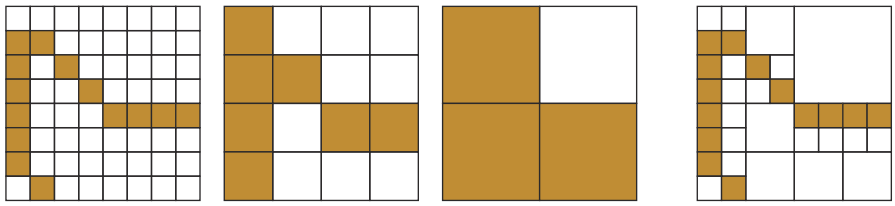

Figure 13.28. Sparse voxel octree, in two-dimensional form. Given a set of voxels on the left, we note which parent nodes have any voxels in them, on up the tree. On the right is a visualization of the final octree, showing the deepest node stored for each grid location. *(Figure after Laine and Karras [963].)*

Figure 13.29. Voxel ray tracing at different levels of detail. From left to right, the resolution is 256, 512, and 1024 along each edge of the voxel grid containing the model. *(Images rendered with Optix and NVIDIA® GVDB Voxels, courtesy of NVIDIA Corporation [753].)*

13.10.3 Generation of Voxels

The input to a voxel model can come from a variety of sources. For example, many scanning devices generate data points at arbitrary locations. The GPU can accelerate *voxelization*, the process of turning a point cloud [930], polygonal mesh, or other representation into a set of voxels. For meshes, one quick but rough method from Karabassi et al. [859] is to render the object from six orthographic views: top, bottom, and the four sides. Each view generates a depth buffer, so each pixel holds where the first visible voxel is from that direction. If a voxel's location is beyond the depth stored in each of the six buffers, it is not visible and so is marked as being inside the object. This method will miss any features that cannot be seen in any of the six views, causing some voxels to improperly be marked as inside. Still, for simple models this method can suffice.

Inspired by visual hulls [1139], Loop et al. [1071] use an even simpler system for creating voxelizations of people in the real world. A set of images of a person are captured and the silhouettes extracted. Each silhouette is used to carve away a set of voxels given its camera location—only pixels where you can see the person will have voxels associated with them.

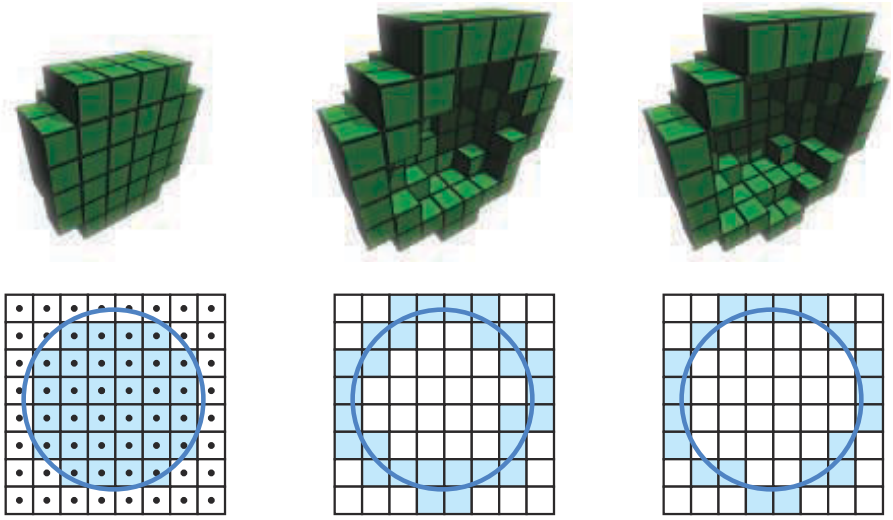

Figure 13.30. A sphere is voxelized three different ways and its cross section is shown. On the left is a solid voxelization, determined by testing the center of each voxel against the sphere. In the middle is a conservative voxelization, where any voxel touched by the sphere's surface is selected. This surface is called a 26-separating voxelization, in which no interior voxel is next to an exterior voxel in its $3 \times 3 \times 3$ neighborhood. In other words, interior and exterior voxels never share a face, edge, or vertex. On the right is a 6-separating voxelization, in which edges and corners can be shared between interior and exterior voxels. *(Figure after Schwarz and Seidel [1594].)*

Voxel grids can also be created from a collection of images, such as with medical image devices that generate slices that are then stacked up. Along the same lines, mesh models can be rendered slice by slice and the voxels found inside the model are duly recorded. The near and far planes are adjusted to bound each slice, which is examined for content. Eisemann and Décoret [409] introduce the idea of a *slicemap*, where the 32-bit target is instead thought of as 32 separate depths, each with a bit flag. The depth of a triangle rendered to this voxel grid is converted to its bit equivalent and stored. The 32 layers can then be rendered in a rendering pass, with more voxel layers available for the pass if wider-channel image formats and multiple render targets are used. Forest et al. [480] give implementation details, noting that on modern GPUs up to 1024 layers can be rendered in a single pass. Note that this slicing algorithm identifies just the surface of the model, its *boundary representation*. The six-views algorithm above also identifies (though sometimes miscategorizes) voxels that are fully inside the model. See Figure 13.30 for three common types of voxelization. Laine [964] provides a thorough treatment of terminology, various voxelization types, and the issues involved in generating and using them.

More efficient voxelization is possible with new functionality available in modern GPUs. Schwarz and Seidel [1594, 1595] and Pantaleoni [1350] present voxelization sys-

tems using compute shaders, which offer the ability to directly build an SVO. Crassin and Green [306, 307] describe their open-source system for regular-grid voxelization, which leverages image load/store operations available starting in OpenGL 4.2. These operations allow random read and write access to texture memory. By using conservative rasterization (Section 23.1.2) to determine all triangles overlapping a voxel, their algorithm efficiently computes voxel occupancy, along with an average color and normal. They can also create an SVO with this method, building from the top down and voxelizing only non-empty nodes as they descend, then populating the structure using bottom-up filtering mipmap creation. Schwarz [1595] gives implementation details for both rasterization and compute kernel voxelization systems, explaining the characteristics of each. Rauwendaal and Bailey [1466] include source code for their hybrid system. They provide performance analysis of parallelized voxelization schemes and details for the proper use of conservative rasterization to avoid false positives. Takeshige [1737] discusses how MSAA can be a viable alternative to conservative rasterization, if a small amount of error is acceptable. Baert et al. [87] present an algorithm for creating SVOs that can efficiently run out-of-core, that is, can voxelize a scene to a high precision without needing the whole model to be resident in memory.

Given the large amount of processing needed to voxelize a scene, dynamic objects—those moving or otherwise animated—are a challenge for voxel-based systems. Gaitatzes and Papaioannou [510] tackle this task by progressively updating their voxel representation of the scene. They use the rendering results from the scene camera and any shadow maps generated to clear and set voxels. Voxels are tested against the depth buffers, with those that are found to be closer than the recorded z-depths being cleared. Depth locations in the buffers are then treated as a set of points and transformed to world space. These points' corresponding voxels are determined and set, if not marked before. This clear-and-set process is view-dependent, meaning that parts of the scene that no camera currently sees are effectively unknown and so can be sources of error. However, this fast approximate method makes computing voxel-based global illumination effects practical to perform at interactive rates for dynamic environments (Section 11.5.6).

13.10.4 Rendering

Voxel data are stored in a three-dimensional array, which can also be thought of as, and indeed stored as, a three-dimensional texture. Such data can be displayed in any number of ways. The next chapter discusses ways to visualize voxel data that are semitransparent, such as fog, or where a slicing plane is positioned to examine the data set, such as an ultrasound image. Here we will focus on rendering voxel data representing solid objects.

Imagine the simplest volume representation, where each voxel contains a bit noting whether it is inside or outside an object. There are a few common ways to display such data [1094]. One method is to directly ray-cast the volume [752, 753, 1908] to

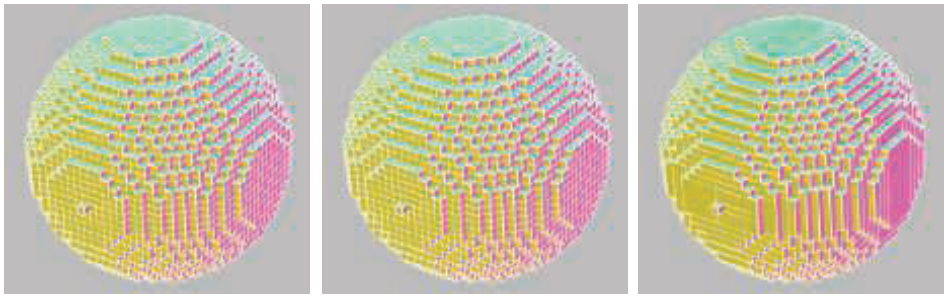

Figure 13.31. Cube culling. On the left, the 17,074 voxel solid sphere is formed of 102,444 quadrilaterals, six per voxel. In the middle, the two quadrilaterals between neighboring solid voxels are removed, reducing the count to 4,770. The look is the same as on left, as the outer shell is untouched. On the right, a fast greedy algorithm merges faces to form larger rectangles, giving 2,100 quadrilaterals. *(Images from Mikola Lysenko's culling program [1094].)*

determine the nearest hit face of each cube. Another technique is to convert the voxel cubes to a set of polygons. Though rendering using a mesh will be fast, this incurs additional cost during voxelization, and is best suited to static volumes. If each voxel's cube is to be displayed as opaque, then we can cull out any faces where two cubes are adjacent, since the shared square between them is not visible. This process leaves us with a hull of squares, hollow on the inside. Simplification techniques (Section 16.5) can reduce the polygon count further. See Figure 13.31.

Shading this set of cube faces is unconvincing for voxels representing curved surfaces. A common alternative to shading the cubes is to create a smoother mesh surface by using an algorithm such as *marching cubes* [558, 1077]. This process is called *surface extraction* or *polygonalization* (a.k.a. *polygonization*). Instead of treating each voxel as a box, we think of it as a point sample. We can then form a cube using the eight neighboring samples in a $2 \times 2 \times 2$ pattern to form the corners. The states of these eight corners can define a surface passing through the cube. For example, if the top four corners of the cube are outside and the bottom four inside, a horizontal square dividing the cube in half is a good guess for the shape of the surface. One corner outside and the rest inside yields a triangle formed by the midpoints of the three cube edges connected to the outside corner. See Figure 13.32. This process of turning a set of cube corners into a corresponding polygonal mesh is efficient, as the eight corner bits can be converted to an index from 0 to 255, which is used to access a table that specifies the number and locations of the triangles for each possible configuration.

Other methods to render voxels, such as level sets [636], are better suited to smooth, curved surfaces. Imagine that each voxel stores the distance to the surface of the object being represented, a positive value for inside and negative for outside. We can use these data to adjust the vertex locations of the mesh formed to more accurately represent the surface, as shown on the right in Figure 13.32. Alternately, we could

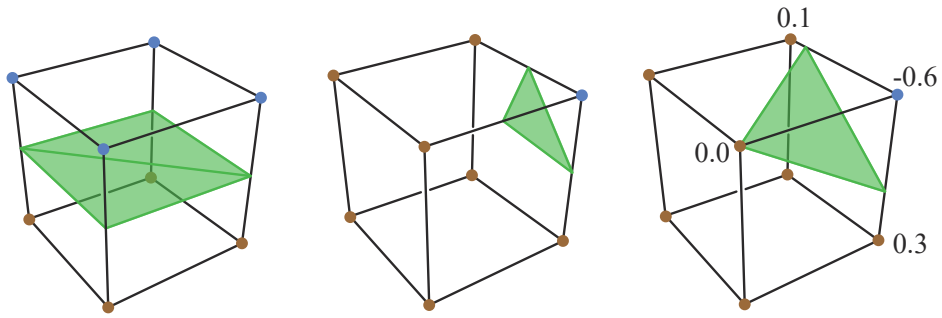

Figure 13.32. Marching cubes. On the left, the four bottom corners are voxel centers inside the object, so a horizontal square of two triangles is formed between the bottom and the top four corners. In the middle, one corner is outside, so a triangle is formed. On the right, if signed distance values are stored at the corners, then we can interpolate the triangle vertices to be at 0.0 along each edge. Note that other cubes sharing a given edge will have a vertex at the same location along that edge, to ensure that the surface has no cracks.

directly ray-trace the level set with an isovalue of zero. This technique is called *level-set rendering* [1249]. It is particularly good at representing the surface and normals of a curved model without any additional voxel attributes.

Voxel data representing differences in density can be visualized in different ways by deciding what forms a surface. For example, some given density may give a good representation of a kidney, another density could show any kidney stones present. Choosing a density value defines an *isosurface*, a set of locations with the same value. Being able to vary this value is particularly useful for scientific visualization. Directly ray tracing any isosurface value is a generalization of level-set ray tracing, where the target value is always zero. Alternatively, one can extract the isosurface and convert it to a polygonal model.

In 2008 Olick [1323] gave an influential talk about how a sparse voxel representation can be rendered directly with ray casting, inspiring further work. Testing rays against regularized voxels is well suited to a GPU implementation, and can be done at interactive frame rates. Many researchers have explored this area of rendering. For an introduction to the subject, start with Crassin's PhD thesis [304] and SIGGRAPH presentation [308], which cover the advantages of voxel-based methods. Crassin exploits the mipmap-like nature of the data by using *cone tracing*. The general idea is to use the regularity and well-defined locality property of voxel representations to define prefiltering schemes for geometry and shading properties, allowing the use of linear filters. A single ray is traced through the scene but is able to gather an approximation of the visibility through a cone emanating from its start point. As the ray moves through space, its radius of interest grows, which means that the voxel hierarchy is sampled higher up the chain, similar to how a mipmap is sampled higher up as more texels fall inside a single pixel. This type of sampling can rapidly compute

Figure 13.33. Cone-traced shadows. Top: a ray-traced spherical area light rendered in Maya in 20 seconds. Bottom: voxelization and cone tracing for the same scene took ∼20 ms. The model is rendered with polygons, with the voxelized version used for shadow computations. *(Images courtesy of Crytek [865].)*

soft shadows and depth of field, for example, as these effects can be decomposed into cone-tracing problems. Area sampling can be valuable for other processes, such as antialiasing and properly filtering varying surface normals. Heitz and Neyret [706] describe previous work and present a new data structure for use with cone tracing that improves visibility calculation results. Kasyan [865] uses voxel cone tracing for area lights, discussing sources of error. A comparison is shown in Figure 13.33. See Figure 7.33 on page 264 for a final result. Cone tracing's use for computing global illumination effects is discussed and illustrated in Section 11.5.7.

