CHAPTER 27

Interactive Ray Tracing Techniques for High-Fidelity Scientific Visualization

John E. Stone
Beckman Institute for Advanced Science and Technology,
University of Illinois at Urbana-Champaign

ABSTRACT

This chapter describes rendering techniques and implementation considerations when using ray tracing for interactive scientific and technical visualization. Ray tracing offers a convenient framework for building high-fidelity rendering engines that can directly generate publication-quality images for scientific manuscripts while also providing high interactivity in a what-you-see-is-what-you-get rendering experience. The combination of interactivity with sophisticated rendering enables scientists who are typically not experts in computer graphics or rendering technologies to be able to immediately apply advanced rendering features in their daily work. This chapter summarizes techniques and practical approaches learned from applying ray tracing techniques to scientific visualization, and molecular visualization in particular.

27.1 INTRODUCTION

Scientific and technical visualizations are used to illustrate complex data, concepts, and physical phenomena to aid in the development of hypotheses, discover design problems, facilitate collaboration, and inform decision making. The scenes that arise in such visualizations incorporate graphical representations of the details of key structures and mechanisms and their relationships, or the dynamics of complex processes under study. High-quality ray tracing techniques have been of great use in the creation of visualizations that elucidate complex scenes. Interactivity is a powerful aid to the effectiveness of scientific visualization because it allows the visualization user to rapidly explore and manipulate data, models, and graphical representations to obtain insights and to help confirm or deny hypotheses.

Some of the challenges that arise in creating easy-to-understand visualizations involve compromises between what is shown in complete detail, what is shown just to provide important visual context, and what has to be eliminated (often sacrificed) for the sake of clarity of the visual communication. Advanced rendering techniques offer a variety of solutions to these kinds of problems. The relative ease with which

E. Haines, T. Akenine-Möller (eds.), *Ray Tracing Gems*, https://doi.org/10.1007/978-1-4842-4427-2_27

ray tracing algorithms can incorporate advanced lighting and shading models, and support a diverse range of geometric primitives and data types, make it a powerful tool for interactive rendering of geometrically complex scenes that arise in scientific and technical visualizations [2, 7, 17, 20, 24, 25].

Although ray tracing has been used for production of such visualizations in an offline or batch mode basis for decades, it has only recently reached performance levels that have made it strongly competitive with incumbent methods based on rasterization, wherein interactivity is a key requirement. The development of high-performance hardware-optimized ray tracing frameworks, and most recently ray tracing–specific hardware acceleration technologies available in commodity GPUs, has created the necessary conditions for broad use of interactive ray tracing for scientific visualization [13, 25, 26]. ParaView, VisIt, Visual Molecular Dynamics (VMD), and Visualization ToolKit (VTK)—several of the most widely used scientific visualization tools in high-performance computing—have each incorporated interactive ray tracing capabilities in the past few years. The performance gains provided by recent and upcoming ray tracing–specific hardware acceleration will hereafter create many new opportunities for interactive ray tracing to be applied in routine scientific and technical visualizations.

The remaining discussions and code samples provided in this chapter are intended to document some of the considerations, practical techniques, and elements of future outlook gained from the experience of developing and integrating three different interactive ray tracing engines within VMD, a widely used molecular visualization tool [5, 17, 19, 20, 21].

27.2 CHALLENGES ASSOCIATED WITH RAY TRACING LARGE SCENES

One of the recurring challenges that frequently arises in scientific visualization is the necessity to render scenes that reach the limits of available physical memory. Visualization approaches based on rasterization benefit from its streaming nature and typically low memory requirements. Conversely, ray tracing methods require the entire scene description to be retained in memory or made available to the ray tracing engine on demand. This is one of the key trade-offs of ray tracing methods in exchange for their flexibility, elegance, and adaptability to a wide range of rendering and visualization problems.

At the time of writing, tremendous gains in ray tracing performance have been achieved on GPUs through dedicated hardware that accelerates both bounding volume hierarchy (BVH) traversal and ray/triangle intersection tests. This advance has increased ray tracing performance to such a degree that, for scientific

visualizations employing relatively low-cost shading, memory bandwidth is now and will likely remain one of the critical factors limiting peak ray tracing performance for the foreseeable future. Considering these issues together, it is clear that the long-term successful application of ray tracing in challenging scientific visualization scenarios will depend on the development and application of techniques that make efficient use of both memory capacity and memory bandwidth.

27.2.1 USING THE RIGHT GEOMETRIC PRIMITIVE FOR THE JOB

Some of the best opportunities for savings in memory capacity and memory bandwidth relate to the choice of geometric primitives used to construct visualizations. As an example, the memory footprint for a sphere position and radius is just 4 floating-point values, whereas an individual triangle with per-vertex normals and no shared vertices requires 18 values. When representing a triangle mesh, shared vertices can be listed explicitly with vertex indices (three vertex array indices per triangle), or better yet, when feasible, they can be implied by triangle strip vertex index ordering (three indices for the first triangle, and only one index for each subsequent triangle). The memory cost of surface normals can be reduced by quantizing or compressing them significantly, further reducing the memory cost per vertex and per triangle. Ultimately, while these and related techniques can significantly reduce the memory cost for triangle meshes, direct ray tracing of spheres, cylinders, or cones rather than small triangle meshes will likely always use less memory and, more importantly in the long term, consume less memory bandwidth. While it is clear that for some domains, such as molecular visualization, large memory efficiency gains can be had through the use of a handful of bespoke geometric primitive implementations, in other scientific domains it is less clear, and the alternative geometric primitives available for consideration might involve numerical precision or convergence challenges in ray/primitive intersection test implementation, or performance attributes or anomalies that make them difficult to use effectively in all cases.

27.2.2 ELIMINATION OF REDUNDANCY, COMPRESSION, AND QUANTIZATION

Once the best choice of geometric primitives has been made, the remaining low-cost opportunities for reducing memory capacity and bandwidth requirements tend to be methods that eliminate high-level redundancies within large batches of geometric primitives. For example, particle advection streamlines used for visualization of fluid flow, magnetic fields, or electrostatic potential fields may contain millions of segments. Why store a radius per cylinder or per sphere when drawing tubular streamlines if all constituent segments have the same radius?

In the same way that rasterization pipelines have supported a broad diversity of triangle mesh formats and per-vertex data, ray tracing engines stand to benefit from similar flexibility, but for a much broader range of potential geometric primitives. For example, a ray tracing engine used to render scenes containing large numbers of streamlines of various types might employ multiple specialized geometry batch types, with radii specified per cylinder and per sphere, and with constant radii for all constituent cylinders and spheres. Depending on the degree of programmability of the underlying ray tracing framework, it might be possible to cause cylinder and sphere primitives to share the same vertex data. Furthermore, it might be possible to implement a fully customized streamline rendering primitive that implements or emulates the effect of a swept sphere following a space curve defined by the original streamline vertices themselves or by computed control points fit to the original data [23]. The more programmability available in the ray tracing framework, the more easily an application can choose the geometric primitives and geometry batching approaches that are most beneficial for resolving the memory capacity and performance issues posed by large visualizations.

