ANARI™

SIGGRAPH 2024 BoF

KHRONOS GROUP

3D APPLICATIONS

ParaView

blender®

VMD
Visual Molecular Dynamics

…

RENDERING ENGINES

Intel® OSPRay

AMD Radeon™ ProRender

NVIDIA VisRTX

Cycles Open Source Production Rendering

…

KHRONOS® GROUP

3D APPLICATIONS

ParaView

blender®

VMD
Visual Molecular Dynamics

...

RENDERING ENGINES

Intel® OSPRay

AMD Radeon™ ProRender

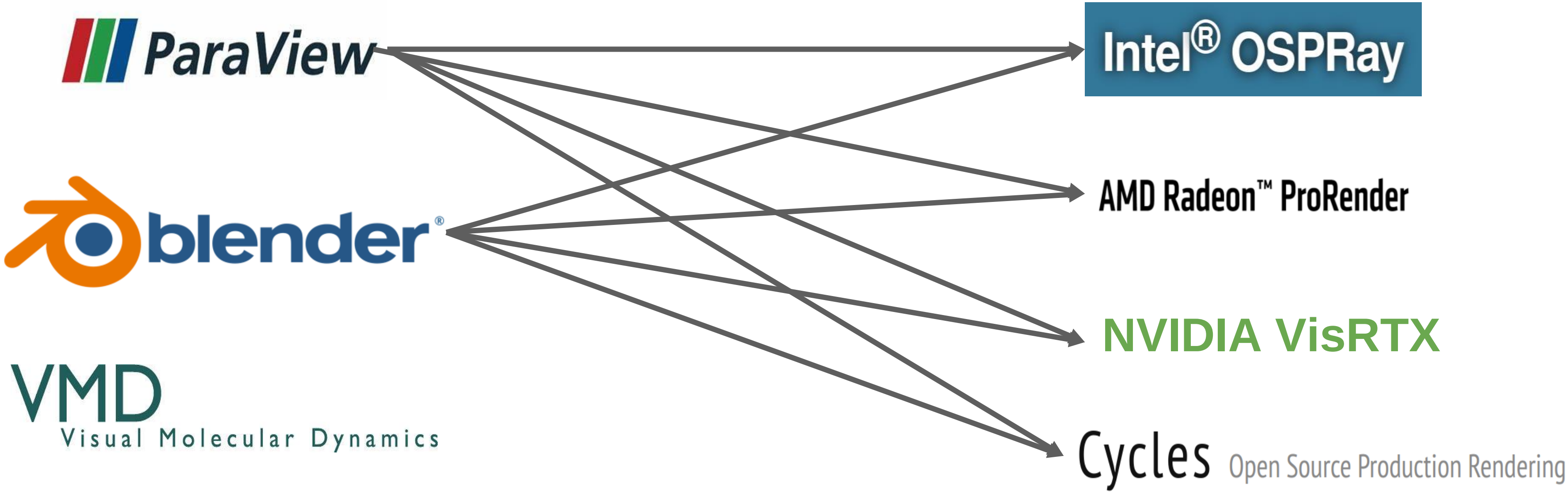
NVIDIA VisRTX

Cycles Open Source Production Rendering

...

3D APPLICATIONS

RENDERING ENGINES

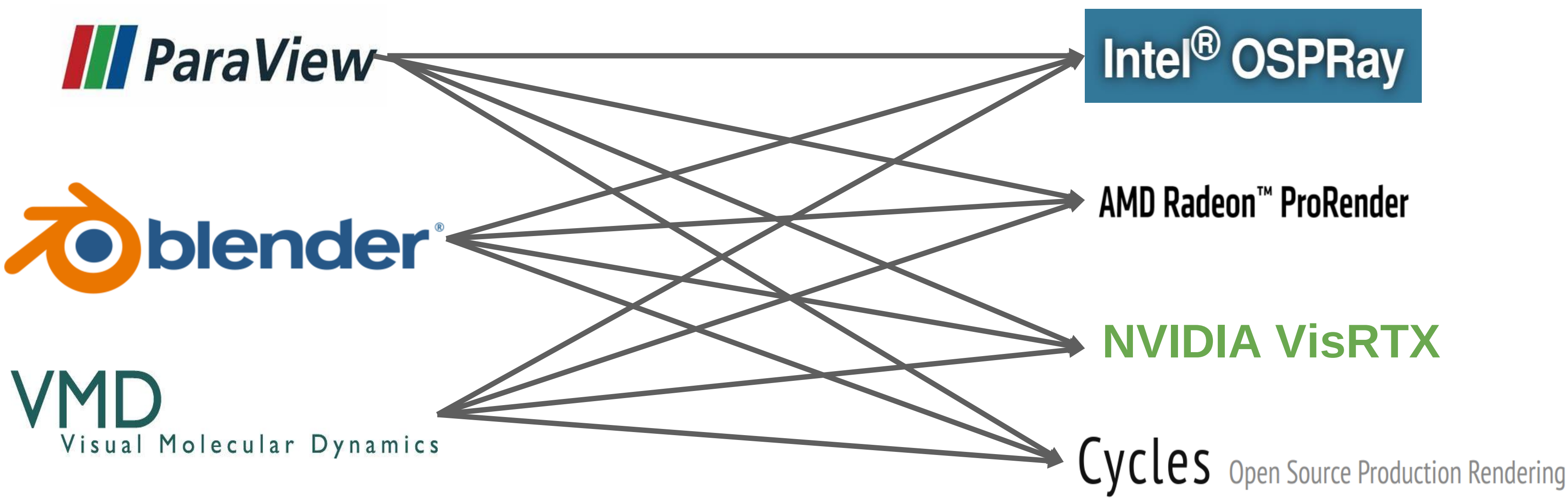


...

...

3D APPLICATIONS

RENDERING ENGINES



...

...

3D APPLICATIONS

ParaView

blender®

VMD
Visual Molecular Dynamics

...

ANARI™

RENDERING ENGINES

Intel® OSPRay

AMD Radeon™ ProRender

NVIDIA VisRTX

Cycles Open Source Production Rendering

...

3D APPLICATIONS

ParaView

blender®

VMD Visual Molecular Dynamics

...

ANARI™

"...for the faint of heart"

RENDERING ENGINES

Intel® OSPRay

AMD Radeon™ ProRender

NVIDIA VisRTX

Cycles Open Source Production Rendering

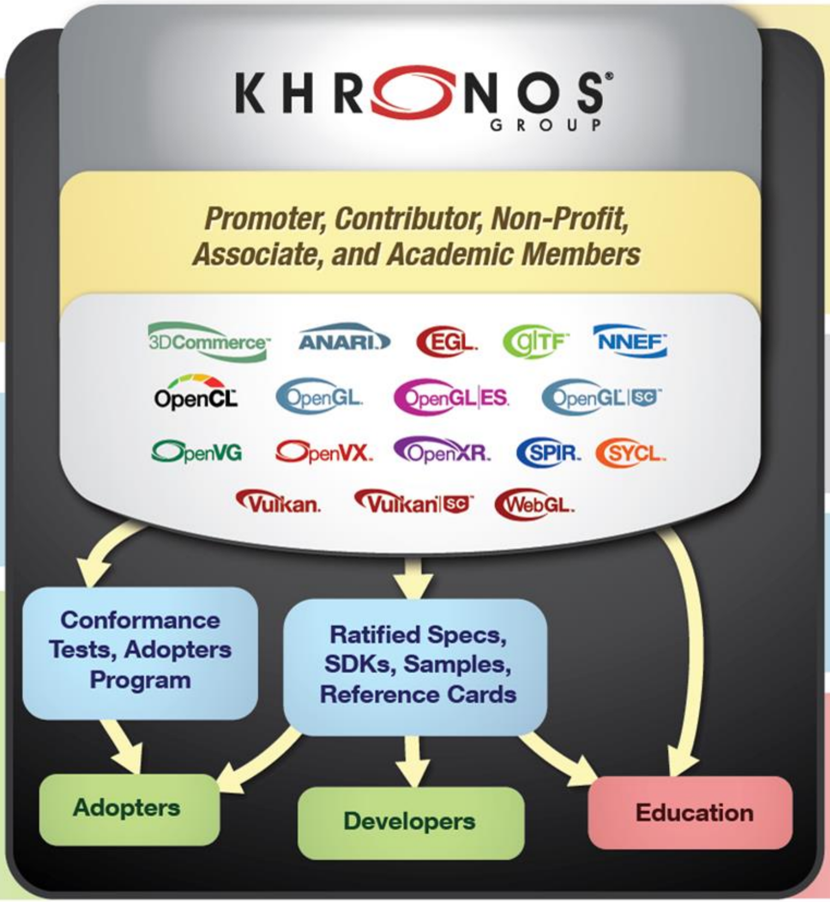
...

Promoter Members
Participate and vote in Working Groups, Board seat for setting strategy and budget

Conformance is Key
Comprehensive testing frameworks available

Adopters
Build conformant implementations

Developers
Freely develop software using Khronos standards



Contributor Members
Participate & vote in Working Groups

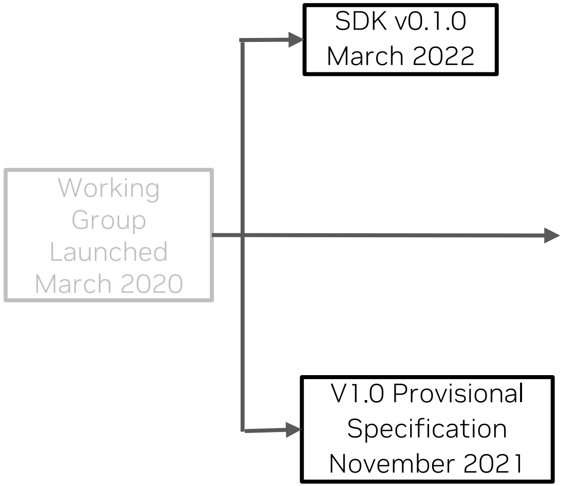
Non-Profit, Associate, and Academic Members
Participate in Working Groups

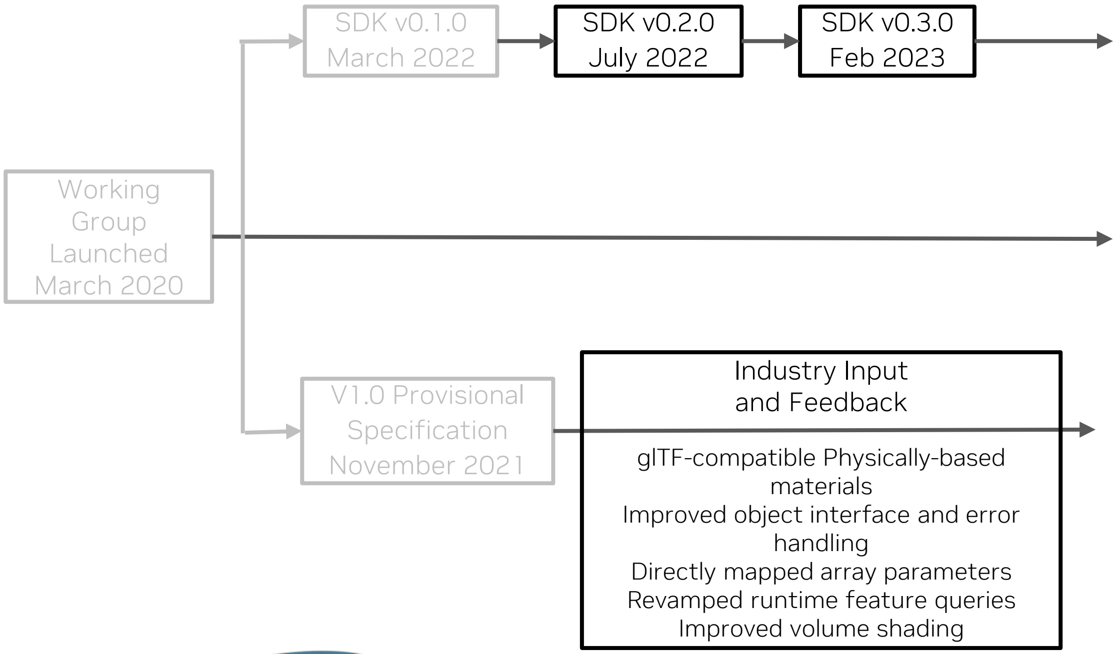
Working Groups
For each Standard, open to all members

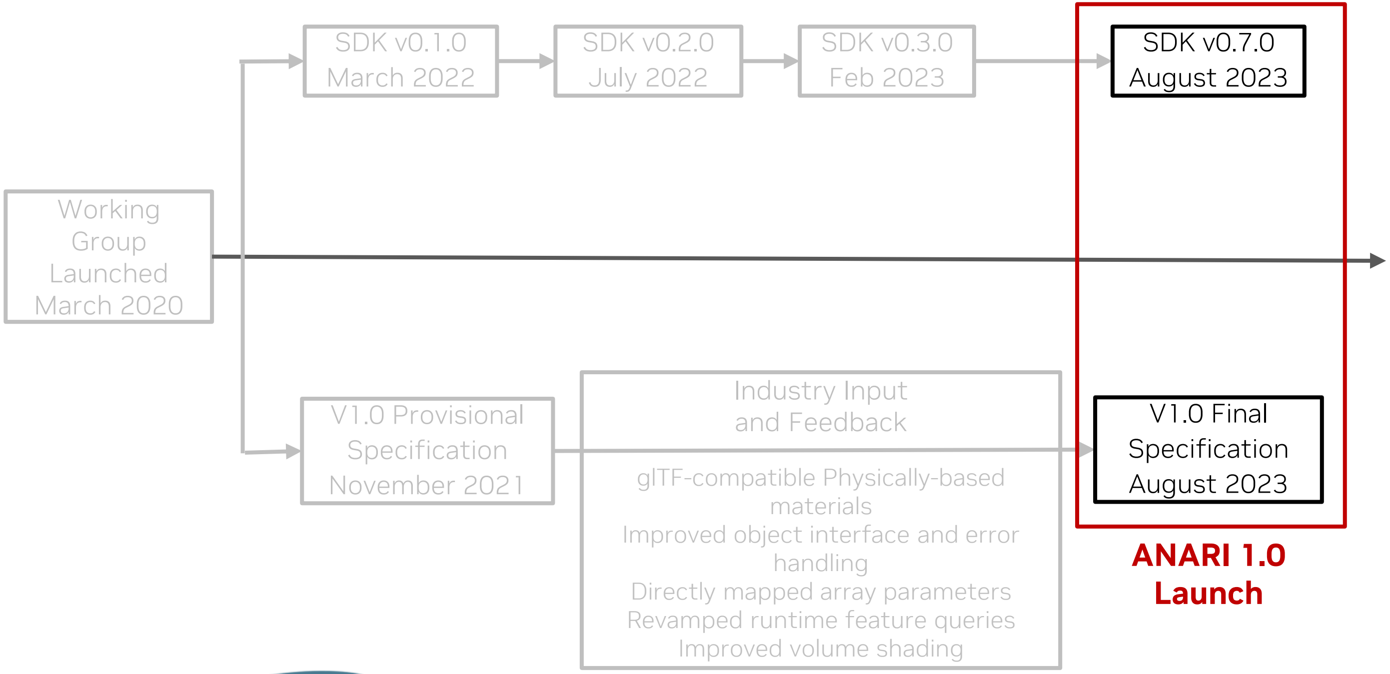
Specifications & Learning Materials
Public & free of charge

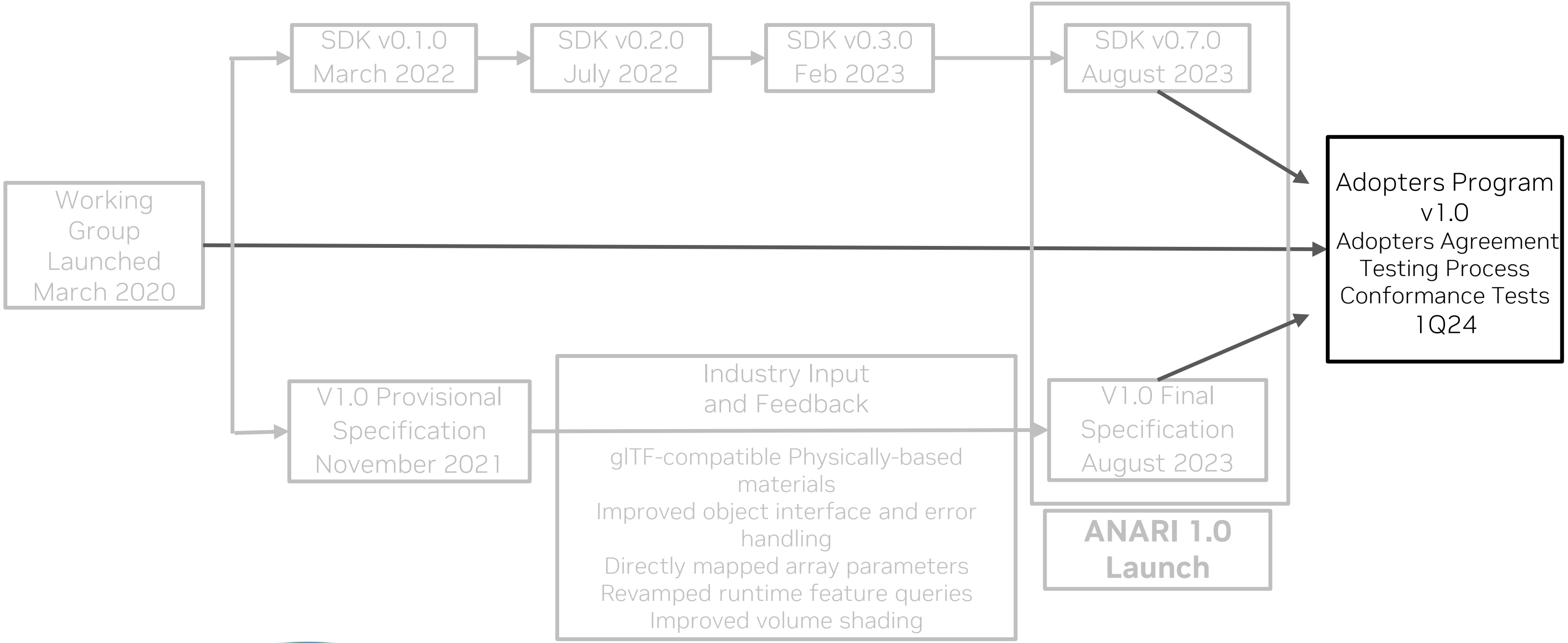
Ecosystem
Samples, tools, webinars, tutorials, meetups

