We can write

$$L(\mathbf{p}) = \sum_i L_i(\mathbf{p})\mathbf{w}_i, \tag{11.36}$$

where $L(\mathbf{p})$ is the final radiance at point $\mathbf{p}$, $L_i(\mathbf{p})$ is the precomputed unit contribution from screen i, and $\mathbf{w}_i$ is its current brightness. This equation defines a *vector space* in the mathematical sense, with L_i being the basis vectors for this space. Any possible lighting can be created by a linear combination of the lights' contributions.

The original PRT paper by Sloan et al. [1651] uses the same reasoning, but in the context of an infinitely distant lighting environment represented using spherical harmonics. Instead of storing how the scene responds to a monitor screen, they store how it responds to surrounding light with a distribution defined by spherical harmonic basis functions. By doing so for some number of SH bands, they can render a scene illuminated by an arbitrary lighting environment. They project this lighting onto spherical harmonics, multiply each resulting coefficient by its respective normalized "unit" contribution, and add these all together, just as we did with the monitors.

Note that the choice of the basis used to "inject" light into the scene is independent of the representation used to express the final lighting. For example, we can describe how the scene is illuminated using spherical harmonics, but pick another basis to store how much radiance arrives at any given point. Say we used an ambient cube for storage. We would calculate how much radiance arrives from the top and how much from the sides. The transfer for each of these directions would be stored separately, instead of as a single scalar value representing the total transfer.

The PRT paper by Sloan et al. [1651] analyzes two cases. The first is when the receiver basis is just a scalar irradiance value of the surface. For this, the receiver needs to be a fully diffuse surface, with a predetermined normal, which means that it cannot use normal maps for fine-scale details. The transfer function takes the form of a dot product between the SH projection of the input lighting and the precomputed *transfer vector*, which varies spatially across the scene.

If we need to render non-Lambertian materials, or allow for normal mapping, we can use the second variation presented. In this case, the SH projection of surrounding lighting is transformed to an SH projection of the incident radiance for a given point. Because this operation provides us the full radiance distribution over the sphere (or hemisphere, if we are dealing with a static opaque object), we can properly convolve it with any BRDF. The transfer function maps SH vectors to other SH vectors, and has a form of matrix multiplication. This multiply operation is expensive, both computationally and in terms of memory. If we use third-order SH for both source and receiver, we need to store a 9×9 matrix for every point in the scene, and these data are for just a monochrome transfer. If we want colors, we need three such matrices—an incredible amount of memory per point.

This problem was addressed by Sloan et al. [1652] one year later. Instead of storing the transfer vectors or matrices directly, their whole set is analyzed using a *principal component analysis* (PCA) technique. The transfer coefficients can be considered points in multi-dimensional space (for example, 81-dimensional in the case of 9×9

matrices), but their sets are not uniformly distributed in that space. They form clusters with lower dimensionality. This clustering is just like how three-dimensional points distributed along a line are, effectively, all in a one-dimensional subspace of a three-dimensional space. PCA can efficiently detect such statistical relationships. Once a subspace is discovered, points can be represented using a much lower number of coordinates, as we can store the position in the subspace with a reduced number of dimensions. Using the line analogy, instead of storing the full position of a point using three coordinates, we could just store its distance along the line. Sloan et al. use this method to reduce the dimensionality of the transfer matrices from 625 dimensions (25×25 transfer matrices) to 256 dimensions. While this is still too high for typical real-time applications, many of the later light transport algorithms have adapted PCA as a way of compressing the data.

This type of dimensionality reduction is inherently lossy. In rare cases the data forms a perfect subspace, but most often it is approximate, so projecting data onto it causes some degradation. To increase quality, Sloan et al. divide the set of transfer matrices into clusters and perform PCA on each separately. The process also includes an optimization step that ensures no discontinuities on the cluster boundaries. An extension that allows for limited deformations of the objects is also presented, called *local deformable precomputed radiance transfer* (LDPRT) [1653].

PRT has been used in various forms in several games. It is especially popular in titles in which gameplay focuses on outdoor areas where time of day and weather conditions change dynamically. *Far Cry 3* and *Far Cry 4* use PRT where the source basis is second-order SH and the receiver basis is a custom, four-direction basis [533, 1154]. *Assassin's Creed 4: Black Flag* uses one basis function as a source (sun color), but precomputes the transfer for different times of the day. This representation can be interpreted as having the source basis functions defined over the time dimension instead of directions. The receiver basis is the same as that used in the *Far Cry* titles.

The SIGGRAPH 2005 course [870] on precomputed radiance transfer provides a good overview of research in the area. Lehtinen [1019, 1020] gives a mathematical framework that can be used to analyze the differences between various algorithms and to develop new ones.

The original PRT method assumes infinitely distant surrounding lighting. While this models lighting of an outdoor scene fairly well, it is too restrictive for indoor environments. However, as we noted earlier, the concept is fully agnostic to the initial source of the illumination. Kristensen et al. [941] describe a method where PRT is computed for a set of lights scattered across the entire scene. This corresponds to having a large number of "source" basis functions. The lights are next combined into clusters, and receiving geometry is segmented into zones, with each zone affected by a different subset of lights. This process results in a significant compression of the transfer data. At runtime, the illumination resulting from arbitrarily placed lights is approximated by interpolating data from the closest lights in the precomputed set. Gilabert and Stefanov [533] use the method to generate indirect illumination in the game *Far Cry 3*. This method in its basic form can handle only point lights. While

Figure 11.26. *Enlighten* by Geomerics can generate global illumination effects in real time. The image shows an example of its integration with the *Unity* engine. The user can freely change the time of day as well as turn lights on and off. All the indirect illumination is updated accordingly in real time. *(Courtyard demo © Unity Technologies, 2015.)*

it could be extended to support other types, the cost grows exponentially with the number of degrees of freedom for each light.

The PRT techniques discussed up to this point precompute the transfer from some number of elements, which are then used to model the lights. Another popular class of methods precomputes transfer between surfaces. In this type of system, the actual source of the illumination becomes irrelevant. Any light source can be used, because the input to these methods is the outgoing radiance from some set of surfaces (or some other related quantity, such as irradiance, if the method assumes diffuse-only surfaces). These direct-illumination calculations can use shadowing (Chapter 7), irradiance environment maps (Section 10.6), or the ambient and directional occlusion methods discussed earlier in this chapter. Any surface can also trivially be made emissive by setting its outgoing radiance to a desired value, turning it into an area light source.

