ANARI™ 1.0.0 – Provisional Specification

The Khronos® ANARI Working Group

Version 1.0.0, 2021-11-02 03:15:03Z: from git branch: main commit: 25d8d34a38b761d7a0cdfb14ed32a409b4181b01

Table of Contents

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 from scientific and technical visualizations containing 3-D 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.

Scientific visualization has diverse rendering needs involving tradeoffs among quality, speed, and scalability to available hardware resources. It is typical for visualization 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 primarily to standardize emerging work found in scientific visualization, and also (where practical) to do the same for related domains, such as 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). 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 Glossary defines the basic meanings of these terms in the context of the Specification. The Specification text provides fuller definitions of the terms and may elaborate, extend, or clarify the Glossary definitions. When a term defined in the Glossary 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 is ommited 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://www.khronos.org/registry/vulkan/ .

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. 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.

Devices may relax this restriction via a core extension.

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.

2.5. Core Features, Core Extensions, and Vendor Extensions

The ANARI specification defines a core foundation of standard features and APIs that must be implemented by all devices. The definition of ANARI core features includes particular minimums for achievable implementation complexity, testing and validation efforts, and ultimately for overall application development cost.

In order to facilitate the inclusion of more advanced features beyond the core, ANARI defines both common core extensions and the semantics for working with vendor specific extensions.

Core extensions provide a well-defined software mechanism for applications and ANARI devices to negotiate the availability of standardized but optional features which ANARI implementations may choose to support. By categorizing extra ANARI functionality among multiple core extensions, implementors are free to choose to support extensions that are well matched to their core rendering algorithms, their target hardware, and their renderer design trade-offs, without taking on implementation of features that would otherwise be prohibitively complex, costly to performance, or impractical to incorporate for other reasons. The set of core extensions defined by ANARI and the syntax to determine their availability are described in section Core Extensions.

ANARI implementations may also define vendor-specific, non-standard extensions that provide additional object subtypes, parameters, and properties. These are expected to be accessible as soon as the calling application initializes the implementing device. Because specific object subtypes, parameters, and properties are all identified with strings, all extensions are accessed through standard API calls and avoid the need for dynamically loaded function call entry points.

2.6. 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

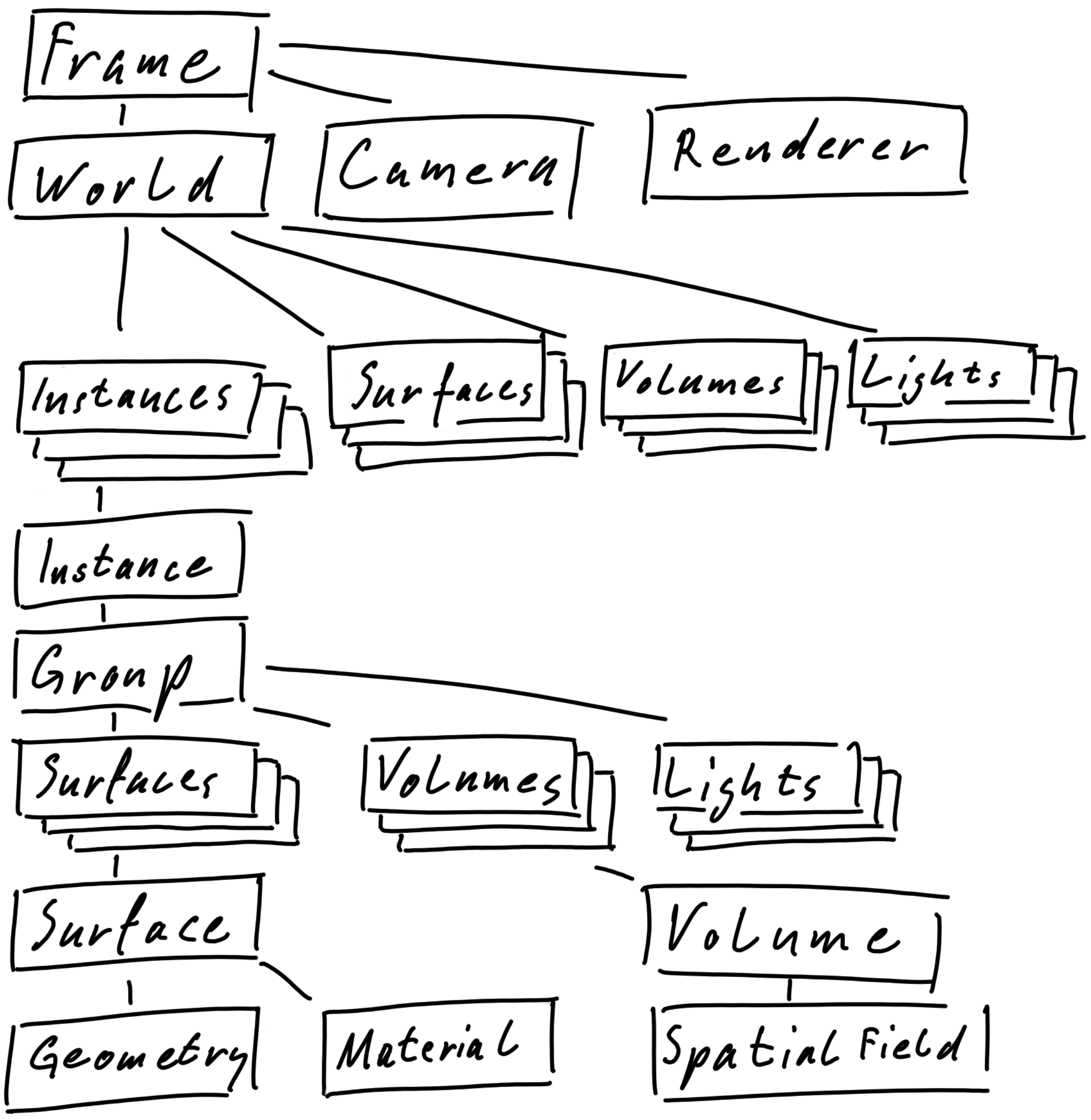
This section describes concepts common to all object types for creation, parameterization, and property queries.