Recent trends explore structures beyond octrees on the GPU. A key drawback of octrees is that operations such as ray tracing require a large number of tree traversal hops and so require storage of a significant number of intermediate nodes. Hoetzlein [752] shows that GPU ray tracing of VDB trees, a hierarchy of grids, can achieve significant performance gains over octrees, and are better suited to dynamic changes in volume data. Fogal et al. [477] demonstrate that index tables, rather than octrees, can be used to render large volumes in real-time using a two-pass approach. The first pass identifies visible sub-regions (bricks), and streams in those regions from disk. The second pass renders the regions currently resident in memory. A thorough survey of large-scale volumetric rendering is provided by Beyer et al. [138].

13.10.5 Other Topics

Surface extraction is commonly used to visualize implicit surfaces (Section 17.3), for example. There are different forms of the basic algorithm and some subtleties to how meshes are formed. For example, if every other corner for a cube is found to be inside, should these corners be joined together in the polygonal mesh formed, or kept separate? See the article by de Araújo et al. [67] for a survey of polygonalization techniques for implicit surfaces. Austin [85] runs through the pros and cons of a variety of general polygonalization schemes, finding cubical marching squares to have the most desirable properties.

Other solutions than full polygonalization are possible when using ray casting for rendering. For example, Laine and Karras [963] attach a set of parallel planes to each voxel that approximate the surface, then use a post-process blur to mask discontinuities between voxels. Heitz and Neyret [706] access the signed distance in a linearly filterable representation that permits reconstructing plane equations and determining coverage in a given direction for any spatial location and resolution.

Eisemann and Décoret [409] show how a voxel representation can be used to perform deep shadow mapping (Section 7.8), for situations where semitransparent overlapping surfaces cast shadows. As Kämpe, Sintorn, and others show [850, 1647], another advantage of a voxelized scene is that shadow rays for all lights can be tested using this one representation, versus generating a shadow map for each light source. Compared to directly visible surface rendering, the eye is more forgiving of small errors in secondary effects such as shadows and indirect illumination, and much less voxel data are needed for these tasks. When only occupancy of a voxel is tracked, there can be extremely high self-similarity among many sparse voxel nodes [849, 1817]. For example, a wall will form sets of voxels that are identical over several levels. This means that various nodes and entire sub-trees in a tree are the same, and so we can use a single instance for such nodes and store them in what is called a *directed acyclic graph* (Section 19.1.5). Doing so often leads to vast reductions in the amount of memory needed per voxel-structure.

Further Reading and Resources

Image-based rendering, light fields, computational photography, and many other topics are discussed in Szeliski's *Computer Vision: Algorithms and Applications* [1729]. See our website realtimerendering.com for a link to the free electronic version of this worthwhile volume. A wide range of acceleration techniques taking advantage of limitations in our visual perceptual system is discussed by Weier et al. [1864] in their state-of-the-art report. Dietrich et al. [352] provide an overview of image-based techniques in the sidebars of their report on massive model rendering.

We have touched upon only a few of the ways images, particles, and other nonpolygonal methods are used to simulate natural phenomena. See the referenced articles for more examples and details. A few articles discuss a wide range of techniques. The survey of crowd rendering techniques by Beacco et al. [122] discusses many variations on impostors, level of detail methods, and much else. Gjøl and Svendsen's presentation [539] gives image-based sampling and filtering techniques for a wide range of effects, including bloom, len flares, water effects, reflections, fog, fire, and smoke.

Chapter 14
Volumetric and Translucency Rendering

"Those you wish should look farthest away you must make proportionately bluer; thus, if one is to be five times as distant, make it five times bluer."
—Leonardo Da Vinci

Participating media is the term used to describe volumes filled with particles. As we can tell by the name, they are media that participate in light transport, in other words they affect light that passes through them via scattering or absorption. When rendering virtual worlds, we usually focus on the solid surfaces, simple yet complex. These surfaces appear opaque because they are defined by light bouncing off of dense participating media, e.g., a dielectric or metal typically modeled using a BRDF. Less dense well-known media are water, fog, steam, or even air, composed of sparse molecules. Depending on its composition, a medium will interact differently with light traveling through it and bouncing off its particles, an event typically referred to as *light scattering*. The density of particles can be homogeneous (uniform), as in the case of air or water. Or it could be heterogeneous (nonuniform, varying with location in space), as in the case of clouds or steam. Some dense materials often rendered as solid surfaces exhibit high levels of light scattering, such as skin or candle wax. As shown in Section 9.1, diffuse surface shading models are the result of light scattering on a microscopic level. Everything is scattering.

14.1 Light Scattering Theory

In this section, we will describe the simulation and rendering of light in participating media. This is a quantitative treatment of the physical phenomena, scattering and absorption, which were discussed in Sections 9.1.1 and 9.1.2. The radiative transfer equation is described by numerous authors [479, 743, 818, 1413] in the context of path tracing with multiple scattering. Here we will focus on *single scattering* and build a good intuition about how it works. Single scattering considers only one bounce of light

Symbol	Description	Unit
σ_a	Absorption coefficient	m^{-1}
σ_s	Scattering coefficient	m^{-1}
σ_t	Extinction coefficient	m^{-1}
ρ	Albedo	unitless
p	Phase function	sr^{-1}

Table 14.1. Notation used for scattering and participating media. Each of these parameters can depend on the wavelength (i.e., RGB) to achieve colored light absorption or scattering. The units of the phase function are inverse steradians (Section 8.1.1).

on the particles that constitute the participating media. Multiple scattering tracks many bounces per light path and so is much more complex [243, 479]. Results with and without multiple scattering can be seen in Figure 14.51 on page 646. Symbols and units used to represent the participating media properties in scattering equations are presented in Table 14.1. Note that many of the quantities in this chapter, such as σ_a, σ_s, σ_t, p, ρ, v, and T_r are wavelength-dependent, which in practice means that they are RGB quantities.

14.1.1 Participating Media Material

There are four types of events that can affect the amount of radiance propagating along a ray through a medium. These are illustrated in Figure 14.1 and summarized as:

- **Absorption** (function of σ_a)—Photons are absorbed by the medium's matter and transformed into heat or other forms of energy.
- **Out-scattering** (function of σ_s)—Photons are scattered away by bouncing off particles in the medium matter. This will happen according to the phase function p describing the distribution of light bounce directions.
- **Emission**—Light can be emitted when media reaches a high heat, e.g., a fire's black-body radiation. For more details about emission, please refer to the course notes by Fong et al. [479].
- **In-scattering** (function of σ_s)—Photons from any direction can scatter into the current light path after bouncing off particles and contribute to the final radiance. The amount of light in-scattered from a given direction also depends on the phase function p for that light direction.

To sum up, adding photons to a path is a function of in-scattering σ_s and emission. Removing photons is a function of *extinction* $\sigma_t = \sigma_a + \sigma_s$, representing both absorption and out-scattering. As explained by the *radiative transfer equation*, the

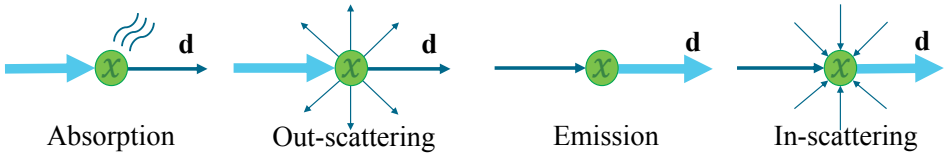

Figure 14.1. Different events change the radiance along a direction $\mathbf{d}$ in participating media.

set of coefficients represents the derivative of radiance at position $\mathbf{x}$ and toward direction $\mathbf{v}$ relative to $L(\mathbf{x}, \mathbf{v})$. That is why these coefficients' values are all in the range $[0, +\infty]$. See the notes by Fong et al. [479] for details. The scattering and absorption coefficients determine the medium's albedo ρ, defined as

$$\rho = \frac{\sigma_s}{\sigma_s + \sigma_a} = \frac{\sigma_s}{\sigma_t}, \tag{14.1}$$

which represents the importance of scattering relative to absorption in a medium for each visible spectrum range considered, i.e., the overall reflectiveness of the medium. The value of ρ is within the range $[0, 1]$. A value close to 0 indicates that most of the light is absorbed, resulting in a murky medium, such as dark exhaust smoke. A value close to 1 indicates that most of the light is scattered instead of being absorbed, resulting in a brighter medium, such as air, clouds, or the earth's atmosphere.

As discussed in Section 9.1.2, the appearance of a medium is a combination of its scattering and absorption properties. Coefficient values for real-world participating media have been measured and published [1258]. For instance, milk has high scattering values, producing a cloudy and opaque appearance. Milk also appears white thanks to a high albedo $\rho > 0.999$. On the other hand, red wine features almost no scattering but instead high absorption, giving it a translucent and colored appearance. See the rendered liquids in Figure 14.2, and compare to the photographed liquids in Figure 9.8 on page 301.

Each of these properties and events are wavelength-dependent. This dependence means that in a given medium, different light frequencies may be absorbed or scattered with differing probabilities. In theory, to account for this we should use spectral values in rendering. For the sake of efficiency, in real-time rendering (and with a few exceptions [660] in offline rendering as well) we use RGB values instead. Where possible, the RGB values of quantities such as σ_a and σ_s should be precomputed from spectral data using color-matching functions (Section 8.1.3).

In earlier chapters, due to the absence of participating media, we could assume that the radiance entering the camera was the same as the radiance leaving the closest surface. More precisely, we assumed (on page 310) that $L_i(\mathbf{c}, -\mathbf{v}) = L_o(\mathbf{p}, \mathbf{v})$, where $\mathbf{c}$ is the camera position, $\mathbf{p}$ is the intersection point of the closest surface with the view ray, and $\mathbf{v}$ is the unit view vector pointing from $\mathbf{p}$ to $\mathbf{c}$.

Once participating media are introduced, this assumption no longer holds and we need to account for the change in radiance along the view ray. As an example, we will

Figure 14.2. Rendered wine and milk, respectively, featuring absorption and scattering at different concentrations. *(Images courtesy of Narasimhan et al. [1258].)*

now describe the computations involved in evaluating scattered light from a punctual light source, i.e., a light source represented by a single infinitesimal point (Section 9.4):

$$L_i(\mathbf{c}, \ -\mathbf{v}) = T_r(\mathbf{c}, \mathbf{p}) L_o(\mathbf{p}, \mathbf{v}) + \int_{t=0}^{\|\mathbf{p}-\mathbf{c}\|} T_r(\mathbf{c}, \mathbf{c} - \mathbf{v}t) L_{\text{scat}}(\mathbf{c} - \mathbf{v}t, \mathbf{v}) \sigma_s dt, \quad (14.2)$$

where $T_r(\mathbf{c}, \mathbf{x})$ is the transmittance between a given point $\mathbf{x}$ and the camera position $\mathbf{c}$ (Section 14.1.2) and $L_{\text{scat}}(\mathbf{x}, \mathbf{v})$ is the light scattered along the view ray (Section 14.1.3) at a given point $\mathbf{x}$ on the ray. The different components of the calculation are shown in Figure 14.3 and explained in the following subsections. More details about how Equation 14.2 is derived from the radiative transfer equation can be found in the course notes by Fong et al. [479].

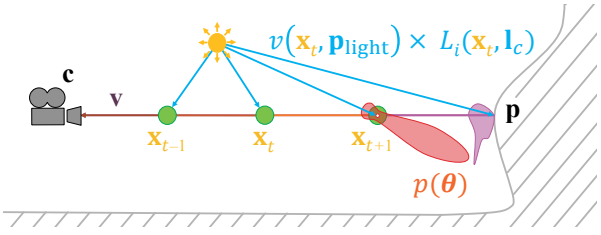

Figure 14.3. Illustration of single scattering integration from a punctual light source. Sample points along the view ray are shown in green, a phase function for one point is shown in red, and the BRDF for an opaque surface S is shown in orange. Here, $\mathbf{l}_c$ is the direction vector to the light center, $\mathbf{p}_{\text{light}}$ is the position of the light, p is the phase function, and the function v is the visibility term.

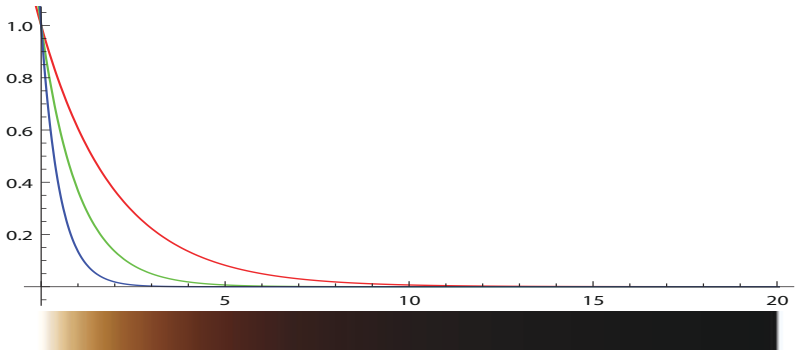

Figure 14.4. Transmittance as a function of depth, with $\sigma_t = (0.5, 1.0, 2.0)$. As expected, a lower extinction coefficient for the red component leads to more red color being transmitted.