The most popular system that works according to these principles is *Enlighten* by Geomerics (Figure 11.26). While the exact details of the algorithm have never been fully publicly revealed, numerous talks and presentations give an accurate picture of this system's principles [315, 1101, 1131, 1435].

The scene is assumed to be Lambertian, but only for the purpose of light transfer. Using Heckbert's notation, the set of paths handled is $LD*(D|S)E$, as the last surface before the eye does not need to be diffuse-only. The system defines a set of "source" elements and another set of "receiver" elements. Source elements exist on surfaces, and share some of their properties, such as diffuse color and normal. The

preprocessing step computes how the light is transferred between the source elements and the receivers. The exact form of this information depends on what the source elements are and what basis is used to gather lighting at the receivers. In the simplest form, the source elements can be points, and we are then interested in generating irradiance in the receiving locations. In this case, the transfer coefficient is just the mutual visibility between source and receiver. At runtime, the outgoing radiance for all source elements is provided to the system. From this information we can numerically integrate the reflectance equation (Equation 11.1), using the precomputed visibility and the known information about the position and orientation of the source and receiver. In this way, a single bounce of light is performed. As the majority of the indirect illumination comes from this first bounce, performing one bounce alone is enough to provide plausible illumination. However, we can use this light and run the propagation step again to generate a second bounce of light. This is usually done over the course of several frames, where the output of one frame is used as an input for the next.

Using points as source elements would result in a large number of connections. To improve performance, clusters of points representing areas of similar normal and color can also be used as source sets. In this case the transfer coefficients are the same as the form factors seen in radiosity algorithms (Section 11.2.1). Note that, despite the similarity, the algorithm is different from classical radiosity, as it computes only one bounce of light at a time and does not involve solving a system of linear equations. It draws upon the idea of *progressive radiosity* [275, 1642]. In this system a single patch can determine how much energy it receives from other patches, in an iterative process. The process of transferring the radiance to the receiving location is called *gathering*.

The radiance at the receiving elements can be gathered in different forms. The transfer to the receiving elements may use any of the directional bases that we described before. In this case the single coefficient becomes a vector of values, with the dimensionality equal to the number of functions in the receiving basis. When performing gathering using a directional representation, the result is the same as for the offline solutions described in Section 11.5.2, so it can be used with normal mapping or to provide a low-gloss specular response.

The same general idea is used in many variants. To save memory, Sugden and Iwanicki [1721] use SH transfer coefficients, quantize them, and store them indirectly as an index to an entry in a palette. Jendersie et al. [820] build a hierarchy of source patches, and store references to higher elements in this tree when the solid angle subtended by the children is too small. Stefanov [1694] introduces an intermediate step, where radiance from surface elements is first propagated to a voxelized representation of the scene, one that later acts as a source for the transport.

The (in some sense) ideal split of surfaces into source patches depends on the receiver's position. For distant elements, considering them as separate entities generates unnecessary storage costs, but they should be treated individually when viewed up close. A hierarchy of source patches mitigates this problem to some extent, but does not solve it entirely. Certain patches that could be combined for particular re-

ceivers may be far enough apart to prevent such merging. A novel approach to the problem is presented by Silvennoinen and Lehtinen [1644]. Their method does not create the source patches explicitly, but rather generates a different set of them for each receiving position. The objects are rendered to a sparse set of environment maps scattered around the scene. Each map is projected to spherical harmonics, and this low-frequency version is "virtually" projected back onto the environment. Receiving points record how much of this projection they can see, and this process is done for each of the senders' SH basis functions separately. Doing so creates a different set of source elements for every receiver, based on the visibility information from both environment probes and receiver points.

Because the source basis is generated from an environment map projected to an SH, it naturally combines surfaces that are farther away. To pick the probes to use, the receivers use a heuristic that favors nearby ones, which makes the receivers "see" the environment at a similar scale. To limit the amount of data that has to be stored, the transfer information is compressed with clustered PCA.

Another form of precomputed transfer is described by Lehtinen et al. [1021]. In this approach neither the source nor the receiving elements exist on the meshes, but rather are volumetric and can be queried at any location in three-dimensional space. This form is convenient for providing lighting consistency between static and dynamic geometry, but the method is fairly expensive computationally.

Loos et al. [1073] precompute transfer within a modular, unit cell, with different configurations of the side walls. Multiple such cells are then stitched and warped to approximate the geometry of the scene. The radiance is propagated first to the cell borders, which act as interfaces, and then to the neighboring cells using the precomputed models. The method is fast enough to efficiently run even on mobile platforms, but the resulting quality might not be sufficient for more demanding applications.

11.5.4 Storage Methods

Regardless of whether we want to use full precomputed lighting or to precalculate the transfer information and allow for some changes in the lighting, the resulting data has to be stored in some form. GPU-friendly formats are a must.

Light maps are one of the most common way of storing precomputed lighting. These are textures that store the precalculated information. Though sometimes terms such as *irradiance map* are used to denote a specific type of data stored, the term *light maps* is used to collectively describe all of these. At runtime, the GPU's built-in texture mechanisms are used. Values are usually bilinearly filtered, which for some representations might not be entirely correct. For example, when using an AHD representation, the filtered D (direction) component will no longer be a unit length after interpolation and so needs to be renormalized. Using interpolation also means that the A (ambient) and H (highlight) are not exactly what they would be if we computed them directly at the sampling point. That said, the results usually look acceptable, even if the representation is nonlinear.

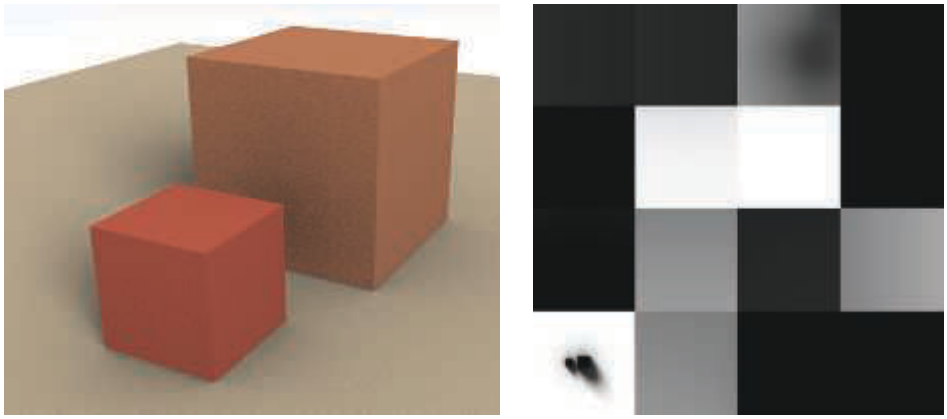

Figure 11.27. Light baked into a scene, along with the light maps applied to the surfaces. Light mapping uses a unique parameterization. The scene is divided into elements that are flattened and packed into a common texture. For example, the section in the lower left corresponds the ground plane, showing the two shadows of the cubes. *(From the three.js example webgl_materials_lightmap [218].)*

In most cases light maps do not use mipmapping, which is normally not needed because the resolution of the light map is small compared to typical albedo maps or normal maps. Even in high-quality applications, a single light-map texel covers the area of at least roughly 20×20 centimeters, often more. With texels of this size, extra mip levels would almost never be needed.