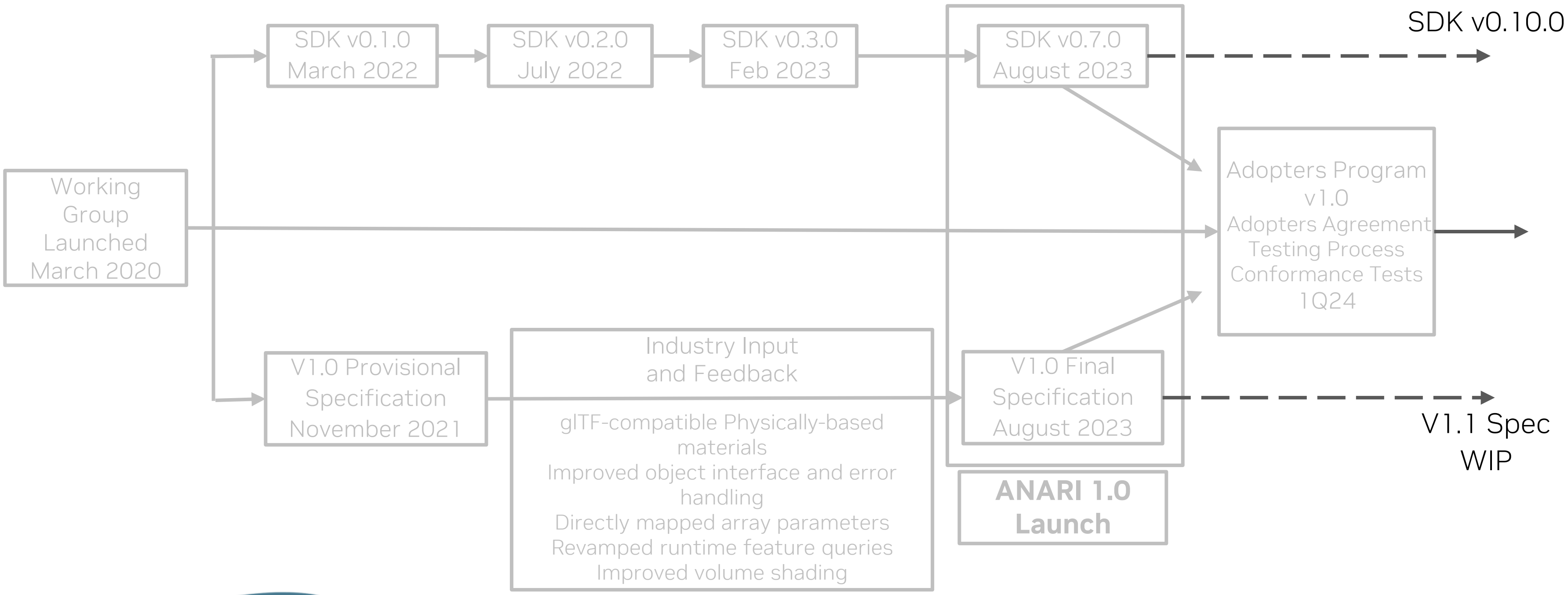
Working Group Launched March 2020


SDK v0.1.0
March 2022
Working Group Launched March 2020
V1.0 Provisional Specification
November 2021


SDK v0.1.0
March 2022
SDK v0.2.0
July 2022
SDK v0.3.0
Feb 2023
Working
Group
Launched
March 2020
V1.0 Provisional
Specification
November 2021
Industry Input
and Feedback
glTF-compatible Physically-based
materials
Improved object interface and error
handling
Directly mapped array parameters
Revamped runtime feature queries
Improved volume shading


SDK v0.1.0
March 2022
SDK v0.2.0
July 2022
SDK v0.3.0
Feb 2023
SDK v0.7.0
August 2023
Working
Group
Launched
March 2020
V1.0 Provisional
Specification
November 2021
Industry Input
and Feedback
glTF-compatible Physically-based
materials
Improved object interface and error
handling
Directly mapped array parameters
Revamped runtime feature queries
Improved volume shading
V1.0 Final
Specification
August 2023
ANARI 1.0
Launch

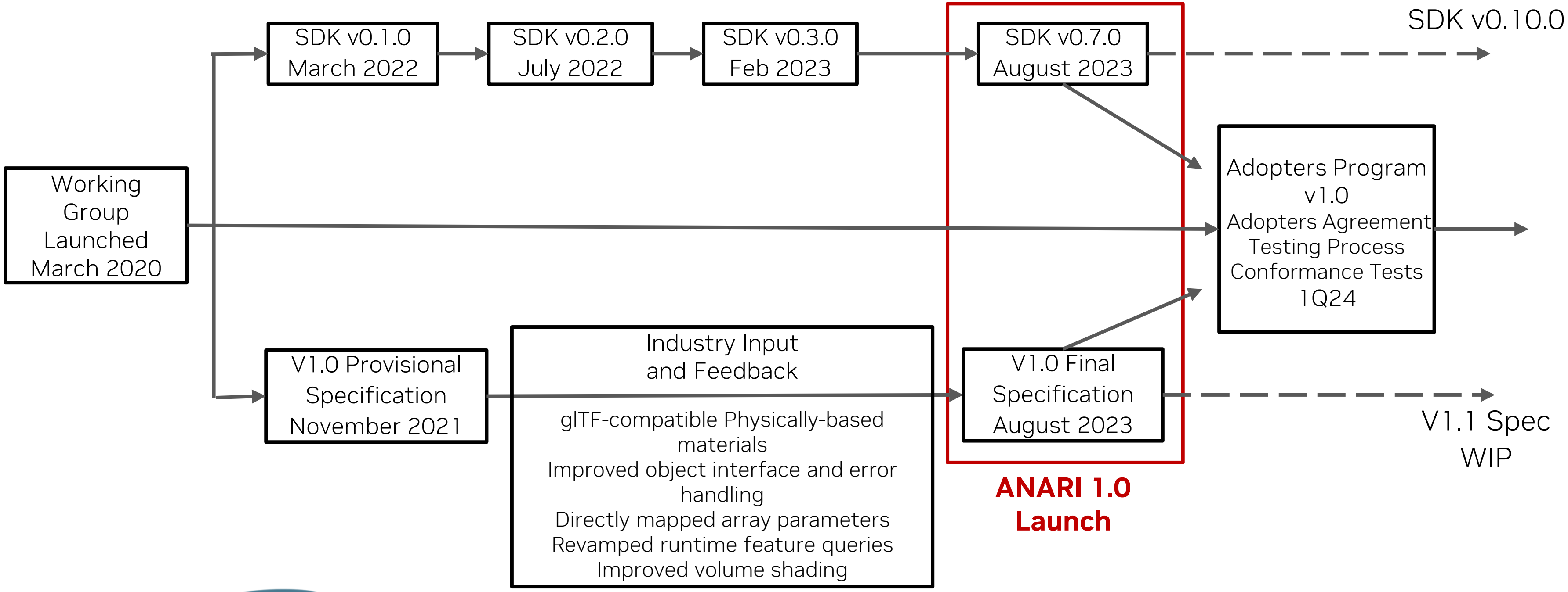

SDK v0.1.0
March 2022
SDK v0.2.0
July 2022
SDK v0.3.0
Feb 2023
SDK v0.7.0
August 2023
Working
Group
Launched
March 2020
Adopters Program
v1.0
Adopters Agreement
Testing Process
Conformance Tests
1Q24
V1.0 Provisional
Specification
November 2021
Industry Input
and Feedback
glTF-compatible Physically-based
materials
Improved object interface and error
handling
Directly mapped array parameters
Revamped runtime feature queries
Improved volume shading
V1.0 Final
Specification
August 2023
ANARI 1.0
Launch



ANARI™

KHRONOS® GROUP

Open-source SDK includes Conformance Test code



All specification, SDK and Conformance Test development work done publicly on GitHub

API Design: Balancing Opposing Forces

API Uniformity

Feature Differentiation

API Design: Balancing Opposing Forces

API Uniformity

Handle-based Objects

Generic Parameters + Arrays

Object/Array Updates

Scene Hierarchy

Concurrency + Parallelism

API Synchronization Semantics

Graphics/Compute API Interop

…

Feature Differentiation

API Design: Balancing Opposing Forces

API Uniformity

Handle-based Objects

Generic Parameters + Arrays

Object/Array Updates

Scene Hierarchy

Concurrency + Parallelism

API Synchronization Semantics

Graphics/Compute API Interop

…

Feature Differentiation

Supported API Extensions

Performance (Frame/Update Latencies)

Supported Hardware Features

Image Quality

Scene Size (Memory overhead, LoD, Out-of-core, Distributed, etc…)

…

API Design: Balancing Opposing Forces

API Uniformity

Handle-based Objects

Generic Parameters + Arrays

Object/Array Updates

Scene Hierarchy

Concurrency + Parallelism

API Synchronization Semantics

Graphics/Compute API Interop

…

Feature Differentiation

Supported API Extensions

Performance (Frame/Update Latencies)

Supported Hardware Features

Image Quality

Scene Size (Memory overhead, LoD, Out-of-core, Distributed, etc…)

…

API Design: Balancing Opposing Forces

API Uniformity | **Feature Differentiation**

Handle-ba

Generic Para

Object/A

Scene

Concurren

API Synchronization Semantics

Graphics/Compute API Interop

…

Extensions

rame/Update

ies)

vare Features

uality

ory overhead, LoD, Out-of-core, Distributed, etc…)