3.1. 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 Instances containing Group. A Group holds arrays of Surface, Volume, and Light to be instanced together. An Instance combines a Group with a transform to 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:



3.2. 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`

3.3. 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.

3.4. 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);
```

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.

3.5. 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 device,
  ANARIObject object,
  const char *parameter,
  ANARIDataType type,
  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 anariCommit has been called on the object. Uncommitted parameters must have no effect on the behavior of the object during rendering operations. Calling anariCommit while an object is participating in a rendering operation is undefined behavior.

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 device,
  ANARIObject object,
  const char *parameter
);
```

Table 1. Data types.

Name	Closest C99-Type	Description
DATA_TYPE	ANARIDataType = int32_t	describes data types in ANARI
STRING	const char*	0-terminated string
VOID_POINTER	void*	void pointer
BOOL	int32_t	boolean represented as int32_t
FUNCTION_POINTER	void(*)(void)	generic function pointer
MEMORY_DELETER	ANARIMemoryDeleter	deleter function pointer
STATUS_CALLBACK	ANARIStatusCallback	status callback function pointer

Name	Closest C99-Type	Description
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
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

Name	Closest C99-Type	Description
`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
`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

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

Name	Closest C99-Type	Description
`UFIXED64_VEC2`	`uint64_t[2]`	two element 64 bit unsigned normalized fixed point vector
`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 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 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 componenet sRGB with linear alpha
`UFIXED8_R_SRGB`	`uint8_t[1]`	single component sRGB
`INT32_BOX1`	`int32_t[2]`	one dimensional 32 bit integer box (lower and upper bounds)
`INT32_BOX2`	`int32_t[4]`	two dimensional 32 bit integer box (lower and upper bound vector)
`INT32_BOX3`	`int32_t[6]`	three dimensional 32 bit integer box (lower and upper bound vector)
`INT32_BOX4`	`int32_t[8]`	four dimensional 32 bit integer box (lower and upper bound vector)
`FLOAT32_BOX1`	`float[2]`	one dimensional 32 bit float box (lower and upper bounds)
`FLOAT32_BOX2`	`float[4]`	two dimensional 32 bit float box (lower and upper bound vector)
`FLOAT32_BOX3`	`float[6]`	three dimensional 32 bit float box (lower and upper bound vector)

Name	Closest C99-Type	Description
FLOAT32_BOX4	float[8]	four dimensional 32 bit float box (lower and upper bound vector)
FLOAT32_MAT2	float[4]	two by two 32 bit float matrix in column major order
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_MAT2x3	float[6]	two by three 32 bit float matrix in column major order
FLOAT32_MAT3x4	float[12]	three 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].

3.5.1. Color

The following data types must be accepted by geometry's color attribute parameter and by Samplers and are grouped here to allow 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

3.6. Object Introspection

ANARI supports introspection of objects, i.e., querying supported 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. Because the meta-information of objects and parameters supported by devices is static, it can be queried without creating a device first by passing the device name.