To store lighting in the texture, objects need to provide an *unique parameterization*. When mapping a diffuse color texture onto a model, it is usually fine for different parts of the mesh to use the same areas of the texture, especially if a model is textured with a general repeating pattern. Reusing light maps is difficult at best. Lighting is unique for every point on the mesh, so every triangle needs to occupy its own, unique area on the light map. The process of creating a parameterization starts with splitting the meshes into smaller chunks. This can be done either automatically, using some heuristics [1036], or by hand, in the authoring tool. Often, the split that is already present for the mapping of other textures is used. Next, each chunk is parameterized independently, ensuring that its parts do not overlap in texture space [1057, 1617]. The resulting elements in texture space are called *charts* or *shells*. Finally, all the charts are packed into a common texture (Figure 11.27). Care must be taken to ensure that not only do charts not overlap, but also their filtering footprints must stay separate. All the texels that can be accessed when rendering a given chart (bilinear filtering accesses four neighboring texels) should be marked as used, so no other chart overlaps them. Otherwise, bleeding between the charts might occur, and lighting from one of them might be visible on the other. Although it is fairly common for light mapping systems to provide a user-controlled "gutter" amount for spacing between the light map charts, this separation is not necessary. The correct filtering footprint of a chart

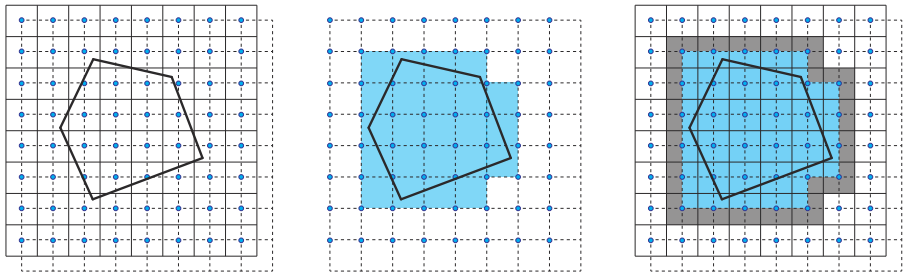

Figure 11.28. To accurately determine the filtering footprint of a chart, we need to find all texels that can be accessed during rendering. If a chart intersects a square spanned between the centers of four neighboring texels, all of them will be used during bilinear filtering. The texel grid is marked with solid lines, texel centers with blue dots, and the chart to be rasterized with thick, solid lines (left). We first conservatively rasterize the chart to a grid shifted by half of a texel size, marked with dashed lined (center). Any texel touching a marked cell is considered occupied (right).

can be determined automatically by rasterizing it in light-map space, using a special set of rules. See Figure 11.28. If shells rasterized this way do not overlap, we are guaranteed that no bleeding will happen.

Avoiding bleeding is another reason why mipmapping is rarely used for light maps. Chart filtering footprints would need to stay separate on all the mip levels, which would lead to excessively large spacing between shells.

Optimally packing charts into textures is an NP-complete problem, which means that there are no known algorithms that can generate an ideal solution with polynomial complexity. As real-time applications may have hundreds of thousands charts in a single texture, all real-world solutions use fine-tuned heuristics and carefully optimized code to generate packing quickly [183, 233, 1036]. If the light map is later block-compressed (Section 6.2.6), to improve the compression quality additional constraints might be added to the packer to ensure that a single block contains only similar values.

A common problem with light maps are *seams* (Figure 11.29). Because the meshes are split into charts and each of these is parameterized independently, it is impossible to ensure that the lighting along split edges is exactly the same on both sides. This manifests as a visual discontinuity. If the meshes are split manually, this problem can be avoided somewhat by splitting them in areas that are not directly visible. Doing so, however, is a laborious process, and cannot be applied when generating the parameterizations automatically. Iwanicki [806] performs a post-process on the final light map that modifies the texels along the split edges to minimize the difference between the interpolated values on both sides. Liu and Ferguson et al. [1058] enforce interpolated values matching along the edge via an equality constraint and solve for texel values that best preserve smoothness. Another approach is to take this constraint into account when creating parameterization and packing charts. Ray et al. [1467] show how *grid-preserving parameterization* can be used to create light maps that do not suffer from seam artifacts.

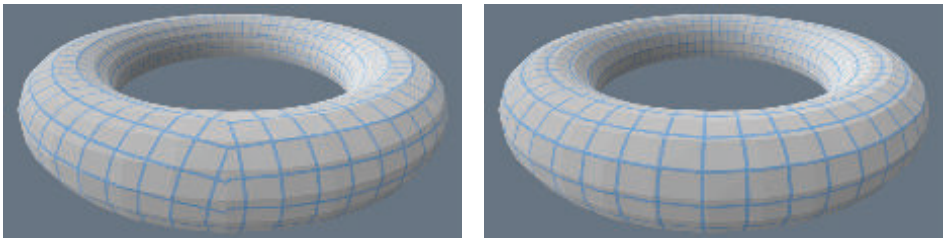

Figure 11.29. To create a unique parameterization for a torus, it needs to be cut and unwrapped. The torus on the left uses a simple mapping, created without considering how the cuts are positioned in texture space. Notice the discontinuities of the grid representing texels on the left. Using more advanced algorithms, we can create a parameterization that ensures that the texel grid lines stay continuous on the three-dimensional mesh, as on the right. Such unwrapping methods are perfect for light mapping, as the resulting lighting does not exhibit any discontinuities.

Precomputed lighting can also be stored at vertices of the meshes. The drawback is that the quality of lighting depends on how finely the mesh is tessellated. Because this decision is usually made at an early stage of authoring, it is hard to ensure that there are enough vertices on the mesh to look good in all expected lighting situations. In addition, tessellation can be expensive. If the mesh is finely tessellated, the lighting signal will be oversampled. If directional methods of storing the lighting are used, the whole representation needs to be interpolated between the vertices by the GPU and passed to the pixel shader stage to perform the lighting calculations. Passing so many parameters between vertex and pixels shaders is fairly uncommon, and generates workloads for which modern GPUs are not optimized, which causes inefficiencies and lower performance. For all these reasons, storing precomputed lighting on vertices is rarely used.