After high-level redundancies have been eliminated from the encoding and parameterization of large batches of geometry, the next areas to approach are techniques that eliminate more-localized data redundancies at the level of groups of neighboring or otherwise related geometric properties. Localized data size reductions can often be made through data compression approaches and reduced-precision quantized representations of geometric attributes, or combinations of the two. When quantization or other lossy compression techniques are used, acceptable error tolerances may depend on the details of the visualization problem at hand. Two representative examples of these techniques are compression of volumetric data, scalar fields, and tensors, e.g., as provided by the ZFP library [8, 9], and quantized representations of surface normals, as in octahedron normal vector encoding [4, 12]. See Listing 27-4 for an example implementation of normal packing and unpacking using octahedron normal encoding.

Listing 27-1. *This code snippet lists the key functions required to implement normal packing and unpacking using octahedron normal vector encoding. The routines convert back and forth between normal vectors represented as three single-precision oating-point values and a single packed 32-bit unsigned integer encoding. Many performance optimizations and improvements are possible here, but these routines are easy to try out in your own ray tracing engine.*

```
1 # include <optixu/optixu_math_namespace.h> // For make_xxx() functions
2
3 // Helper routines that implement the floating-point stages of
4 // octahedron normal vector encoding
```

```
 5 static __host__ __device__ __inline__
 6 float3 OctDecode(float2 projected) {
 7   float3 n;
 8   n = make_float3(projected.x, projected.y,
 9                   1.0f - (fabsf(projected.x) + fabsf(projected.y)));
10   if (n.z < 0.0f) {
11     float oldX = n.x;
12     n.x = copysignf(1.0f - fabsf(n.y), oldX);
13     n.y = copysignf(1.0f - fabsf(oldX), n.y);
14   }
15   return n;
16 }
17
18 static __host__ __device__ __inline__
19 float2 OctEncode(float3 n) {
20   const float invL1Norm = 1.0f / (fabsf(n.x)+fabsf(n.y)+fabsf(n.z));
21   float2 projected;
22   if (n.z < 0.0f) {
23     float2 tmp = make_float2(fabsf(n.y), fabsf(n.x));
24     projected = 1.0f - tmp * invL1Norm;
25     projected.x = copysignf(projected.x, n.x);
26     projected.y = copysignf(projected.y, n.y);
27   } else {
28     projected = make_float2(n.x, n.y) * invL1Norm;
29   }
30   return projected;
31 }
32
33 // Helper routines to quantize to or invert the quantization
34 // to and from packed unsigned integer representations
35 static __host__ __device__ __inline__
36 uint convfloat2uint32(float2 f2) {
37   f2 = f2 * 0.5f + 0.5f;
38   uint packed;
39   packed = ((uint) (f2.x * 65535)) | ((uint) (f2.y * 65535) << 16);
40   return packed;
41 }
42
43 static __host__ __device__ __inline__
44 float2 convuint32float2(uint packed) {
45   float2 f2;
46   f2.x = (float)((packed      ) & 0x0000ffff) / 65535;
47   f2.y = (float)((packed >> 16) & 0x0000ffff) / 65535;
48   return f2 * 2.0f - 1.0f;
49 }
50
51 // The routines to be called when preparing geometry buffers prior
52 // to ray tracing and when decoding them on-the-fly during rendering
```

```
53 static __host__ __device__ __inline__
54 uint packNormal(const float3& normal) {
55   float2 octf2 = OctEncode(normal);
56   return convfloat2uint32(octf2);
57 }
58
59 static __host__ __device__ __inline__
60 float3 unpackNormal(uint packed) {
61   float2 octf2 = convuint32float2(packed);
62   return OctDecode(octf2);
63 }
```

The atomic-detail molecular structure shown in Figure 27-1 demonstrates the use of all the techniques described in this section, using both triangle meshes and bespoke geometric primitive implementations, with redundancy elimination approaches applied to geometry encoding and batching, along with octahedron normal vectors. An example implementation of normal packing using octahedron normal encoding is included to demonstrate the value and application of the technique in interactive ray tracing. Vertex normals are not required for ray/triangle intersection tests. Normals are only referenced when the closest-hit result has been found and must be shaded. As such, the costs of on-the-fly inverse quantization or decompression during shading are low, and for interactive ray tracing of large, geometrically complex scenes, they tend to have negligible impact on frame rates while providing substantial memory savings. Similar approaches can be applied to per-vertex colors and other attributes, potentially with even greater practical effect.

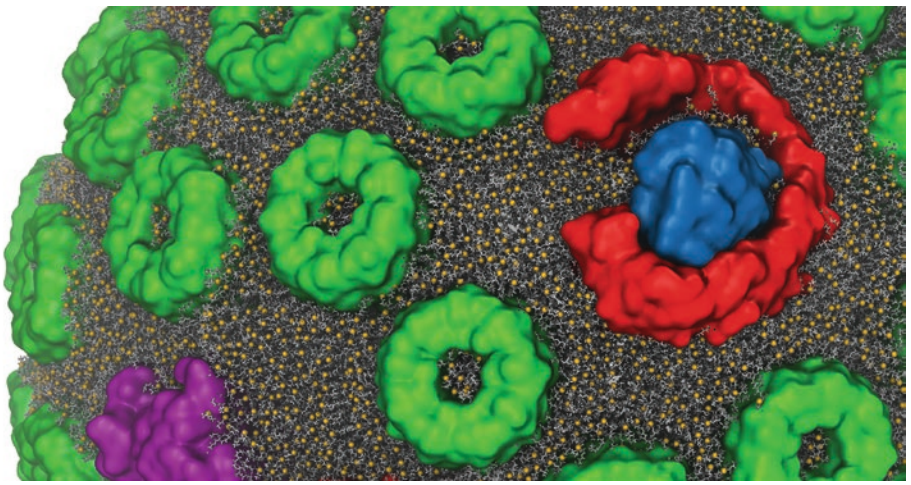

Figure 27-1. *Closeup visualization of an atomic-detail model of the lipid membrane in a photosynthetic chromatophore structure. Contextual parts of the model are visualized with triangle mesh surface representations using octahedron normal vectors. The atomic details shown in the lipid membrane are composed of tens of millions of individual spheres and cylinders. The memory savings associated with the use of direct ray tracing of custom sphere and cylinder arrays makes interactive ray tracing of this large structure feasible while maintaining high performance on commodity GPUs [20].*

27.2.3 CONSIDERATIONS FOR RAY TRACING ACCELERATION STRUCTURES

Beyond the direct memory cost associated with a given geometric primitive, it is important to consider the per-primitive memory costs associated with the BVH or other ray tracing acceleration structure that ultimately contains them. It can be surprising that, despite the use of data compression techniques in state-of-the-art ray tracing acceleration structures, the acceleration structures themselves can sometimes end up being as large or larger in size than the scene geometry they encode. Acceleration structures and their space-versus-time trade-offs are therefore an area of significant concern for applications of ray tracing to scientific visualizations. Since acceleration structure construction, storage, and traversal are all performance-critical aspects of ray tracing, they are frequently proprietary, highly hardware-optimized, and therefore often less flexible than one might prefer.