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

To get a list of device subtypes implemented in a library, call function `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 **anariGetObjectSubtypes(
    ANARILibrary, const char *deviceSubtype, 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. Allowed `ANARIDataType` of `objectType` are

- GEOMETRY
- SAMPLER
- MATERIAL
- VOLUME
- LIGHT
- CAMERA
- RENDERER

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

const ANARIParameter *anariGetObjectParameters(ANARILibrary,
    const char *deviceSubtype,
    const char *objectSubtype,
    ANARIDataType objectType);
```

An object subtype can be queried for supported parameters with `anariGetObjectParameters`, where `objectType` is used for consistency checking and to disambiguate name collisions (object subtypes of different type with the same name). The function returns `NULL` if there are no parameters, or an array of `ANARIParameter` structs, where the last element has `NULL` as name and `ANARI_UNKNOWN` as type.

```
const void *anariGetParameterInfo(ANARILibrary,
    const char *deviceSubtype,
    const char *objectSubtype,
    ANARIDataType objectType,
    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` on error (unknown property or mismatching types). The following infos of parameters can be queried:

Table 2. 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 `minimum`..`maximum` if present
required	`BOOL`	Yes	whether the parameter must be set for an object to be valid, must be `FALSE` if a `default` is present

3.7. 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.

3.8. Arrays and Memory Ownership

`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. Input data has dimensions encoded in its handle type, where each dimension can have a sparse stride.

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

```
typedef void (*ANARIMemoryDeleter)(void *userPtr, void *appMemory);
```

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

```
ANARIArray1D anariNewArray1D(ANARIDevice,
   void *appMemory,
   ANARIMemoryDeleter,
   void *userPtr,
   ANARIDataType elementType,
   uint64_t numItems,
   uint64_t byteStride);

ANARIArray2D anariNewArray2D(ANARIDevice,
   void *appMemory,
   ANARIMemoryDeleter,
   void *userPtr,
   ANARIDataType elementType,
   uint64_t numItems1,
   uint64_t numItems2,
   uint64_t byteStride1,
   uint64_t byteStride2);

ANARIArray3D anariNewArray3D(ANARIDevice,
   void *appMemory,
   ANARIMemoryDeleter,
   void *userPtr,
   ANARIDataType elementType,
   uint64_t numItems1,
   uint64_t numItems2,
   uint64_t numItems3,
   uint64_t byteStride1,
   uint64_t byteStride2,
   uint64_t byteStride3);
```

The number of elements `numItemsN` must be positive (there cannot be an empty array object). The distance between consecutive elements (per dimension) is given in bytes with `byteStrideN`. If a `byteStrideN` is zero, it will be determined automatically (e.g., as `sizeof(type)`). Strides do not need to be ordered. For example, `byteStride2` can be smaller than `byteStride1`, which is equivalent to a transpose. However, if the stride should be calculated, then an ordering in dimensions is assumed to disambiguate, i.e., `byteStride1 < byteStride2 < byteStride3`.

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. Device managed arrays can only have dense layouts (`byteStrideN = 0`).

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 an array is created with a deleter callback, it is implementation defined when an implementation frees host memory after the array object has been released.

Applications are only permitted to write to the memory visible to `ANARIArray` objects if all array objects involved are mapped. Mapping an 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.

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.

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. The byte strides of arrays allow such "transformation" without changing the actual memory layout. An example are the vertex attributes of the triangle geometry, which all start with `vertex.` followed by the attribute name.

3.9. 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,
  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.

3.10. 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 implemetation. 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 3. Properties queryable on devices.

Name	Type	Required	Description
version	INT32	Yes	unique version number guaranteed to increase between versions
version.major	INT32	No	semantic version major value (major.minor.patch)
version.minor	INT32	No	semantic version minor value (major.minor.patch)
version.patch	INT32	No	semantic version patch value (major.minor.patch)
version.name	STRING	No	human readable name/title
geometryMaxIndex	UINT64	Yes	largest supported index into vertex arrays in geometries

3.11. 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)(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 ANARIStatusSeverity are defined:

- SEVERITY_FATAL_ERROR

- `SEVERITY_ERROR`
- `SEVERITY_WARNING`
- `SEVERITY_PERFORMANCE_WARNING`
- `SEVERITY_INFO`
- `SEVERITY_DEBUG`

Table 4. Parameters accepted 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.

3.12. Attributes

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

Surface attributes are passed from geometries and volumes 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 or volume. Unspecified (components of) attributes default to zero for the first three components and to one for the fourth component.

Table 5. Attributes

Identifier	Internal Type	Description
`color`	`FLOAT32_VEC4`	color
`worldPosition`	`FLOAT32_VEC4`	world space position
`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. Object Types

This section describes the base object types that make up a scene in ANARI. An overview is that objects are connected starting from the bottom-up: leaf objects like geometries, volumes and lights are instanced in space via instance objects, which are in turn added to the top-level root object, the world.

4.1. 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);
```

For supported subtypes see Section Geometries.

Note

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

4.2. 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);
```

For supported subtypes see Section Samplers.

4.3. Material

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

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

For supported subtypes see Section Materials.