Even though information about the incoming radiance is needed on the surfaces (except when doing volumetric rendering, discussed in Chapter 14), we can precompute and store it volumetrically. Doing so, lighting can be queried at an arbitrary point in space, providing illumination for objects that were not present in the scene during the precomputation phase. Note, however, that these objects will not correctly reflect or occlude lighting.

Greger et al. [594] present the *irradiance volume*, which represents the five-dimensional (three spatial and two directional) irradiance function with a sparse spatial sampling of irradiance environment maps. That is, there is a three-dimensional grid in space, and at each grid point is an irradiance environment map. Dynamic objects interpolate irradiance values from the closest maps. Greger et al. use a two-level adaptive grid for the spatial sampling, but other volume data structures, such as octrees [1304, 1305], can be used.

In the original irradiance volume, Greger et al. stored irradiance at each sample point in a small texture, but this representation cannot be filtered efficiently on the GPU. Today, volumetric lighting data are most often stored in three-dimensional

textures, so sampling the volume can use the GPU's accelerated filtering. The most common representations for the irradiance function at the sample points include:

- Spherical harmonics (SH) of second- and third-order, with the former being more common, as the four coefficients needed for a single color channel conveniently pack into four channels of typical texture format.
- Spherical Gaussians.
- Ambient cube or ambient dice.

The AHD encoding, even though technically capable of representing spherical irradiance, generates distracting artifacts. If SH is used, spherical harmonic gradients [54] can further improve quality. All of the above representations have been successfully used in many games [766, 808, 1193, 1268, 1643].

Evans [444] describes a trick used for the irradiance volumes in *LittleBigPlanet*. Instead of a full irradiance map representation, an average irradiance is stored at each point. An approximate directionality factor is computed from the gradient of the irradiance field, i.e., the direction in which the field changes most rapidly. Instead of computing the gradient explicitly, the dot product between the gradient and the surface normal $\mathbf{n}$ is computed by taking two samples of the irradiance field, one at the surface point $\mathbf{p}$ and another at a point displaced slightly in the direction of $\mathbf{n}$, and subtracting one from the other. This approximate representation is motivated by the fact that the irradiance volumes in *LittleBigPlanet* are computed dynamically.

Irradiance volumes can also be used to provide lighting for static surfaces. Doing so has the advantage of not having to provide a separate parameterization for the light map. The technique also does not generate seams. Both static and dynamic objects can use the same representation, making lighting consistent between the two types of geometry. Volumetric representations are convenient for use in deferred shading (Section 20.1), where all lighting can be performed in a single pass. The main drawback is memory consumption. The amount of memory used by light maps grows with the square of their resolution; for a regular volumetric structure it grows with the cube. For this reason, considerably lower resolutions are used for grid volume representations. Adaptive, hierarchical forms of lighting volumes have better characteristics, but they still store more data than light maps. They are also slower than a grid with regular spacing, as the extra indirections create load dependencies in the shader code, which can result in stalls and slower execution.

Storing surface lighting in volumetric structures is somewhat tricky. Multiple surfaces, sometimes with vastly different lighting characteristics, can occupy the same voxel, making it unclear what data should be stored. When sampling from such voxels, the lighting is frequently incorrect. This happens particularly often near the walls between brightly lit outdoors and dark indoors, and results in either dark patches outside or bright ones inside. The remedy for this is to make the voxel size small enough to never straddle such boundaries, but this is usually impractical because of

Figure 11.30. *Unity* engine uses a tetrahedral mesh to interpolate lighting from a set of probes. *(Book of the Dead © Unity Technologies, 2018.)*

the amount of data needed. The most common ways of dealing with the problem are shifting the sampling position along the normal by some amount, or tweaking the trilinear blending weights used during interpolation. This is often imperfect and manual tweaks to the geometry to mask the problem might be needed. Hooker [766] adds extra clipping planes to the irradiance volumes, which limit their influence to the inside of a convex polytope. Kontkanen and Laine [926] discuss various strategies to minimize bleeding.

The volumetric structure that holds the lighting does not have to be regular. One popular option is to store it in an irregular cloud of points that are then connected to form a Delaunay tetrahedralization (Figure 11.30). This approach was popularized by Cupisz [316]. To look up the lighting, we first find the tetrahedron the sampling position is in. This is an iterative process and can be somewhat expensive. We traverse the mesh, moving between neighboring cells. Barycentric coordinates of the lookup point with respect to the current tetrahedron corners are used to pick the neighbor to visit in the next step (Figure 11.31). Because typical scenes can contain thousands of positions in which the lighting is stored, this process can potentially be time consuming. To accelerate it, we can record a tetrahedron used for lookup in the previous frame (when possible) or use a simple volumetric data structure that provides a good "starting tetrahedron" for an arbitrary point in the scene.

Once the proper tetrahedron is located, the lighting stored on its corners is interpolated, using the already-available barycentric coordinates. This operation is not accelerated by the GPU, but it requires just four values for interpolation, instead of the eight needed for trilinear interpolation on a grid.

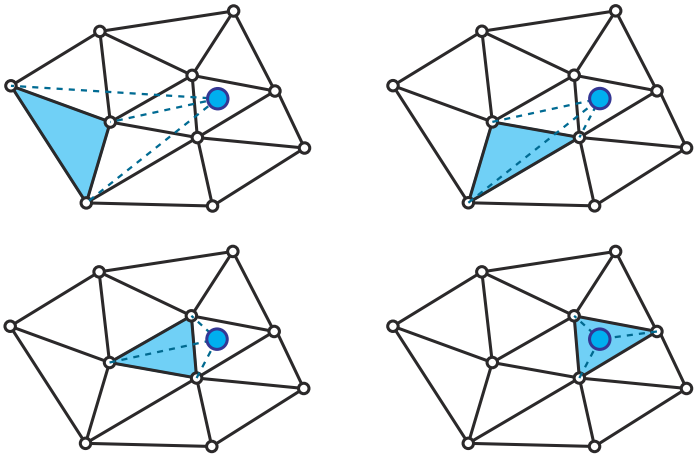

Figure 11.31. Process of a lookup in a tetrahedral mesh illustrated in two dimensions. Steps are shown left to right, top to bottom. Given some starting cell (marked in blue), we evaluate the barycentric coordinates of the lookup point (blue dot) with respect to the cell's corners. In the next step, we move toward the neighbor across the edge opposite to the corner with the most negative coordinate.