14.1.2 Transmittance

The transmittance T_r represents the ratio of light that is able to get through a medium over a certain distance according to

$$T_r(\mathbf{x}_a, \mathbf{x}_b) = e^{-\tau}, \quad \text{where} \quad \tau = \int_{\mathbf{x}=\mathbf{x}_a}^{\mathbf{x}_b} \sigma_t(\mathbf{x}) \, \|d\mathbf{x}\|. \tag{14.3}$$

This relationship is also known as the Beer-Lambert Law. The optical depth τ is unitless and represents the amount of light attenuation. The higher the extinction or distance traversed, the larger the optical depth will be and, in turn, the less light will travel through the medium. An optical depth $\tau = 1$ will remove approximately 60% of the light. For instance, if $\sigma_t = (0.5, 1, 2)$ in RGB, then the light coming through depth $d = 1$ meter will be $T_r = e^{-d\sigma_t} \approx (0.61, 0.37, 0.14)$. This behavior is presented in Figure 14.4. Transmittance needs to be applied on (i) the radiance $L_o(\mathbf{p}, \mathbf{v})$ from opaque surfaces, (ii) the radiance $L_{\text{scat}}(\mathbf{x}, \mathbf{v})$ resulting from an in-scattering event, and (iii) each path from a scattering event to the light source. Visually, (i) will result in some fog-like occlusion of surfaces, (ii) will result in occlusion of scattered light, giving another visual cue about the media thickness (see Figure 14.6), and (iii) will result in volumetric self-shadowing by the participating media (see Figure 14.5). Since $\sigma_t = \sigma_a + \sigma_s$, it is expected that the transmittance is influenced by both the absorption and out-scattering components.

14.1.3 Scattering Events

Integrating in-scattering from punctual light sources in a scene for a given position $\mathbf{x}$ and from direction $\mathbf{v}$ can be done as follows:

$$L_{\text{scat}}(\mathbf{x}, \mathbf{v}) = \pi \sum_{i=1}^{n} p(\mathbf{v}, \mathbf{l}_{c_i}) v(\mathbf{x}, \mathbf{p}_{\text{light}_i}) c_{\text{light}_i}(\|\mathbf{x} - \mathbf{p}_{\text{light}_i}\|), \tag{14.4}$$

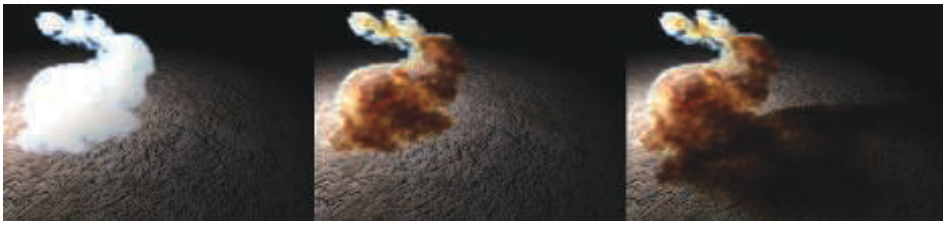

Figure 14.5. Example of volumetric shadows from a Stanford bunny made of participating media [744]. Left: without volumetric self-shadowing; middle: with self-shadowing; right: with shadows cast on other scene elements. *(Model courtesy of the Stanford Computer Graphics Laboratory.)*

where n is the number of lights, $p()$ is the phase function, $v()$ is the visibility function, $\mathbf{l}_{c_i}$ is the direction vector to the ith light, and $\mathbf{p}_{\text{light}_i}$ is the position of the ith light. In addition, $c_{\text{light}_i}()$ is the radiance from the ith light as a function of the distance to its position, using the definition from Section 9.4 and the inverse square falloff function from Section 5.2.2. The visibility function $v(\mathbf{x}, \mathbf{p}_{\text{light}_i})$ represents the ratio of light reaching a position $\mathbf{x}$ from a light source at $\mathbf{p}_{\text{light}_i}$ according to

$$v(\mathbf{x}, \mathbf{p}_{\text{light}_i}) = \texttt{shadowMap}(\mathbf{x}, \mathbf{p}_{\text{light}_i}) \cdot \texttt{volShad}(\mathbf{x}, \mathbf{p}_{\text{light}_i}), \tag{14.5}$$

where $\texttt{volShad}(\mathbf{x}, \mathbf{p}_{\text{light}_i}) = T_r(\mathbf{x}, \mathbf{p}_{\text{light}_i})$. In real-time rendering, shadows result from two kinds of occlusion: opaque and volumetric. Shadows from opaque objects (`shadowMap`) are traditionally computed by using shadow mapping or other techniques from Chapter 7.

The volumetric shadow term $\texttt{volShad}(\mathbf{x}, \mathbf{p}_{\text{light}_i})$ of Equation 14.5 represents the transmittance from light source position $\mathbf{p}_{\text{light}_i}$ to sampled point $\mathbf{x}$, with values in the range $[0, 1]$. Occlusion produced by a volume is a crucial component of volumetric rendering, where the volume element can self-shadow or cast shadows on other scene elements. See Figure 14.5. This result is usually achieved by performing ray marching along the primary ray from the eye through the volume to the first surface, and then along a secondary ray path from each of these samples toward each light source. "Ray marching" refers to the act of sampling the path between two points using n samples, integrating scattered light and transmittance along the way. See Section 6.8.1 for more details on this method of sampling, which in that case was for rendering a heightfield. Ray marching is similar for three-dimensional volumes, with each ray progressing step by step and sampling the volume material or lighting at each point along the way. See Figure 14.3, which shows sample points on a primary ray in green and secondary shadow rays in blue. Many other publications also describe ray marching in detail [479, 1450, 1908].

Being of $O(n^2)$ complexity, where n is the number of samples along each path, ray marching quickly becomes expensive. As a trade-off between quality and performance, specific volumetric shadow representation techniques can be used to store transmit-

Figure 14.6. Stanford dragon with increasing media concentration. From left to right: 0.1, 1.0, and 10.0, with $\sigma_s = (0.5, 1.0, 2.0)$. *(Model courtesy of the Stanford Computer Graphics Laboratory.)*

tance for outgoing directions from a light. These techniques will be explained in the appropriate sections throughout the rest of this chapter.

To gain some intuition about the behavior of light scattering and extinction within a medium, consider $\sigma_s = (0.5, 1, 2)$ and $\sigma_a = (0, 0, 0)$. For a short light path within the medium, in-scattering events will dominate over extinction, e.g., out-scattering in this case, where $T_r \approx 1$ for a small depth. The material will appear blue, since this channel's σ_s value is highest. The deeper that light penetrates into the medium, the fewer photons will get through, due to extinction. In this case, the transmittance color from extinction will start to dominate. This can be explained by the fact that we have $\sigma_t = \sigma_s$, since $\sigma_a = (0, 0, 0)$. As a result, $T_r = e^{-d\sigma_t}$ will decrease much faster than the linear integration of scattered light as a function of the optical depth $d\sigma_s$ using Equation 14.2. For this example, the red light channel will be less subject to extinction through the medium, since this channel's σ_t value is lowest, so it will dominate. This behavior is depicted in Figure 14.6, and is exactly what happens in the atmosphere and sky. When the sun is high (e.g., short light path through the atmosphere, perpendicular to the ground), blue light scatters more, giving the sky its natural blue color. However, when the sun is at the horizon, so that there is a long light path through the atmosphere, the sky will appear redder since more red light is transmitted. This results in the beautiful sunrise and sunset transitions we all know. See Section 14.4.1 for more details about the atmosphere's material composition. For another example of this effect, see the opalescent glass on the right side of Figure 9.6 on page 299.

14.1.4 Phase Functions

A participating medium is composed of particles with varying radii. The distribution of the size of these particles will influence the probability that light will scatter in a given direction, relative to the light's forward travel direction. The physics behind this behavior is explained in Section 9.1.

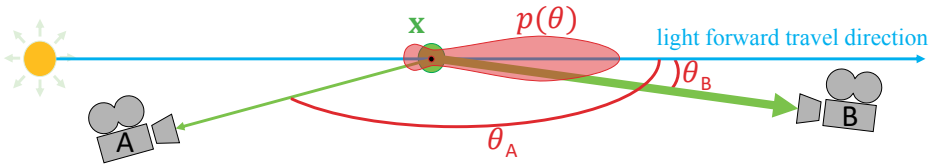

Figure 14.7. Illustration of a phase function (in red) and its influence on scattered light (in green) as a function of θ.

Describing the probability and distribution of scattering directions at a macro level is achieved using a phase function when evaluating in-scattering, as shown in Equation 14.4. This is illustrated in Figure 14.7. The phase function in red is expressed using the parameter θ as the angle between the light's forward travel path in blue and toward direction $\mathbf{v}$ in green. Notice the two main lobes in this phase function example: a small backward-scattering lobe in the opposite direction of the light path, and a large forward-scattering lobe. Camera B is in the direction of the large forward-scattering lobe, so it will receive much more scattered radiance as compared to camera A. To be energy-conserving and -preserving, i.e., no energy gain or loss, the integration of a phase function over the unit sphere must be 1.

A phase function will change the in-scattering at a point according to the directional radiance information reaching that point. The simplest function is isotropic: Light will be scattered uniformly in all directions. This perfect but unrealistic behavior is presented as

$$p(\theta) = \frac{1}{4\pi}, \tag{14.6}$$

where θ is the angle between the incoming light and out-scattering directions, and 4π is the area of the unit sphere.

Physically based phase functions depend on the relative size s_p of a particle according to

$$s_p = \frac{2\pi r}{\lambda}, \tag{14.7}$$

where r is the particle radius and λ the considered wavelength [743]:

- When $s_p \ll 1$, there is Rayleigh scattering (e.g., air).
- When $s_p \approx 1$, there is Mie scattering.
- When $s_p \gg 1$, there is geometric scattering.

Rayleigh Scattering

Lord Rayleigh (1842–1919) derived terms for the scattering of light from molecules in the air. These expressions are used, among other applications, to describe light scattering in the earth's atmosphere. This phase function has two lobes, as shown

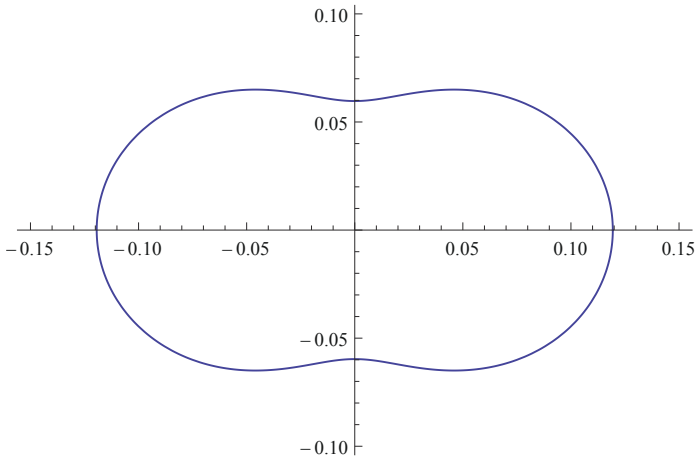

Figure 14.8. Polar plot of the Rayleigh phase as a function of θ. The light is coming horizontally from the left, and the relative intensity is shown for angle θ, measured counterclockwise from the x-axis. The chance of forward and backward scattering is the same.

in Figure 14.8, referred to as *backward and* forward scattering, relative to the light direction. This function is evaluated at θ, the angle between the incoming light and out-scattering directions. The function is

$$p(\theta) = \frac{3}{16\pi}(1 + \cos^2\theta). \tag{14.8}$$

Rayleigh scattering is highly wavelength-dependent. When viewed as a function of the light wavelength λ, the scattering coefficient σ_s for Rayleigh scattering is proportional to the inverse fourth power of the wavelength:

$$\sigma_s(\lambda) \propto \frac{1}{\lambda^4}. \tag{14.9}$$

This relationship means that short-wavelength blue or violet light is scattered much more than long-wavelength red light. The spectral distribution from Equation 14.9 can be converted to RGB using spectral color-matching functions (Section 8.1.3): $\sigma_s = (0.490, 1.017, 2.339)$. This value is normalized to a luminance of 1, and should be scaled according to the desired scattering intensity. The visual effects resulting from blue light being scattered more in the atmosphere are explained in Section 14.4.1.

Mie Scattering

Mie scattering [776] is a model that can be used when the size of particles is about the same as the light's wavelength. This type of scattering is not wavelength-dependent. The MiePlot software is useful for simulating this phenomenon [996]. The Mie phase

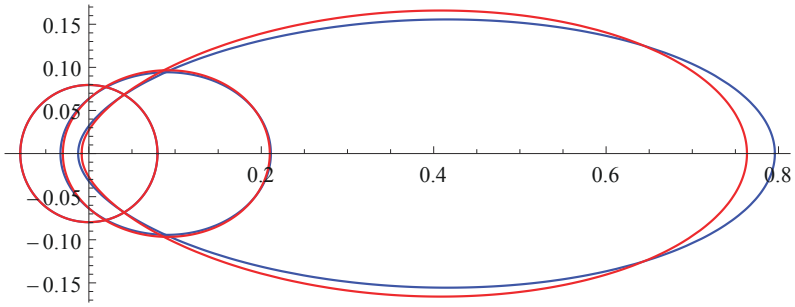

Figure 14.9. Polar plot of the Henyey-Greenstein (in blue) and Schlick approximation (in red) phase as a function of θ. The light is coming horizontally from the left. Parameter g increases from 0 to 0.3 and 0.6, resulting in a strong lobe to the right, meaning that light will be scattered more along its forward path, from the left to the right.

function for a specific particle size is typically a complex distribution with strong and sharp directional lobes, i.e., representing a high probability to scatter photons in specific directions relative to the photon travel direction. Computing such phase functions for volume shading is computationally expensive, but fortunately it is rarely needed. Media typically have a continuous distribution of particle sizes. Averaging the Mie phase functions for all these different sizes results in a smooth average phase function for the overall medium. For this reason, relatively smooth phase functions can be used to represent Mie scattering.

One phase function commonly used for this purpose is the Henyey-Greenstein (HG) phase function, which was originally proposed to simulate light scattering in interstellar dust [721]. This function cannot capture the complexity of every real-world scattering behavior, but it can be a good match to represent one of the phase function lobes [1967], i.e., toward the main scatter direction. It can be used to represent any smoke, fog, or dust-like participating media. Such media can exhibit strong backward or forward scattering, resulting in large visual halos around light sources. Examples include spotlights in fog and the strong silver-lining effect at the edges of clouds in the sun's direction.