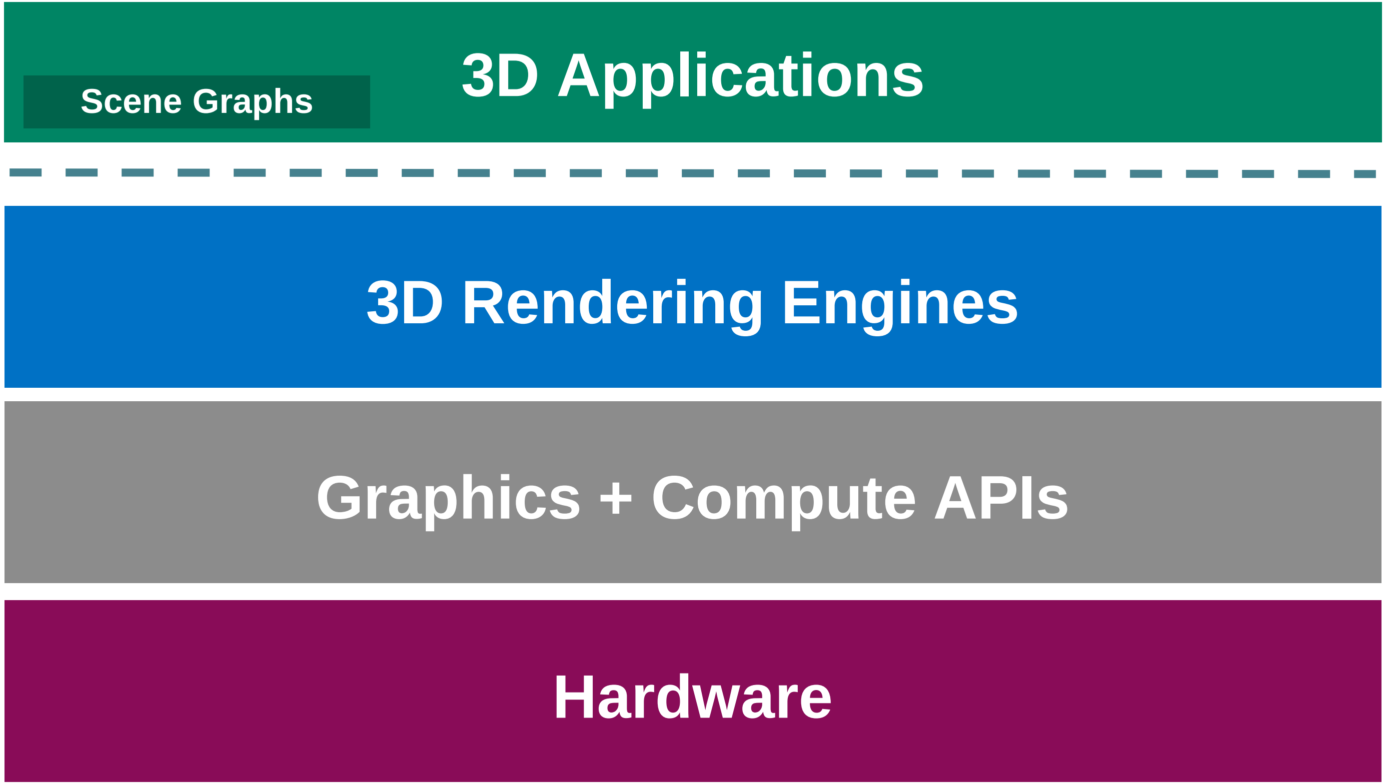
…

only ***"what"*** and ***"when"***

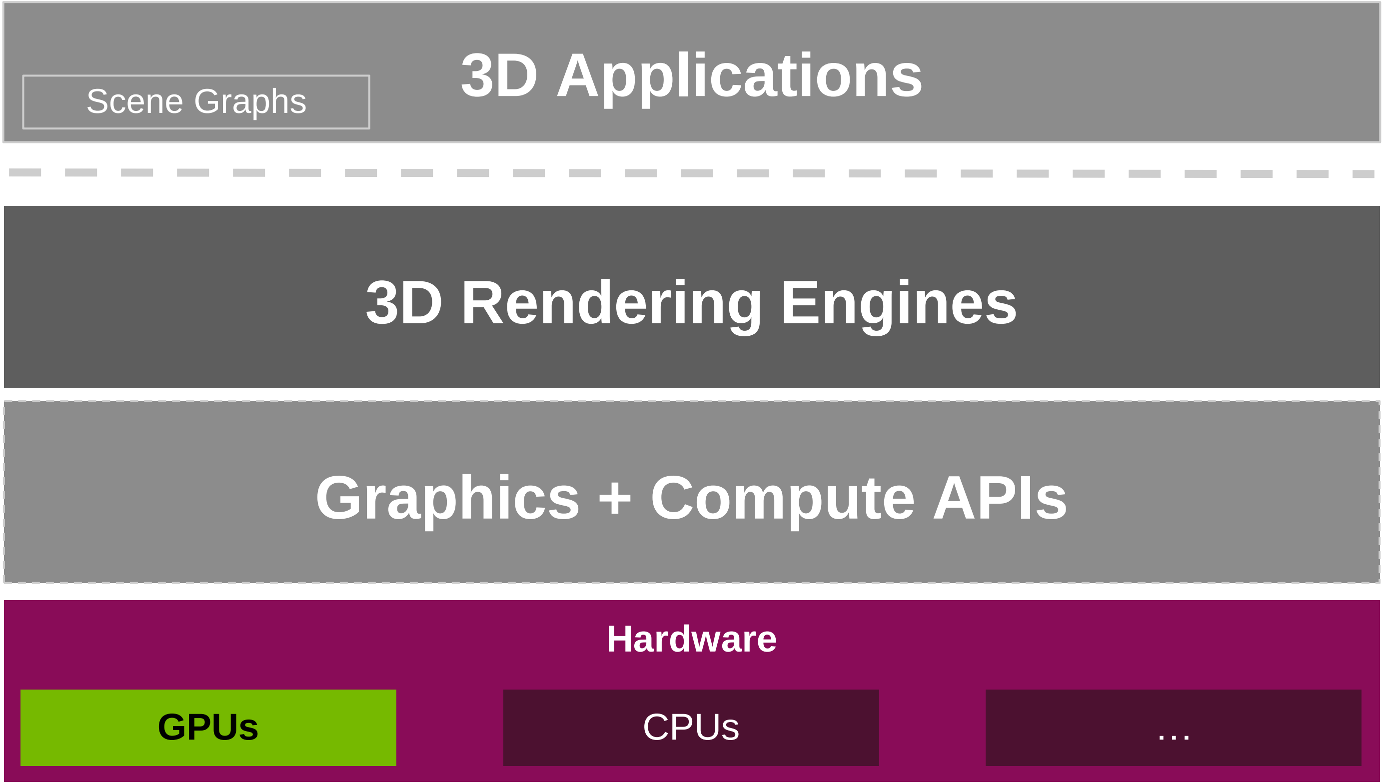
not *"how"*

ANARI Development Stack

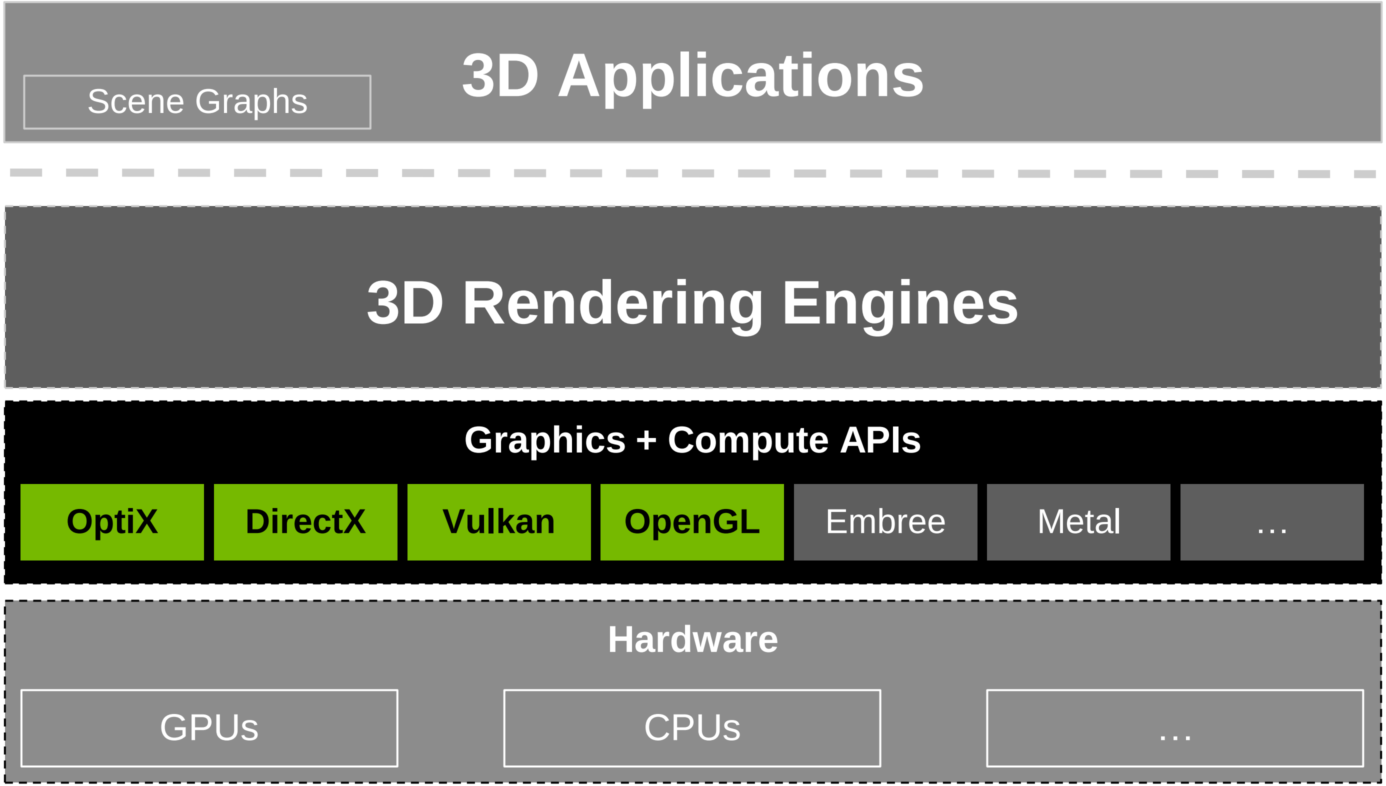
C99 | C++ | Python | ...



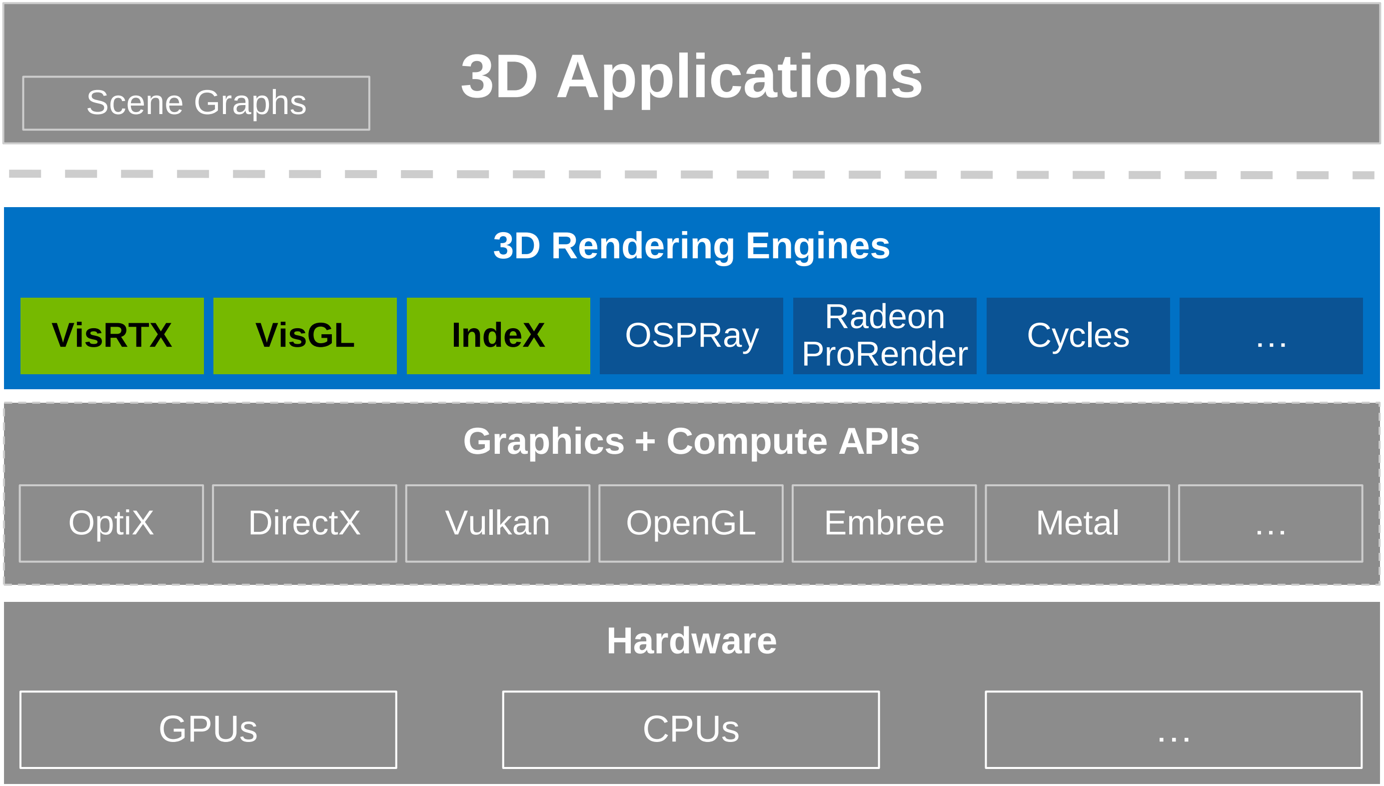
ANARI Development Stack



ANARI Development Stack

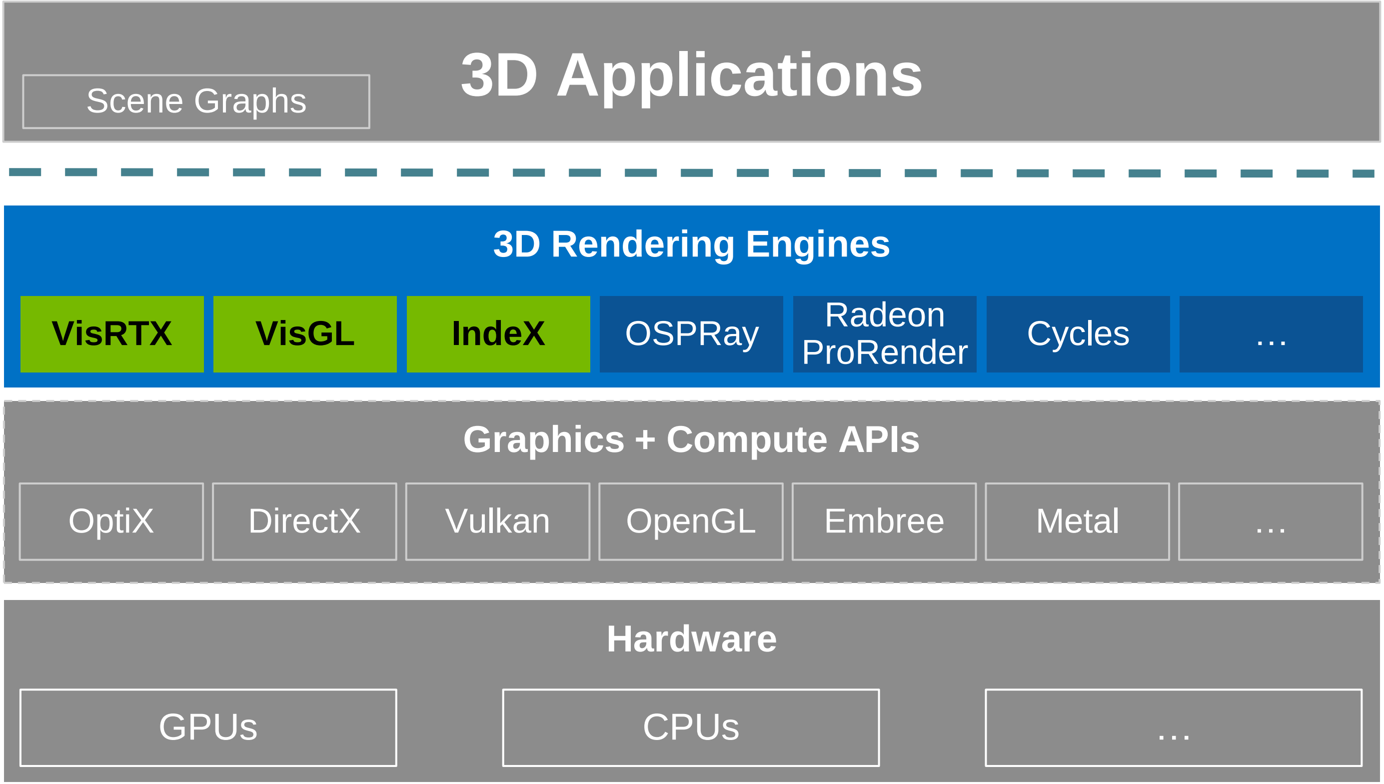


ANARI Development Stack

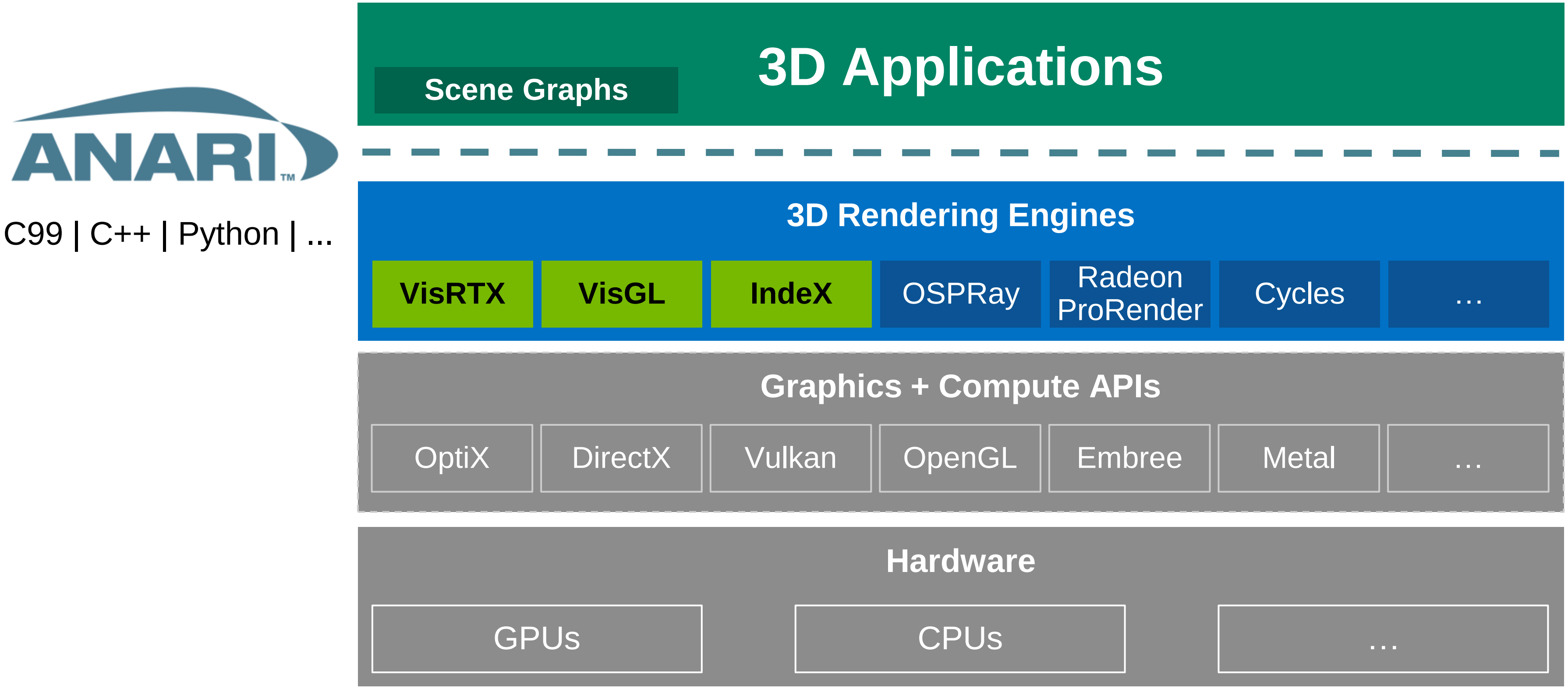


ANARI Development Stack

C99 | C++ | Python | ...

