Experiences with Adopting ANARI in Existing Visualization Applications

John E. Stone, University of Illinois
Analytic Rendering API
Portable access to live rendering systems

- Specification development ongoing
- Exploratory implementations
- Identifying friction points
- Anyone welcome to join!

https://www.khronos.org/anari
ANARI API Position Within Visualization Stack

Visualization Apps, Frameworks, Libraries: ParaView, VisIt, VMD, VTK, ...

ANARI

Renderers: OSPRay, Radeon ProRender, VisRTX, ...

Acceleration APIs: Embree, OptiX, Vulkan, ...

Comparison of API Abstraction Levels

**NVIDIA APIs**

- OptiX $\leqslant 6.0$
- OptiX $\geqslant 7.0$

**Intel APIs**

- Embree

**Open Standards APIs**

- Vulkan
- OpenGL $\leqslant 2.0$
- OpenGL $\geqslant 3.0$
- ANARI

---

VMD – “Visual Molecular Dynamics”

100,000 active users worldwide

Visualization and analysis of:
- Molecular dynamics simulations
- Lattice cell simulations
- Quantum chemistry calculations
- Cryo-EM densities, volumetric data

User extensible scripting and plugins

http://www.ks.uiuc.edu/Research/vmd/

Cell-Scale Modeling

MD Simulation
VMD Software Decomposition

- Rendering code is 25% core VMD lines of code
  - 358,000 LoC total
  - 92,000 LoC rendering!
- Need to evolve VMD toward higher level rendering abstractions
VMD Rendering Code

• 92,000 lines in total
• Roughly 20,000 are high level abstractions
• Remaining ~72,000 lines are various (internal) full rendering engines and their support code
• Tremendous specialized expertise required to write and maintain renderers anywhere close to current state-of-the-art
VMD DisplayDevice Subclasses

- Interactive display renderer originally written using Silicon Graphics IRIS GL
- C++ DisplayDevice subclasses
  - Ease support for new interactive or offline/batch renderers
  - CAVE VR displays
  - Desktop windowed OpenGL
  - Interactive ray tracing
  - Heavily used for coupling VMD to offline/batch renderers

NVIDIA OptiX
Tachyon
Intel® OSPRay
VMD Offline Renderer Subclasses

- 20,000+ lines of code in 22+ `FileRenderer` subclasses
  - Every implementation is different despite the common VMD class hierarchy they operate within
  - Development, debugging, testing, and maintenance are all very costly in time and require significant expertise

- VMD scheme to reduce programming complexity:
  Subclasses implement virtual methods for features supported by their associated back-end renderer
  - Unimplemented virtual methods revert to superclass implementations:
  - Provide emulation / alternative / workarounds, e.g. triangulation of spheres, cylinders, cones
  - Runtime warning for anything that remains unimplemented in the scene
VMD ANARI Offline Renderer Subclass

- VMD C++ subclasses and virtual methods simplify porting and incremental bringup of new renderers
  - Emulation of missing features always comes at a performance or memory cost, but for offline renderers, these are reasonable trade-offs
  - Differences in shading and material properties tend to remain an area where different renderers still diverge and have a unique “look”

- ANARI has been EASY to implement incrementally within the VMD offline rendering framework
  - Added virtual method for material handling
  - Incrementally add virtual methods for geometric primitive types associated with ANARI subtypes
  - NEXT: better integration of ANARI with VMD’s methods for geometric instancing
VMD Examples from Early In-Progress ANARI Impl.

- Tachyon Ray Tracer (AO)
- OSPRay Path Tracer
- OptiX Path Tracer
// Use the `example` library if not overridden, this is where the impl(s) come from
const char *libname = "example";
if (getenv("ANARI_LIBRARY"))
    libname = getenv("ANARI_LIBRARY");
ANARILibrary lib = anariLoadLibrary(libname, statusFunc, NULL);
ANARIDevice dev = anariNewDevice(lib, "default");
if (!dev) {
    printf("\n\nERROR: could not load default device in example library\n");
    return -1;
}
anariCommit(dev, dev);

// .... abridged application code ...

