ANARI: ANAlytic Rendering Interface

K. Griffin[1], J. Amstutz[†1], D. DeMarle[2], J. Günther[‡2], J. Progsch[1], B. Sherman[4]
J. E. Stone[3], W. Usher[2] and K. van Kooten[1]

[1]NVIDIA, USA
[2]Intel Corporation, USA
[3]University of Illinois at Urbana-Champaign, USA
[4]National Institute of Standards and Technology, USA

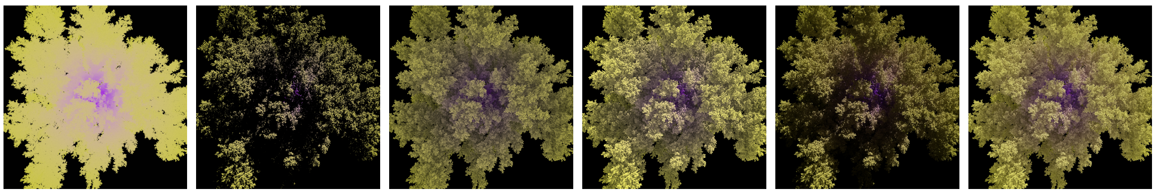

Figure 1: *Comparison of lighting techniques for a complex and crowded visualization of the results of a diffusion-limited aggregation simulation, rendered using ANARI and the VisRTX back-end device. The lighting techniques, from left, are: raycasting of surface color, directional lighting and shadows, ambient occlusion lighting, directional lighting with ambient occlusion, directional lighting with path traced indirect lighting, and directional lighting combined with ambient occlusion and path traced indirect lighting.*

Abstract
The ANARI API enables users to build the description of a scene to generate imagery, rather than specifying the details of the rendering process, providing simplified visualization application development and cross-vendor portability to diverse rendering engines, including those using state-of-the-art ray tracing.

CCS Concepts
• ***Computing methodologies*** $\rightarrow$ ***Graphics systems and interfaces; Rendering; Scientific visualization;***

1. Introduction

The fundamental problem being solved by the ANARI [SGA*22] 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 with the goal of significantly reducing overall application development-time cost.

Although many renderers and APIs already exist [WWB*14, WJA*17, PBD*10, HMCA15], 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. 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

† ANARI Working Group Chair
‡ ANARI Specification Editor

DOI: 10.2312/evp.20221126

and other more general functionality must be obtained through other APIs or applications which use ANARI.

Scientific visualization has diverse rendering needs involving trade offs among quality, speed, and scalability to available hardware resources. It is typical for visualization applications [HDS96, CBW*12, SB92, AGL05] 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.

2. API Design

The ANARI API is specified as a C99 API in order to provide compiler-independent linkage, which allows 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. It has 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. 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.

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 predominantly 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 costs and storage requirements 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.

ANARI scenes are represented as hierarchies of objects (Figure 2). The **Frame** is the root object of the scene. It holds the frame buffer configuration and the **World**, **Camera**, and **Renderer** objects. The **Camera** configures the projection of the rendering used

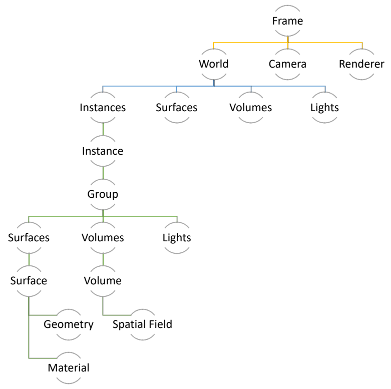

Figure 2: *ANARI scene description in memory.*

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** each containing a **Group**. A **Group** holds arrays of **Surface**, **Volume**, and **Light** to be instanced together. An **Instance** combines a **Group** with a transform for placement of the same collection of objects at multiple locations within the same **World**. A **Surface** represents drawable surfaces containing a **Geometry** and a **Material**. A **Geometry** specifies drawable primitives and data associated with them. A **Material** specifies the surface's appearance related to the data from the **Geometry**. A **Volume** represents volumetric drawable objects and may contain **Spatial Field** objects. A **Spatial Field** represents a field of values in 3D space. A **Light** represents sources of illumination.