For visualization of static structures, large and highly optimized acceleration structures yield the best performance since construction and update costs are relatively unimportant. For interactive display of time series data such as simulation trajectories, time spent on geometry buffer updates and acceleration structure (re)builds becomes an important factor in interactivity. Time series animation is a much more complex case that can benefit significantly from increased concurrency, e.g., via multithreading techniques. To completely decouple geometry updates and acceleration structure (re)builds from ongoing interactive rendering and display, it is necessary to employ double- or multi-buffering of key ray tracing data structures. Multi-buffering of ray tracing data structures permits scene updates to occur concurrently and asynchronously with ongoing rendering.

The need for flexibility in ray tracing acceleration structure optimization is of particular interest for both large, static scenes and for dynamic time series visualizations. When visualizing large scientific scenes that have extremely high geometric complexity, often the memory required by the acceleration structure exceeds available capacity. In such cases it is usually preferable to build a moderately coarser acceleration structure that sacrifices some performance in favor of increased geometric capacity. The use of a coarser acceleration structure may also turn out to be a desirable trade-off for time series visualizations. Some existing ray tracing frameworks provide simple controls over acceleration structure construction heuristics and tunables for these purposes. This remains an area of active development where one can expect future ray tracing engines to make significant advances.

27.3 VISUALIZATION METHODS

In this section, several simple but extremely useful ray tracing–compatible shading techniques are described, along with descriptions of their practical use and implementation. Scientists and technicians who use visualization tools have tremendous domain expertise, but they often have only moderate familiarity with optics, lighting, shading, and computer graphics techniques in general. A key component of the techniques described here is that they are easily used by nonexpert visualization practitioners, particularly when implemented in a fully interactive ray tracing engine with progressive refinement and other niceties. A panoply of excellent shading techniques are available for scientific visualization applications based on rasterization. However, many of these depend on rasterization-specific techniques or API features, and they may not be compatible with the range of lighting and shading techniques commonly used in interactive ray tracing visualization engines. The techniques described next have low performance costs, can be combined with other ray tracing features, and, most importantly, have seen ongoing use in the creation of effective visualizations.

The ray tracing methods described here provide several useful scientific visualization tools for ambient occlusion lighting, non-photorealistic transparent surfaces, edge outlining of opaque surfaces, and clipping planes and spheres, each of which can contribute to improving the clarity and interpretation of resulting visualizations.

27.3.1 AMBIENT OCCLUSION LIGHTING IN SCIENTIFIC VISUALIZATION

A key value of ambient occlusion (AO) lighting for scientific and technical visualization is its tremendous time savings, particularly when paired with complex scenes and other high-fidelity ray tracing techniques. AO can be useful for interactive viewing of complex models, but especially for time series data such as simulation trajectories, when it is impractical for a user to continually adjust manually placed lights to achieve a desirable lighting outcome [19, 22]. The "ambient" aspect of AO lighting is what makes it such a convenient tool for nonexpert users. With interactive use of AO and progressive ray tracing, users need not become experts at lighting design and can instead achieve a "good" lighting arrangement by adjusting one or two key ambient occlusion lighting parameters, typically in combination with one or two manually positioned directional or point light sources. This is particularly true in domains such as molecular visualization, where the visualization lighting design is solely for elucidating details of molecular structure and is not an attempt to replicate a photorealistic scene of some sort. One way in which the application of AO can be made easy for beginners is to provide independent light scaling factors for both

AO ("ambient") and manually placed ("direct") light sources. By providing separate easy-to-use global intensity scaling factors for ambient and direct lighting, beginners find it easier to balance their lighting design and avoid both over-lit and under-lit conditions that can otherwise easily occur in geometrically complex scenes that contain pockets, pores/tunnels, or cavities that each pose lighting challenges.

27.3.1.1 AO WITH LIMITED OCCLUSION DISTANCE

A problem with AO that often arises when exploring scenes with densely packed geometry is that there are few paths for the "ambient" light to get deep within a complex structure, such as within a virus capsid or a cell membrane. A simple but effective solution to this problem is to compute AO lighting with a maximum occlusion distance, beyond which ambient occlusions are ignored. Using this technique, one can choose a maximum occlusion distance that comfortably fits within the confined viewing spaces of interest, maintaining the key benefits of AO for visualization purposes, as shown in Figure 27-2. While a camera-centered point light could be used to light dark interiors of largely or fully enclosed structures, it would result in an undesirable flat-looking surface. This too could be resolved by careful manual or offset placement of multiple point lights or area lights, but such tasks are ultimately undesirable distractions that take away from unrestricted interactive exploration of complex models or simulation results. The use of AO with a limited occlusion distance avoids these undesirable issues while maintaining unrestricted interactive scene navigation. A further, perhaps unanticipated, benefit of this type of approach is that the maximum AO occlusion distance can also be used to shadow only pores, pockets, and cavities of a particular maximum diameter range, converting AO lighting into a tool capable of highlighting particular geometric features with a mild degree of selectivity. This technique can be refined further by incorporating user-specified AO falloff attenuation coefficients, if desired. See Listing 27-2 for a simple example implementation.