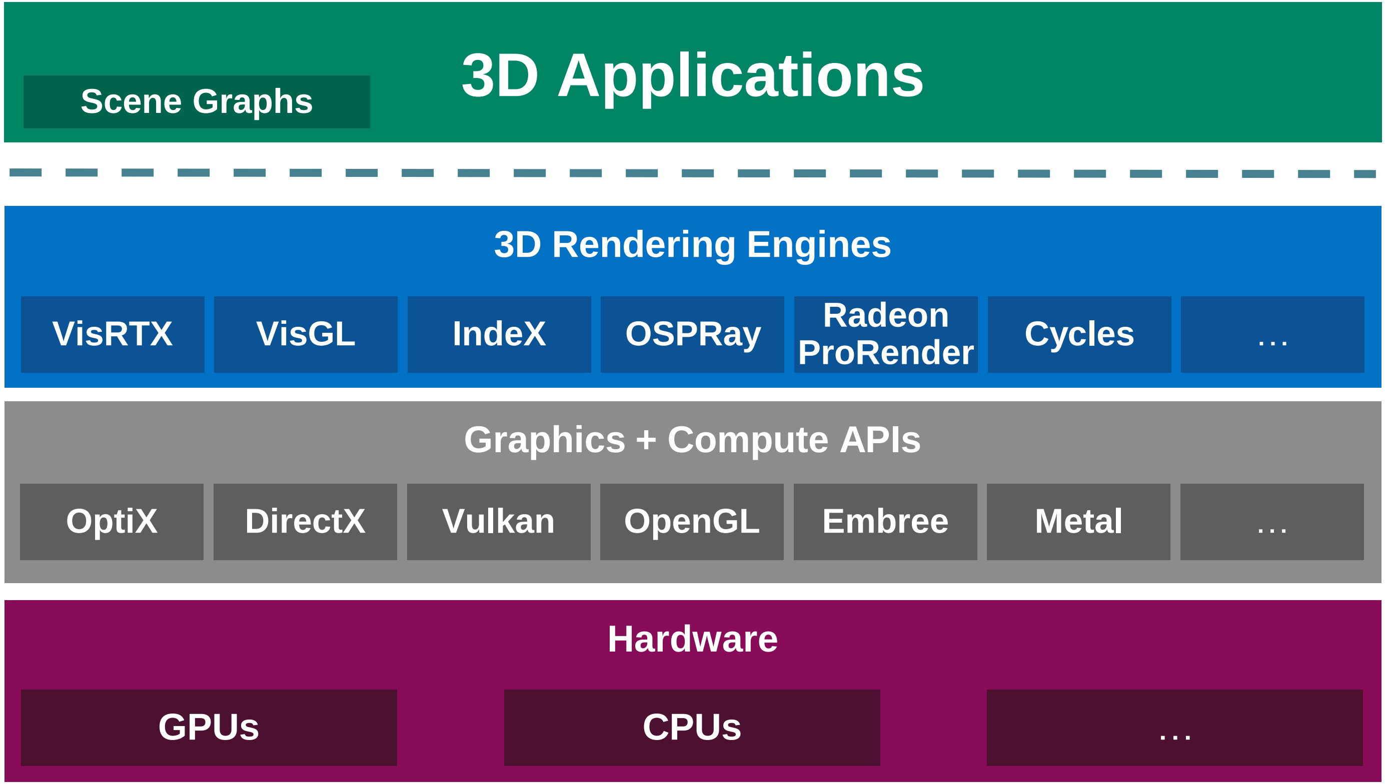


ANARI Development Stack

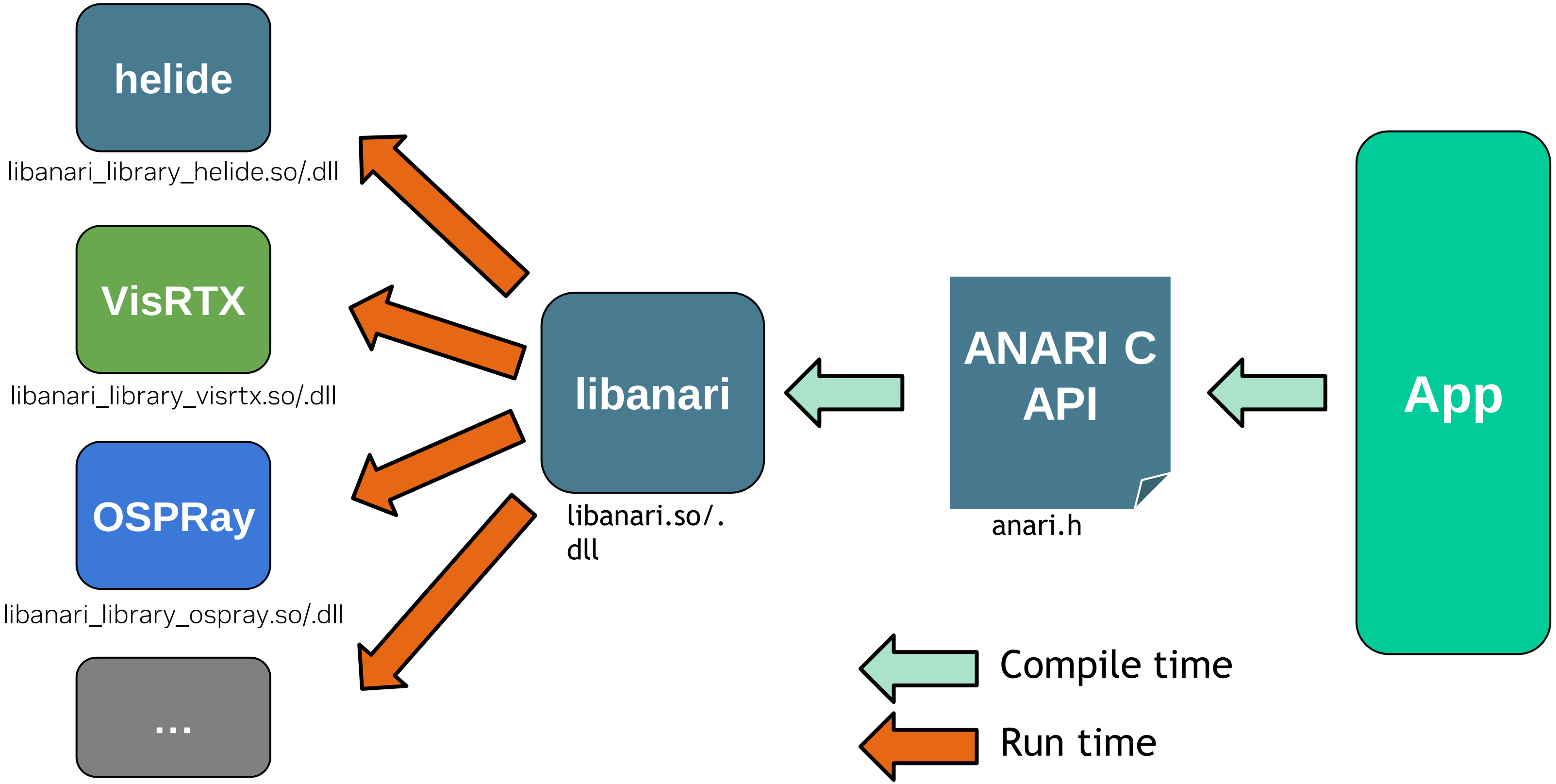


ANARI Development Stack

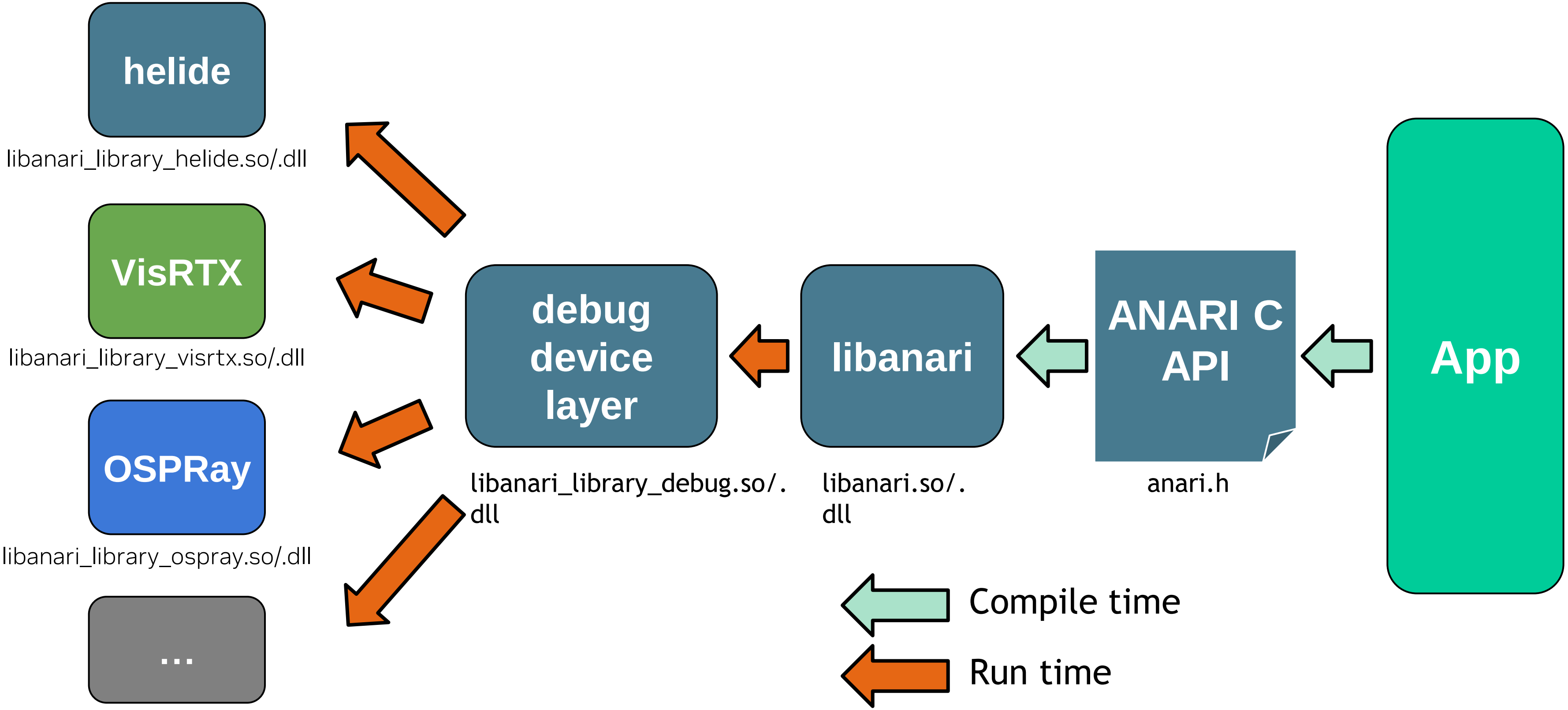
C99 | C++ | Python | ...



ANARI Library Usage



Transparently Adding Layers



What's in the ANARI-SDK?

What’s in the ANARI-SDK?

Code Gen Tools
(Extension Queries + Headers)

Blender Add-on
CTS
hdAnari
Simple Examples
anariViewer
anari_viewer
anari_test_scenes

Python Bindings
C++ Wrappers (anari_cpp)

ANARI Front-end Library (From Specification)

Helium
Helide (Example Implementation)
Debug Layer
Sink Device
Remote Device

KHRONOS GROUP

What's in the ANARI-SDK?

Code Gen Tools
(Extension Queries + Headers)

Blender Add-on | CTS | hdAnari | Simple Examples | anariViewer (anari_viewer, anari_test_scenes)

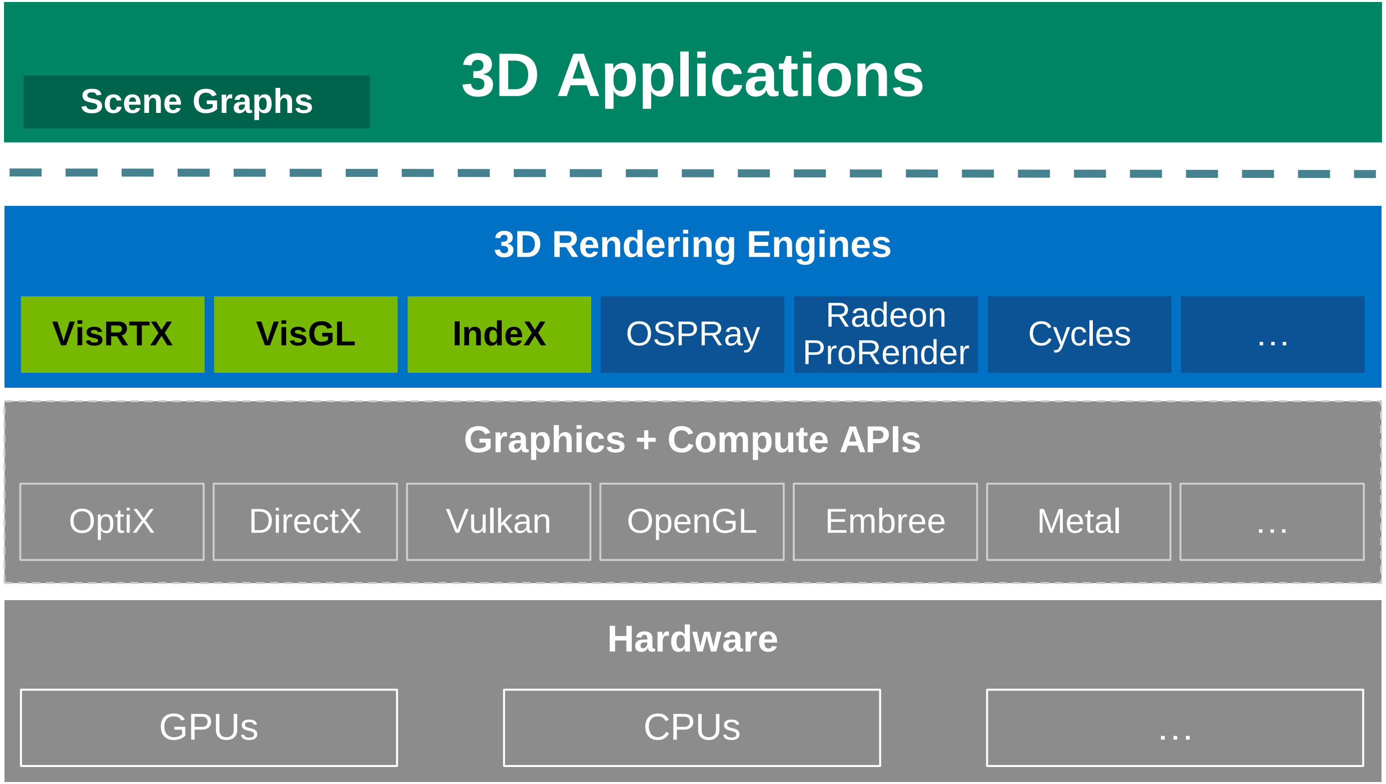
Python Bindings | C++ Wrappers (anari_cpp)

ANARI Front-end Library (From Specification)

Helium | Helide (Example Implementation) | Debug Layer | Sink Device | Remote Device

ANARI Development Stack

C99 | C++ | Python | ...



What’s in the ANARI-SDK?

Code Gen Tools
(Extension Queries + Headers)

Blender Add-on | CTS | hdAnari | Simple Examples | anariViewer

anari_viewer | anari_test_scenes

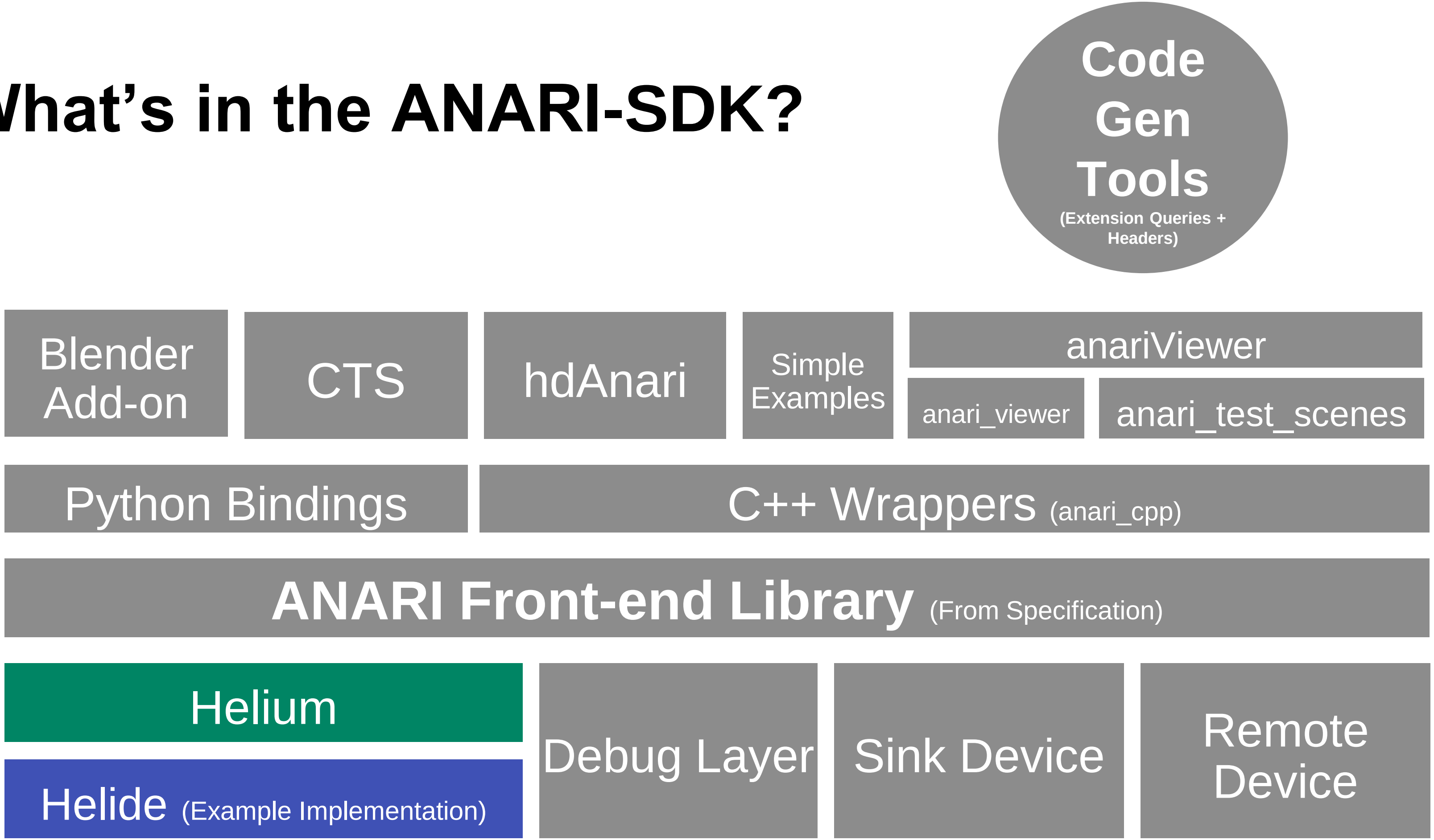
Python Bindings | C++ Wrappers (anari_cpp)

ANARI Front-end Library (From Specification)

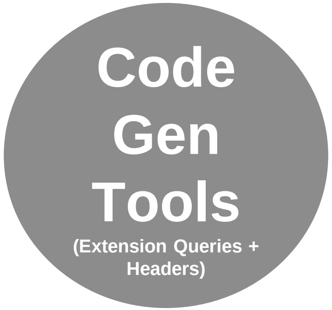
Helium | Debug Layer | Sink Device | Remote Device

