype is supported
- `KHR_VOLUME_TRANSFER_FUNCTION1D`: `transferFunction1D` volume subtype is supported

Appendix C: Example Algorithm for Rendering a TransferFunction1D Volume (Informative)

The following is a possible ray-marching implementation of the `transferFunction1D` volume. Implementations are not required to implement this but are encouraged to use it as a reference when adapting other algorithms to produce visually similar results.

```
Color = 0
Transmittance = 1

while x in Volume
    StepValue = value(x)
    StepColor = color(StepValue).rgb
    StepOpacity = color(StepValue).a * opacity(StepValue)
    StepTransmittance = pow(1 - StepOpacity, Step / unitDistance)

    Color += Transmittance * (1 - StepTransmittance) * StepColor
    Transmittance *= StepTransmittance

    x += Direction * Step
```

Any transformations the volume is subject to are assumed to be taken into account during sampling of `value(x)`.

The current sample location `x` is in world space. The `Direction` vector is the view/ray direction and is assumed to be normalized. The variables `value`, `opacity`, `color` and `unitDistance` are parameters of the `transferFunction1D` volume object.

Appendix D: Credits (Informative)

ANARI 1.0 is the result of contributions from many people and companies participating in the Khronos ANARI Working Group, as well as input from the ANARI Advisory Panel.

Members of the Working Group, including the company that they represented at the time of their most recent contribution, are listed in the following section. Some specific contributions made by individuals are listed together with their name.

Working Group Contributors to ANARI

- Bill Sherman, NIST
- Brian Savery, AMD
- Dave DeMarle, Intel
- Jakob Progsch, NVIDIA
- Jefferson Amstutz, NVIDIA
- Johannes Günther, Intel
- John Stone, NVIDIA
- Kees Van Kooten, NVIDIA
- Kevin Griffin, NVIDIA
- Neil Trevett, NVIDIA
- Mirosław Pawłowski, Intel
- Peter Messmer, NVIDIA
- Will Usher, Intel

Other Credits

The ANARI Advisory Panel members provided important real-world usage information and advice that helped guide design decisions.

Administrative support to the Working Group for ANARI 1.0 was provided by Khronos staff.

Technical support was provided by James Riordon, webmaster of Khronos.org.