// create and setup camera
ANARICamera camera = anariNewCamera(dev, "perspective");
float aspect = imgSize[0] / (float)imgSize[1];
anariSetParameter(dev, camera, "aspect", ANARI_FLOAT32, &aspect);
anariSetParameter(dev, camera, "position", ANARI_FLOAT32_VEC3, cam_pos);
anariSetParameter(dev, camera, "direction", ANARI_FLOAT32_VEC3, cam_view);
anariSetParameter(dev, camera, "up", ANARI_FLOAT32_VEC3, cam_up);
anariCommit(dev, camera); // commit camera object to indicate mods are done

Figure 11. Example ANARI API calls required to load and instantiate an ANARI “device” implementation, and to create a perspective camera, and set its associated parameters.

ANARISurface gen_knot(ANARIDevice dev, float *coords, float *colors,
    int numpts, float radius, ANARIMaterial mat) {
    ANARIGeometry spheres = anariNewGeometry(dev, ”sphere”);
    ANARIArray1D array;
    array = anariNewArray1D(dev, coords, 0, 0, ANARI_FLOAT32_VEC3, numpts, 0);
    anariCommit(dev, array);
    anariSetParameter(dev, spheres, ”vertex.position”, ANARI_ARRAY1D, &array);
    anariRelease(dev, array); // we are done using this handle

    anariSetParameter(dev, spheres, ”radius”, ANARI_FLOAT32, &radius);

    ANARIArray1D cols = anariNewArray1D(dev, colors, 0, 0, ANARI_FLOAT32_VEC4, numpts, 0);
    anariSetParameter(dev, spheres, ”vertex.color”, ANARI_ARRAY1D, &cols);
    anariCommit(dev, spheres);

    ANARISurface surface = anariNewSurface(dev);
    anariSetParameter(dev, surface, ”geometry”, ANARI_GEOMETRY, &spheres);
    anariSetParameter(dev, surface, ”material”, ANARI_MATERIAL, &mat);
    anariCommit(dev, surface);
    anariRelease(dev, surface); // we are done using this handle

    return surface;
}

Figure 12. ANARI API calls that add an array of spheres to the scene, adapted from the knot example.

ANARI: A 3D Rendering Interface Standard. J. E. Stone, K. Griffin, J. Amstutz, D. DeMarle,
// The world to be populated with renderable objects
ANARIWorld world = anariNewWorld(dev);

// Set the material rendering parameters
ANARIMaterial mat = anariNewMaterial(dev, "matte");
// route the primitive/vertex colors to the material’s color input
anariSetParameter(dev, mat, "color", ANARI_STRING, "color");
anariCommit(dev, mat);

// generate surface geometry
int numsurfs = 0;
ANARISurface surf[2] = { 0 };  
surfs[numsurfs++] = gen_knot(dev, coords, colors, numpts, 0.33f, mat);
surfs[numsurfs++] = gen_floor(dev, 200.0f, 3.5f, 200.0f, mat);
anariRelease(dev, mat);

// put the surfaces directly onto the world
ANARIArrayID array = anariNewArrayID(dev, surfs, 0, 0, ANARI_SURFACE, numsurfs, 0);
anariCommit(dev, array);
anariSetParameter(dev, world, "surface", ANARI_ARRAYID, &array);
anariCommit(dev, world); // finalize the completed world
for (int i=0; i<numsurfs; i++)
    anariRelease(dev, surfs[i]);
anariRelease(dev, array);

// setup renderer and associated params...
ANARIRenderer rend = anariNewRenderer(dev, "default"); // could use pathtracer, etc
float bgColor[4] = { 1.f, 1.f, 1.f, 1.f }; // white
anariSetParameter(dev, rend, "backgroundColor", ANARI_FLOAT32_VEC4, bgColor);
anariCommit(dev, rend); // finalize renderer

// create and setup frame
ANARIFrame frame = anariNewFrame(dev);
anariSetParameter(dev, frame, "size", ANARI_UINT32_VEC2, imgSize);
ANARIDataType fbFormat = ANARI_UINT32_E4B4B8;
anariSetParameter(dev, frame, "color", ANARI_DATA_TYPE, &fbFormat);
anariSetParameter(dev, frame, "renderer", ANARI_RENDERER, &rend);
anariSetParameter(dev, frame, "camera", ANARI_CAMERA, &camera);
anariSetParameter(dev, frame, "world", ANARI_WORLD, &world);
anariCommit(dev, frame);

// render and accumulate frames
for (int frames = 0; frames < 100; frames++) {
    anariRenderFrame(dev, frame); // launch async frame rendering...
    anariFrameReady(dev, frame, ANARI_WAIT); // wait for async frame completion...
}

Figure 13. ANARI example scene setup and rendering loop, adapted from the knot example.