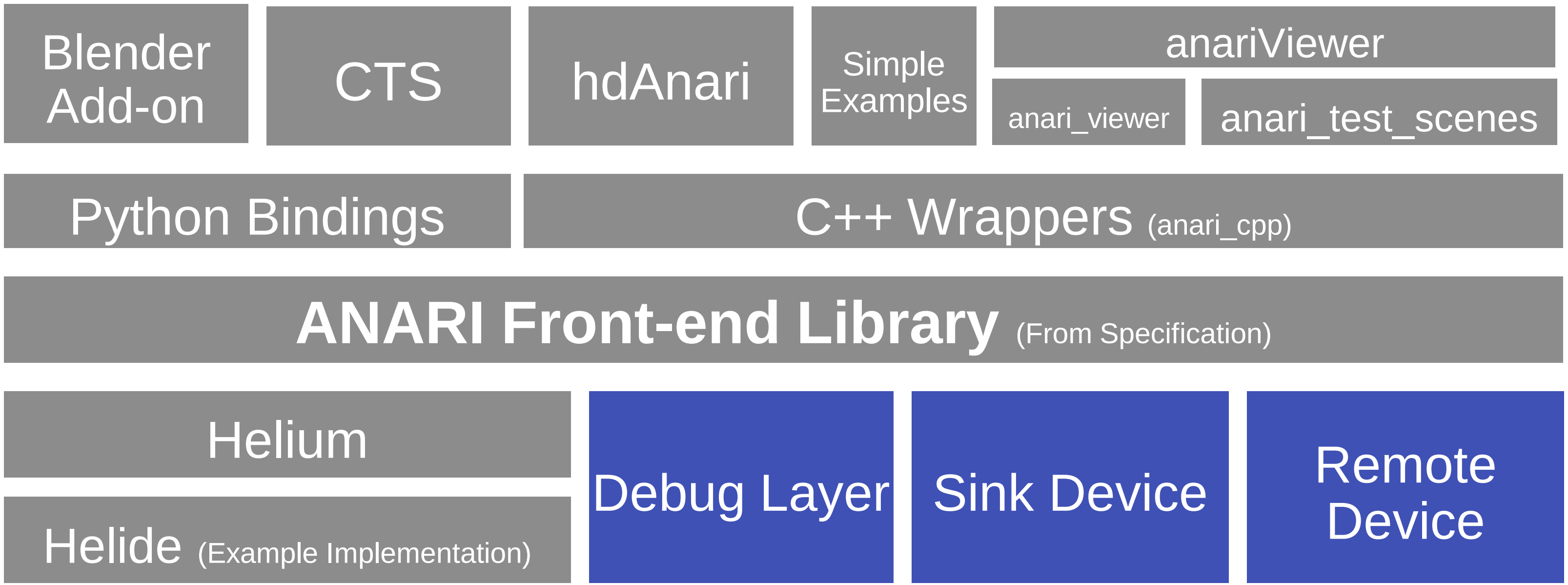
Helide (Example Implementation)

What’s in the ANARI-SDK?

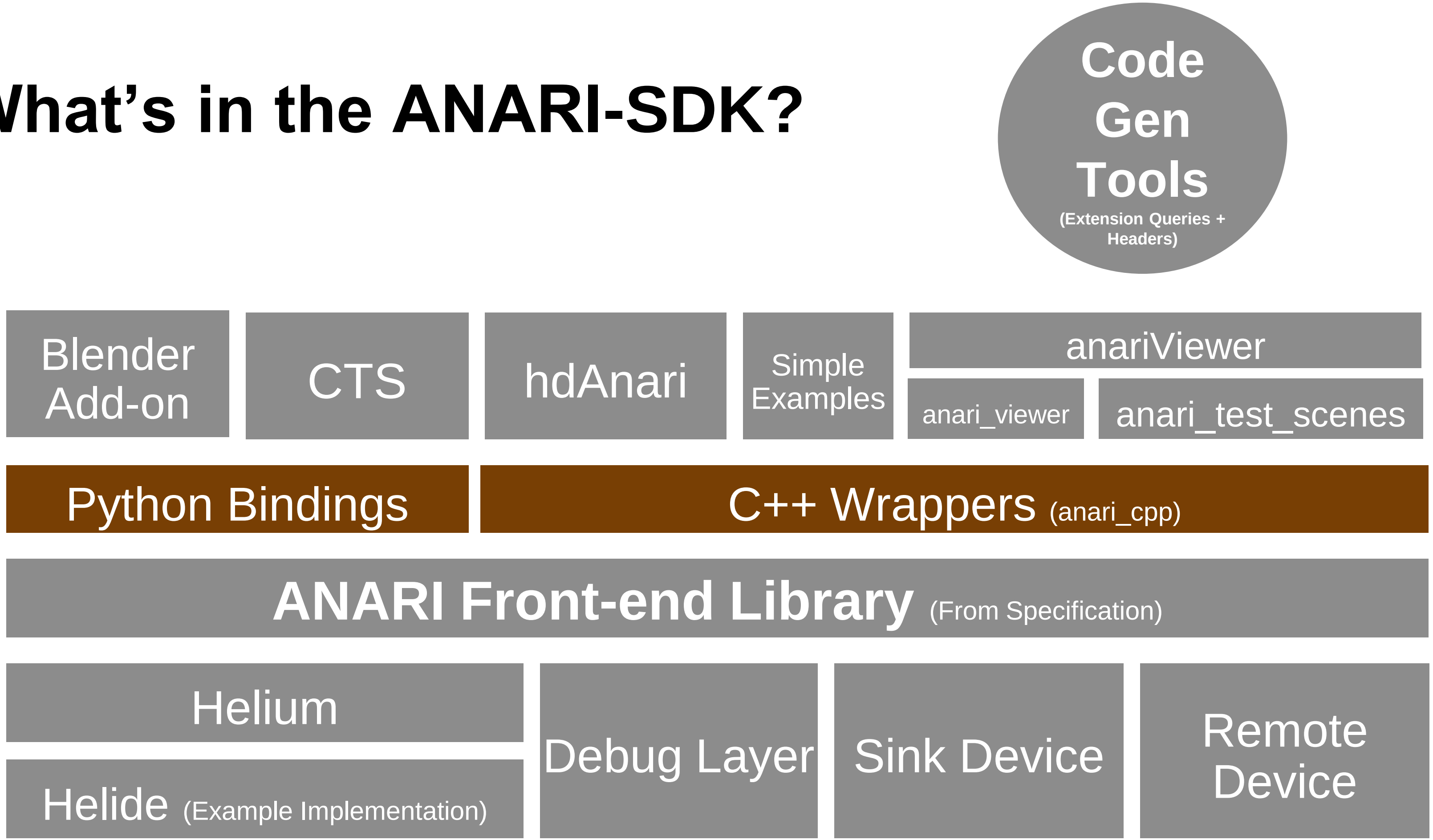


What’s in the ANARI-SDK?

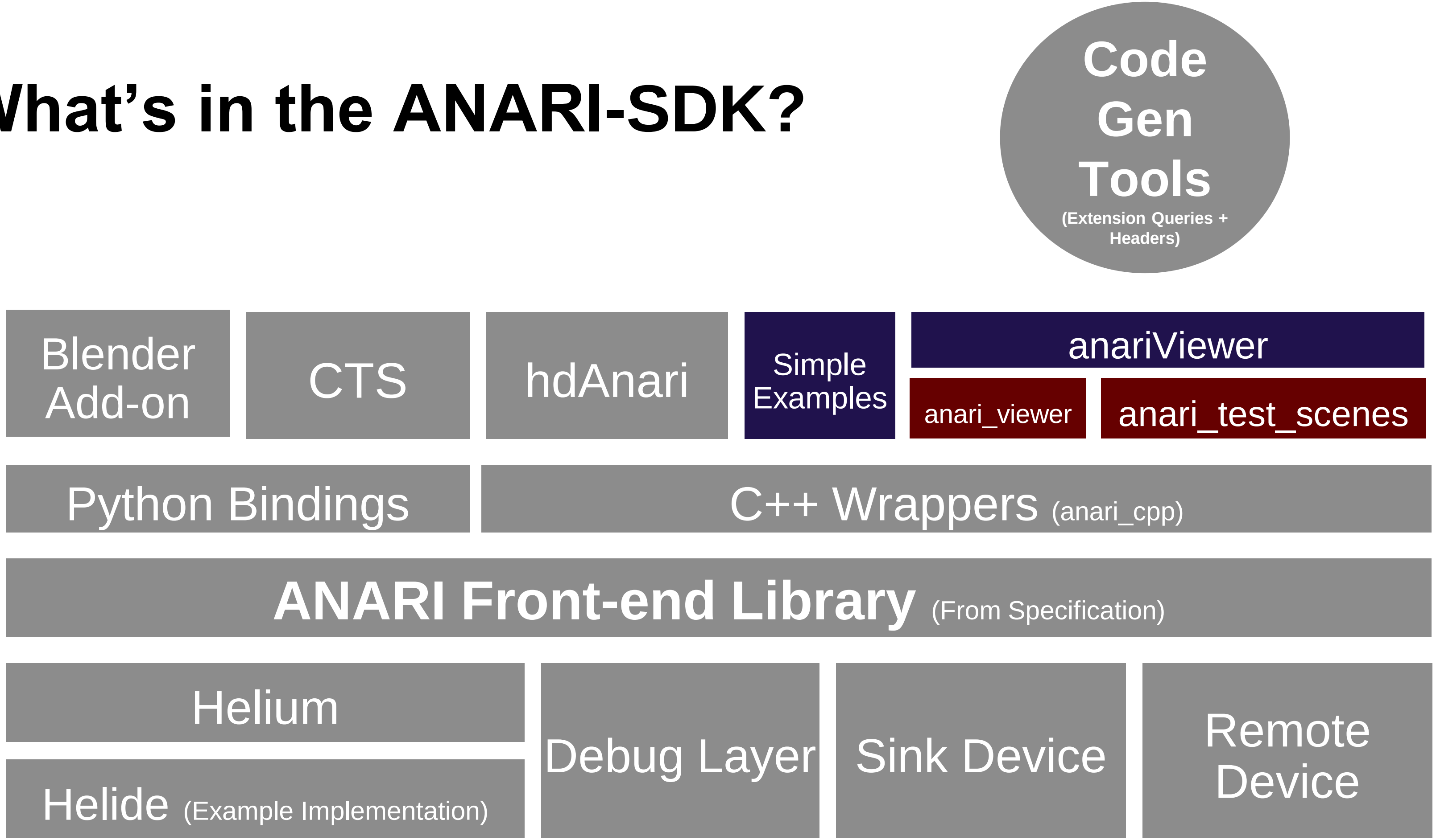




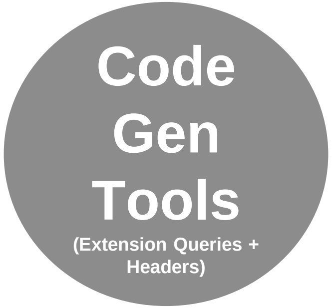
What’s in the ANARI-SDK?

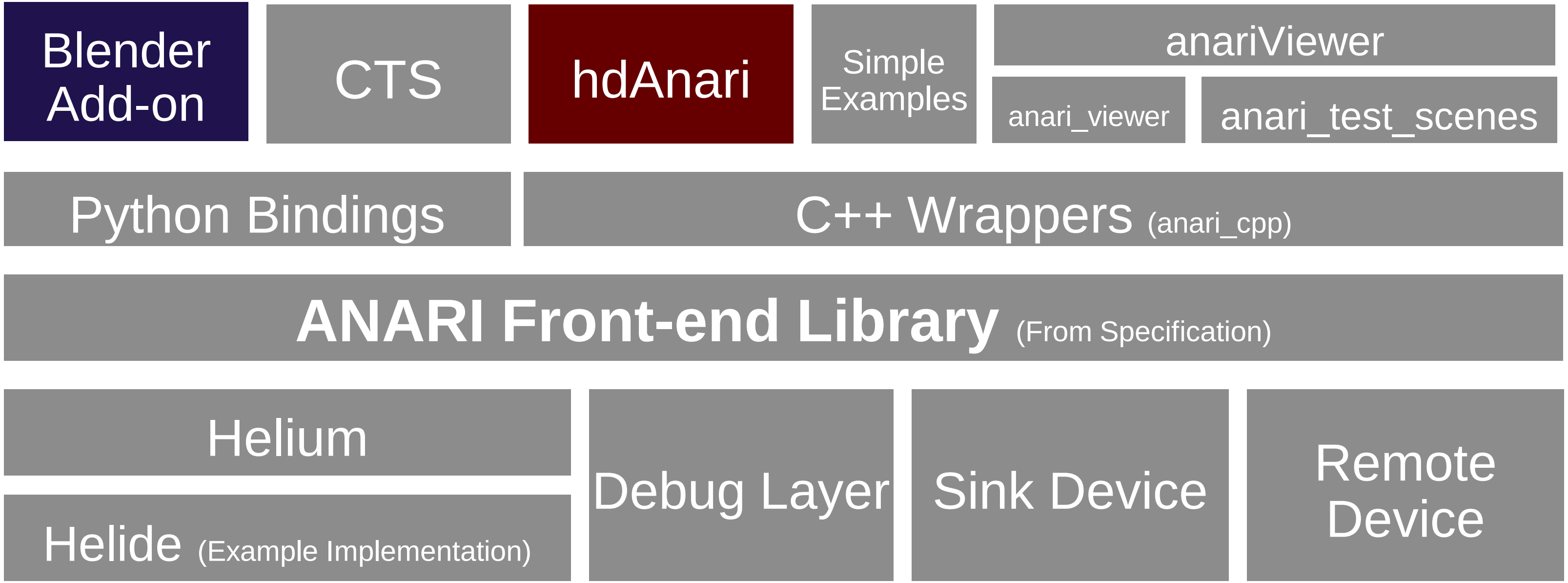


What’s in the ANARI-SDK?

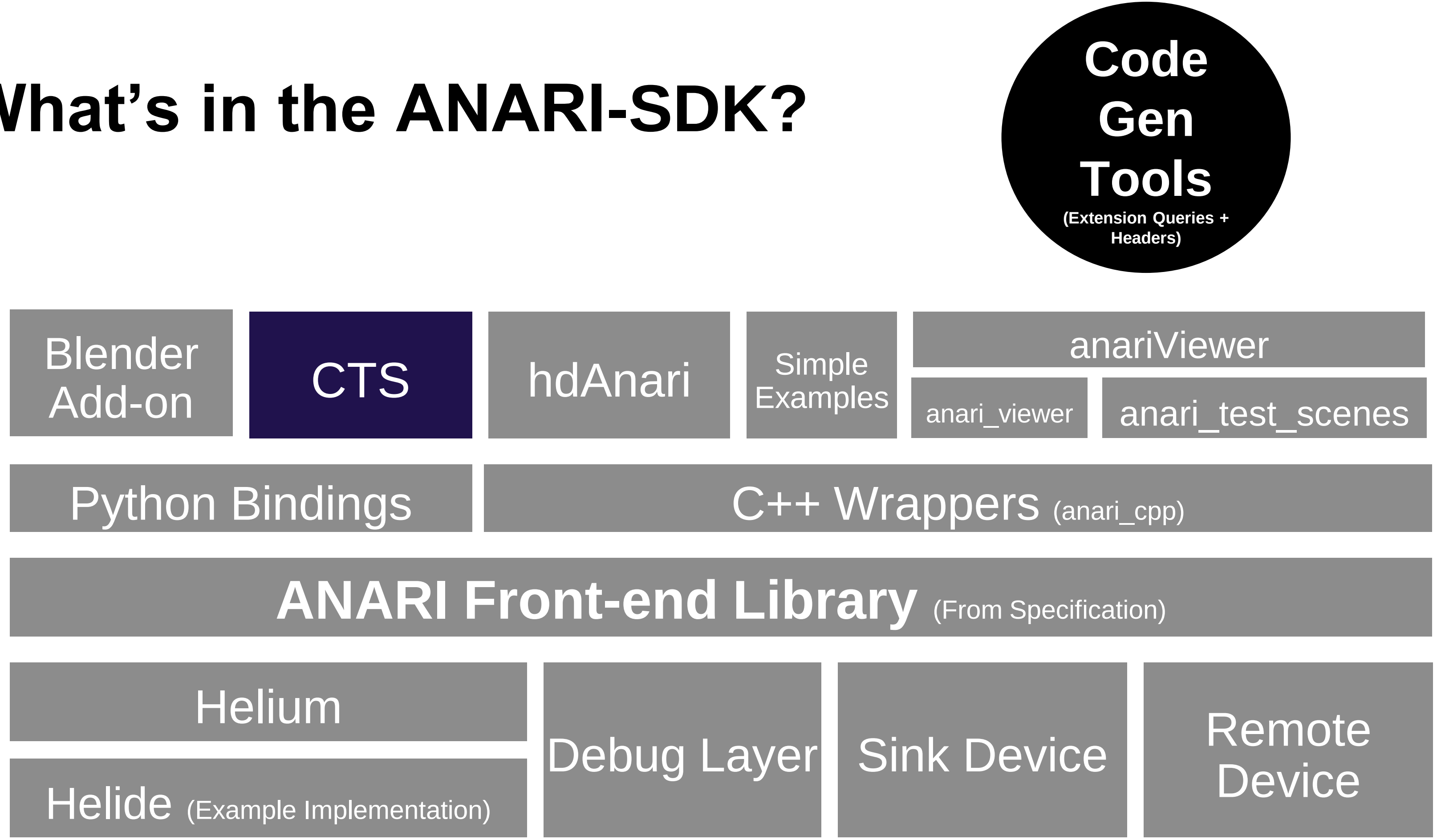


What's in the ANARI-SDK?