The positions for which the lighting gets precomputed and stored can be placed manually [134, 316] or automatically [809, 1812]. They are often referred to as *lighting probes*, or *light probes*, as they probe (or sample) the lighting signal. The term should not be confused with a "light probe" (Section 10.4.2), which is the distant lighting recorded in an environment map.

The quality of the lighting sampled from a tetrahedral mesh is highly dependent upon the structure of that mesh, not just the overall density of the probes. If they are distributed nonuniformly, the resulting mesh can contain thin, elongated tetrahedrons that generate visual artifacts. If probes are placed by hand, problems can be easily corrected, but it is still a manual process. The structure of the tetrahedrons is not related to the structure of the scene geometry, so if not properly handled, lighting will be interpolated across the walls and generate bleeding artifacts, just as with irradiance volumes. In the case of manual probe placement, users can be required to insert additional probes to stop this from happening. When automated placement of probes is used, some form of visibility information can be added to the probes or tetrahedrons to limit their influence to only relevant areas [809, 1184, 1812].

It is a common practice to use different lighting storage methods for static and dynamic geometry. For example, static meshes could use light maps, while dynamic objects could get lighting information from volumetric structures. While popular, this scheme can create inconsistencies between the looks of different types of geometry. Some of these differences can be eliminated by regularization, where lighting information is averaged across the representations.

When baking the lighting, care is needed to compute its values only where they are truly valid. Meshes are often imperfect. Some vertices may be placed inside geometry, or parts of the mesh may self-intersect. The results will be incorrect if we compute incident radiance in such flawed locations. They will cause unwanted darkening or bleeding of incorrectly unshadowed lighting. Kontkanen and Laine [926] and Iwanicki and Sloan [809] discuss different heuristics that can be used to discard invalid samples.

Ambient and directional occlusion signals share many of the spatial characteristics of diffuse lighting. As mentioned in Section 11.3.4, all of the above methods can be used for storing them as well.

11.5.5 Dynamic Diffuse Global Illumination

Even though precomputed lighting can produce impressive-looking results, its main strength is also its main weakness—it requires precomputation. Such offline processes can be lengthy. It is not uncommon for lighting bakes to take many hours for typical game levels. Because lighting computations take so long, artists are usually forced to work on multiple levels at the same time, to avoid downtime while waiting for the bakes to finish. This, in turn, often results in an excessive load on the resources used for rendering, and causes the bakes to take even longer. This cycle can severely impact productivity and cause frustration. In some cases, it is not even possible to precompute the lighting, as the geometry changes at runtime or is created to some extent by the user.

Several methods have been developed to simulate global illumination in dynamic environments. Either they do not require any preprocessing, or the preparation stage is fast enough to be executed every frame.

One of the earliest methods of simulating global illumination in fully dynamic environments was based on "Instant Radiosity" [879]. Despite the name, the method has little in common with the radiosity algorithm. In it, rays are cast outward from the light sources. For each location where a ray hits, a light is placed, representing the indirect illumination from that surface element. These sources are called *virtual point lights* (VPLs). Drawing upon this idea, Tabellion and Lamorlette [1734] developed a method used during production of *Shrek 2* that performs a direct lighting pass of scene surfaces and stores the results in textures. Then, during rendering, the method traces rays and uses the cached lighting to create one-bounce indirect illumination. Tabellion and Lamorlette show that, in many cases, a single bounce is enough to create believable results. This was an offline method, but it inspired a method by Dachsbacher and Stamminger [321] called *reflective shadow maps* (RSM).

Similar to regular shadow maps (Section 7.4), reflective shadow maps are rendered from the point of view of the light. Beyond just depth, they store other information about visible surfaces, such as their albedo, normal, and direct illumination (flux). When performing the final shading, texels of the RSM are treated as point lights to provide a single bounce of indirect lighting. Because a typical RSM contains hundreds of thousands of pixels, only a subset of these are chosen, using an importance-driven

Figure 11.32. The game *Uncharted 4* uses reflective shadow maps to provide indirect illumination from the player's flashlight. The image on the left shows the scene without the indirect contribution. The image on the right has it enabled. The insets show a close-up of the frame rendered without (top) and with (bottom) temporal filtering enabled. It is used to increase the effective number of VPLs that are used for each image pixel. (*UNCHARTED 4 A Thief's End* ©/^{TM} *2016 SIE. Created and developed by Naughty Dog LLC.)*

heuristic. Dachsbacher and Stamminger [322] later show how the method can be optimized by reversing the process. Instead of picking the relevant texels from the RSM for every shaded point, some number of lights are created based on the entire RSM and splatted (Section 13.9) in screen space.

The main drawback of the method is that it does not provide occlusion for the indirect illumination. While this is a significant approximation, results look plausible and are acceptable for many applications.

To achieve high-quality results and maintain temporal stability during light movement, a large number of indirect lights need to be created. If too few are created, they tend to rapidly change their positions when the RSM is regenerated, and cause flickering artifacts. On the other hand, having too many indirect lights is challenging from a performance perspective. Xu [1938] describes how the method was implemented in the game *Uncharted 4*. To stay within the performance constraints, he uses a small number (16) of lights per pixel, but cycles through different sets of them over several frames and filters the results temporally (Figure 11.32).

Different methods have been proposed to address the lack of indirect occlusion. Laine et al. [962] use a dual-paraboloid shadow map for the indirect lights, but add them to the scene incrementally, so in any single frame only a handful of the shadow maps are rendered. Ritschel et al. [1498] use a simplified, point-based representation of the scene to render a large number of *imperfect shadow maps*. Such maps are small and contain many defects when used directly, but after simple filtering provide enough fidelity to deliver proper occlusion effects for indirect illumination.

Some games have used methods that are related to these solutions. *Dust 514* renders a top-down view of the world, with up to four independent layers when required [1110]. These resulting textures are used to perform gathering of indirect illumination, much like the method by Tabellion and Lamorlette. A similar method is used to provide indirect illumination from the terrain in the *Kite* demo, showcasing the Unreal Engine [60].

11.5.6 Light Propagation Volumes

Radiative transfer theory is a general way of modeling how electromagnetic radiation is propagated through a medium. It accounts for scattering, emission, and absorption. Even though real-time graphics strives to show all these effects, except for the simplest cases the methods used for these simulations are orders of magnitude too costly to be directly applied in rendering. However, some of the techniques used in the field have proved useful in real-time graphics.