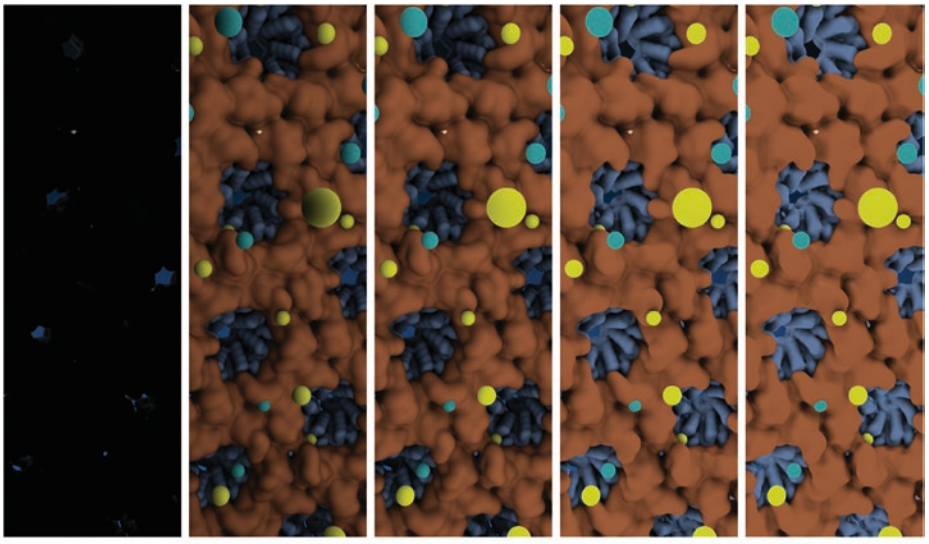

Figure 27-2. *Visualization of the interior of the HIV-1 capsid at various settings of the AO lighting maximum occlusion distance. (a) Conventional AO lighting: since the virus capsid completely encloses the viewpoint, only a few thin shafts of light enter the interior through pores in the capsid structure, leaving it almost completely dark. (b) The user-specified maximum occlusion distance was set to slightly less than the minor interior diameter of the capsid. The remaining images show this distance decreased by a factor of(c) 2, (d) 8, and (e) 16.*

Listing 27-2. *This closest-hit shader code snippet skips shading of transparent surfaces when the incident ray has crossed through a user-defined maximum number of transparent surfaces, proceeding instead by shooting a transmission ray and continuing as though there had been no ray/surface intersection.*

```
1 struct PerRayData_radiance {
2   float3 result;    // Final shaded surface color
3   // ...
4 }
5
6 struct PerRayData_shadow {
7   float3 attenuation;
8 };
9
10 rtDeclareVariable(PerRayData_radiance, prd, rtPayload, );
11 rtDeclareVariable(PerRayData_shadow, prd_shadow, rtPayload, );
12
13 rtDeclareVariable (float, ao_maxdist, , ); // max AO occluder distance
14
15 static __device__
16 float3 shade_ambient_occlusion(float3 hit, float3 N,
17                                float aoimportance) {
18   // Skipping boilerplate AO shadowing material here ...
19
```

```
20   for (int s=0; s<ao_samples; s++) {
21     Ray aoray;
22     // Skipping boilerplate AO shadowing material here ...
23     aoray = make_Ray (hit, dir, shadow_ray_type,
24                       scene_epsilon, ao_maxdist);
25
26     shadow_prd.attenuation = make_float3(1.0f);
27     rtTrace(root_shadower, ambray, shadow_prd);
28     inten += ndotambl * shadow_prd.attenuation;
29   }
30
31   return inten * lightscale;
32 }
33
34 RT_PROGRAM void closest_hit_shader( ... ) {
35   // Skipping boilerplate closest-hit shader material here ...
36
37   // Add ambient occlusion diffuse lighting, if enabled.
38   if (AO_ON && ao_samples > 0) {
39     result *= ao_direct;
40     result += ao_ambient * col * p_Kd *
41               shade_ambient_occlusion(hit_point, N, fogf * p_opacity);
42   }
43
44   // Continue with typical closest-hit shader contents ...
45
46   prd.result = result; // Pass the resulting color back up the tree.
47 }
```

27.3.1.2 REDUCING MONTE CARLO SAMPLING NOISE

Scientists who use visualization tools frequently need to generate quick "snapshot" renderings for routine use in team meetings and presentations. Being perpetually short of time, there is a tendency for users to prefer high-fidelity rendering approaches, but with the condition that rendering can be halted at any point, providing them with an image that is free of "grain" or "speckle," albeit without having fully converged lighting or depth of field focal blur.

A particularly promising class of state-of-the-art techniques for real-time denoising employs carefully trained deep neural networks to eliminate grain and speckle noise in undersampled regions of images produced by Monte Carlo rendering [3, 6, 10, 15, 16]. The success of so-called artificially intelligent (AI) denoisers often depends on the availability of auxiliary image data buffers containing depth, surface normals, albedo, and other types of information that help the denoiser do a better job of identifying noise and undersampled image

regions. The interactive-rate performance of AI denoisers also hinges upon the availability of hardware-accelerated AI inferencing, which enables the denoiser to outrun brute-force sampling, even on hardware platforms with dedicated ray tracing hardware acceleration. It appears likely that AI denoising will remain one of best and most broadly used approaches for denoising in sophisticated path tracing, and in ray tracing engines more generally, because the techniques can be tuned or trained specifically for particular renderers and scene content.

Besides sophisticated denoising techniques, one can also make potentially beneficial trade-offs between high-frequency noise content and the correlation of stochastic samples, e.g., resulting in visible AO shadow boundary edges in undersampled interactive renderings. In conventional ray tracing technique, ambient occlusion lighting and other Monte Carlo sampling implementations typically use completely uncorrelated pseudo-random or quasi-random number sequences to generate directions for AO lighting shadow feeler rays within the hemisphere normal to the surface being shaded. With an uncorrelated sampling approach, when a sufficient number of AO lighting samples have been taken, a smooth grain-free image results. However, early termination of an unconverged sampling process results in a grainy looking image. By purposefully correlating AO samples in all image pixels, e.g., by seeding AO random number generators or quasi-random sequence generators with the same seed, all pixels in the image will choose the same AO shadow feeler directions, and there will be no image grain from AO. This approach is particularly well suited for interactive ray tracing of geometrically complex scenes that would otherwise require a large number of samples to achieve grain-free images.

27.3.2 EDGE-ENHANCED TRANSPARENT SURFACES

A common problem that arises in molecular visualizations is the need to clearly display the boundaries of molecular complexes or their constituent substructures, while making it easy to see the details of their internal structures. Molecular scientists spend significant effort selecting what should be shown and how it should be displayed. Raster3D [11], Tachyon [18], and VMD [5, 20] employ special shaders that make it easy to see the interior of a structure by making viewer-directed surfaces entirely transparent, while leaving the boundary regions that are seen edge-on largely opaque. The surface shader instantly adapts to changes in viewing orientation, permitting the user to freely rotate the molecular complex while maintaining an unobscured view of interior details. This technique is demonstrated effectively in Figure 27-3, where it is applied to light-harvesting complexes and photosynthetic reaction centers, and in Figure 27-4, where it is applied to a solvent box and solvent/protein interface. See Listing 27-3 for the details of the shader implementation.