The HG phase function can represent more complex behavior than Rayleigh scattering and is evaluated using

$$p_{hg}(\theta, g) = \frac{1 - g^2}{4\pi(1 + g^2 - 2g\cos\theta)^{1.5}}. \tag{14.10}$$

It can result in varied shapes, as shown in Figure 14.9. The g parameter can be used to represent backward ($g < 0$), isotropic ($g = 0$), or forward ($g > 0$) scattering, with g in $[-1, 1]$. Examples of scattering results using the HG phase function are shown in Figure 14.10.

A faster way to obtain similar results to the Henyey-Greenstein phase function is to use an approximation proposed by Blasi et al. [157], which is usually named for the

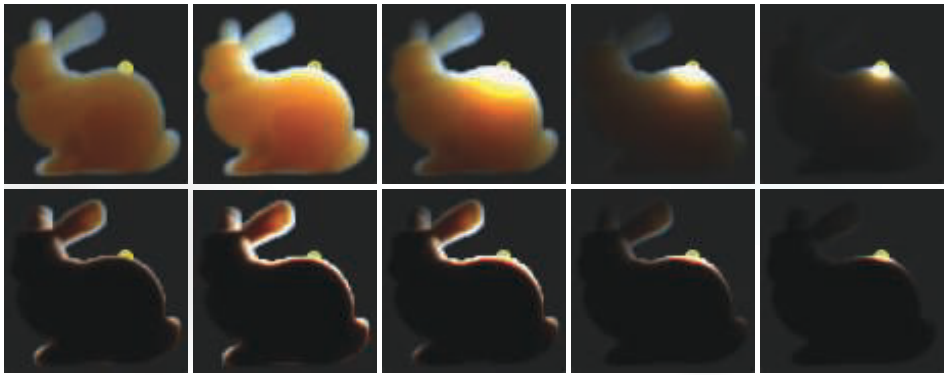

Figure 14.10. Participating-media Stanford bunny showing HG phase function influence, with g ranging from isotropic to strong forward scattering. From left to right: $g = 0.0$, 0.5, 0.9, 0.99, and 0.999. The bottom row uses a ten-times denser participating media. *(Model courtesy of the Stanford Computer Graphics Laboratory.)*

third author as the Schlick phase function:

$$p(\theta, k) = \frac{1 - k^2}{4\pi(1 + k\cos\theta)^2}, \quad \text{where} \quad k \approx 1.55g - 0.55g^3. \tag{14.11}$$

It does not include any complex power function but instead only a square, which is much faster to evaluate. In order to map this function onto the original HG phase function, the k parameter needs to be computed from g. This has to be done only once for participating media having a constant g value. In practical terms, the Schlick phase function is an excellent energy-conserving approximation, as shown in Figure 14.9.

It is also possible to blend multiple HG or Schlick phase functions in order to represent a more complex range of general phase functions [743]. This allows us to represent a phase function having strong forward- and backward-scattering lobes at the same time, similar to how clouds behave, as described and illustrated in Section 14.4.2.

Geometric Scattering

Geometric scattering happens with particles significantly larger than the light's wavelength. In this case, light can refract and reflect within each particle. This behavior can require a complex scattering phase function to simulate it on a macro level. Light polarization can also affect this type of scattering. For instance, a real-life example is the visual rainbow effect. It is caused by internal reflection of light inside water particles in the air, dispersing the sun's light into a visible spectrum over a small visual angle ($\sim$ 3 degrees) of the resulting backward scattering. Such complex phase functions can be simulated using the MiePlot software [996]. An example of such a phase function is described in Section 14.4.2.

Figure 14.11. Fog used to accentuate a mood. *(Image courtesy of NVIDIA Corporation.)*

14.2 Specialized Volumetric Rendering

This section presents algorithms for rendering volumetric effects in a basic, limited way. Some might even say these are old school tricks, often relying on ad hoc models. The reason they are used is that they still work well.

14.2.1 Large-Scale Fog

Fog can be approximated as a depth-based effect. Its most basic form is the alpha blending of the fog color on top of a scene according to the distance from the camera, usually called *depth fog*. This type of effect is a visual cue to the viewer. First, it can increase the level of realism and drama, as seen in Figure 14.11. Second, it is an important depth cue helping the viewer of a scene determine how far away objects are located. See Figure 14.12. Third, it can be used as a form of occlusion culling. If objects are completely occluded by the fog when too far away, their rendering can safely be skipped, increasing application performance.

One way to represent an amount of fog is to have f in $[0, 1]$ representing a transmittance, i.e., $f = 0.1$ means 10% of the background surface is visible. Assuming that the input color of a surface is $\mathbf{c}_i$ and the fog color is $\mathbf{c}_f$, then the final color, $\mathbf{c}$, is determined by

$$\mathbf{c} = f\mathbf{c}_i + (1 - f)\mathbf{c}_f. \tag{14.12}$$

The value f can be evaluated in many different ways. The fog can increase linearly using

$$f = \frac{z_{\text{end}} - z_s}{z_{\text{end}} - z_{\text{start}}}, \tag{14.13}$$

Figure 14.12. Fog is used in this image of a level from *Battlefield 1*, a DICE game, to reveal the complexity of the gameplay area. Depth fog is used to reveal the large-scale nature of the scenery. Height fog, visible on the right at the ground level, reveals the large amount of buildings raising up from the valley. *(Courtesy of DICE, © 2018 Electronic Arts Inc.)*

where z_{start} and z_{end} are user parameters that determine where the fog is to start and end (i.e., become fully foggy), and z_s is the linear depth from the viewer to the surface where fog is to be computed. A physically accurate way to evaluate fog transmittance is to have it increase exponentially with distance, thus following the Beer-Lambert Law for transmittance (Section 14.1.2). This effect can be achieved using

$$f = e^{-d_f z_s}, \tag{14.14}$$

where the scalar d_f is a user parameter that controls the density of the fog. This traditional large-scale fog is a coarse approximation to the general simulation of light scattering and absorption within the atmosphere (Section 14.4.1), but it is still used in games today to great effect. See Figure 14.12.

This is how hardware fog was exposed in legacy OpenGL and DirectX APIs. It is still worthwhile to consider using these models for simpler use cases on hardware such as mobile devices. Many current games rely on more advanced post-processing for atmospheric effects such as fog and light scattering. One problem with fog in a perspective view is that the depth buffer values are computed in a nonlinear fashion (Section 23.7). It is possible to convert the nonlinear depth buffer values back to linear depths z_s using inverse projection matrix mathematics [1377]. Fog can then be applied as a full-screen pass using a pixel shader, enabling more advanced results to be achieved, such as height-dependent fog or underwater shading.

Height fog represents a single slab of participating media with a parameterized height and thickness. For each pixel on screen, the density and scattered light is evaluated as a function of the distance the view ray has traveled through the slab before hitting a surface. Wenzel [1871] proposes a closed-form solution evaluating f for an exponential fall-off of participating media within the slab. Doing so results in a smooth fog transition near the edges of the slab. This is visible in the background fog on the left side of Figure 14.12.

Many variations are possible with depth and height fog. The color $\mathbf{c}_f$ can be a single color, be read from a cube map sampled using the view vector, or even can be the result of complex atmospheric scattering with a per-pixel phase function applied for directional color variations [743]. It is also possible to combine depth f_d and height f_h fog transmittance using $f = f_d f_h$ and have both types of fog interleaved together in a scene.

Depth and height fog are large-scale fog effects. One might want to render more local phenomena such as separated fog areas, for example, in caves or around a few tombs in a cemetery. Shapes such as ellipsoids or boxes can be used to add local fog where needed [1871]. These fog elements are rendered from back to front using their bounding boxes. The front d_f and back d_b intersection along the view vector of each shape is evaluated in the pixel shader. Using volume depth as $d = \max(0, \min(z_s, d_b) - d_f)$, where z_s is the linear depth representing the closest opaque surface, it is possible to evaluate a transmittance T_r (Section 14.1.2), with coverage as $\alpha = 1.0 - T_r$. The amount of scattered light $\mathbf{c}_f$ to add on top can then be evaluated as $\alpha\mathbf{c}_f$. To allow more varied shapes evaluated from meshes, Oat and Scheuermann [1308] give a clever single-pass method of computing both the closest entry point and farthest exit point in a volume. They save the surface distance, d_s, to a surface in one channel, and $1 - d_s$ in another channel. By setting the alpha blending mode to save the minimum value found, after the volume is rendered, the first channel will have the closest value d_f and the second channel will have the farthest value d_b, encoded as $1 - d$, allowing recovery of d.

Water is a participating medium and, as such, exhibits the same type of depth-based color attenuation. Coastal water has a transmittance of about $(0.3, 0.73, 0.63)$ per meter [261], thus using Equation 14.23 we can recover $\sigma_t = (1.2, 0.31, 0.46)$. When rendering dark water using an opaque surface, it is possible to enable the fog logic when the camera is under the water surface, and to turn it off when above. A more advanced solution has been proposed by Wenzel [1871]. If the camera is under water, scattering and transmittance are integrated until a solid or the water surface is hit. If above water, these are integrated from only the distance between the water's top surface to the solid geometry of the seabed.

14.2.2 Simple Volumetric Lighting

Light scattering within participating media can be complex to evaluate. Thankfully, there are many efficient techniques that can be used to approximate such scattering convincingly in many situations.

The simplest way to obtain volumetric effects is to render transparent meshes blended over the framebuffer. We refer to this as a *splatting* approach (Section 13.9). To render light shafts shining through a window, through a dense forest, or from a spotlight, one solution is to use camera-aligned particles with a texture on each. Each textured quad is stretched in the direction of the light shaft while always facing the camera (a cylinder constraint).

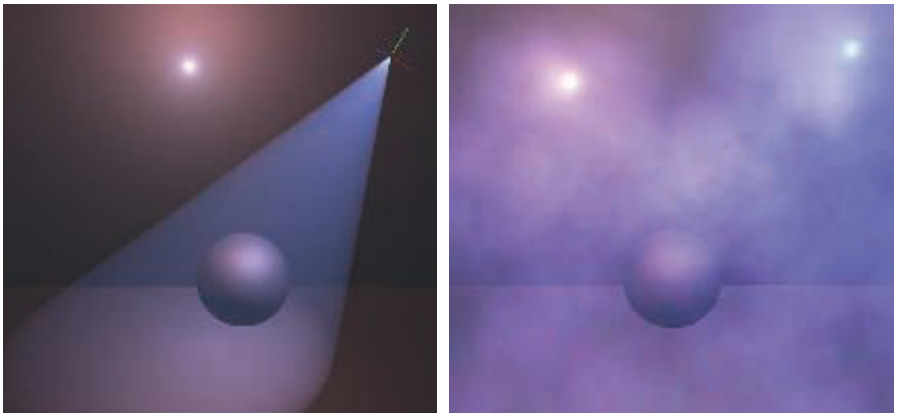

Figure 14.13. Volumetric light scattering from light sources evaluated using the analytical integration from the code snippet on page 603. It can be applied as a post effect assuming homogeneous media (left) or on particles, assuming each of them is a volume with depth (right). *(Image courtesy of Miles Macklin [1098].)*

The drawback with mesh splatting approaches is that accumulating many transparent meshes will increase the required memory bandwidth, likely causing a bottleneck, and the textured quad facing the camera can sometimes be visible. To work around this issue, post-processing techniques using closed-form solutions to light single scattering have been proposed. Assuming a homogeneous and spherical uniform phase function, it is possible to integrate scattered light with correct transmittance along a path assuming a constant medium. The result is visible in Figure 14.13. An example implementation of this technique is shown here in a GLSL shader code snippet [1098]:

```
float inScattering(
vec3 rayStart, vec3 rayDir,
vec3 lightPos, float rayDistance)
{
    // Calculate coefficients.
    vec3  q = rayStart - lightPos;
    float b = dot(rayDir, q);
    float c = dot(q, q);
    float s = 1.0f / sqrt(c - b*b);

    // Factorize some components.
    float x = s * rayDistance;
    float y = s * b;
    return s * atan( (x) / (1.0 + (x + y) * y));
}
```

where `rayStart` is the ray start position, `rayDir` is the ray normalized direction, `rayDistance` the integration distance along the ray, and `lightPos` the light source

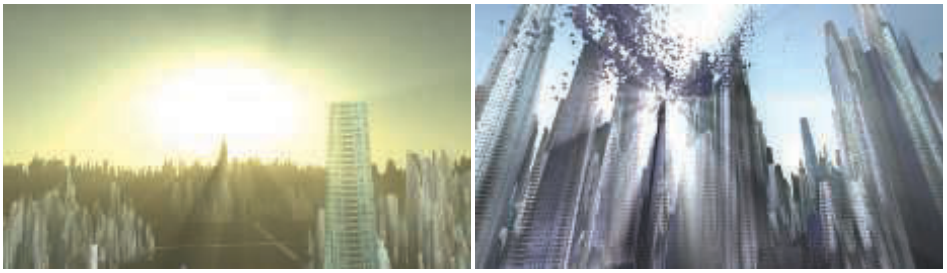

Figure 14.14. Light shafts rendered using a screen-space post-process. *(Image courtesy of Kenny Mitchell [1225].)*

position. The solution by Sun et al. [1722] additionally takes into account the scattering coefficient σ_s. It also describes how the diffuse and specular radiance bouncing off Lambertian and Phong surfaces should be affected by the fact that the light would have scattered in non-straight paths before hitting any surface. In order to take into account transmittance and phase functions, a more ALU-heavy solution can be used [1364]. All these models are effective at what they do, but are not able to take into account shadows from depth maps or heterogeneous participating media.

It is possible to approximate light scattering in screen space by relying on a technique known as bloom [539, 1741]. Blurring the framebuffer and adding a small percentage of it back onto itself [44] will make every bright object leak radiance around it. This technique is typically used to approximate imperfections in camera lenses, but in some environments, it is an approximation that works well for short distances and non-occluded scattering. Section 12.3 describes bloom in more detail.

Dobashi et al. [359] present a method of rendering large-scale atmospheric effects using a series of planes sampling the volume. These planes are perpendicular to the view direction and are rendered from back to front. Mitchell [1219] also proposes the same approach to render spotlight shafts, using shadow maps to cast volumetric shadows from opaque objects. Rendering volume by splatting slices is described in detail in Section 14.3.1.