4.4. 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 6. Parameters understood by Surface.

Name	Type	Description
geometry	`GEOMETRY`	Geometry object used by this surface
material	`MATERIAL`	optional Material applied to the geometry

Surfaces require a valid Geometry to be set as the `geometry` parameter.

4.5. 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);
```

For supported subtypes see Section Spatial Fields.

4.6. 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);
```

For supported subtypes see Section Volumes.

4.7. 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);
```

For supported subtypes see Section Lights.

4.8. Group

Groups in ANARI represent collections of Surface, Volume, and Light 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 7. 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 8. Properties supported by groups.

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

4.9. Instance

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

```
ANARIInstance anariNewInstance(ANARIDevice);
```

Instances take an array of Group representing all the objects which share the instances world-space transform.

Table 9. Parameters understood by instances.

Name	Type	Default	Description
group	GROUP		Group to be instanced, required
transform	FLOAT32_MAT3x4	((1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 0, 0))	world-space transformation matrix for all attached objects (overridden by motion.transform, extension KHR_TRANSFORMATION_MOTION_BLUR)

Table 10. Properties supported by instances.

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

Extension KHR_TRANSFORMATION_MOTION_BLUR: Uniformly (in time) distributed transformation keys can be given with motion.transform to achieve transformation motion blur (in combination with time and shutter). Alternatively, the transformation keys can also be given as decomposed motion.scale, motion.rotation and motion.translation. Camera shutter also needs to be set appropiately.

Table 11. Additional parameters understood by instances with extension KHR_TRANSFORMATION_MOTION_BLUR.

Name	Type	Default	Description
motion.transform	ARRAY1D of FLOAT32_MAT3x4		additional uniformly distributed world-space transformations
motion.scale	ARRAY1D of FLOAT32_VEC3		additional uniformly distributed scale, overridden by motion.transform
motion.rotation	ARRAY1D of FLOAT32_QUAT_IJKW		additional uniformly distributed quaternion rotation, overridden by motion.transform
motion.translation	ARRAY1D of FLOAT32_VEC3		additional uniformly distributed transformlation, overridden by motion.transform
time	FLOAT32_BOX1	[0, 1]	time associated with first and last key in motion.* arrays

4.10. 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 12. Parameters understood by the world.

Name	Type	Description
instance	ARRAY1D of INSTANCE	optional array with handles of instances
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 13. Properties supported by the world.

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

4.11. Camera

To create a new camera of given type subtype use

```
ANARICamera anariNewCamera(ANARIDevice, const char *subtype);
```

For supported subtypes see Section Cameras.

4.12. Renderer

A renderer is the central object for rendering in ANARI. Different renderers implement different features, 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.

4.13. 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.

Table 14. Parameters accepted by the frame.

Name	Type	Description
world	`WORLD`	required world to be rendererd
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)
color	`DATA_TYPE`	enable mapping the `color` channel and specify its observable type; RGB color including alpha; possible values: `UFIXED8_VEC4`, `UFIXED8_RGBA_SRGB`, `FLOAT32_VEC4`
depth	`DATA_TYPE`	enable mapping the `depth` channel and specify its observable type; euclidean distance to the camera (*not* to the image plane), for multiple samples per pixel their minimum is taken; possible values: `FLOAT32`

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

Table 15. Additional parameters accepted by the frame with extension `KHR_FRAME_COMPLETION_CALLBACK`.

Name	Type	Description
frameCompletionCallback	`FRAME_COMPLETION_CALLBACK`	callback to invoke as continuation when the rendered frame is complete
frameCompletionCallbackUserData	`VOID_POINTER`	optional user pointer passed as the first argument of the frame completion callback

Table 16. Additional parameters accepted by the frame with extension `KHR_AUXILIARY_BUFFERS`.

Name	Type	Description
normal	`DATA_TYPE`	enable mapping the `normal` channel and specify its observable type; average world-space normal of the first hit; possible values: `FIXED16_VEC3`, `FLOAT32_VEC3`
albedo	`DATA_TYPE`	enable mapping the `albedo` channel and specify its observable type; average material albedo (color without illumination) at the first hit; possible values: `UFIXED8_VEC3`, `UFIXED8_RGB_SRGB`, `FLOAT32_VEC3`

Table 17. Additional parameters accepted by the frame with extension `KHR_STOCHASTIC_RENDERING`.

Name	Type	Default	Description
accumulation	`BOOL`	`FALSE`	whether additional internal buffers are created to potentially improve the image quality when multiple subsequent calls to `anariRenderFrame` are made
variance	`BOOL`	`FALSE`	whether to estimate the `variance` property

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);
```

Only channels that have been set as parameters to the frame can be mapped, the type of the pixels matches the corresponding parameter value.