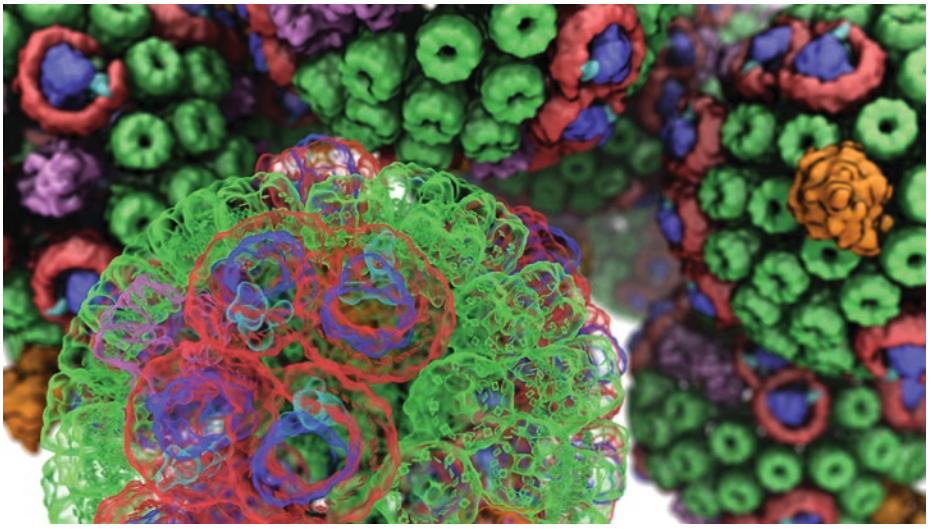

Figure 27-3. *Visualization of the intracellular packing of chromatophore light-harvesting vesicles that use photosynthesis to produce ATP, the chemical fuel for living cells. The foreground chromatophore vesicle is shown with transparent molecular surfaces to reveal selected interior atomic structures of the rings of chlorophyll pigments within each of its individual photosynthetic complexes and reaction centers. Background instances of opaque chromatophores show the crowded packing of chromatophore vesicles within the cytoplasm of a purple bacterium.*

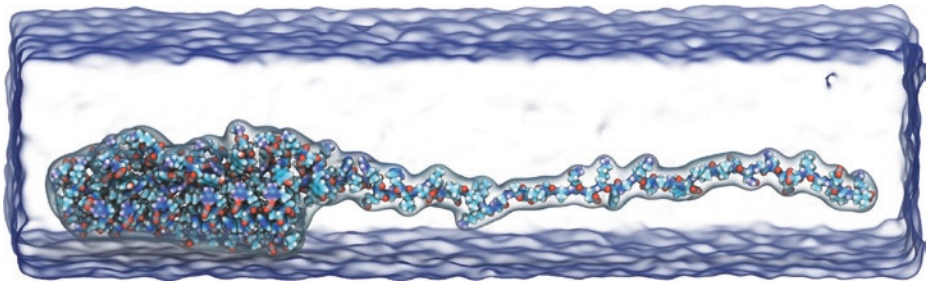

Figure 27-4. *Visualization of the molecular dynamics of an unfolding Ankyrin protein, with solvent (water and ions) surfaces rendered using the edge-enhanced transparent surface shading technique [1].*

Listing 27-3. *This example code snippet makes viewer-facing surfaces appear completely transparent while leaving surfaces seen edge-on more visible and opaque. This type of rendering is extremely useful to facilitate views into the interior of crowded scenes, such as densely packed biomolecular complexes.*

```
1 RT_PROGRAM void closest_hit_shader( ... ) {
2   // Skipping boilerplate closest-hit shader material here ...
3
4   // Exemplary simplified placeholder for typical
5   // transmission ray launch code
```

```
 6   if (alpha < 0.999f) {
 7     // Emulate Tachyon/Raster3D's angle-dependent surface opacity
 8     if (transmode) {
 9       alpha = 1.0f + cosf(3.1415926f * (1.0f-alpha) *
10                       dot(N, ray.direction));
11       alpha = alpha*alpha * 0.25f;
12     }
13     result *= alpha; // Scale down lighting by any new transparency
14
15     // Skipping boilerplate code to prepare a new transmission ray ...
16     rtTrace(root_object, trans_ray, new_prd);
17   }
18   result += (1.0f - alpha) * new_prd.result;
19
20   // Continue with typical closest-hit shader contents ...
21
22   prd.result = result; // Pass the resulting color back up the tree.
23 }
```

27.3.3 PEELING AWAY EXCESS TRANSPARENT SURFACES

Many domains within scientific visualization produce scenes that incorporate significant amounts of partially transparent geometry, often to display surfaces within volumetric data of various types, e.g., electron density maps, medical images, tomograms from cryo-electron microscopy, or flow fields from computational fluid dynamics simulations. When rendering scenes containing complex or noisy volumetric data, transparent isosurfaces and contained geometry may become more difficult to interpret visually, and it is often helpful to create purposefully non-photorealistic renderings that "peel away" all but the first, or first few, layers of transparent surfaces so they do not create a distracting background behind features of particular interest. See Figure 27-5. Transparent surfaces can be peeled as described by making a small modification to a canonical closest-hit program: store an additional counter for transparent surface crossing as an extra per-ray data item. When primary rays are generated, the crossing counter is initially set to the maximum number of transparent surfaces to be shown. As the ray is traced through the scene, the per-ray transparent surface crossing counter is decremented on each transparent surface until it reaches zero. Once this happens, all subsequent intersections with transparent surfaces are ignored, i.e., they are not shaded and do not contribute to the final color, and transmission rays are generated to continue as if no intersection had occurred. See Listing 27-4 for an example implementation.

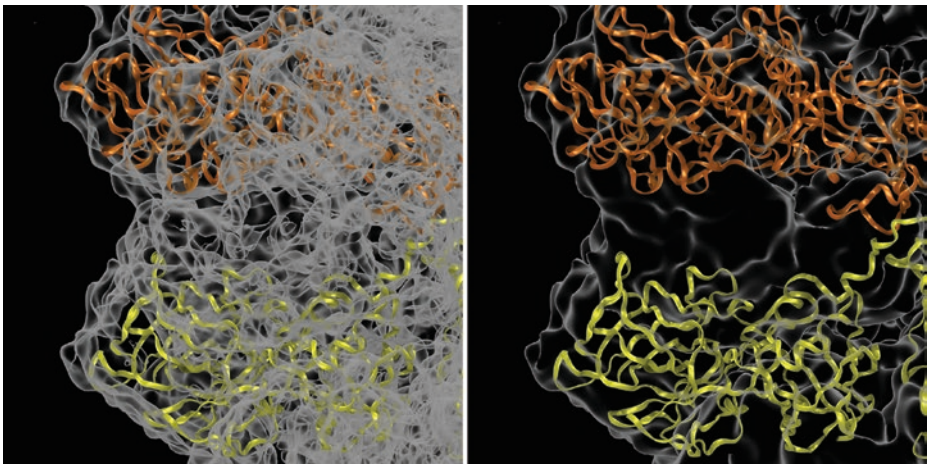

Figure 27-5. *Closeup visualization of an atomic-detail structure of rabbit hemorrhagic disease virus, obtained through X-ray crystallography and computational modeling and fit into a low-resolution electron density map from cryo-electron microscopy using molecular dynamics flexible fitting: the results of conventional ray traced transparency (left), and the transparency peeling approach that eliminates obscuration of details of the fitted interior atomic structures (right).*