Light propagation volumes (LPV), introduced by Kaplanyan [854], draw inspiration from the *discrete ordinate methods* in radiative transfer. In his method, the scene is discretized into a regular grid of three-dimensional cells. Each cell will hold a directional distribution of radiance flowing through it. He uses second-order spherical harmonics for these data. In the first step, lighting is injected to the cells that contain directly lit surfaces. Reflective shadow maps are accessed to find these cells, but any other method could be used as well. The injected lighting is the radiance reflected off the lit surfaces. As such, it forms a distribution around the normal, facing away from the surface, and gets its color from the material's color. Next, the lighting is propagated. Each cell analyzes the radiance fields of its neighbors. It then modifies its own distribution to account for the radiance arriving from all the directions. In a single step radiance gets propagated over a distance of only a single cell. Multiple iterations are required to distribute it further (Figure 11.33).

The important advantage of this method is that it generates a full radiance field for each cell. This means that we can use an arbitrary BRDF for the shading, even

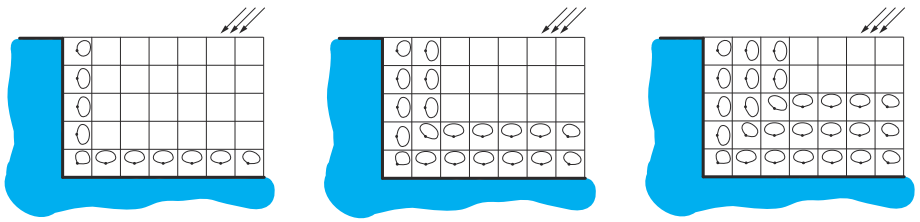

Figure 11.33. Three steps of the propagation of the light distribution through a volumetric grid. The left image shows the distribution of the lighting reflected from the geometry illuminated by a directional light source. Notice that only cells directly adjacent to the geometry have a nonzero distribution. In the subsequent steps, light from neighboring cells is gathered and propagated through the grid.

though the quality of the reflections for a glossy BRDF will be fairly low when using second-order spherical harmonics. Kaplanyan shows examples with both diffuse and reflective surfaces.

To allow for the propagation of light over larger distances, as well as to increase the area covered by the volume, while keeping memory usage reasonable, a *cascaded* variant of the method was developed by Kaplanyan and Dachsbacher [855]. Instead of using a single volume with cells of uniform size, they use a set of these with progressively larger cells, nested in one another. Lighting is injected into all the levels and propagated independently. During the lookup, they select the most detailed level available for a given position.

The original implementation did not account for any occlusion of the indirect lighting. The revised approach uses the depth information from the reflective shadow map, as well as the depth buffer from the camera's position, to add information about the light blockers to the volumes. This information is incomplete, but the scene could also be voxelized during the preprocess and so use a more precise representation.

The method shares the problems of other volumetric approaches, the greatest of which is bleeding. Unfortunately, increasing the grid resolution to fix it causes other problems. When using a smaller cell size, more iterations are required to propagate light over the same world-space distance, making the method significantly more costly. Finding a balance between the resolution of the grid and performance is not trivial. The method also suffers from aliasing problems. Limited resolution of the grid, combined with the coarse directional representation of the radiance, causes degradation of the signal as it moves between the neighboring cells. Spatial artifacts, such as diagonal streaks, might appear in the solution after multiple iterations. Some of these problems can be removed by performing spatial filtering after the propagation pass.

11.5.7 Voxel-Based Methods

Introduced by Crassin [304], *voxel cone tracing global illumination* (VXGI) is also based on a voxelized scene representation. The geometry itself is stored in the form of a *sparse voxel octree*, described in Section 13.10. The key concept is that this structure provides a mipmap-like representation of the scene, so that a volume of space can be rapidly tested for occlusion, for example. Voxels also contain information about the amount of light reflected off the geometry they represent. It is stored in a directional form, as the radiance is reflected in six major directions. Using reflective shadow maps, the direct lighting is injected to the lowest levels of the octree first. It is then propagated up the hierarchy.

The octree is used for estimation of the incident radiance. Ideally, we would trace a ray to get an estimate of the radiance coming from a particular direction. However, doing so requires many rays, so whole bundles of these are instead approximated with a cone traced in their average direction, returning just a single value. Exactly testing the cone for intersections with an octree is not trivial, so this operation is approximated with a series of lookups into the tree along the cone's axis. Each lookup reads the

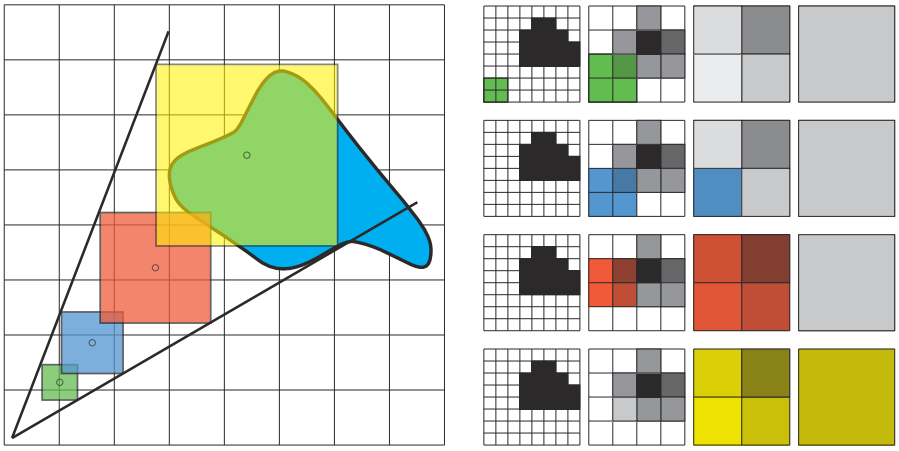

Figure 11.34. Voxel cone tracing approximates an exact cone trace with a series of filtered lookups into a voxel tree. The left side shows a two-dimensional analog of a three-dimensional trace. Hierarchical representation of the voxelized geometry is shown on the right, with each column showing an increasingly coarser level of the tree. Each row shows the nodes of the hierarchy used to provide coverage for a given sample. The levels used are picked so that the size of a node in the coarser level is larger than the lookup size, and in the finer level smaller. A process analogous to trilinear filtering is used to interpolate between these two chosen levels.

level of the tree with a node size corresponding to the cross section of a cone at the given point. The lookup provides the filtered radiance reflected in the direction of the cone origin, as well as the percentage of the lookup footprint that is occupied by geometry. This information is used to attenuate the lighting from subsequent points, in a fashion similar to alpha blending. The occlusion of the entire cone is tracked. In each step it is reduced to account for the percentage of the current sample occupied by geometry. When accumulating the radiance, it is first multiplied by the combined occlusion factor (Figure 11.34). This strategy cannot detect full occlusions that are a result of multiple partial ones, but the results are believable nonetheless.