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);
```

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

Table 18. Properties queryable on frames.

Name	Type	Required	Description
duration	`FLOAT32`	Yes	time between start and completion of the frame

Table 19. Additional properties queryable on frames with extension `KHR_STOCHASTIC_RENDERING`.

Name	Type	Required	Description
progress	`FLOAT32`	No	progress of rendering the frame, in [0..1]
variance	`FLOAT32`	No	estimated variance, can be used as a quality indicator how well the image converged (and thus whether to continue progressive rendering); only computed when enabled via `variance` parameter

4.14. Extension Objects

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

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

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

Chapter 5. 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. If a call to `anariCommit(ANARIDevice)` happens while a frame is being rendered, the result is undefined behavior and should be avoided.

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.

Chapter 6. Standard Subtypes

6.1. Geometries

This section outlines geometry subtypes which are supplied by complete ANARI device implementations.

All geometries support the following attribute setting parameters.

Table 20. Parameters accepted by all geometries.

Name	Type	Description
primitive.color	`ARRAY1D` of Color	primitive color attribute
primitive.attribute0	`ARRAY1D` of `FLOAT32` / `FLOAT32_VEC2` / `FLOAT32_VEC3` / `FLOAT32_VEC4`	primitive attribute0
primitive.attribute1	`ARRAY1D` of `FLOAT32` / `FLOAT32_VEC2` / `FLOAT32_VEC3` / `FLOAT32_VEC4`	primitive attribute1
primitive.attribute2	`ARRAY1D` of `FLOAT32` / `FLOAT32_VEC2` / `FLOAT32_VEC3` / `FLOAT32_VEC4`	primitive attribute2
primitive.attribute3	`ARRAY1D` of `FLOAT32` / `FLOAT32_VEC2` / `FLOAT32_VEC3` / `FLOAT32_VEC4`	primitive attribute3
primitive.id	`ARRAY1D` of `UINT32` / `UINT64`	per-primitive re-mapping of the `primitiveId` attribute

Values in `primitive.id` must be at most device limit `geometryMaxIndex`. If geometries have additional per-vertex parameters specifying the same attribute, then the geometry specific `vertex.*` parameters take precedence.

6.1.1. Triangle

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

Table 21. Parameters defining a triangle geometry.

Name	Type	Default	Description
vertex.position	`ARRAY1D` of `FLOAT32_VEC3`		required vertex positions

Name	Type	Default	Description
vertex.normal	ARRAY1D of FIXED16_VEC3 / FLOAT32_VEC3		vertex normals
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_VEC3 / UINT64_VEC3	[(0, 1, 2), (4, 5, 6), …]	optional indices (into the vertex array(s)), each 3-tupel defines one triangle

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

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`.

6.1.2. Quad

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

Table 22. Parameters defining a quad geometry.

Name	Type	Default	Description
vertex.position	ARRAY1D of FLOAT32_VEC3		required vertex positions

Name	Type	Default	Description
vertex.normal	`ARRAY1D` of `FIXED16_VEC3` or `FLOAT32_VEC3`		vertex normals
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_VEC4` / `UINT64_VEC4`	[(0, 1, 2, 4), …]	optional indices (into the vertex array(s)), each 4-tupel defines one quad

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

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`.

6.1.3. 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 23. Parameters defining a 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 `sphere.radius` 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`.

6.1.4. 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 24. Parameters defining a curve geometry.

Name	Type	Default	Description
vertex.position	ARRAY1D of FLOAT32_VEC3		required vertex positions
vertex.radius	ARRAY1D of FLOAT32		per-vertex 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 `vertex.*` arrays, each index defines the start of one segment of a curve (the end is implicitly given with index+1)
radius	FLOAT32	1	default radius for all curve vertices (if `vertex.radius` 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, conneting 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.

6.1.5. 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 25. Parameters defining a 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 `vertex.*` arrays, each 2-tuple defines one cone
caps	STRING	`none`	default vertex caps for all cones if `vertex.cap` is not set, possible values: `none`, `first`, `second`, `both`

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`.

6.1.6. 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 26. Parameters defining a cylinder geometry.

Name	Type	Default	Description
vertex.position	`ARRAY1D` of `FLOAT32_VEC3`		required center positions of the cylinders
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 `vertex.*` arrays, each 2-tuple defines one cylinder
primitive.radius	`ARRAY1D` of `FLOAT32`		per-cylinder radius
radius	`FLOAT32`	1	default radius for all cylinders (if `cylinder.radius` is not set)
caps	`STRING`	`none`	default vertex caps for all cylinders if `vertex.cap` is not set, possible values: `none`, `first`, `second`, `both`

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.

6.2. Samplers

This section outlines sampler subtypes which are supplied by complete ANARI device implementations.

6.2.1. Image1D

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

Table 27. Parameters accepted by image1D samplers.

Name	Type	Default	Description
inAttribute	STRING	attribute0	surface attribute used as texture coordinate, possible values: float Attributes
inTransform	FLOAT32_MAT4x4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the input transform before sampling
image	ARRAY1D of Color		array backing the sampler
filter	STRING	nearest	filter of the sampler, possible values: nearest, linear
wrapMode	STRING	clampToEdge	wrap mode of the sampler, possible values: clampToEdge, repeat, mirrorRepeat
outTransform	FLOAT32_MAT4x4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the sampled values

The attribute values indicated by inAttribute are first transformed via inTransform, 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) and transformed via outTransform to yield the final result of the sampler.

6.2.2. Image2D

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

Table 28. Parameters accepted by `image2D` samplers.

Name	Type	Default	Description
inAttribute	`STRING`	`attribute0`	surface attribute used as texture coordinate, possible values: float Attributes
inTransform	`FLOAT32_MAT4x4`	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the input transform before sampling