Listing 27-4. *This closest-hit shader code snippet skips the shading of transparent surfaces when the incident ray has crossed through a user-defined maximum number of transparent surfaces, proceeding instead by shooting a transmission ray and continuing as though there had been no ray/surface intersection.*

```
struct PerRayData_radiance {
  float3 result;     // Final shaded surface color
  int transcnt;      // Transmission ray surface count/depth
  int depth;         // Current ray recursion depth
  // ...
}

rtDeclareVariable(PerRayData_radiance, prd, rtPayload, );

RT_PROGRAM void closest_hit_shader( ... ) {
  // Skipping boilerplate closest-hit shader material here ...

  // Do not shade transparent surface if the maximum
  // transcnt has been reached.
  if ((opacity < 1.0) && (transcnt < 1)) {
    // Spawn transmission ray; shading behaves as if there
    // had been no intersection.
    PerRayData_radiance new_prd;
    new_prd.depth = prd.depth; // Do not increment recursion depth.
    new_prd.transcnt = prd.transcnt - 1;
    // Set/update various other properties of the new ray.

```

```
23      // Shoot the new transmission ray and return its color as if
24      // there had been no intersection with this transparent surface.
25      Ray trans_ray = make_Ray(hit_point, ray.direction,
26                               radiance_ray_type, scene_epsilon,
27                               RT_DEFAULT_MAX);
28      rtTrace(root_object, trans_ray, new_prd);
29    }
30
31    // Otherwise, continue shading this
32    // transparent surface hit point normally ...
33
34    // Continue with typical closest-hit shader contents ...
35    prd.result = result; // Pass the resulting color back up the tree.
36  }
```

27.3.4 EDGE OUTLINES

The addition of edge outlining on opaque geometry is often helpful in making the depth and spatial relationships between nearby objects or surfaces of the same color much more obvious and easy to interpret. Edge outlining can be used both to further enhance the visibility of salient details of surface structure, such as protrusions, pores, or pockets, and can be used either with light effects for detailed renderings or with a much stronger effect to remain visible when blurred or faded by depth of field or depth cueing. Figure 27-6 shows two examples of edge outlining applied to both foreground and background contextual structures in combination with depth of field focal blur and depth cueing.

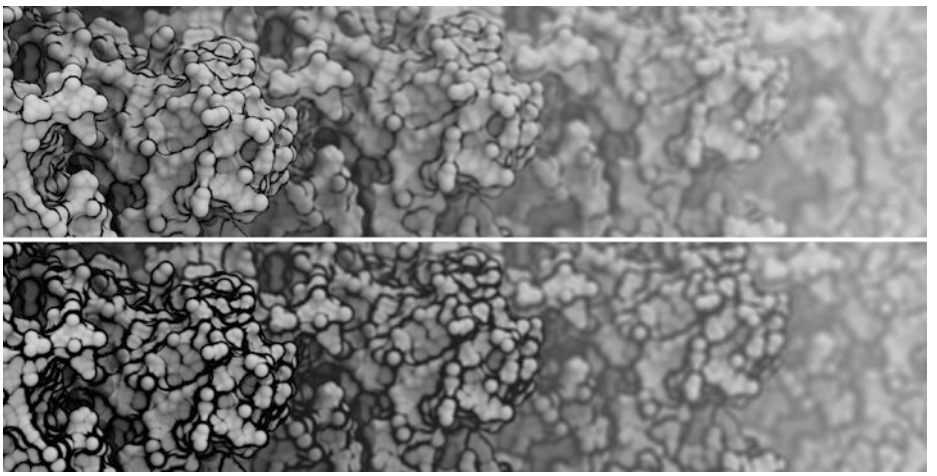

Figure 27-6. *Visualization of molecular surfaces with edge outlining applied to enhance the visibility of significant structural features and with depth of field and depth cueing (fog) used. Top: edge outlining has been applied relatively sparingly and is only easily visible on the in-focus foreground molecular surfaces. Bottom: the edge outline width has been significantly increased. Although the wide edge outline might be excessive when applied to in-focus foreground structures, it allows salient features of the molecular structure to be seen even in the most distant structures that have been blurred and faded.*

While many outlining techniques exist for conventional rasterization pipelines, they are usually implemented in multi-pass rendering approaches that often require access to a depth buffer, which is not well suited to the internal workings of most ray tracing engines. For many years, VMD and Tachyon have implemented an easy-to-use outline shader that is simple to implement within ray tracing engines as it does not require depth buffer access, deferred shading, or other extra rendering passes. See Listing 27-5 for an example implementation.

Listing 27-5. *This example code snippet adds a dark outline on the edges of geometry to help accentuate objects that are packed closely together and may not otherwise be visually distinct.*

```
struct PerRayData_radiance {
  float3 result;      // Final shaded surface color
  // ...
}

rtDeclareVariable(PerRayData_radiance, prd, rtPayload, );

// Example of instantiating a shader with outlining enabled ...
RT_PROGRAM void closest_hit_shader_outline( ... ) {
  // Skipping boilerplate closest-hit shader material here ...

  // Add edge shading, if applicable.
  if (outline > 0.0f) {
    float edgefactor = dot(N, ray.direction);
    edgefactor *= edgefactor;
    edgefactor = 1.0f - edgefactor;
    edgefactor = 1.0f - powf(edgefactor, (1.0f-outlinewidth) * 32.0f);
    result *= __saturatef((1.0f-outline) + (edgefactor * outline));
  }

  // Continue with typical closest -hit shader contents ...

  prd.result = result; // Pass the resulting color back up the tree.
}
```

27.3.5 CLIPPING PLANES AND SPHERES

One of the powerful rendering capabilities long enjoyed by users of advanced ray tracing engines is constructive solid geometry (CSG), which models complex geometry with unions, intersections, and differences between arbitrary numbers of basic geometric primitives [14]. CSG can be a powerful tool for modeling complex shapes, but in scientific visualization a user frequently needs easy-to-use tools for cutting away visual obscuration, which can be performed using just CSG differences. When interactively visualizing large scenes, it is often impractical to make significant changes to the underlying model or data in the scene within the

available frame rate budget. However, approaches that leave the model unchanged and instead manipulate only the low-level rendering process are often still feasible under such constraints. Fully general CSG implementations require somewhat extensive bookkeeping, but clipping geometry is a special case that can be achieved far more simply. Since ray tracing engines do their work by computing and sorting intersections, it is usually easy to implement user-defined clipping planes, spheres, or other clipping geometry within the intersection management logic. This is particularly true if clipping geometry applies globally to everything in the scene, since that case incurs insignificant bookkeeping overhead. Global clipping geometry can typically be added to any ray tracing engine by computing the clipping geometry intersection distances and storing them in per-ray data for use when rendering the rest of the scene geometry. See Listing 27-6 for an example implementation.