Mitchell [1225] and Rohleder and Jamrozik [1507] present an alternative method working in screen space; see Figure 14.14. It can be used to render light shafts from a distant light such as the sun. First, a fake bright object is rendered around the sun on the far plane in a buffer cleared to black, and a depth buffer test is used to accept non-occluded pixels. Second, a directional blur is applied on the image in order to leak the previously accumulated radiance outward from the sun. It is possible to use a separable filtering technique (Section 12.1) with two passes, each using n samples, to get the same blur result as n^2 samples but rendered faster [1681]. To finish, the final blurred buffer can be added onto the scene buffer. The technique is efficient and, despite the drawback that only the light sources visible on screen can cast light shafts, it provides a significant visual result at a small cost.

14.3 General Volumetric Rendering

In this section, we present volumetric rendering techniques that are more physically based, i.e., that try to represent the medium's material and its interaction with light sources (Section 14.1.1). General volumetric rendering is concerned with spatially varying participating media, often represented using voxels (Section 13.10), with volumetric light interactions resulting in visually complex scattering and shadowing phenomena. A general volumetric rendering solution must also account for the correct composition of volumes with other scene elements, such as opaque or transparent surfaces. The spatially varying media's properties might be the result of a smoke and fire simulation that needs to be rendered in a game environment, together with volumetric light and shadow interactions. Alternately, we may wish to represent solid materials as semitransparent volumes, for applications such as medical visualization.

14.3.1 Volume Data Visualization

Volume data visualization is a tool used for the display and analysis of volume data, usually scalar fields. Computer tomography (CT) and magnetic resonance image (MRI) techniques can be used to create clinical diagnostic images of internal body structures. A data set may be, say, 256^3 voxels, each location holding one or more values. This voxel data can be used to form a three-dimensional image. Voxel rendering can show a solid model, or make various materials (e.g., the skin and skull) appear partially or fully transparent. Cutting planes can be used to show only a sub-volume or parts of the source data. In addition to its use for visualization in such diverse fields as medicine and oil prospecting, volumetric rendering can also produce photorealistic imagery.

There are many voxel rendering techniques [842]. It is possible to use regular path tracing or photon mapping to visualize volumetric data under complex lighting environments. Several less-expensive methods have been proposed to achieve real-time performance.

For solid objects, implicit surface techniques can be used to turn voxels into polygonal surfaces, as described in Section 17.3. For semitransparent phenomena, the volume data set can be sampled by a set of equally spaced slices in layers perpendicular to the view direction. Figure 14.15 shows how this works. It is also possible to render opaque surfaces with this approach [797]. In this case, the solid volume is considered present when the density is greater than a given threshold, and the normal $\mathbf{n}$ can be evaluated as the three-dimensional gradient of the density field.

For semitransparent data, it is possible to store color and opacity per voxel. To reduce the memory footprint and enable users to control the visualization, transfer functions have been proposed. A first solution is to map a voxel density scalar to color and opacity using a one-dimensional transfer texture. However, this does not allow identifying specific material transitions, for instance, human sinuses bone-to-air or bone-to-soft tissue, independently, with separate colors. To solve this issue, Kniss

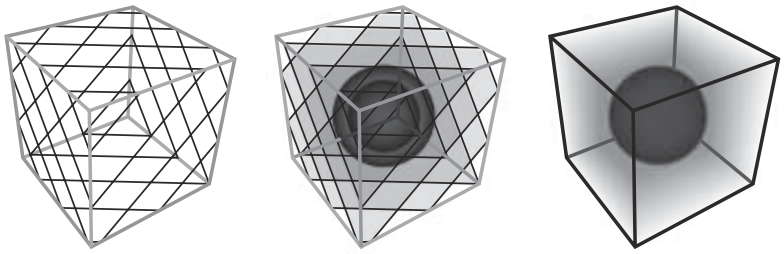

Figure 14.15. A volume is rendered by a series of slices parallel to the view plane. Some slices and their intersection with the volume are shown on the left. The middle shows the result of rendering just these slices. On the right the result is shown when a large series of slices are rendered and blended. *(Figures courtesy of Christof Rezk-Salama, University of Siegen, Germany.)*

et al. [912] suggest using a two-dimensional transfer function that is indexed based on density d and the gradient length of the density field $||\nabla d||$. Regions of change have high gradient magnitudes. This approach results in more meaningful colorization of density transitions. See Figure 14.16.

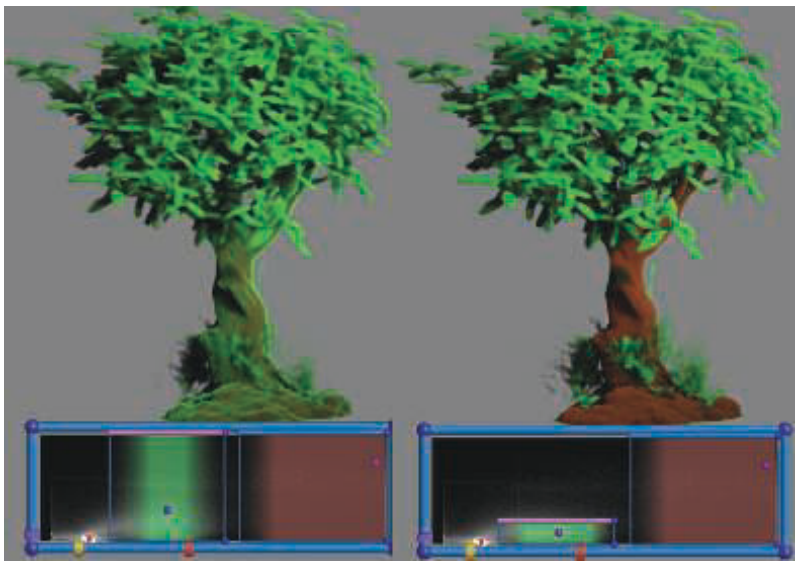

Figure 14.16. Volume material and opacity evaluated using a one-dimensional (left) and two-dimensional (right) transfer function [912]. In the second case, it is possible to maintain the brown color of the trunk without having it covered with the green color of lighter density representing the leaves. The bottom part of the image represents the transfer functions, with the x-axis being density and the y-axis the gradient length of the density field $||\nabla d||$. *(Image courtesy of Joe Michael Kniss [912].)*

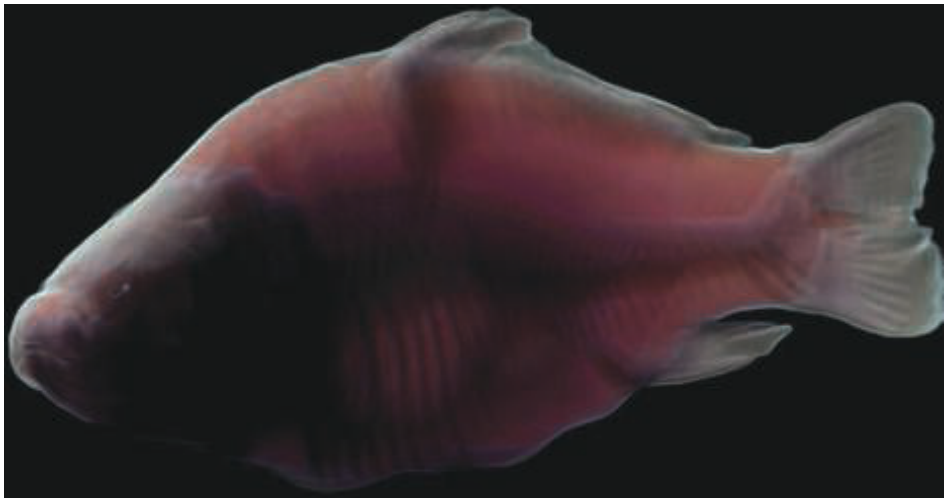

Figure 14.17. Volumetric rendering with forward subsurface scattering using light propagation through half-angle slices. *(Image courtesy of Ikits et al. [797].)*

Ikits et al. [797] discuss this technique and related matters in depth. Kniss et al. [913] extend this approach, instead slicing according to the half-angle. Slices are still rendered back to front but are oriented halfway between the light and view directions. Using this approach, it is possible to render radiance and occlusion from the light's point of view and accumulate each slice in view space. The slice texture can be used as input when rendering the next slice, using occlusion from the light direction to evaluate volumetric shadows, and using radiance to estimate multiple scattering, i.e., light bouncing multiple times within a medium before reaching the eye. Because the previous slice is sampled according to multiple samples in a disk, this technique can synthesize only subsurface phenomena resulting from forward scattering within a cone. The final image is of high quality. See Figure 14.17. This half-angle approach has been extended by Schott et al. [1577, 1578] to evaluate ambient occlusion and depth-of-field blur effects, which improves the depth and volume perception of users viewing the voxel data.

As seen in Figure 14.17, half-angle slicing can render high-quality subsurface scattering. However, the memory bandwidth cost due to rasterization has to be paid for each slice. Tatarchuk and Shopf [1744] perform medical imaging using ray marching in shaders and so pay the rasterization bandwidth cost only once. Lighting and shadowing can be achieved as described in the next section.

14.3.2 Participating Media Rendering

Real-time applications can portray richer scenes by rendering participating media. These effects become more demanding to render when factors such as the time of

day, weather, or environment changes such as building destruction are involved. For instance, fog in a forest will look different if it is noon or dusk. Light shafts shining in between trees should adapt to the sun's changing direction and color. Light shafts should also be animated according to the trees' movement. Removing some trees via, say, an explosion would result in a change in the scattered light in that area due to fewer occluders and to the dust produced. Campfires, flashlights, and other sources of light will also generate scattering in the air. In this section, we discuss techniques that can simulate the effects of these dynamic visual phenomena in real time.

A few techniques are focused on rendering shadowed large-scale scattering from a single source. One method is described in depth by Yusov [1958]. It is based on sampling in-scattering along epipolar lines, rays of light that project onto a single line on the camera image plane. A depth map from the light's point of view is used to determine whether a sample is shadowed. The algorithm performs a ray march starting from the camera. A min/max hierarchy along rays is used to skip empty space, while only ray-marching at depth discontinuities, i.e., where it is actually needed to accurately evaluate volumetric shadows. Instead of sampling these discontinuities along epipolar lines, it is possible to do it in view space by rendering a mesh generated from the light-space depth map [765]. In view space, only the volume between front- and backfaces is needed to evaluate the final scattered radiance. To this end, the in-scattering is computed by adding the scattered radiance resulting from frontfaces to the view, and subtracting it for backfaces.

These two methods are effective at reproducing single-scattering events with shadows resulting from opaque surface occlusions [765, 1958]. However, neither can represent heterogeneous participating media, since they both assume that the medium is of a constant material. Furthermore, these techniques cannot take into account volumetric shadows from non-opaque surfaces, e.g., self-shadowing from participating media or transparent shadows from particles (Section 13.8). They are still used in games to great effect, since they can be rendered at high resolution and they are fast, thanks to empty-space skipping [1958].

Splatting approaches have been proposed to handle the more general case of a heterogeneous medium, sampling the volume material along a ray. Without considering any input lighting, Crane et al. [303] use splatting for rendering smoke, fire, and water, all resulting from fluid simulation. In the case of smoke and fire, at each pixel a ray is generated that is ray-marched through the volume, gathering color and occlusion information from the material at regular intervals along its length. In the case of water, the volume sampling is terminated once the ray's first hit point with the water surface is encountered. The surface normal is evaluated as the density field gradient at each sample position. To ensure a smooth water surface, tricubic interpolation is used to filter density values. Examples using these techniques are shown in Figure 14.18.

Taking into account the sun, along with point lights and spotlights, Valient [1812] renders into a half-resolution buffer the set of bounding volumes where scattering from each source should happen. Each of the light volumes is ray-marched with a per-pixel random offset applied to the ray marching start position. Doing so adds a bit of

Figure 14.18. Fog and water rendered using volumetric rendering techniques in conjunction with fluid simulation on the GPU. *(Image on left from "Hellgate: London," courtesy of Flagship Studios, Inc.; image on right courtesy of NVIDIA Corporation [303].)*

noise, which has the advantage of removing banding artifacts resulting from constant stepping. The use of different noise values each frame is a means to hide artifacts. After reprojection of the previous frame and blending with the current frame, the noise will be averaged and thus will vanish. Heterogeneous media are rendered by voxelizing flat particles into a three-dimensional texture mapped onto the camera frustum at one eighth of the screen resolution. This volume is used during ray marching as the material density. The half-resolution scattering result can be composited over the full-resolution main buffer using first a bilateral Gaussian blur and then a bilateral up-sampling filter [816], taking into account the depth difference between pixels. When the depth delta is too high compared to the center pixel, the sample is discarded. This Gaussian blur is not mathematically separable (Section 12.1), but it works well in practice. The complexity of this algorithm depends on the number of light volumes splatted on screen, as a function of their pixel coverage.

This approach has been extended by using blue noise, which is better at producing a uniform distribution of random values over a frame pixel [539]. Doing so results in smoother visuals when up-sampling and blending samples spatially with a bilateral filter. Up-sampling the half-resolution buffer can also be achieved using four stochastic samples blended together. The result is still noisy, but because it gives full-resolution per-pixel noise, it can easily be resolved by a temporal antialiasing post-process (Section 5.4).

The drawback of all these approaches is that depth-ordered splatting of volumetric elements with any other transparent surfaces will never give visually correct ordering of the result, e.g., with large non-convex transparent meshes or large-scale particle effects. All these algorithms need some special handling when it comes to applying volumetric lighting on transparent surfaces, such as a volume containing in-scattering and transmittance in voxels [1812]. So, why not use a voxel-based representation from

the start, to represent not only spatially varying participating media properties but also the radiance distribution resulting from light scattering and transmittance? Such techniques have long been used in the film industry [1908].