image	`ARRAY2D` of Color / `FIXED8_VEC3` / `FIXED8_VEC4`		array backing the sampler
filter	`STRING`	`nearest`	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_MAT4x4`	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the sampled values

The attribute values indicated by `inAttribute` are first transformed via `inTransform`, then its first two components are used as normalized coordinates 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) and transformed via `outTransform` to yield the final result of the sampler.

6.2.3. Image3D

A three dimensional image sampler is created by calling `anariNewSampler` with subtype string `image3D`. It accepts the following parameters.

Table 29. Parameters accepted by `image3D` samplers.

Name	Type	Default	Description
inAttribute	`STRING`	`attribute0`	surface attribute used as texture coordinate, possible values: float Attributes
inTransform	`FLOAT32_MAT4x4`	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the input transform before sampling

Name	Type	Default	Description
image	ARRAY3D of Color		array backing the sampler
filter	STRING	nearest	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_MAT4x4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to the sampled values

The attribute values indicated by inAttribute are first transformed via inTransform, then its first three components are used as normalized coordinates 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) and transformed via outTransform to yield the final result of the sampler.

6.2.4. Primitive

The primitive sampler samples an Array1D at unnormalized integer coordinates based on the primitiveId surface attribute, shifted by non-negative offset.

Table 30. Parameters accepted by all primitive samplers.

Name	Type	Description
array	ARRAY1D	backing array of the sampler

The array size must be at most device limit geometryMaxIndex.

6.2.5. Transform

The transform sampler applies a transform to the input attribute and returns the result as the sampled value.

Table 31. Parameters accepted by all primitive samplers.

Name	Type	Default	Description
inAttribute	STRING	attribute0	surface attribute used as texture coordinate

Name	Type	Default	Description
transform	FLOAT32_MAT4x4	((1, 0, 0, 0), (0, 1, 0, 0), (0, 0, 1, 0), (0, 0, 0, 1))	transform applied to coordinates before sampling

6.3. Materials

This section outlines material subtypes which are supplied by complete ANARI device implementations.

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.

6.3.1. 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 is always treated as entirely opaque.

Table 32. Parameters of the matte material.

Name	Type	Default	Description
color	FLOAT32_VEC3 / SAMPLER / STRING	(0.8, 0.8, 0.8)	diffuse color

6.3.2. Transparent Matte

The `transparentMatte` material extends the Matte material by adding support for (partial) cut-out transparency.

Table 33. Parameters of the transparentMatte 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

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.

6.4. Spatial Fields

This section outlines spatial field subtypes which are supplied by complete ANARI device implementations.

6.4.1. 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 34. Configuration parameters for structured regular spatial fields.

Name	Type	Default	Description
data	`ARRAY3D` of `UINT8` / `INT16` / `UINT16` / `FLOAT32` / `FLOAT64`		the actual field values for the 3D grid, i.e., the scalars are vertex-centered; the size and value type of the spatial field is inferred from `data`
origin	`FLOAT32_VEC3`	(0, 0, 0)	origin of the grid in object-space
spacing	`FLOAT32_VEC3`	(1, 1, 1)	size of the grid cells in object-space
filter	`STRING`	`linear`	filter used for reconstructing the field, possible values: `nearest`, `linear`

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`].

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.

6.5. Volumes

This section outlines volume subtypes which are supplied by complete ANARI device implementations.

6.5.1. Scivis

The scivis volume is created by passing the subtype string `scivis` to `anariNewVolume`. It supports the folowing parameter:

Table 35. Parameters understood by the scivis volume.

Name	Type	Default	Description
field	`SPATIAL_FIELD`		Spatial field used for the field values of the volume
valueRange	`FLOAT32_BOX1`	[0, 1]	sampled values of `field` are clamped to this range
color	`ARRAY1D` of Color		array to map sampled and clamped field values to color

Name	Type	Default	Description
color.position	ARRAY1D of FLOAT32		optional array to position the elements of color values in valueRange
opacity	ARRAY1D of FLOAT32		array to map sampled and clamped field values to opacity
opacity.position	ARRAY1D of FLOAT32		optional array to position the elements of opacity values in valueRange
densityScale	FLOAT32	1	makes volumes uniformly thinner or thicker

The color and the opacity array map values sampled in field (clamped to valueRange) into output colors and opacities used for volume rendering, respectively.

Note

The color and opacity arrays together represent a transfer function, which is used to visually emphasize the structure or certain features in the field data. Both arrays can be of different size.

If present, color.position and opacity.position arrays must contain monotonically increasing values within (inclusive) valueRange and their array size must not be larger than the corresponding color or opacity array. The *.position arrays contain the numeric position which the elements of corresponding color or opacity reside within valueRange.

If a *.position array is not set, the position values are assumed to be uniformly distributed within valueRange, with the first element representing the color/opacity of the lowest value in valueRange, and the last element representing the color/opacity of the last element in valueRange.

6.6. Lights

This section outlines light subtypes which are supplied by complete ANARI device implementations.

All light sources accept the following parameters:

Table 36. Parameters accepted by all lights.

Name	Type	Default	Description
color	FLOAT32_VEC3	(1, 1, 1)	color of the light, in 0..1

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

Table 37. Additional parameters accepted by all lights with extension KHR_AREA_LIGHTS.

Name	Type	Default	Description
visible	BOOL	TRUE	whether the light can be directly seen

6.6.1. 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 38. Special parameters accepted by the directional light.

Name	Type	Default	Description
direction	`FLOAT32_VEC3`	(0, 0, -1)	main emission direction of the directional light