Listing 27-6. *This excerpt from Tachyon shows the simplicity with which one can implement a basic user-defined clipping plane feature (that globally clips all objects, when enabled) by storing clipping plane information in per-ray data and adding a simple distance comparison for each of the clipping plane(s) to be tested.*

```
/* Only keeps closest intersection, no clipping, no CSG */
void add_regular_intersection(flt t, const object * obj, ray * ry) {
  if (t > EPSILON) {
    /* if we hit something before maxdist update maxdist */
    if (t < ry->maxdist) {
      ry->maxdist = t;
      ry->intstruct.num=1;
      ry->intstruct.closest.obj = obj;
      ry->intstruct.closest.t = t;
    }
  }
}

/* Only keeps closest intersection, also handles clipping, no CSG */
void add_clipped_intersection(flt t, const object * obj, ray * ry) {
  if (t > EPSILON) {
    /* if we hit something before maxdist update maxdist */
    if (t < ry->maxdist) {

      /* handle clipped object tests */
      if (obj->clip != NULL) {
        vector hit;
        int i;

```

```
25          RAYPNT(hit, (*ry), t); /* find hit point for further tests */
26          for (i =0; i<obj->clip->numplanes; i++) {
27            if ((obj->clip->planes[i * 4    ] * hit.x +
28                 obj->clip->planes[i * 4 + 1] * hit.y +
29                 obj->clip->planes[i * 4 + 2] * hit.z) >
30                 obj->clip->planes[i * 4 + 3]) {
31              return; /* hit point was clipped */
32            }
33          }
34        }
35
36        ry->maxdist = t;
37        ry->intstruct.num=1;
38        ry->intstruct.closest.obj = obj;
39        ry->intstruct.closest.t = t;
40      }
41    }
42 }
43
44 /* Only meant for shadow rays, unsafe for anything else */
45 void add_shadow_intersection(flt t, const object * obj, ray * ry) {
46   if (t > EPSILON) {
47     /* if we hit something before maxdist update maxdist */
48     if (t < ry->maxdist) {
49       /* if this object doesn't cast a shadow, and we aren't  */
50       /* limiting the number of transparent surfaces to less  */
51       /* than 5, then modulate the light by its opacity value */
52       if (!(obj->tex->flags & RT_TEXTURE_SHADOWCAST)) {
53         if (ry->scene->shadowfilter)
54           ry->intstruct.shadowfilter *= (1.0 - obj->tex->opacity);
55         return;
56       }
57
58       ry->maxdist = t;
59       ry->intstruct.num=1;
60
61       /* if we hit *anything* before maxdist, and we're firing a */
62       /* shadow ray, then we are finished ray tracing the shadow */
63       ry->flags |= RT_RAY_FINISHED;
64     }
65   }
66 }
67
68 /* Only meant for clipped shadow rays, unsafe for anything else */
69 void add_clipped_shadow_intersection(flt t, const object * obj,
70                                      ray * ry) {
71   if (t > EPSILON) {
72     /* if we hit something before maxdist update maxdist */
73     if (t < ry->maxdist) {
74       /* if this object doesn't cast a shadow, and we aren't */
75       /* limiting the number of transparent surfaces to less */
```

```
76        /* than 5, then modulate the light by its opacity value */
77        if (!(obj->tex->flags & RT_TEXTURE_SHADOWCAST)) {
78          if (ry->scene->shadowfilter)
79            ry->intstruct.shadowfilter *= (1.0 - obj->tex->opacity);
80          return;
81        }
82
83        /* handle clipped object tests */
84        if (obj->clip != NULL) {
85          vector hit;
86          int i;
87
88          RAYPNT(hit, (*ry), t); /* find hit point for further tests */
89          for (i=0; i<obj->clip->numplanes; i++) {
90            if ((obj->clip->planes[i * 4    ] * hit.x +
91                 obj->clip->planes[i * 4 + 1] * hit.y +
92                 obj->clip->planes[i * 4 + 2] * hit.z) >
93                 obj->clip->planes[i * 4 + 3]) {
94              return; /* hit point was clipped */
95            }
96          }
97        }
98
99        ry->maxdist = t;
100       ry->intstruct.num=1;
101
102       /* if we hit *anything* before maxdist, and we're firing a */
103       /* shadow ray, then we are finished ray tracing the shadow */
104       ry->flags |= RT_RAY_FINISHED;
105     }
106   }
107 }
```

27.4 CLOSING THOUGHTS

This chapter has described many of the benefits and challenges associated with the use of interactive ray tracing techniques for scientific visualization. Since the major strengths of ray tracing are well known, this chapter included a few unconventional techniques that combine non-photorealistic approaches with the classic strengths of ray tracing to solve tricky visualization problems. Although most of the example images and motivations given are biomolecular in nature, these approaches are of value in many other areas as well.

An exciting area of my own and others' research is the ongoing development of using techniques such as interactive path tracing for scientific visualization. Path tracing used to be too costly to be practical for many routine visualization

tasks that a scientist might perform on a daily basis. However, when the ray tracing performance provided by state-of-the-art hardware is combined with the latest techniques for Monte Carlo image denoising, interactive path tracing becomes feasible for a wide spectrum of visualization workloads without having to compromise on either interactivity or image quality. These developments are of particular value for scientific and technical visualizations where improved photorealism is important.

The code examples provided with the chapter are intended to serve as exemplary starting points for further specialization. Each of the techniques can be significantly extended to add new capabilities far beyond what is demonstrated here, and I have tried to strike a balance between simplicity, reusability, and completeness.

ACKNOWLEDGMENTS

This work was supported in part by the National Institutes of Health, under grant P41-GM104601. The author thanks Melih Sener and Angela Barragan for the use of the chromatophore models. The author wishes to thank many current and former colleagues in the Theoretical and Computational Biophysics Group at the University of Illinois for years of collaboration on the design of the VMD molecular visualization software and the use of advanced rendering techniques for production of effective visualizations.