Wronski [1917] proposes a method where the scattered radiance from the sun and lights in a scene is voxelized into a three-dimensional volume texture V_0 mapped over the view clip space. Scattered radiance is evaluated for each voxel-center world-space position, where the x- and y-axes of the volume correspond to screen coordinates, while the z-coordinate is mapped over camera frustum depth. This volume texture is considerably lower resolution than the final image. A typical implementation of this technique uses a voxel resolution that is one eighth of the screen resolution in the x- and y-axes. Subdivision along the z-coordinate depends on a quality and performance trade-off, with 64 slices being a typical choice. This texture contains the in-scattered radiance $L_{\text{scat}_{\text{in}}}$ in RGB as well as extinction σ_t in alpha. From this input data, the final scattering volume V_f is generated by iterating over each slice from near to far using

$$V_f[x, y, z] = (L'_{\text{scat}} + T'_r L_{\text{scat}_{\text{in}}} d_s, T_{r_{\text{slice}}} T'_r), \tag{14.15}$$

where $L'_{\text{scat}} = V_0[x, y, z-1]_{rgb}$, $T'_r = V_0[x, y, z-1]_a$, and $T_{r_{\text{slice}}} = e^{-\sigma_t d_s}$. This updates slice z from the previous slice $z-1$ data over world-space slice depth d_s. Doing so will result in V_f containing the scattered radiance reaching the viewer and transmittance over the background in each voxel. In Equation 14.15, notice that $L_{\text{scat}_{\text{in}}}$ is affected by only the transmittance from previous slices T'_r. This behavior is incorrect, since $L_{\text{scat}_{\text{in}}}$ should also be affected by the transmittance resulting from σ_t within the current slice.

This problem is discussed by Hillaire [742, 743]. He proposes an analytical solution to the integration of $L_{\text{scat}_{\text{in}}}$ for a constant extinction σ_t over a given depth:

$$V_f[x, y, z] = \left(L'_{\text{scat}} + \frac{L_{\text{scat}_{\text{in}}} - L_{\text{scat}_{\text{in}}} T_{r_{\text{slice}}}}{\sigma_t}, T_{r_{\text{slice}}} T'_r \right). \tag{14.16}$$

The final pixel radiance L_o of an opaque surface with radiance L_s will be modified by L_{scat} and T_r from V_f, sampled with clip-space coordinates as $L_o = T_r L_s + L_{\text{scat}}$. Because V_f is coarse, it is subject to aliasing from camera motion and from high-frequency bright lights or shadows. The previous frame V_f can be reprojected and combined with the new V_f using an exponential moving average [742].

Building over this framework, Hillaire [742] presents a physically based approach for the definition of participating media material as follows: scattering σ_s, absorption σ_a, phase function parameter g, and emitted radiance L_e. This material is mapped to the camera frustum and stored into a participating media material volume texture V_{pm}, being the three-dimensional version of a G-buffer storing an opaque surface material (Section 20.1). Considering single scattering only, and despite voxel discretization, Hillaire shows that using such a physically based material representation results in visuals that are close to path tracing. Analogous to meshes, participating media volumes that are positioned in the world are voxelized into V_{pm} (see Figure 14.19). In each of these volumes, a single material is defined and variation is added, thanks

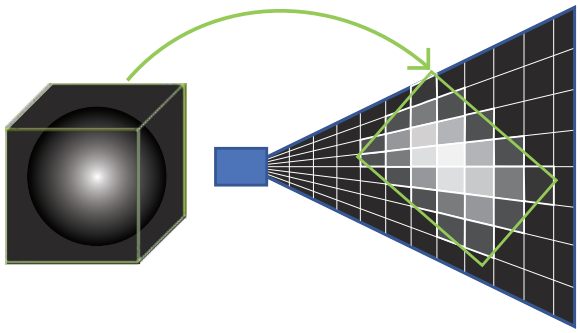

Figure 14.19. An example of a participating media volume placed by an artist in a level and voxelized into camera frustum space [742, 1917]. On the left, a three-dimensional texture, in the shape of a sphere in this case, is mapped onto the volume. The texture defines the volume's appearance, similar to textures on triangles. On the right, this volume is voxelized into the camera frustum by taking into account its world transform. A compute shader accumulates the contribution into each voxel the volume encompass. The resulting material can then be used to evaluate light scattering interactions in each voxel [742]. Note that, when mapped onto camera clip space, voxels take the shape of small frusta and are referred to as *froxels*.

to density sampled from a three-dimensional input texture, resulting in heterogeneous participating media. The result is shown in Figure 14.20. This same approach is also implemented in the Unreal Engine [1802], but instead of using box volumes as sources of participating media, particles are used, assuming a spherical volume instead of a box. It is also possible to represent the material volume texture using a sparse structure [1191], using a topmost volume with each voxel being empty or pointing to a finer-grained volume containing the participating media material data.

The only drawback with camera frustum volume-based approaches [742, 1917] is the low screen-space resolution that is required in order to reach acceptable performance on less powerful platforms (and use a reasonable amount of memory). This is where the previously explained splatting approaches excel, as they produce sharp visual details. As noted earlier, splatting requires more memory bandwidth and provides less of a unified solution, e.g., it is harder to apply on any other transparent surfaces without sorting issues or have participating media cast volume shadows on itself.

Not only direct light, but also illumination that has already bounced, or scattered, can scatter through a medium. Similar to Wronski [1917], the Unreal Engine makes it possible to bake volume light maps, storing irradiance in volumes, and have it scatter back into the media when voxelized in the view volume [1802]. In order to achieve dynamic global illumination in participating media, it is also possible to rely on light propagation volumes [143].

An important feature is the use of volumetric shadows. Without them, the final image in a fog-heavy scene can look too bright and flatter than it should be [742]. Furthermore, shadows are an important visual cue. They help the viewer with perception of depth and volume [1846], produce more realistic images, and can lead to better

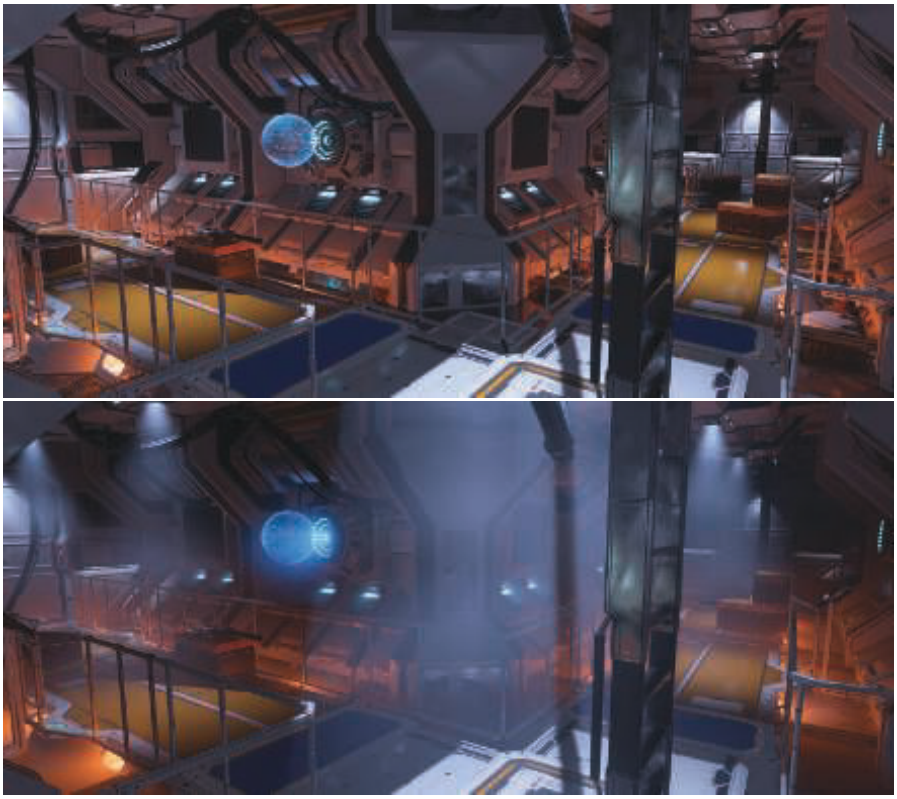

Figure 14.20. A scene rendered without (top) and with (bottom) volumetric lighting and shadowing. Every light in the scene interacts with participating media. Each light's radiance, IES profile, and shadow map are used to accumulate its scattered light contribution [742]. *(Image courtesy of Frostbite, © 2018 Electronic Arts Inc.)*

immersion. Hillaire [742] presents a unified solution to achieve volumetric shadows. Participating media volumes and particles are voxelized into three volumes cascaded around the camera, called *extinction volumes*, according to a clipmap distribution scheme [1777]. These contain extinction σ_t values needed to evaluate T_r and represent a single unified source of data to sample in order to achieve volumetric shadows using an opacity shadow map [742, 894]. See Figure 14.21. Such a solution enables particles and participating media to self-shadow and to cast shadows on each other, as well as any other opaque and transparent elements in a scene.

Volumetric shadows can be represented using opacity shadow maps. However, using a volume texture can quickly become a limitation if high resolutions are needed to catch details. Thus, alternative representations have been proposed to represent T_r more efficiently, for instance using an orthogonal base of functions such as a Fourier [816] or discrete cosine transform [341]. Details are given in Section 7.8.

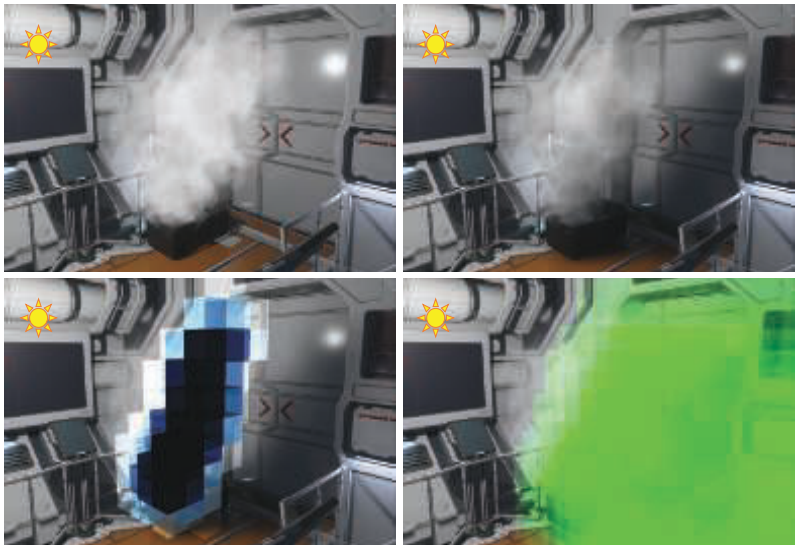

Figure 14.21. At the top, the scene is rendered without (left) and with (right) volumetric shadows. On the bottom, debug views of voxelized particle extinction (left) and volume shadows (right). Greener means less transmittance [742]. *(Image courtesy of Frostbite, © 2018 Electronic Arts Inc.)*

14.4 Sky Rendering

Rendering a world inherently requires a planet sky, atmospheric effects, and clouds. What we call the blue sky on the earth is the result of sunlight scattering in the atmosphere's participating media. The reason why the sky is blue during day and red when the sun is at the horizon is explained in Section 14.1.3. The atmosphere is also a key visual cue since its color is linked to the sun direction, which is related to the time of day. The atmosphere's (sometimes) foggy appearance helps viewers with the perception of relative distance, position, and size of elements in a scene. As such, it is important to accurately render these components required by an increasing number of games and other applications featuring dynamic time of day, evolving weather affecting cloud shapes, and large open worlds to explore, drive around, or even fly over.

14.4.1 Sky and Aerial Perspective

To render atmospheric effects, we need to take into account two main components, as shown in Figure 14.22. First, we simulate the sunlight's interaction with air particles, resulting in wavelength-dependent Rayleigh scattering. This will result in the sky color and a thin fog, also called *aerial perspective*. Second, we need the effect of large particles concentrated near the ground on the sunlight. The concentration of these

Figure 14.22. The two different types of atmospheric light scattering: Rayleigh only at the top and Mie with regular Rayleigh scattering at the bottom. From left to right: density of 0, regular density as described in [203], and exaggerated density. *(Image courtesy of Frostbite, © 2018 Electronic Arts Inc. [743].)*

large particles depends on such factors as weather conditions and pollution. Large particles cause wavelength-independent Mie scattering. This phenomenon will cause a bright halo around the sun, especially with a heavy particle concentration.

The first physically based atmosphere model [1285] rendered the earth and its atmosphere from space, simulating single scattering. Similar results can by achieved using the method proposed by O'Neil [1333]. The earth can be rendered from ground to space using ray marching in a single-pass shader. Expensive ray marching to integrate Mie and Rayleigh scattering is done per vertex when rendering the sky dome. The visually high-frequency phase function is, however, evaluated in the pixel shader. This makes the appearance smooth and avoids revealing the sky geometry due to interpolation. It is also possible to achieve the same result by storing the scattering in a texture and to distribute the evaluation over several frames, accepting update latency for better performance [1871].

Analytical techniques use fitted mathematical models on measured sky radiance [1443] or reference images generated using expensive path tracing of light scattering in the atmosphere [778]. The set of input parameters is generally limited compared to those for a participating media material. For example, *turbidity* represents the contribution of particles resulting in Mie scattering, instead of σ_s and σ_t coefficients. Such a model presented by Preetham et al. [1443] evaluates the sky radiance in any direction using turbidity and sun elevation. It has been improved by adding support for spectral output, better directionality to the scattered radiance around the sun, and a new ground albedo input parameter [778]. Analytical sky models are fast to evaluate. They are, however, limited to ground views, and atmosphere parameters cannot be changed to simulate extra-terrestrial planets or achieve specific art-driven visuals.

Figure 14.23. Real-time rendering of the earth's atmosphere from the ground (left) and from space (right) using a lookup table. *(Image courtesy of Bruneton and Neyret [203].)*

Another approach to rendering skies is to assume that the earth is perfectly spherical, with a layer of atmosphere around it composed of heterogeneous participating media. Extensive descriptions of the atmosphere's composition are given by Bruneton and Neyret [203] as well as Hillaire [743]. Leveraging these facts, precomputed tables can be used to stored transmittance and scattering according to the current view altitude r, the cosine of the view vector angle relative to the zenith μ_v, the cosine of the sun direction angle relative to the zenith μ_s, and the cosine of the view vector angle relative to the sun direction in the azimuthal plane ν. For instance, transmittance from the viewpoint to the atmosphere's boundary can be parameterized by two parameters, r and μ_v. During a precompute step, the transmittance can be integrated in the atmosphere and stored in a two-dimensional lookup table (LUT) texture T_{lut} that can be sampled at runtime using the same parameterization. This texture can be used to apply atmosphere transmittance to sky elements such as the sun, stars, or other celestial bodies.