To compute diffuse lighting, a number of cones are traced. How many are generated and cast is a compromise between performance and precision. Tracing more cones provides higher-quality results, at the cost of more time spent. It is assumed that the cosine term is constant over the entire cone, so that this term can be factored out of the reflectance equation integral. Doing so makes the calculation of the diffuse lighting as simple as computing a weighted sum of the values returned by cone traces.

The method was implemented in a prototype version of the Unreal Engine, as described by Mittring [1229]. He gives several optimizations that the developers needed to make it run as a part of a full rendering pipeline. These improvements include performing the traces at a lower resolution and distributing the cones spatially. This process was done so that each pixel traces only a single cone. The full radiance for the diffuse response is obtained by filtering the results in screen space.

A major problem with using a sparse octree for storing lighting is the high lookup cost. Finding the leaf node containing the given location corresponds to a series of memory lookups, interleaved with a simple logic that determines which subtree to traverse. A typical memory read can take in the order of a couple hundred cycles. GPUs try to hide this latency by executing multiple groups of shader threads (warps or wavefronts) in parallel (Chapter 3). Even though only one group performs ALU operations at any given time, when it needs to wait for a memory read, another group takes its place. The number of warps that can be active at the same time is determined by different factors, but all of them are related to the amount of resources a single group uses (Section 23.3). When traversing hierarchical data structures, most of the time is spent waiting for the next node to be fetched from memory. However, other warps that will be executed during this wait will most likely also perform a memory read. Since there is little ALU work compared to the number of memory accesses, and because the total number of warps in flight is limited, situations where all groups are waiting for memory and no actual work is executed are common.

Having large numbers of stalled warps gives suboptimal performance, and methods that try to mitigate these inefficiencies have been developed. McLaren [1190] replaces the octree with a cascaded set of three-dimensional textures, much like cascaded light propagation volumes [855] (Section 11.5.6). They have the same dimensions, but cover progressively larger areas. In this way, reading the data is accomplished with just a regular texture lookup—no dependent reads are necessary. Data stored in the textures are the same as in the sparse voxel octree. They contain the albedo, occupancy, and bounced lighting information in six directions. Because the position of the cascades changes with the camera movement, objects constantly go in and out of the high-resolution regions. Due to memory constraints, it is not possible to keep these voxelized versions resident all the time, so they are voxelized on demand, when needed. McLaren also describes a number of optimizations that made the technique viable for a 30 FPS game, *The Tomorrow Children* (Figure 11.35).

11.5.8 Screen-Space Methods

Just like screen-space ambient occlusion (Section 11.3.6), some diffuse global illumination effects can be simulated using only surface values stored at screen locations [1499]. These methods are not as popular as SSAO, mainly because the artifacts resulting from the limited amount of data available are more pronounced. Effects such as color bleeding are a result of a strong direct light illuminating large areas of fairly constant color. Surfaces such as this are often impossible to entirely fit in the view. This condition makes the amount of bounced light strongly depend on the current framing, and fluctuate with camera movement. For this reason, screen-space methods are used only to augment some other solution at a fine scale, beyond the resolution achievable by the primary algorithm. This type of system is used in the game *Quantum Break* [1643]. Irradiance volumes are used to model large-scale global illumination effects, and a screen-space solution provides bounced light for limited distances.

Figure 11.35. The game *The Tomorrow Children* uses voxel cone tracing to render indirect illumination effects. *(© 2016 Sony Interactive Entertainment Inc. The Tomorrow Children is a trademark of Sony Interactive Entertainment America LLC.)*

11.5.9 Other Methods

Bunnell's method for calculating ambient occlusion [210] (Section 11.3.5) also allows for dynamically computing global-illumination effects. The point-based representation of the scene (Section 11.3.5) is augmented by storing information about the reflected radiance for each disk. In the gather step, instead of just collecting occlusion, a full incident radiance function can be constructed at each gather location. Just as with ambient occlusion, a subsequent step must be performed to eliminate lighting coming from occluded disks.

11.6 Specular Global Illumination

The methods presented in the previous sections were mainly tailored to simulate diffuse global illumination. We will now look at various methods that can be used to render view-dependent effects. For glossy materials, specular lobes are much tighter than the cosine lobe used for diffuse lighting. If we want to display an extremely shiny material, one with a thin specular lobe, we need a radiance representation that can deliver such high-frequency details. Alternatively, these conditions also mean that evaluation of the reflectance equation needs only the lighting incident from a limited solid angle, unlike a Lambertian BRDF that reflects illumination from the entire hemisphere. This is a completely different requirement than those imposed by diffuse materials. These

characteristics explain why different trade-offs need to be made to deliver such effects in real time.

Methods that store the incident radiance can be used to deliver crude view-dependent effects. When using AHD encoding or the HL2 basis, we can compute the specular response as if the illumination was from a directional light arriving from the encoded direction (or three directions, in the case of the HL2 basis). This approach does deliver some specular highlights from indirect lighting, but they are fairly imprecise. Using this idea is especially problematic for AHD encoding, where the directional component can change drastically over small distances. The variance causes the specular highlights to deform in unnatural ways. Artifacts can be reduced by filtering the direction spatially [806]. Similar problems can be observed when using the HL2 basis if the tangent space changes rapidly between neighboring triangles.

Artifacts can also be reduced by representing incoming lighting with higher precision. Neubelt and Pettineo use spherical Gaussian lobes to represent incident radiance in the game *The Order: 1886* [1268]. To render specular effects, they use a method by Xu et al. [1940], who developed an efficient approximation to a specular response of a typical microfacet BRDF (Section 9.8). If the lighting is represented with a set of spherical Gaussians, and the Fresnel term and the masking-shadowing function are assumed constant over their supports, then the reflectance equation can be approximated by

$$L_o(\mathbf{v}) \approx \sum_k \left(M(\mathbf{l}_k, \mathbf{v})(\mathbf{n} \cdot \mathbf{l}_k)^+ \int_{\mathbf{l} \in \Omega} D(\mathbf{l}, \mathbf{v}) L_k(\mathbf{l}) d\mathbf{l} \right), \tag{11.37}$$

where L_k is the kth spherical Gaussian representing incident radiance, M is the factor combining the Fresnel and masking-shadowing function, and D is the NDF. Xu et al. introduce an *anisotropic spherical Gaussian* (ASG), which they use to model the NDF. They also provide an efficient approximation for computing the integral of a product of SG and ASG, as seen in Equation 11.37.