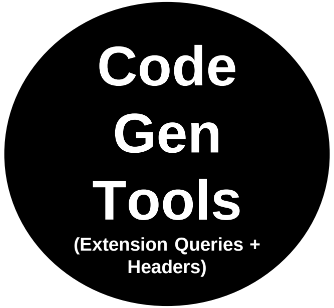


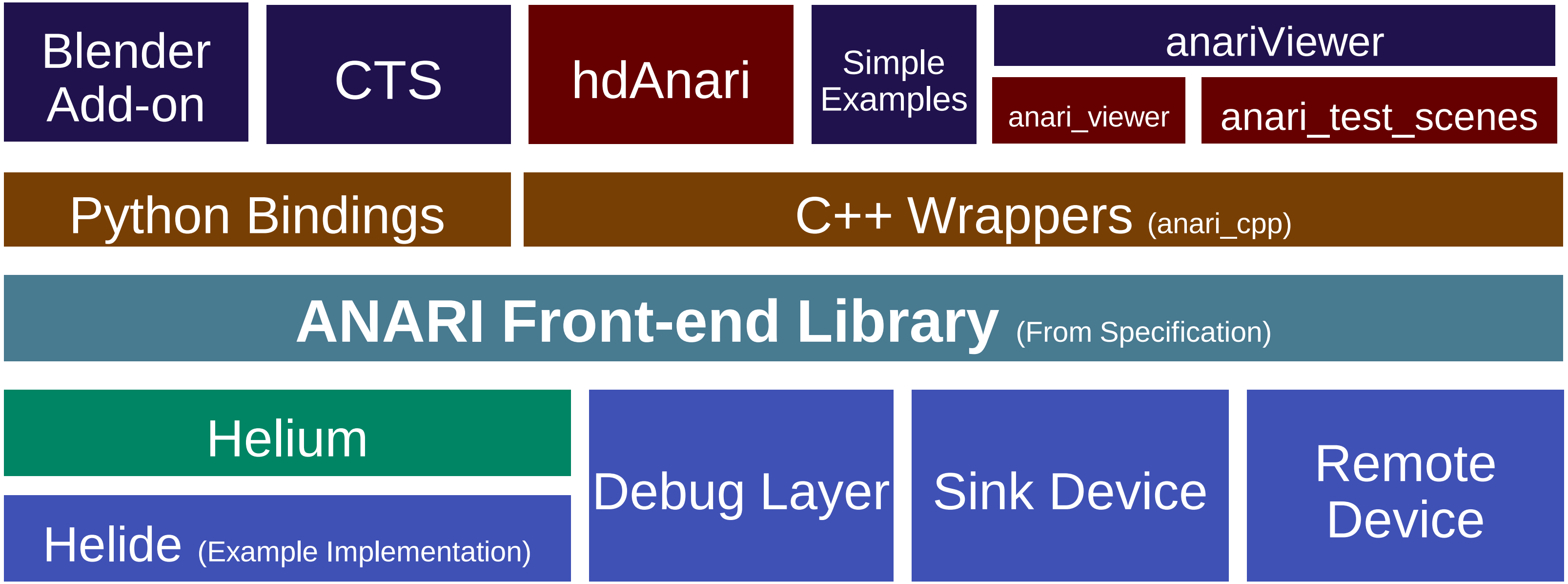


What’s in the ANARI-SDK?

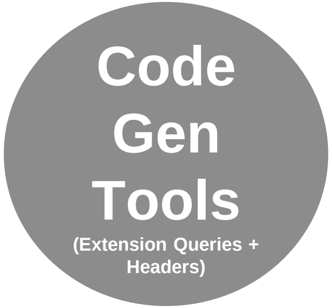


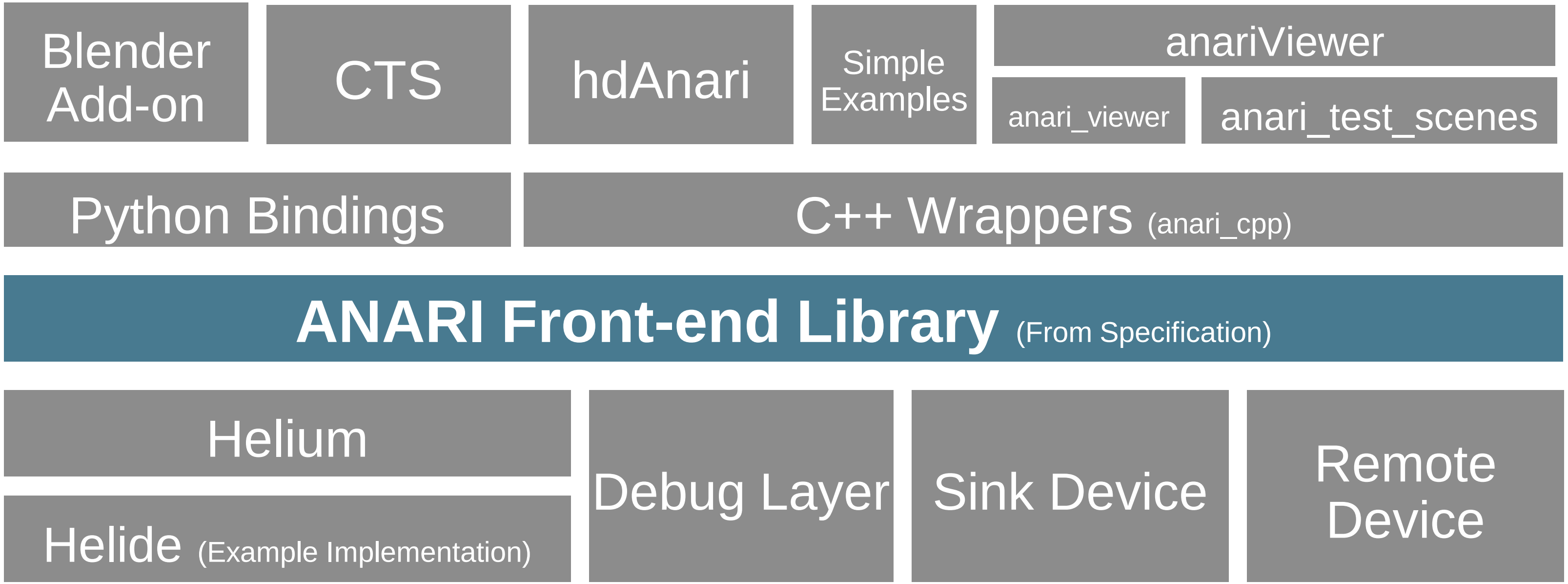
What's in the ANARI-SDK?





What’s in the ANARI-SDK?





API Design: Devices

- ANARI is a C API, with available C++ type safe wrappers

API Design: Devices

- ANARI is a C API, with available C++ type safe wrappers
- **Devices are the main object which handles all API calls from the application**
  - Devices are the instance of the 3D engine that the app is making API calls against
  - Devices (usually) come from shared libraries loaded at runtime

API Design: Devices

- ANARI is a C API, with available C++ type safe wrappers
- **Devices are the main object which handles all API calls from the application**
  - Devices are the instance of the 3D engine that the app is making API calls against
  - Devices (usually) come from shared libraries loaded at runtime

```
// Load implementation from
libanari_library_visrtx.so/.dll
ANARILibrary lib = anariLoadLibrary("visrtx");

// Create instance of VisRTX from the library

ANARIDevice device = anariNewDevice(lib, "default");
```

API Design: Objects

- Objects are represented by *opaque handles* and are:
  - Reference counted
  - Configured with ***parameters*** (from app to device)
  - Introspected with ***properties*** (from device to app)

API Design: Objects

- Objects are represented by *opaque handles* and are:
  - Reference counted
  - Configured with ***parameters*** (from app to device)
  - Introspected with ***properties*** (from device to app)
- Parameter updates are *transactional* using object “commits” to signal object state change

API Design: Objects

- Objects are represented by *opaque handles* and are:
  - Reference counted
  - Configured with ***parameters*** (from app to device)
  - Introspected with ***properties*** (from device to app)
- Parameter updates are *transactional* using object “commits” to signal state change
- Parameters are *unidirectional*: values flow into the object, not out
  - Applications are responsible for keeping values around they want to “remember” (e.g. to display in a UI)

API Design: Objects

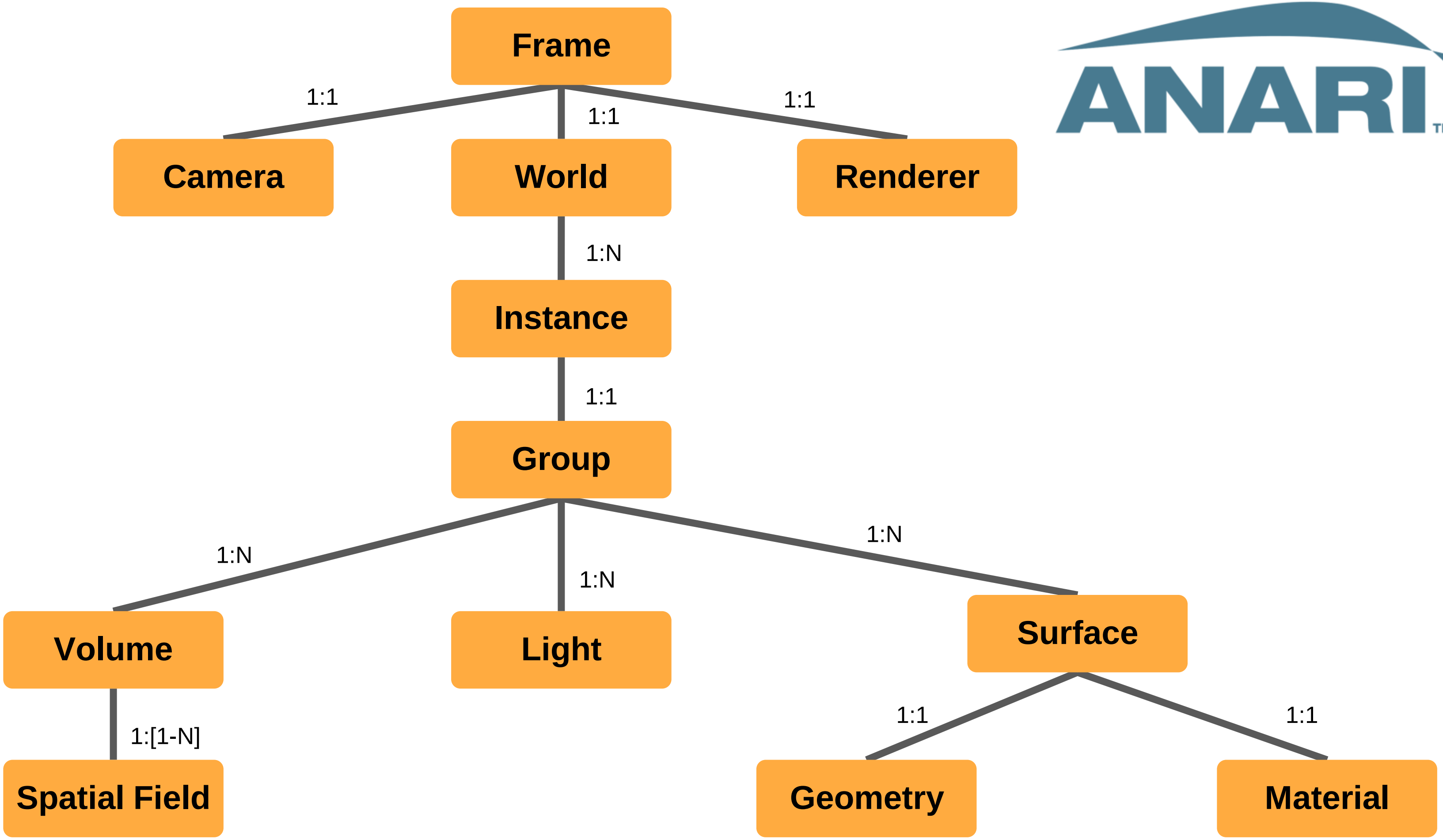
```
// Create an object that does not need a subtype

ANARIWorld world = anariNewWorld(device);

// Create an object that is subtyped

ANARICamera camera = anariNewCamera(device, "perspective");
```

API Design: Objects

```
// Create an object that does not need a subtype
ANARIWorld world = anariNewWorld(device);
// Create an object that is subtyped
ANARICamera camera = anariNewCamera(device, "perspective");
// Parameterize camera using values from the application
anariSetParameter(device, camera, "position", ANARI_FLOAT32_VEC3, &cam_pos);
anariSetParameter(device, camera, "direction", ANARI_FLOAT32_VEC3, &cam_view);
anariSetParameter(device, camera, "up", ANARI_FLOAT32_VEC3, &cam_up);
```

API Design: Objects

```
// Create an object that does not need a subtype
ANARIWorld world = anariNewWorld(device);
// Create an object that is subtyped
ANARICamera camera = anariNewCamera(device, "perspective");
// Parameterize camera using values from the application
anariSetParameter(device, camera, "position", ANARI_FLOAT32_VEC3, &cam_pos);
anariSetParameter(device, camera, "direction", ANARI_FLOAT32_VEC3, &cam_view);
anariSetParameter(device, camera, "up", ANARI_FLOAT32_VEC3, &cam_up);
// Commit set parameters to the camera for use in the next rendered frame
anariCommitParameters(device, camera);
```


Frame
1:1
1:1
1:1
Camera
World
Renderer
1:N
Instance
1:1
Group
1:N
1:N
1:N
Volume
Light
Surface
1:[1-N]
1:1
1:1
Spatial Field
Geometry
Material

API Design: Rendering Frames

```
// Render one frame
anariRenderFrame(device, frame);

// Wait on the frame to be completed (anariMapFrame() will block if needed)
anariFrameReady(device, frame, ANARI_WAIT);

// Get pointer to the pixels in the color channel
uint32_t width = 0, height = 0;
ANARIDataType type = ANARI_UNKNOWN;
uint32_t *pixels =
    (uint32_t *)anariMapFrame(device, frame, "channel.color", &width, &height, &type);

// Consume the pixels, in this case writing them to a file
writePNG("anari_frame.png", pixels, type, width, height);

// Unmap the pixel buffer and move on to the next frame
anariUnmapFrame(device, frame, "channel.color");


// ...
```

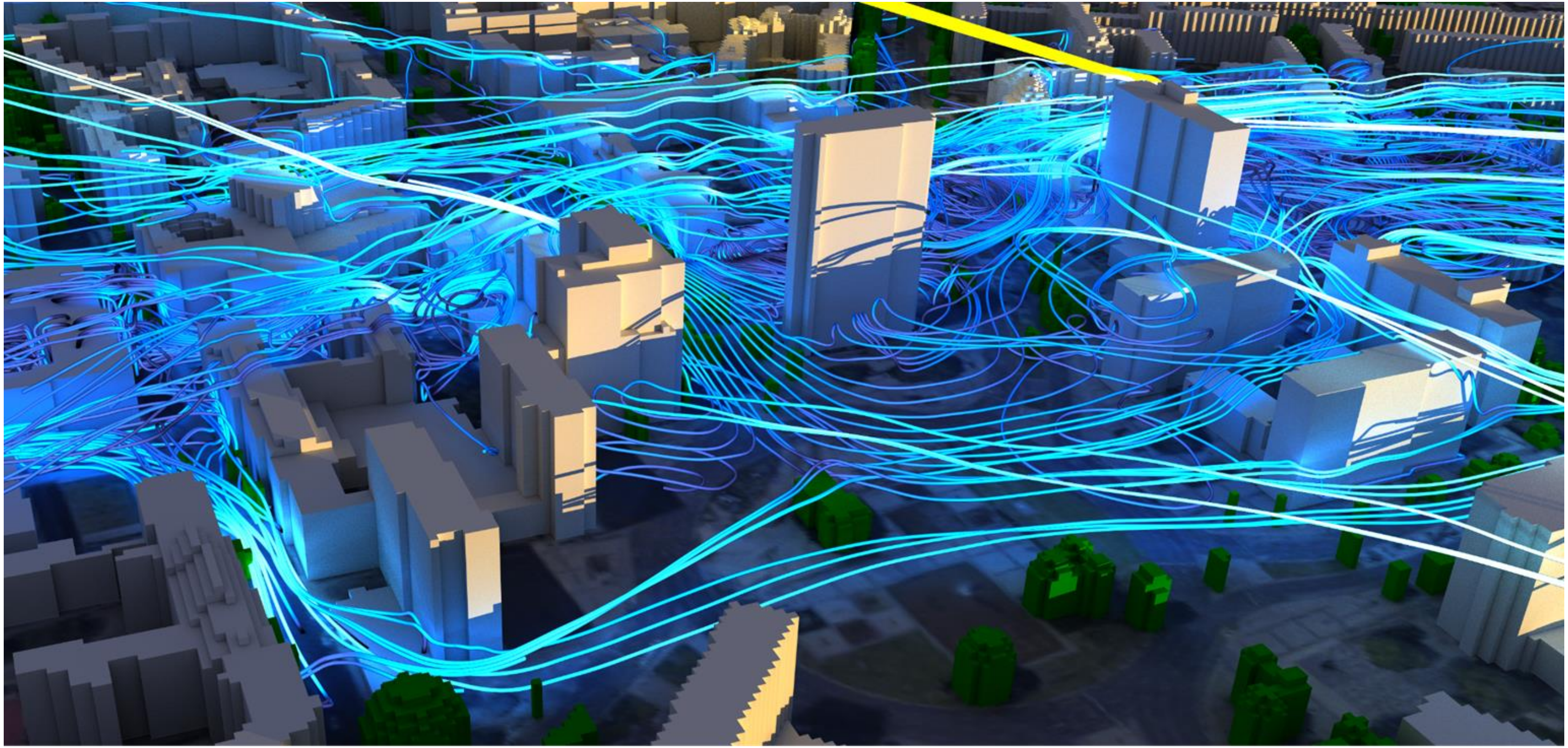
API Design: Rendering Frames

```
// Render one frame
anariRenderFrame(device, frame);

// Wait on the frame to be completed (anariMapFrame() will block if needed)
anariFrameReady(device, frame, ANARI_WAIT);

// Get pointer to the pixels in the color channel
uint32_t width = 0, height = 0;
ANARIDataType type = ANARI_UNKNOWN;
uint32_t *pixels =
    (uint32_t *)anariMapFrame(device, frame, "channel.color", &width, &height, &type);

// Consume the pixels, in this case writing them to a file
writePNG("anari_frame.png", pixels, type, width, height);

// Unmap the pixel buffer and move on to the next frame
anariUnmapFrame(device, frame, "channel.color");

// ...
```