irradiance	`FLOAT32`	1	the amount of light arriving at a surface point, assuming the light is oriented towards to the surface, in W/m^2

Table 39. Additional parameters accepted by the directional light with extension `KHR_AREA_LIGHTS`.

Name	Type	Default	Description
angularDiameter	`FLOAT32`	0	apparent size (angle in radians) of the light
radiance	`FLOAT32`	1	the amount of light emitted in a direction, in W/sr/m^2; `irradiance` takes precedence if also specified

Note

Extension `KHR_AREA_LIGHTS`: Setting the angular diameter to a value greater than zero will result in soft shadows when supported by the implementation. For instance, the apparent size of the sun is about 0.53° or 0.00925 rad.

6.6.2. Point

The point light (or with extension 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 40. Special parameters accepted by the point light.

Name	Type	Default	Description
position	`FLOAT32_VEC3`	(0, 0, 0)	the position of the point light
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

Extension `KHR_AREA_LIGHTS`: Setting the radius to a value greater than zero will result in soft shadows when the renderer support it.

Table 41. Additional parameters accepted by the point light with extension `KHR_AREA_LIGHTS`.

Name	Type	Default	Description
radius	`FLOAT32`	0	the size of the sphere light
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

6.6.3. 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 42. Special parameters accepted 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`	1	the overall amount of light energy emitted, in W; `intensity` takes precedence if also specified

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

6.6.4. Ring (Extension `KHR_AREA_LIGHTS`)

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 43. Special parameters accepted 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

Name	Type	Default	Description
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–π]; 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 [0–2π]; 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`.

6.6.5. Quad (Extension KHR_AREA_LIGHTS)

The quad[1] light is a planar, procedural area light source 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 44. Special parameters accepted 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–π]; 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 [0–2π]; 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` × `edge1`.

Note

Only renderers that use stochastic sampling (like the path tracer) will compute soft shadows from the quad light. Other renderers will just sample the center of the quad light, which results in hard shadows.

6.6.6. HDRI (Extension KHR_AREA_LIGHTS)

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 45. Special parameters accepted 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`	`equirectangular`	possible values: `equirectangular`
scale	`FLOAT32`	1	scale factor for `radiance`

6.7. Cameras

This section outlines camera subtypes which are supplied by complete ANARI device implementations.

All cameras sources accept the following parameters:

Table 46. Parameters accepted by all 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
transform	`FLOAT32_MAT3x4`	((1, 0, 0), (0, 1, 0), (0, 0, 1), (0, 0, 0))	additional world-space transformation matrix (overridden by `motion.transform`, extension `KHR_TRANSFORMATION_MOTION_BLUR`)
imageRegion	`FLOAT32_BOX2`	((0, 0), (1, 1))	region of the sensor in normalized screen-space coordinates
apertureRadius	`FLOAT32`	0	size of the aperture, controls the depth of field
focusDistance	`FLOAT32`	1	distance at where the image is sharpest when depth of field is enabled
stereoMode	`STRING`	`none`	for stereo rendering, possible values: `none`, `left`, `right`, `sideBySide`, `topBottom` (left eye at top half)

Name	Type	Default	Description
interpupillaryDistance	FLOAT32	0.0635	distance between left and right eye when stereo is enabled

The camera is placed and oriented in the world with position, direction and up. Additionally, an extra transformation transform can be specified, which will only be applied to 3D vectors (i.e. position, direction and up), but does *not* affect any sizes (e.g., apertureRadius or interpupillaryDistance). 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.

All cameras support depth of field via the apertureRadius and focusDistance parameters and stereo 3D rendering by setting stereoMode and interpupillaryDistance.

Table 47. Additional parameters accepted by all cameras with extension KHR_TRANSFORMATION_MOTION_BLUR.

Name	Type	Default	Description
motion.transform	ARRAY1D of FLOAT32_MAT3x4		additional uniformly distributed world-space transformations
motion.scale	ARRAY1D of FLOAT32_VEC3		additional uniformly distributed scale, overridden by motion.transform
motion.rotation	ARRAY1D of FLOAT32_QUAT_IJKW		additional uniformly distributed quaternion rotation, overridden by motion.transform
motion.translation	ARRAY1D of FLOAT32_VEC3		additional uniformly distributed transformlation, overridden by motion.transform
time	FLOAT32_BOX1	[0, 1]	time associated with first and last key in motion.* arrays
shutter	FLOAT32_BOX1	[0.5, 0.5]	start and end of shutter time

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

6.7.1. Perspective Camera

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 48. Additional parameters accepted by the perspective camera.

Name	Type	Default	Description
fovy	`FLOAT32`	π/3	the field of view of the frame's height
aspect	`FLOAT32`	1	ratio of width by height of the frame (and image region)

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

6.7.2. Orthographic Camera

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 49. Additional parameters accepted by the orthographic camera.

Name	Type	Default	Description
aspect	`FLOAT32`	1	ratio of width by height of the frame (and image region)

Size of the camera sensor, and thus the extent of the scene that is captured in the image, can be controlled with the `imageRegion` parameter. The `aspect` ratio needs to be set accordingly to get an undistorted image.

6.7.3. Omnidirectional Camera

The omnidirectional camera captures the complete surrounding It is 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 50. Additional parameters accepted by the omnidirectional camera.

Name	Type	Default	Description
layout	`STRING`	`equirectangular`	pixel layout, possible values: `equirectangular`

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

[1] actually a parallelogram

Chapter 7. Core Extensions

This section describes the core extensions which ANARI specifies. The various ANARI implementing devices may or may not support any of the features described here. Applications can check whether a specific feature is supported after initializing a device by calling.