Considering scattering, Bruneton and Neyret [203] describe a way to store it in a four-dimensional LUT S_{lut} parameterized by all the parameters in the preceding paragraph. They also provide a way to evaluate multiple scattering of order n by iterating n times: (i) Evaluate the single-scattering table S_{lut}, (ii) evaluate S^n_{lut} using S^{n-1}_{lut}, and (iii) add the result to S_{lut}. Do (ii) and (iii) $n-1$ times. More details about the process, as well as source code, are provided by Bruneton and Neyret [203]. See Figure 14.23 for examples of the result. Bruneton and Neyret's parameterization can sometimes exhibit visual artifacts at the horizon. Yusov [1957] has proposed an improved transformation. It is also possible to use an only three-dimensional LUT by ignoring ν [419]. Using this scheme, the earth will not cast shadows in the atmosphere, which can be an acceptable trade-off. The advantage is that this LUT will be much smaller and less expensive to update and sample.

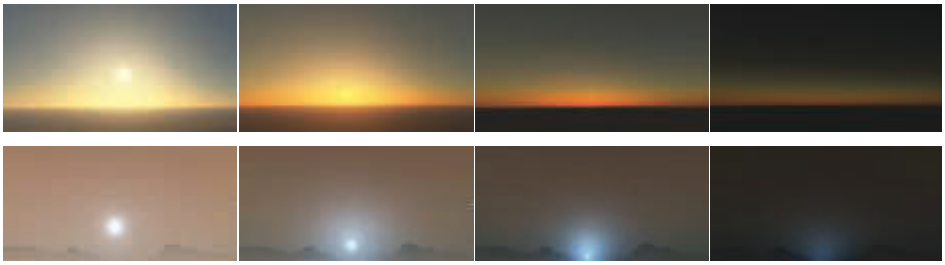

Figure 14.24. Real-time rendering using a fully parameterized model enables the simulation of the earth's atmosphere (top) and the atmosphere of other planets, such as Mars's blue sunset (bottom). *(Top images courtesy of Bruneton and Neyret [203] and bottom images courtesy of Frostbite, © 2018 Electronic Arts Inc. [743].)*

This last three-dimensional LUT approach is used by many Electronic Arts Frostbite real-time games, such as *Need for Speed*, *Mirror's Edge Catalyst*, and *FIFA* [743]. In this case, artists can drive the physically based atmosphere parameters to reach a target sky visual and even simulate extra-terrestrial atmosphere. See Figure 14.24. The LUT has to be recomputed when atmosphere parameters are changed. To update these LUTs more efficiently, it is also possible to use a function that approximates the integral of the material in the atmosphere instead of ray-marching through it [1587]. The cost of updating the LUTs can be amortized down to 6% of the original by temporally distributing the evaluation of the LUTs and multiple scattering. This is achievable by updating only a sub-part of S^n_{lut} for a given scattering order n, while interpolating the last two solved LUTs, accepting a few frames of latency. As another optimization, to avoid sampling the different LUTs multiple times per pixel, Mie and Rayleigh scattering are baked in voxels of a camera-frustum-mapped low-resolution volume texture. The visually high-frequency phase function is evaluated in the pixel shader in order to produce smooth scattering halos around the sun. Using this type of volume texture also permits applying aerial perspective per vertex on any transparent objects in the scene.

14.4.2 Clouds

Clouds are complex elements in the sky. They can look menacing when representing an incoming storm, or alternately appear discreet, epic, thin, or massive. Clouds change slowly, with both their large-scale shapes and small-scale details evolving over time. Large open-world games with weather and time-of-day changes are more complex cases that require dynamic cloud-rendering solutions. Different techniques can be used depending on the target performance and visual quality.

Clouds are made of water droplets, featuring high-scattering coefficients and complex phase functions that result in a specific look. They are often simulated using participating media, as described in Section 14.1, and their materials have been measured as having a high single-scattering albedo $\rho = 1$ and extinction coefficients σ_t in

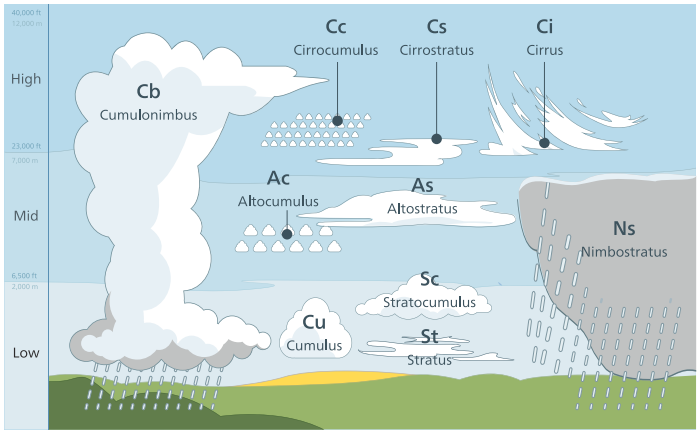

Figure 14.25. Different types of clouds on the earth. *(Image courtesy of Valentin de Bruyn.)*

the range $[0.04, 0.06]$ for stratus (low-level horizontal cloud layers) and $[0.05, 0.12]$ for cumulus [743] (isolated low-level cotton-like fluffy clouds). See Figure 14.25. Given the fact that ρ is close to 1, $\sigma_s = \sigma_t$ can be assumed.

A classic approach to cloud rendering is to use a single panoramic texture composited over the sky using alpha blending. This is convenient when rendering a static sky. Guerrette [620] presents a visual flow technique that gives the illusion of cloud motion in the sky affected by a global wind direction. This is an efficient method that improves over the use of a static set of panoramic cloud textures. However, it will not be able to represent any change to the cloud shapes and lighting.

Clouds as Particles

Harris renders clouds as volumes of particles and impostors [670]. See Section 13.6.2 and Figure 13.9 on page 557.

Another particle-based cloud rendering method is presented by Yusov [1959]. He uses rendering primitives that are called *volume particles*. Each of these is represented by a four-dimensional LUT, allowing retrieval of scattered light and transmittance on the view-facing quad particle as a function of the sun light and view directions. See Figure 14.26. This approach is well suited to render stratocumulus clouds. See Figure 14.25.

When rendering clouds as particles, discretization and popping artifacts can often be seen, especially when rotating around clouds. These problems can be avoided by using *volume-aware* blending. This ability is made possible by using a GPU feature called rasterizer order views (Section 3.8). Volume-aware blending enables the synchronization of pixel shader operations on resources per primitive, allowing deterministic custom blending operations. The closest n particles' depth layers are kept in a buffer at the same resolution as the render target into which we render. This

Figure 14.26. Clouds rendered as volumes of particles. *(Image courtesy of Egor Yusov [1959].)*

buffer is read and used to blend the currently rendered particle by taking into account intersection depth, then finally written out again for the next particle to be rendered. The result is visible in Figure 14.27.

Clouds as Participating Media

Considering clouds as isolated elements, Bouthors et al. [184] represent a cloud with two components: a mesh, showing its overall shape, and a hypertexture [1371], adding high-frequency details under the mesh surface up to a certain depth inside the cloud. Using this representation, a cloud edge can be finely ray-marched in order to gather details, while the inner part can be considered homogeneous. Radiance is integrated while ray marching the cloud structure, and different algorithms are used to gather scattered radiance according to scattering order. Single scattering is integrated using an analytical approach described in Section 14.1. Multiple scattering evaluation is accelerated using offline precomputed transfer tables from disk-shaped light collectors positioned at the cloud's surface. The final result is of high visual quality, as shown in Figure 14.28.

Figure 14.27. On the left, cloud particles rendered the usual way. On the right, particles rendered with volume-aware blending. *(Images courtesy of Egor Yusov [1959].)*

Figure 14.28. Clouds rendered using meshes and hypertextures. *(Image courtesy of Bouthors et al. [184].)*

Instead of rendering clouds as isolated elements, it is possible to model them as a layer of participating media in the atmosphere. Relying on ray marching, Schneider and Vos presented an efficient method to render clouds in this way [1572]. With only a few parameters, it is possible to render complex, animated, and detailed cloud shapes under dynamic time-of-day lighting conditions, as seen in Figure 14.29. The layer is

Figure 14.29. Clouds rendered using a ray-marched cloud layer using Perlin-Worley noise, and featuring dynamic volumetric lighting and shadowing. *(Results by Schneider and Vos [1572], copyright ©2017 Guerrilla Games.)*

built using two levels of procedural noise. The first level gives the cloud its base shape. The second level adds details by eroding this shape. In this case, a mix of Perlin [1373] and Worley [1907] noise is reported to be a good representation of the cauliflower-like shape of cumulus and similar clouds. Source code and tools to generate such textures have been shared publicly [743, 1572]. Lighting is achieved by integrating scattered light from the sun using samples distributed in the cloud layer along the view ray.

Volumetric shadowing can be achieved by evaluating the transmittance for a few samples within the layer, testing toward the sun [743, 1572] as a secondary ray marching. It is possible to sample the lower mipmap levels of the noise textures for these shadow samples in order to achieve better performance and to smooth out artifacts that can become visible when using only a few samples. An alternative approach to avoid secondary ray marching per sample is to encode the transmittance curve from the sun once per frame in textures, using one of the many techniques available (Section 13.8). For instance, the game *Final Fantasy XV* [416] uses transmittance function mapping [341].

Rendering clouds at high resolution with ray marching can become expensive if we want to capture every little detail. To achieve better performance, it is possible to render clouds at a low resolution. One approach is to update only a single pixel within each 4×4 block and reproject the previous frame data to fill up the rest [1572]. Hillaire [743] proposes a variation that always renders at a fixed lower resolution and adds noise on the view ray marching start position. The previous frame result can be reprojected and combined with the new frame using an exponential moving average [862]. This approach renders at lower resolution but can converge faster.

Clouds' phase functions are complex [184]. Here we present two methods that can be used to evaluate them in real time. It is possible to encode the function as a texture and sample it based on θ. If doing so requires too much memory bandwidth, it is possible to approximate the function by combining two Henyey-Greenstein phase functions from Section 14.1.4 [743] as

$$p_{\text{dual}}(\theta, g_0, g_1, w) = p_{\text{dual}_0} + w(p_{\text{dual}_1} - p_{\text{dual}_0}), \tag{14.17}$$

where the two main scattering eccentricities g_0 and g_1, as well as the blend factor w, can be authored by an artist. This is important in representing both the main forward and backward scattering directions, revealing details in clouds when looking both away from and toward the light source, e.g., the sun or moon. See Figure 14.30.

There are different ways to approximate scattered light from ambient lighting in clouds. A straightforward solution is to use a single radiance input uniformly integrated from a render of the sky into a cube map texture. A bottom-up, dark-to-light gradient can also be used to scale the ambient lighting to approximate occlusion from clouds themselves. It is also possible to separate this input radiance as bottom and top, e.g., ground and sky [416]. Then ambient scattering can analytically be integrated for both contributions, assuming constant media density within the cloud layer [1149].

Figure 14.30. Clouds rendered using a ray-marched cloud layer with dynamic lighting and shadowing using a physically based representation of participating media as described by Hillaire [743]. *(Images courtesy of Sören Hesse (top) and Ben McGrath (bottom) from BioWare, ©2018 Electronic Arts Inc.)*

Multiple Scattering Approximation

Clouds' bright and white look is the result of light scattering multiple times within them. Without multiple scattering, thick clouds would mostly be lit at the edge of their volumes, and they would appear dark everywhere else. Multiple scattering is a key component for clouds to not look smoky or murky. It is excessively expensive to evaluate multiple scattering using path tracing. A way to approximate this phenomenon when ray marching has been proposed by Wrenninge [1909]. It integrates o octaves of scattering and sums them as

$$L_{\text{multiscat}}(\mathbf{x}, \mathbf{v}) = \sum_{n=0}^{o-1} L_{\text{scat}}(\mathbf{x}, \mathbf{v}), \tag{14.18}$$

where the following substitutions are made when evaluating L_{scat} (for instance, using σ_s' instead of σ_s): $\sigma_s' = \sigma_s a^n$, $\sigma_e' = \sigma_e b^n$, and $p'(\theta) = p(\theta c^n)$, where a, b, and c are user-control parameters in $[0, 1]$ that will let the light punch through the participating

Figure 14.31. Clouds rendered using Equation 14.18 as an approximation to multiple scattering. From left to right, n is set to 1, 2, and 3. This enable the sun light to punch through the clouds in a believable fashion. *(Image courtesy of Frostbite, © 2018 Electronic Arts Inc. [743].)*

media. Clouds look softer when these values are closer to 0. In order to make sure this technique is energy-conserving when evaluating $L_{\text{multiscat}}(\mathbf{x}, \mathbf{v})$, we must ensure that $a \leq b$. Otherwise, more light can scatter, because the equation $\sigma_t = \sigma_a + \sigma_s$ would not be respected, as σ_s could end up being larger than σ_t. The advantage of this solution is that it can integrate the scattered light for each of the different octaves on the fly while ray marching. The visual improvement is presented in Figure 14.31. The drawback is that it does a poor job at complex multiple scattering behavior when light could scatter in any direction. However, the look of the clouds is improved, and this method allows lighting artists to easily control the visuals with a few parameters and express their vision, thanks to a wider range of achievable results. With this approach, light can punch through the medium and reveal more internal details.

Clouds and Atmosphere Interactions

When rendering a scene with clouds, it is important to take into account interactions with atmospheric scattering for the sake of visual coherency. See Figure 14.32.

Since clouds are large-scale elements, atmospheric scattering should be applied to them. It is possible to evaluate the atmospheric scattering presented in Section 14.4.1

Figure 14.32. Clouds entirely covering the sky are rendered by taking into account the atmosphere [743]. Left: without atmospheric scattering applied on the clouds, leading to incoherent visuals. Middle: with atmospheric scattering, but the environment appears too bright without shadowing. Right: with clouds occluding the sky, thus affecting the light scattering in the atmosphere and resulting in coherent visuals. *(Image courtesy of Frostbite, © 2018 Electronic Arts Inc. [743].)*

for each sample taken through the cloud layer, but doing so quickly becomes expensive. Instead it is possible to apply the atmospheric scattering on the cloud according to a single depth representing the mean cloud depth and transmittance [743].