3. Conclusion

The ANARI API enables developers to build a scene description to generate imagery, rather than specifying the details of the rendering process. Unlike more general scene graph APIs, ANARI focuses primarily on rendering operations and leaves other domain-specific scene operations in the hands of the application itself. ANARI back-end renderers are free to expose new ANARI extensions, e.g., those that add new geometric primitives, load custom shaders, or provide enhanced efficiency with other APIs. ANARI extensions can be candidates for adoption into the core ANARI API if there's enough synergy between other back-ends. As of this writing, the ANARI specification is in provisional status. Future work will consist of finalizing the remaining features needed for ANARI 1.0.

If interested, there are a couple of ways to get involved in the development of the ANARI (https://www.khronos.org/anari) specification. The first is to become a Khronos member and join the ANARI working group. The second way is to be invited by the ANARI working group to join the ANARI Advisory Panel. The ANARI SDK is available on GitHub (https://github.com/KhronosGroup/ANARI-SDK) and includes links to vendor-specific back-end renderer implementations. The latest ANARI specification is also available on GitHub (https://github.com/KhronosGroup/ANARI-Registry).

References

[AGL05] Ahrens J. P., Geveci B., Law C. C.: ParaView: An End-User Tool for Large-Data Visualization. In *The Visualization Handbook* (2005), Elsevier. ISBN: 978-0123875822. 2

[CBW*12] Childs H., Brugger E., Whitlock B., Meredith J., Ahern S., Pugmire D., Biagas K., Miller M., Harrison C., Weber G. H., Krishnan H., Fogal T., Sanderson A., Garth C., Bethel E. W., Camp D., Rübel O., Durant M., Favre J. M., Navrátil P.: VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*. Oct 2012, pp. 357–372. doi: 10.1145/2535571.2535595. 2

[HDS96] Humphrey W., Dalke A., Schulten K.: VMD – Visual Molecular Dynamics. *Journal of Molecular Graphics 14*, 1 (1996), 33–38. doi: 10.1016/0263-7855(96)00018-5. 2

[HMCA15] Hanwell M. D., Martin K. M., Chaudhary A., Avila L. S.: The Visualization Toolkit (VTK): Rewriting The Rendering Code for Modern Graphics Cards. *SoftwareX 1-2* (2015), 9–12. doi: 10.1016/j.softx.2015.04.001. URL: https://www.sciencedirect.com/science/article/pii/S2352711015000035, doi:https://doi.org/10.1016/j.softx.2015.04.001. 1

[PBD*10] Parker S. G., Bigler J., Dietrich A., Friedrich H., Hoberock J., Luebke D., McAllister D., McGuire M., Morley K., Robison A., Stich M.: OptiX: A General Purpose Ray Tracing Engine. In *ACM SIGGRAPH 2010 papers* (New York, NY, USA, 2010), SIGGRAPH '10, ACM, pp. 66:1–66:13. doi: 10.1145/1778765.1778803. 1

[SB92] Sayle R., Bissel A.: RasMol: A Program for Fast Realistic Rendering of Molecular Structures with Shadows. In *Proceedings of the 10th Eurographics UK '92 Conference* (1992). 2

[SGA*22] Stone J. E., Griffin K., Amstutz J., Demarle D. E., Sherman W., Guenther J.: ANARI: A 3D Rendering API Standard. *Computing in Science Engineering* (2022), 1–1. doi: 10.1109/MCSE.2022.3163151. 1

[WJA*17] Wald I., Johnson G., Amstutz J., Brownlee C., Knoll A., Jeffers J., Gunther J., Navratil P.: OSPRay – A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 1–1. doi: 10.1109/TVCG.2016.2599041. doi:10.1109/TVCG.2016.2599041. 1

[WWB*14] Wald I., Woop S., Benthin C., Johnson G. S., Ernst M.: Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Trans. Graph. 33*, 4 (July 2014), 143:1–143:8. doi: . 1