API Design: Rendering Frames

```
// Render one frame
anariRenderFrame(device, frame);

// Wait on the frame to be completed (anariMapFrame() will block if needed)
anariFrameReady(device, frame, ANARI_WAIT);

// Get pointer to the pixels in the color channel
uint32_t width = 0, height = 0;
ANARIDataType type = ANARI_UNKNOWN;
uint32_t *pixels =
    (uint32_t *)anariMapFrame(device, frame, "channel.color", &width, &height, &type);

// Consume the pixels, in this case writing them to a file
writePNG("anari_frame.png", pixels, type, width, height);

// Unmap the pixel buffer and move on to the next frame
anariUnmapFrame(device, frame, "channel.color");

// ...
```

API Design: Rendering Frames



Additional Topics

- Details of specific object subtype extensions
  - Geometries, materials, samplers, lights, spatial fields, volumes, cameras…
- Device introspection – detecting extensions + parameter information
- Asynchronous operations: rendering vs. scene updates, thread safety
- Multi-frame and multi-device application architecture
- Array ownership semantics + content updates
- Performance considerations
- Diversity of implementation approaches and design choices

API Design: Balancing Opposing Forces

API Uniformity

Handle-based Objects

Generic Parameters + Arrays

Object/Array Updates

Scene Hierarchy

Concurrency + Parallelism

API Synchronization Semantics

Graphics/Compute API Interop

…

Feature Differentiation

Supported API Extensions

Performance (Frame/Update Latencies)

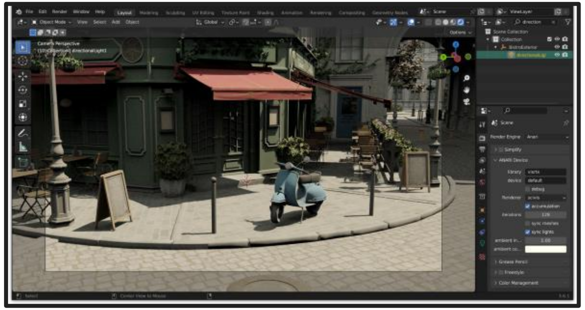
Supported Hardware Features

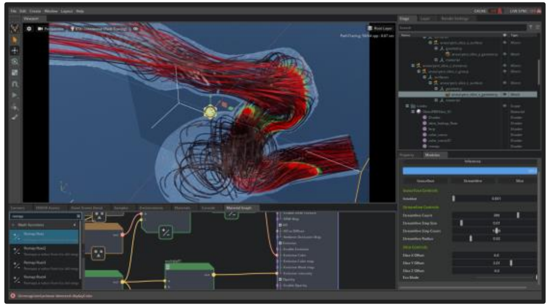
Image Quality

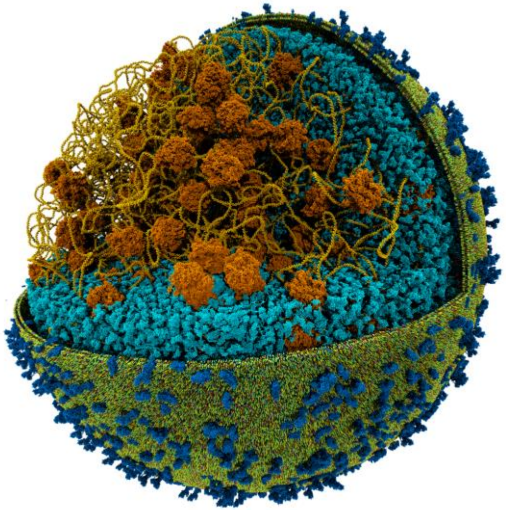
Scene Size (Memory overhead, LoD, Out-of-core, Distributed, etc…)

…

ANARI is *Portable* (API Uniformity)



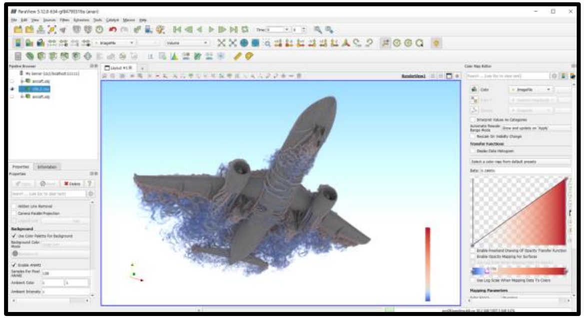


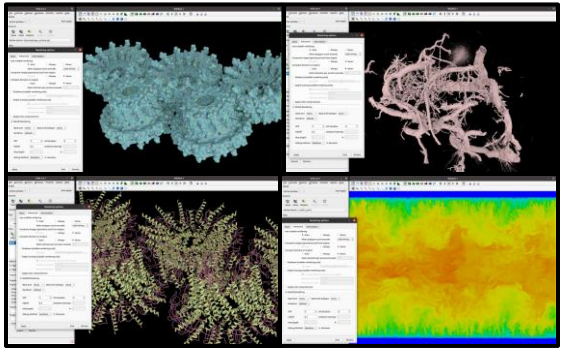


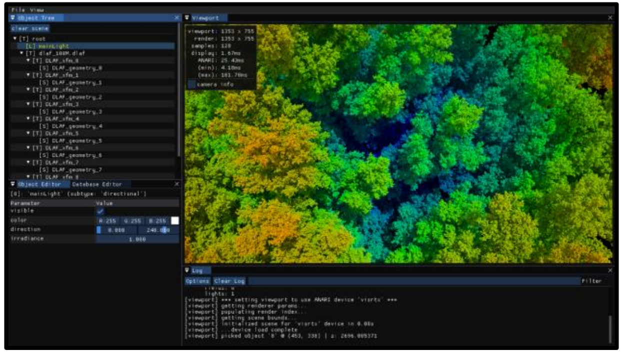
blender

NVIDIA OMNIVERSE

VMD
Visual Molecular Dynamics







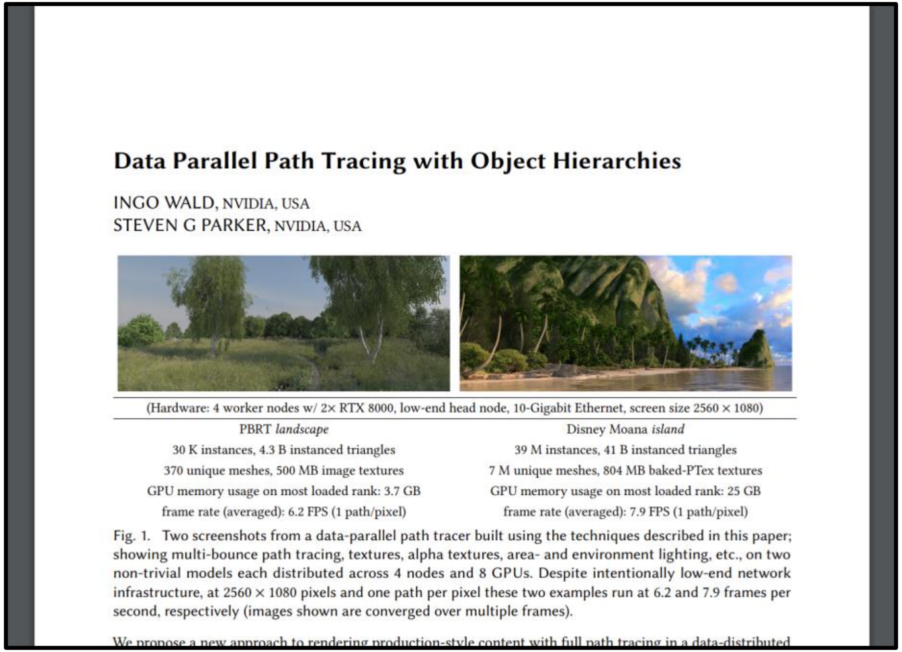
ParaView
Parallel Visualization Application

VisIt

ANARI is *Scalable* (Feature Differentiation)

- By not prescribing “how” things are rendered, implementations can scale…
  - Image Quality (lighting, materials, etc.)
  - Available HW (multi-GPU, multi-node)
  - Render Rate
  - Scene Size (geometry, volumes, instances)
  - Animation Update Rate

Elevating Research with ANARI



Data Parallel Path Tracing with Object Hierarchies

INGO WALD, NVIDIA, USA
STEVEN G PARKER, NVIDIA, USA

(Hardware: 4 worker nodes w/ 2× RTX 8000, low-end head node, 10-Gigabit Ethernet, screen size 2560 × 1080)

PBRT *landscape*	Disney Moana *island*
30 K instances, 4.3 B instanced triangles	39 M instances, 41 B instanced triangles
370 unique meshes, 500 MB image textures	7 M unique meshes, 804 MB baked-PTex textures
GPU memory usage on most loaded rank: 3.7 GB	GPU memory usage on most loaded rank: 25 GB
frame rate (averaged): 6.2 FPS (1 path/pixel)	frame rate (averaged): 7.9 FPS (1 path/pixel)

Fig. 1. Two screenshots from a data-parallel path tracer built using the techniques described in this paper; showing multi-bounce path tracing, textures, alpha textures, area- and environment lighting, etc., on two non-trivial models each distributed across 4 nodes and 8 GPUs. Despite intentionally low-end network infrastructure, at 2560 × 1080 pixels and one path per pixel these two examples run at 6.2 and 7.9 frames per second, respectively (images shown are converged over multiple frames).

We propose a new approach to rendering production-style content with full path tracing in a data-distributed

HPG 2022

Data Parallel Multi-GPU Path Tracing using Ray Queue Cycling (author's pre-print, with some addtl material)

Ingo Wald[†] Milan Jaroš[‡] Stefan Zellmann[◦]
[†]NVIDIA
[‡]IT4Innovations, VSB – Technical University of Ostrava, Ostrava, Czech Republic
[◦]University of Cologne

Fig. 1. A high-resolution version of the Disney Moana *island* model, with nearly 600 million triangles before instancing, 31 million instances, and 33 GB of textures, for a total of 84 GBs of model data excluding acceleration structures. At 2560 × 1080 pixels and 8 paths per pixel, our method runs this at 2.9 frames per second (FPS) on a DGX-2 (with 16 Volta class GPUs and NVLink and NVSwitch), at 2.9 FPS on an HGX (similar architecture, but with 8 A100 GPUs), and at 6.0 FPS, respectively, on an RTX Server with 8 Ampere class GPUs with ray tracing cores on PCIe. An important feature of our method is that it is almost entirely oblivious to how geometry gets partitioned across GPUs, and does not require any spatially or object-space coherent assignment whatsoever. Right: A false-color image where an object's color encodes which GPU it is on; showing a near-random assignment that works just fine in our method.

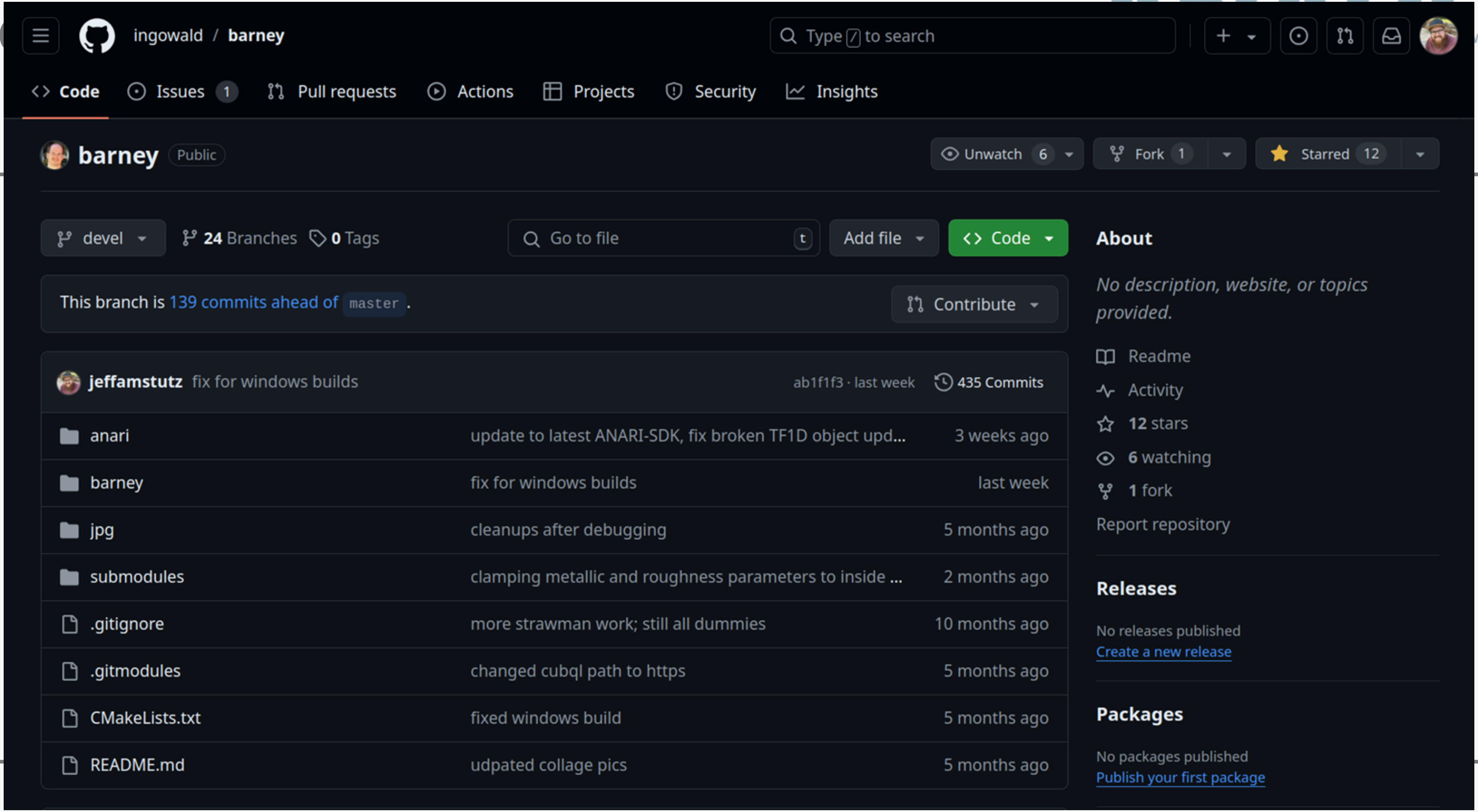
Abstract
We propose a novel approach to data-parallel path tracing on single-node/multi-GPU hardware that builds on ray forwarding, but which aims—above all else—at generality and practicability. We do this by *avoiding* any attempts at reducing the number of traces or forward operations performed, and instead focus on always using all GPUs' aggregate compute and bandwidth to effectively trace each ray on every GPU. We show that—counter-intuitively—this is both feasible and desirable; and that when run on typical data-center/cloud hardware, the resulting framework not only achieves good performance and scalability, but also comes with significantly fewer limitations, assumptions,

than one GPU is either already the norm, or an easy-to-adopt option. However, despite a rich history of data parallel rendering research, in practice this technology seems to be entirely confined to scientific visualization, and hardly used at all outside of that field.