If cloud coverage is increased to simulate rainy weather, sunlight scattering in the atmosphere should be reduced under the cloud layer. Only light scattered through the clouds should scatter in the atmosphere under them. The illumination can be modified by reducing the sky's lighting contribution to the aerial perspective and adding scattered light back into the atmosphere [743]. The visual improvement is shown in Figure 14.32.

To conclude, cloud rendering can be achieved with advanced physically based material representation and lighting. Realistic cloud shapes and details can be achieved by using procedural noise. Finally, as presented in this section, it is also important to keep in mind the big picture, such as interactions of clouds with the sky, in order to achieve coherent visual results.

14.5 Translucent Surfaces

Translucent surfaces typically refer to materials having a high absorption together with low scattering coefficients. Such materials include glass, water, or the wine shown in Figure 14.2 on page 592. In addition, this section will also discuss translucent glass with a rough surface. These topics are also covered in detail in many publications [1182, 1185, 1413].

14.5.1 Coverage and Transmittance

As discussed in Section 5.5, a transparent surface can be treated as having a coverage represented by α, e.g., opaque fabric or tissue fibers hiding a percentage of what lies behind. For glass and other materials, we want to compute the translucency, where a solid volume lets a percentage of each light wavelength pass through, acting as a filter over the background as a function of transmittance T_r (Section 14.1.2). With the output color $\mathbf{c}_o$, the surface radiance $\mathbf{c}_s$, and the background color $\mathbf{c}_b$, the blending operation for a transparency-as-coverage surface is

$$\mathbf{c}_o = \alpha\mathbf{c}_s + (1 - \alpha)\mathbf{c}_b. \tag{14.19}$$

In the case of a translucent surface, the blending operation will be

$$\mathbf{c}_o = \mathbf{c}_s + \mathbf{T}_r\mathbf{c}_b, \tag{14.20}$$

where $\mathbf{c}_s$ contains the specular reflection of the solid surface, i.e., glass or a gel. Note that $\mathbf{T}_r$ is a three-valued transmittance color vector. To achieve colored translucency, one can use the dual-source color blending feature of any modern graphics API in order to specify these two output colors to blend with the target buffer color $\mathbf{c}_b$. Drobot [386]

Figure 14.33. Translucency with different absorption factors through multiple layers of a mesh [115]. *(Images courtesy of Louis Bavoil [115].)*

presents the different blending operations that can be used depending on whether, for a given surface, reflection and transmittance are colored or not.

In the general case, it is possible to use a common blending operation for coverage and translucency specified together [1185]. The blend function to use in this case is

$$\mathbf{c}_o = \alpha(\mathbf{c}_s + \mathbf{T}_r\mathbf{c}_b) + (1-\alpha)\mathbf{c}_b. \tag{14.21}$$

When the thickness varies, the amount of light transmitted can be computed using Equation 14.3, which can be simplified to

$$\mathbf{T}_r = e^{-\boldsymbol{\sigma}_t d}, \tag{14.22}$$

where d is the distance traveled through the material volume. The physical extinction parameter $\boldsymbol{\sigma}_t$ represents the rate at which light drops off as it travels through the medium. For intuitive authoring by artists, Bavoil [115] sets the target color $\mathbf{t}_c$ to be the amount of transmittance at some given distance d. Then extinction $\boldsymbol{\sigma}_t$ can be recovered as

$$\boldsymbol{\sigma}_t = \frac{-\log(\mathbf{t}_c)}{d}. \tag{14.23}$$

For example, with target transmittance color $\mathbf{t}_c = (0.3, 0.7, 0.1)$ and distance $d = 4.0$ meters, we recover

$$\boldsymbol{\sigma}_t = \frac{1}{4}(-\log 0.3, -\log 0.7, -\log 0.1) = (0.3010, 0.0892, 0.5756). \tag{14.24}$$

Note that a transmittance of 0 needs to be handled as a special case. A solution is to subtract a small epsilon, e.g., 0.000001, from each component of $\mathbf{T}_r$. The effect of color filtering is shown in Figure 14.33.

In the case of an empty shell mesh whose surface consists of a single thin layer of translucent material, the background color should be occluded as a function of

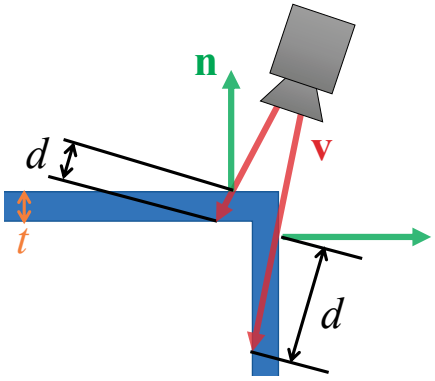

Figure 14.34. Colored transmittance computed from the distance d a view ray $\mathbf{v}$ has traveled within a transparent surface of thickness t. *(Right image courtesy of Activision Publishing, Inc. 2018.)*

the path length d that the light has traveled within the medium. So, viewing such as surface along its normal or tangentially will result in a different amount of background occlusion as a function of its thickness t, because the path length changes with angle. Drobot [386] proposes such an approach where the transmittance $\mathbf{T}_r$ is evaluated as

$$\mathbf{T}_r = e^{-\boldsymbol{\sigma}_t d}, \quad \text{where} \quad d = \frac{t}{\max(0.001, \mathbf{n} \cdot \mathbf{v})}. \tag{14.25}$$

Figure 14.34 shows the result. See Section 9.11.2 for more details about thin-film and multi-layer surfaces.

In the case of solid translucent meshes, computing the actual distance that a ray travels through a transmitting medium can be done in many ways. A common method is to first render the surface where the view ray exits the volume. This surface could be the backface of a crystal ball, or could be the sea floor (i.e., where the water ends). The depth or location of this surface is stored. Then the volume's surface is rendered. The stored exit depth is accessed in the shader, and the distance between it and the current pixel surface is computed. This distance is then used to compute the transmittance to apply over the background.

This method works if it is guaranteed that the volume is closed and convex, i.e., it has one entry and one exit point per pixel, as with a crystal ball. Our seabed example also works because once we exit the water we encounter an opaque surface, so further transmittance will not occur. For more elaborate models, e.g., a glass sculpture or other object with concavities, two or more separate spans may absorb incoming light. Using depth peeling, as discussed in Section 5.5, we can render the volume surfaces in precise back-to-front order. As each frontface is rendered, the distance through the volume is computed and used to compute transmittance. Applying each of these in turn gives the proper final transmittance. Note that if all volumes are made of the

Figure 14.35. Water rendered taking into account the transmittance and reflectance effects. Looking down, we can see into the water with a light blue tint since transmittance is high and blue. Near the horizon the seabed becomes less visible due to a lower transmittance (because light has to travel far into the water volume) and reflection increasing at the expense of transmission, due to the Fresnel effect. *(Image from "Crysis," courtesy of Crytek.)*

same material at the same concentration, the transmittance could be computed once at the end using the summed distances, if the surface has no reflective component. A-buffer or K-buffer methods directly storing object fragments in a single pass can also be used for more efficiency on recent GPUs [115, 230]. Such an example of multi-layer transmittance is shown in Figure 14.33.

In the case of large-scale sea water, the scene depth buffer can be directly used as a representation of the backface seabed. When rendering transparent surfaces, one must consider the Fresnel effect, as described in Section 9.5. Most transmitting media have an index of refraction significantly higher than that of air. At glancing angles, all the light will bounce back from the interface, and none will be transmitted. Figure 14.35 shows this effect, where underwater objects are visible when looking directly into the water, but looking farther out, at a grazing angle, the water surface mostly hides what is beneath the waves. Several articles explain handling reflection, absorption, and refraction for large bodies of water [261, 977].

14.5.2 Refraction

For transmittance we assume that the incoming light comes from directly beyond the mesh volume, in a straight line. This is a reasonable assumption when the front and back surfaces of the mesh are parallel and the thickness is not great, e.g., for a pane

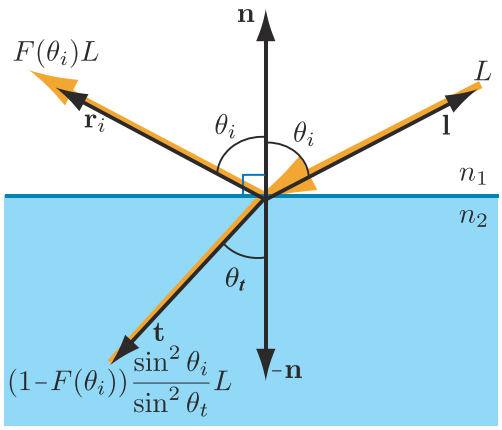

Figure 14.36. Refracted and transmitted radiance as a function of incident angle θ_i and transmission angle θ_t.

of glass. For other transparent media, the index of refraction plays an important role. Snell's law, which describes how light changes direction when a mesh's surface is encountered, is described in Section 9.5.

Due to conservation of energy, any light not reflected is transmitted, so the proportion of transmitted flux to incoming flux is $1 - f$, where f is the amount of reflected light. The proportion of transmitted-to-incident radiance, however, is different. Due to differences in projected area and solid angle between the incident and transmitted rays, the radiance relationship is

$$L_t = (1 - F(\theta_i))\frac{\sin^2\theta_i}{\sin^2\theta_t}L_i. \tag{14.26}$$

This behavior is illustrated in Figure 14.36. Snell's law combined with Equation 14.26 yields a different form for transmitted radiance:

$$L_t = (1 - F(\theta_i))\frac{n_2^2}{n_1^2}L_i. \tag{14.27}$$

Bec [123] presents an efficient method to compute the refraction vector. For readability (because n is traditionally used for the index of refraction in Snell's equation), we define $\mathbf{N}$ as the surface normal and $\mathbf{l}$ as the direction to the light:

$$\mathbf{t} = (w - k)\mathbf{N} - n\mathbf{l}, \tag{14.28}$$

where $n = n_1/n_2$ is the relative index of refraction, and

$$\begin{aligned} w &= n(\mathbf{l} \cdot \mathbf{N}), \\ k &= \sqrt{1 + (w - n)(w + n)}. \end{aligned} \tag{14.29}$$

Figure 14.37. Left: refraction by glass angels of a cubic environment map, with the map itself used as a skybox background. Right: reflection and refraction by a glass ball with chromatic aberration. *(Left image from the three.js example webgl_materials_cube map_refraction [218], Lucy model from the Stanford 3D scanning repository, texture by Humus. Right image courtesy of Lee Stemkoski [1696].)*

The resulting refraction vector $\mathbf{t}$ is returned normalized. Water has an index of refraction of approximately 1.33, glass typically around 1.5, and air effectively 1.0.

The index of refraction varies with wavelength. That is, a transparent medium will bend each color of light at a different angle. This phenomenon is called *dispersion*, and explains why prisms spread out white light into a rainbow-colored light cone and why rainbows occur. Dispersion can cause a problem in lenses, termed *chromatic aberration*. In photography, this phenomenon is called *purple fringing*, and can be particularly noticeable along high contrast edges in daylight. In computer graphics we normally ignore this effect, as it is usually an artifact to be avoided. Additional computation is needed to properly simulate the effect, as each light ray entering a transparent surface generates a set of light rays that must then be tracked. As such, normally a single refracted ray is used. It is worthwhile to note that some virtual reality renderers apply an inverse chromatic aberration transform, to compensate for the headset's lenses [1423, 1823].

A general way to give an impression of refraction is to generate a cubic environment map (EM) from the refracting object's position. When this object is then rendered, the EM can be accessed by using the refraction direction computed for the frontfacing surfaces. An example is shown in Figure 14.37. Instead of using an EM, Sousa [1675] proposes a screen-space approach. First, the scene is rendered as usual without any refractive objects into a scene texture $\mathbf{s}$. Second, refractive objects are rendered into the alpha channel of $\mathbf{s}$ that was first cleared to 1. If pixels pass the depth test, a value of 0 is written. Finally, refractive objects are rendered fully, and in the pixel shader $\mathbf{s}$ is sampled according to pixel position on the screen with a perturbed offset coming from,

Figure 14.38. Transparent glasses at the bottom of the image features roughness-based background scattering. Elements behind the glass appear more or less blurred, simulating the spreading of refracted rays. *(Image courtesy of Frostbite, © 2018 Electronic Arts Inc.)*

for instance, the scaled surface normal tangent xy-components, simulating refraction. In this context, the color of the perturbed samples is taken into account only if $\alpha = 0$. This test is done to avoid using samples from surfaces that are in front of the refractive object and so having their colors pulled in as if they were behind it. Note that instead of setting $\alpha = 0$, the scene depth map could be used to compare the pixel shader depth against the perturbed scene sample's depth [294]. If the center pixel is farther away, the offset sample is closer; it is then ignored and replaced by the regular scene sample as if there was no refraction.

These techniques give the impression of refraction, but bear little resemblance to physical reality. The ray gets redirected when it enters the transparent solid, but the ray is never bent a second time, when it is supposed to leave the object. This exit interface never comes into play. This flaw sometimes does not matter, because human eyes are forgiving for what the correct appearance should be [1185].

Many games feature refraction through a single layer. For rough refractive surfaces, it is important to blur the background according to material roughness, to simulate the spreading of refracted ray directions caused by the distribution of microgeometry normals. In the game *DOOM* (2016) [1682], the scene is first rendered as usual. It is then downsampled to half resolution and further down to four mipmap levels. Each mipmap level is downsampled according to a Gaussian blur mimicking a GGX BRDF lobe. In the final step, refracting meshes are rendered over the full-resolution scene [294]. The background is composited behind the surfaces by sampling the scene's mipmapped texture and mapping the material roughness to the mipmap level. The rougher the surface, the blurrier the background. Using a general material representation, the same approach is proposed by Drobot [386]. A similar technique is also used within a unified transparency framework from McGuire and Mara [1185]. In this case, a Gaussian point-spread function is used to sample the background in a single pass. See Figure 14.38.