```
int anariDeviceImplements(ANARIDevice, const char *extension);
```

Where `extension` is a string name of the extension. Preprocessor defines are also declared in the `anari_enums.h` with a value for each of the features describe here.

7.1. Frame Completion Callback

Devices which implement `ANARI_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 complete before the `ANARI_TASK_COMPLETE` event is signaled by `anariWait()` on the frame.

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

```
typedef void (*ANARIFrameCompletionCallback)(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 `anariWait()` with `ANARI_TASK_COMPLETE`, as this will incur a deadlock.

7.2. Area Lights

The extension `ANARI_KHR_AREA_LIGHTS` indicates that implementations support spatially extended Lights and thus potentially soft shadows.

7.3. Relaxed Device Synchronization

Devices that implement `ANARI_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 as before.

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

7.4. Transformation Motion Blur

The extension `ANARI_KHR_TRANSFORMATION_MOTION_BLUR` indicates that implementations support motion blur via time-varying transformations on instances and cameras.

7.5. Auxiliary Buffers

The extension `ANARI_KHR_AUXILIARY_BUFFERS` indicates that implementations support additional albedo and normal buffers on the Frame which can be mapped.

7.6. Stochastic Rendering

The extension `ANARI_KHR_STOCHASTIC_RENDERING` indicates that implementations use stochastic rendering techniques like Monte Carlo integration and thus that multiple subsequent calls to `anariRenderFrame` without changes to Frame improve the image quality (e.g., reduce noise levels). Additional properties can be queried on Frame.

Glossary

The terms defined in this section are used consistently throughout this Specification and may be used with or without capitalization.

Attributes

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

Device

The object implementing the ANARI API, which is used as the first parameter in nearly all API calls.

Frame

Contains all the objects necessary to render, and holds the resulting rendered 2D image (and optionally auxilliary information associated with pixels).

Sampler

Maps attributes into other object inputs such as materials.

Appendix A: 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, UIUC
- Kees Van Kooten, NVIDIA
- Kevin Griffin, NVIDIA
- Neil Trevett, NVIDIA
- 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.