REFERENCES

[1] Borkiewicz, K., Christensen, A. J., and Stone, J. E. Communicating Science Through Visualization in an Age of Alternative Facts. In *ACM SIGGRAPH Courses* (2017), pp. 8:1–8:204.

[2] Brownlee, C., Patchett, J., Lo, L.-T., DeMarle, D., Mitchell, C., Ahrens, J., and Hansen, C. D. A Study of Ray Tracing Large-Scale Scientific Data in Two Widely Used Parallel Visualization Applications. In *Eurographics Symposium on Parallel Graphics and Visualization* (2012), pp. 51–60.

[3] Chaitanya, C. R. A., Kaplanyan, A. S., Schied, C., Salvi, M., Lefohn, A., Nowrouzezahrai, D., and Aila, T. Interactive Reconstruction of Monte Carlo Image Sequences Using a Recurrent Denoising Autoencoder. *ACM Transactions on Graphics 36*, 4 (July 2017), 98:1–98:12.

[4] Cigolle, Z. H., Donow, S., Evangelakos, D., Mara, M., McGuire, M., and Meyer, Q. A Survey of Efficient Representations for Independent Unit Vectors. *Journal of Computer Graphics Techniques 3*, 2 (April 2014), 1–30.

[5] Humphrey, W., Dalke, A., and Schulten, K. VMD—Visual Molecular Dynamics. *Journal of Molecular Graphics 14*, 1 (1996), 33–38.

[6] Kalantari, N. K., Bako, S., and Sen, P. A Machine Learning Approach for Filtering Monte Carlo Noise. *ACM Transactions on Graphics 34*, 4 (July 2015), 122:1–122:12.

[7] Knoll, A., Wald, I., Navrátil, P. A., Papka, M. E., and Gaither, K. P. Ray Tracing and Volume Rendering Large Molecular Data on Multi-Core and Many-Core Architectures. In *International Workshop on Ultrascale Visualization* (2013), pp. 5:1–5:8.

[8] Lindstrom, P. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (Dec. 2014), 2674–2683.

[9] Lindstrom, P., and Isenburg, M. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics 12*, 5 (Sept. 2006), 1245–1250.

[10] Mara, M., McGuire, M., Bitterli, B., and Jarosz, W. An Efficient Denoising Algorithm for Global Illumination. In *Proceedings of High-Performance Graphics* (2017), pp. 3:1–3:7.

[11] Merritt, E. A., and Murphy, M. E. P. Raster3D Version 2.0—A Program for Photorealistic Molecular Graphics. *Acta Crystallography 50*, 6 (1994), 869–873.

[12] Meyer, Q., Süßmuth, J., Sußner, G., Stamminger, M., and Greiner, G. On Floating-Point Normal Vectors. In *Eurographics Symposium on Rendering* (2010), pp. 1405–1409.

[13] Parker, S. G., Bigler, J., Dietrich, A., Friedrich, H., Hoberock, J., Luebke, D., McAllister, D., McGuire, M., Morley, K., Robison, A., and Stich, M. OptiX: A General Purpose Ray Tracing Engine. *ACM Transactions on Graphics 29*, 4 (2010), 66:1–66:13.

[14] Roth, S. D. Ray Casting for Modeling Solids. *Computer Graphics and Image Processing 18*, 2 (1982), 109–144.

[15] Santos, J. D., Sen, P., and Oliveira, M. M. A Framework for Developing and Benchmarking Sampling and Denoising Algorithms for Monte Carlo Rendering. *The Visual Computer 34*, 6-8 (June 2018), 765–778.

[16] Schied, C., Kaplanyan, A., Wyman, C., Patney, A., Chaitanya, C. R. A., Burgess, J., Liu, S., Dachsbacher, C., Lefohn, A., and Salvi, M. Spatiotemporal Variance-Guided Filtering: Real-Time Reconstruction for Path-Traced Global Illumination. In *Proceedings of High-Performance Graphics* (2017), pp. 2:1–2:12.

[17] Sener, M., Stone, J. E., Barragan, A., Singharoy, A., Teo, I., Vandivort, K. L., Isralewitz, B., Liu, B., Goh, B. C., Phillips, J. C., Kourkoutis, L. F., Hunter, C. N., and Schulten, K. Visualization of Energy Conversion Processes in a Light Harvesting Organelle at Atomic Detail. In *International Conference on High Performance Computing, Networking, Storage and Analysis* (2014).

[18] Stone, J. E. An Efficient Library for Parallel Ray Tracing and Animation. Master's thesis, Computer Science Department, University of Missouri-Rolla, April 1998.

[19] Stone, J. E., Isralewitz, B., and Schulten, K. Early Experiences Scaling VMD Molecular Visualization and Analysis Jobs on Blue Waters. In *Extreme Scaling Workshop* (Aug. 2013), pp. 43–50.

[20] Stone, J. E., Sener, M., Vandivort, K. L., Barragan, A., Singharoy, A., Teo, I., Ribeiro, J. V., Isralewitz, B., Liu, B., Goh, B. C., Phillips, J. C., MacGregor-Chatwin, C., Johnson, M. P., Kourkoutis, L. F., Hunter, C. N., and Schulten, K. Atomic Detail Visualization of Photosynthetic Membranes with GPU-Accelerated Ray Tracing. *Parallel Computing 55* (2016), 17–27.

[21] Stone, J. E., Sherman, W. R., and Schulten, K. Immersive Molecular Visualization with Omnidirectional Stereoscopic Ray Tracing and Remote Rendering. In *IEEE International Parallel and Distributed Processing Symposium Workshop* (2016), pp. 1048–1057.

[22] Stone, J. E., Vandivort, K. L., and Schulten, K. GPU-Accelerated Molecular Visualization on Petascale Supercomputing Platforms. In *International Workshop on Ultrascale Visualization* (2013), pp. 6:1–6:8.

[23] Van Wijk, J. J. Ray Tracing Objects Defined by Sweeping a Sphere. *Computers & Graphics 9*, 3 (1985), 283–290.

[24] Wald, I., Friedrich, H., Knoll, A., and Hansen, C. Interactive Isosurface Ray Tracing of Time-Varying Tetrahedral Volumes. *IEEE Transactions on Visualization and Computer Graphics 13*, 6 (11 2007), 1727–1734.

[25] Wald, I., Johnson, G., Amstutz, J., Brownlee, C., Knoll, A., Jeffers, J., Gunther, J., and Navratil, P. OSPRay—A CPU Ray Tracing Framework for Scientific Visualization. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 931–940.

[26] Wald, I., Woop, S., Benthin, C., Johnson, G. S., and Ernst, M. Embree: A Kernel Framework for Efficient CPU Ray Tracing. *ACM Transactions on Graphics 33*, 4 (July 2014), 143:1–143:8.