Why this may be so is an interesting topic for debate; however, we believe the three most important reasons are the following: first, most existing approaches to data parallel rendering have focused on multi-node cluster/MPI setups, but those are often constrained in terms of bandwidth, and are too complicated to set up and use for the average

HPG 2023

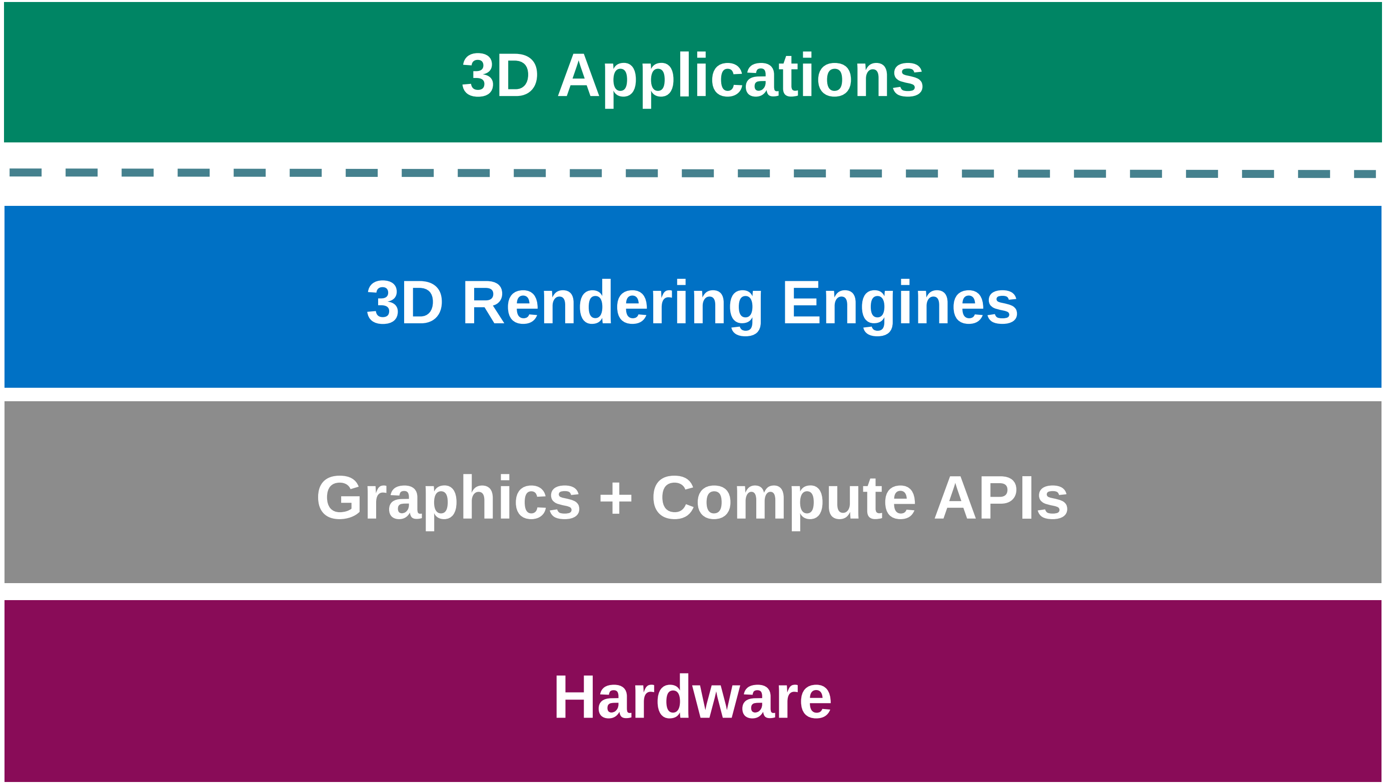
El



ingowald / barney
Type / to search
Code
Issues 1
Pull requests
Actions
Projects
Security
Insights
barney
Public
Unwatch 6
Fork 1
Starred 12
devel
24 Branches
0 Tags
Go to file
t
Add file
Code
About
No description, website, or topics provided.
This branch is 139 commits ahead of master.
Contribute
jeffamstutz fix for windows builds
ab1f1f3 · last week
435 Commits
anari
update to latest ANARI-SDK, fix broken TF1D object upd...
3 weeks ago
barney
fix for windows builds
last week
jpg
cleanups after debugging
5 months ago
submodules
clamping metallic and roughness parameters to inside ...
2 months ago
.gitignore
more strawman work; still all dummies
10 months ago
.gitmodules
changed cubql path to https
5 months ago
CMakeLists.txt
fixed windows build
5 months ago
README.md
udpated collage pics
5 months ago
Readme
Activity
12 stars
6 watching
1 fork
Report repository
Releases
No releases published
Create a new release
Packages
No packages published
Publish your first package

ANARI Software Stack

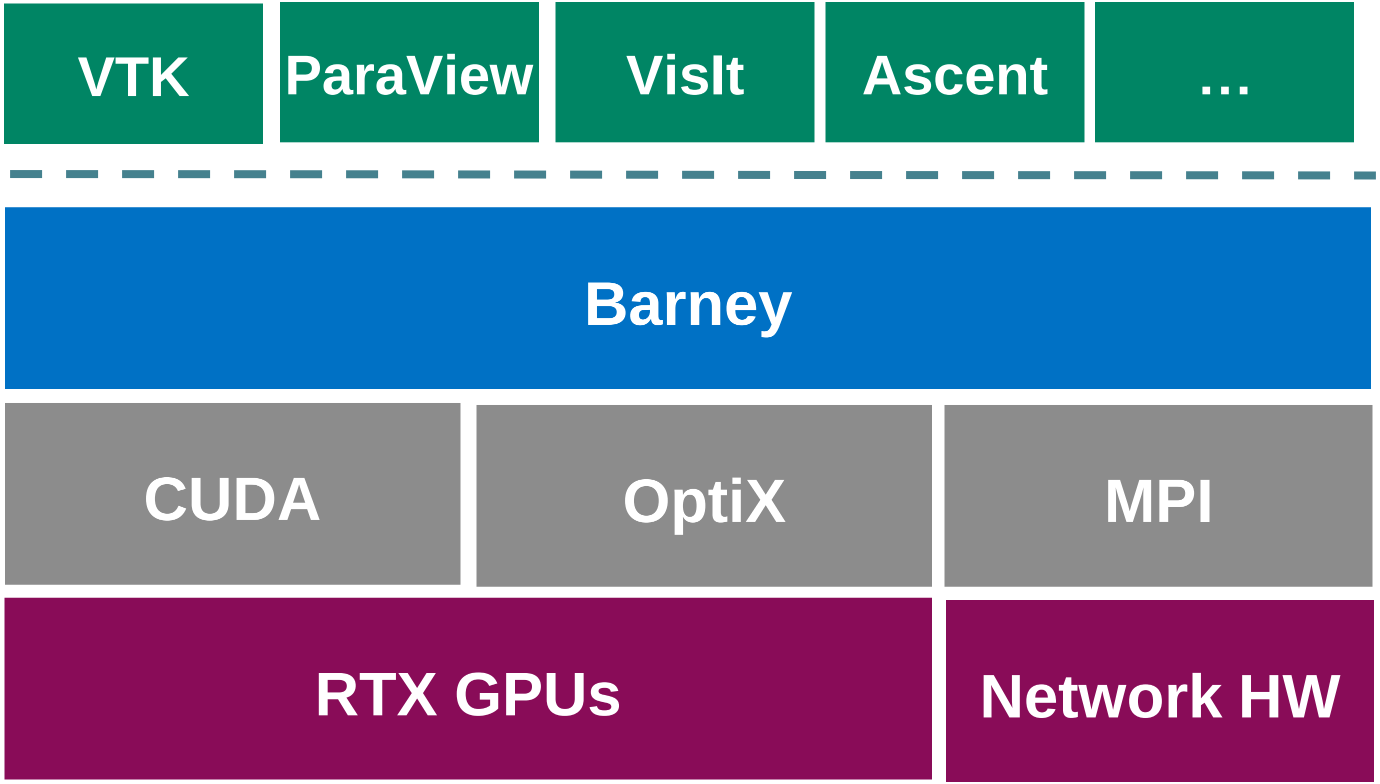
C99 | C++ | Python | ...



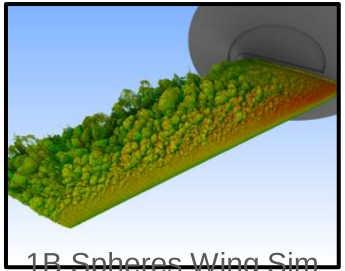
Barney Software Stack

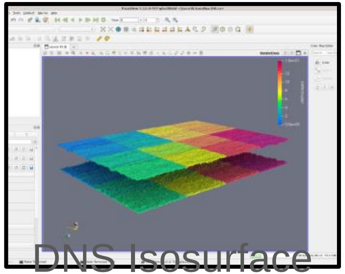
ANARI™

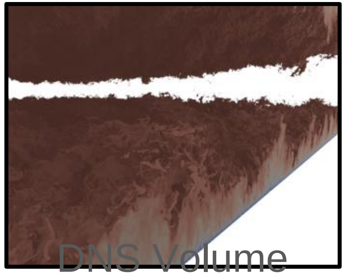
C99 | C++ | Python | ...

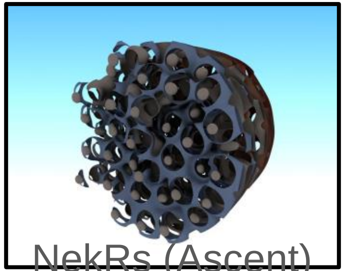


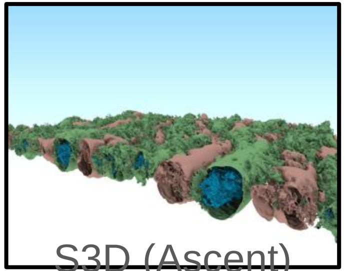
Elevating Research with ANARI

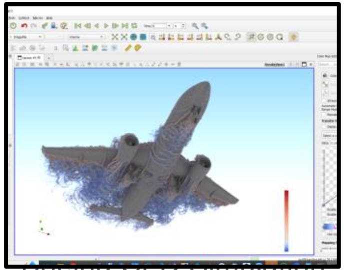

1B Spheres Wing Sim

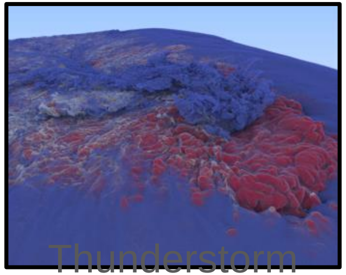

DNS Isosurface


DNS Volume

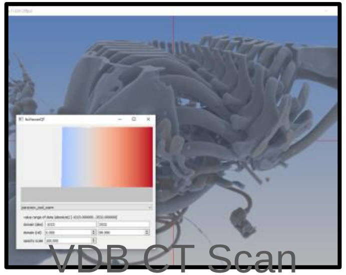

NekRs (Ascent)


S3D (Ascent)


Boeing CFD Simulation


Thunderstorm


200M DLA Particles


VDB CT Scan

Elevating Research with ANARI

Standardized Data-Parallel Rendering Using ANARI

Ingo Wald* NVIDIA
Stefan Zellmann† University of Cologne
Jefferson Amstutz‡ NVIDIA
Qi Wu§ University of California, Davis
Kevin Griffin ¶ NVIDIA
Milan Jaros‖ IT4Innovations, VSB – Technical University of Ostrava
Stefan Wesner** University of Cologne

9v1 [cs.GR] 28 Jun 2024

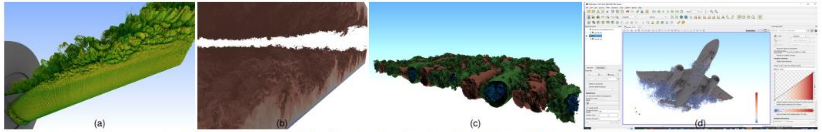


Figure 1: Several examples of large sci-vis data being rendered using the data-parallel ANARI paradigm proposed in this paper. From left to right: a) Roughly one billion color-mapped spheres, rendered using HayStack and BANARI. b) The roughly 500GB DNS data set, with volume path tracing on 128 GPUs, also using HayStack and BANARI. c) An iso-surface rendered during an in-situ Ascent session, while attached to an S3D simulation. d) ParaView performing data-parallel rendering on the `airplane` data set, using our data-parallel ANARI integration in `pvserver`.

Abstract

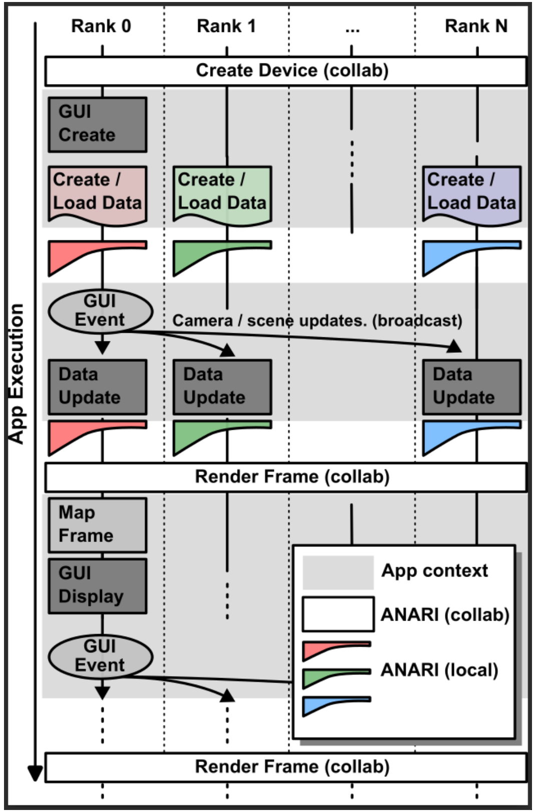
We propose and discuss a paradigm that allows for expressing *data-parallel* rendering with the classically non-parallel ANARI API. We propose this as a new standard for data-parallel sci-vis rendering, describe two different implementations of this paradigm, and use multiple sample integrations into existing apps to show how easy it is to adopt this paradigm, and what can be gained from doing so.

1 Introduction

Visualization is about more than rendering, but rendering nevertheless plays a large role in many vis tools. Rendering is hard: it was already a hard problem when all such tools could rely on a single common API (e.g. OpenGL); today, it is further complicated

involved in rendering, such as cameras or data arrays containing geometry, materials, colors, etc. These objects ultimately represent a generic interface to the private implementation of the back-end, where the mechanics of rendering frames is left up to the implementation.

ANARI is not a silver bullet, though. Even with a single agreed-upon API, different implementations can and will still differ in what features exactly they will support (and in which form). Thus, applications still need to be aware of which specific implementation they may be running on—and either adopt a least common denominator approach, or have some application features only available from specific ANARI vendors. Still, this standardization is encouraging as ANARI is already seeing adoption even in VTK and VTK-m, and



Call to Action

- Try out the ANARI-SDK: (https://github.com/KhronosGroup/ANARI-SDK)
    - Make a “hello world” ANARI program with C++ or Python
    - Integrate the API with your research application(s)
    - Try out the various implementations: VisRTX/GL, OSPRay, Visionaray, Barney, Cycles, …

Call to Action

- Try out the ANARI-SDK: (https://github.com/KhronosGroup/ANARI-SDK)
    - Make a “hello world” ANARI program with C++ or Python
    - Integrate the API with your research application(s)
    - Try out the various implementations: VisRTX/GL, OSPRay, Visionaray, Barney, Cycles, …
- Explore implementing ANARI using your renderer/engine
    - Reference the ‘helide’ device inside the SDK or other FOSS implementations

Call to Action

- Try out the ANARI-SDK: (https://github.com/KhronosGroup/ANARI-SDK)
  - Make a "hello world" ANARI program with C++ or Python
  - Integrate the API with your research application(s)
  - Try out the various implementations: VisRTX/GL, OSPRay, Visionaray, Barney, Cycles, …
- Explore implementing ANARI using your renderer/engine
  - Reference the 'helide' device inside the SDK or other FOSS implementations
- Get involved with the standard
  - Join the ANARI Working Group (for Khronos members)
    - Weekly WG meetings are Wednesdays @ 10am Pacific
  - Join the ANARI Advisory Panel (for Khronos non-members)
    - Mailing list + on-demand online discussions
  - Open issues on GitHub, both the SDK + specification

Call to Action

- Try out the ANARI-SDK: (https://github.com/KhronosGroup/ANARI-SDK)
  - Make a “hello world” ANARI program with C++ or Python
  - Integrate the API with your research application(s)
  - Try out the various implementations: VisRTX/GL, OSPRay, Visionaray, Barney, Cycles, …
- Explore implementing ANARI using your renderer/engine
  - Reference the ‘helide’ device inside the SDK or other FOSS implementations
- Get involved with the standard
  - Join the ANARI Working Group (for Khronos members)
    - Weekly WG meetings are Wednesdays @ 10am Pacific
  - Join the ANARI Advisory Panel (for Khronos non-members)
    - Mailing list + on-demand online discussions
  - Open issues on GitHub, both the SDK + specification