VMD Interactive Renderer Subclasses

- Current VMD hierarchy branches off from an OpenGL renderer class (several omitted in the figure below)
  - Interactive ray tracing engines (RTRT), real-time video streaming variations and various VR display subclasses
  - RTRT mode only uses OpenGL for windowed image presentation
  - Subclasses “piggyback” on the OpenGL superclass methods, and apply further specialization
  - Performance is critical, so there’s no emulation of missing features, geometric primitives
VMD Interactive Renderer Subclasses

- Implementation of ANARI in the interactive hierarchy is FAR simpler than what was done historically for IRIS GL, OpenGL, Vulkan, etc...
- Primary consideration going forward is ensuring maximally efficient use of ANARI objects, so that ANARI back-end implementations can:
  - Use high-performance data transfer/caching methods, particularly on GPU accelerated back-ends
  - Maintain persistent data structures for unchanging scene components
  - Minimize geometry acceleration structure (BVH) (re)builds
  - Exploit efficient internal methods for object instancing
VMD Molecular Structure Data and Global State

Scene Graph

Graphical Representations
- DrawMolecule
- Non-Molecular Geometry

User Interface Subsystem
- Tcl/Python Scripting
- Mouse + Windows
- VR Input “Tools”

Display Subsystem
- VMDDisplayList
- OpenGLDisplayDevice
- FileRenderer

Display Subsystem
- Windowed OpenGL GPU
- ANARI Batch + Interactive

Scene Graph

Display Subsystem

Graphical Representations

User Interface Subsystem
ANARI VMD Integration Experiences

- ANARI 1.0 (provisional) includes complete coverage of key required camera, light, and geometry subtypes for molecular visualization
- Special VMD-specific materials will be an interesting challenge
  - VMD will likely co-evolve to exploit ANARI opportunities
  - VMD could adopt support for industry standard materials that get exposed in ANARI either as core features or extensions
- Early ANARI devices that implement the geometric subtypes needed by VMD already do well as offline renderers, both rendering performance and memory use are acceptable
- Early experiences with real-time interactive rendering shows that the ANARI back ends will work very well for static scenes
- VMD ANARI implementation for interactive fully dynamic scene geometry (simulation “trajectory” time series playback, etc) is in early stages:
  - Teaching VMD how to selectively create/destroy individual ANARI scene elements to facilitate higher interactive renderings when only a small part of a scene evolves dynamically
ANARI VMD Integration Experiences

- Integration with specific types of back-end devices/renderers:
  - ANARI devices that use OpenGL require special consideration related to OpenGL context sharing or management
  - ANARI devices using other APIs (Intel Embree, NVIDIA CUDA+OptiX, Vulkan, ...) haven’t yet required special handling since they insulate of internal state, threading, and context management
  - ANARI USD device makes use of ANARI “name” tags on the various ANARI objects, enabling the tags to be visible in GUI treeview displays of scene contents
    - Required a little finesse to convert VMD-internal scene component names into something usable in an outboard GUI/renderer combo
    - Enables VMD scenes to retain scientifically relevant object, group, instance name tags within tools like NVIDIA Omniverse Create
VMD+NVIDIA Omniverse Using ANARI USD Impl.

ANARI Developer Tools

- Special ANARI “devices” for debugging and API tracing
- Debug device interposed in-between active back-end device/renderer and application
  - Catch common ANARI programming mistakes:
    - Catch/warn on bad ANARI parameter strings
    - Use of uncommitted ANARI object
    - Leaked ANARI object (fail to make release calls)
    - Warnings for redundant commits
- ANARI API Tracing:
  - Capture+log all ANARI API calls and arrays, data, to disk files
  - Use captured API trace to (with some editing) create standalone compilable source code to replay API sequence and associated data
- Future ideas?:
  - Includable ANARI-specific profiling tag macros for popular performance analysis tools
Examples from In-Progress ANARI Impls.

ParaView visualization rendered with ANARI OSPRay back-end.

Examples from In-Progress ANARI Impls.

Knots drawn using ANARI sphere primitives with OSPRay back-end.
CAVE VR Display at NIST

Examples from In-Progress ANARI Impls.

Parametric Surfaces with ANARI VisRTX back-end.

Examples from In-Progress ANARI_Impls.

VisIt distributed memory MPI volume rendering using the ANARI VisRTX back-end, with IceT image compositing.

Examples from In-Progress ANARI Impls.

San Miguel scene using ANARI VisRTX back-end.

Q&A Panel