Neubelt and Pettineo use nine to twelve Gaussian lobes to represent the lighting, which lets them model only moderately glossy materials. They were able to represent most of the game lighting using this method because the game takes place in 19th-century London, and highly polished materials, glass, and reflective surfaces are rare.

11.6.1 Localized Environment Maps

The methods discussed so far are not enough to believably render polished materials. For these techniques the radiance field is too coarse to precisely encode fine details of the incoming radiance, which makes the reflections look dull. The results produced are also inconsistent with the specular highlights from analytical lights, if used on the same material. One solution is to use more spherical Gaussians or a much higher-order SH to get the details we need. This is possible, but we quickly face a performance problem: Both SH and SGs have *global support*. Each basis function is nonzero over the entire sphere, which means that we need all of them to evaluate lighting in a

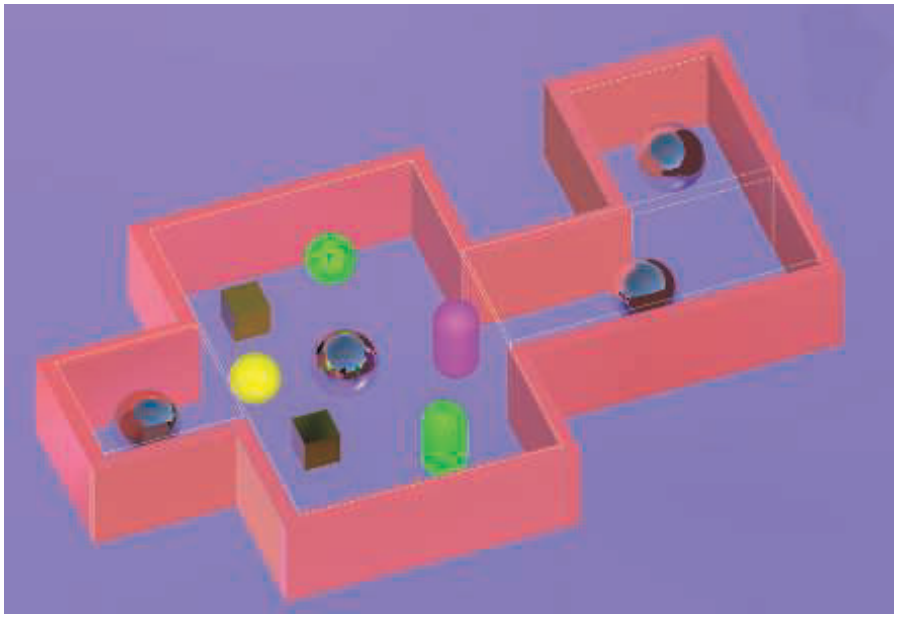

Figure 11.36. A simple scene with localized reflection probes set up. Reflective spheres represent probe locations. Yellow lines depict the box-shaped reflection proxies. Notice how the proxies approximate the overall shape of the scene.

given direction. Doing so becomes prohibitively expensive with fewer basis functions than are needed to render sharp reflections, as we would need thousands. It is also impossible to store that much data at the resolution that is typically used for diffuse lighting.

The most popular solution for delivering specular components for global illumination in real-time settings are *localized environment maps*. They solve both of our earlier problems. The incoming radiance is represented as an environment map, so just a handful of values is required to evaluate the radiance. They are also sparsely distributed throughout the scene, so the spatial precision of the incident radiance is traded for increased angular resolution. Such environment maps, rendered at specific points in the scene, are often called *reflection probes*. See Figure 11.36 for an example.

Environment maps are a natural fit for rendering perfect reflections, which are specular indirect illumination. Numerous methods have been developed that use textures to deliver a wide range of specular effects (Section 10.5). All of these can be used with localized environment maps to render the specular response to indirect illumination.

One of the first titles that tied environment maps to specific points in space was *Half-Life 2* [1193, 1222]. In their system, artists first place sampling locations across

the scene. In a preprocessing step a cube map is rendered from each of these positions. Objects then use the nearest location's result as the representation of the incoming radiance during specular lighting calculations. It can happen that the neighboring objects use different environment maps, which causes a visual mismatch, but artists could manually override the automated assignment of cube maps.

If an object is small and the environment map is rendered from its center (after hiding the object so it does not appear in the texture), the results are fairly precise. Unfortunately, this situation is rare. Most often the same reflection probe is used for multiple objects, sometimes with significant spatial extents. The farther the specular surface's location is from the environment map's center, the more the results can vary from reality.

One way of solving this problem was suggested by Brennan [194] and Bjorke [155]. Instead of treating the incident lighting as if it is coming from an infinitely distant surrounding sphere, they assume that it comes from a sphere with a finite size, with the radius being user-defined. When looking up the incoming radiance, the direction is not used directly to index the environment map, but rather is treated as a ray originating from the evaluated surface location and intersected with this sphere. Next, a new direction is computed, one from the environment map's center to the intersection location. This vector serves as the lookup direction. See Figure 11.37. The procedure has the effect of "fixing" the environment map in space. Doing so is often referred to as *parallax correction*. The same method can be used with other primitives, such as boxes [958]. The shapes used for the ray intersection are often called *reflection proxies*. The proxy object used should represent the general shape and size of the geometry rendered into the environment map. Though usually not possible, if they match exactly—for example when a box is used to represent a rectangular room—the method provides perfectly localized reflections.

This technique has gained great popularity in games. It is easy to implement, is fast at runtime, and can be used in both forward and deferred rendering schemes. Artists have direct control over both the look as well as the memory usage. If certain areas need more precise lighting, they can place more reflection probes and fit proxies better. If too much memory is used to store the environment maps, it is easy to remove the probes. When using glossy materials, the distance between the shading point and the intersection with the proxy shape can be used to determine which level of the prefiltered environment map to use (Figure 11.38). Doing so simulates the growing footprint of the BRDF lobe as we move away from the shading point.

When multiple probes cover the same areas, intuitive rules on how to combine them can be established. For example, probes can have a user-set priority parameter that makes those with higher values take precedence over lower ones, or they can smoothly blend into each other.

Unfortunately, the simplistic nature of the method causes a variety of artifacts. Reflection proxies rarely match the underlying geometry exactly. This makes the reflections stretch in unnatural ways in some areas. This is an issue mainly for highly reflective, polished materials. In addition, reflective objects rendered into the envi-

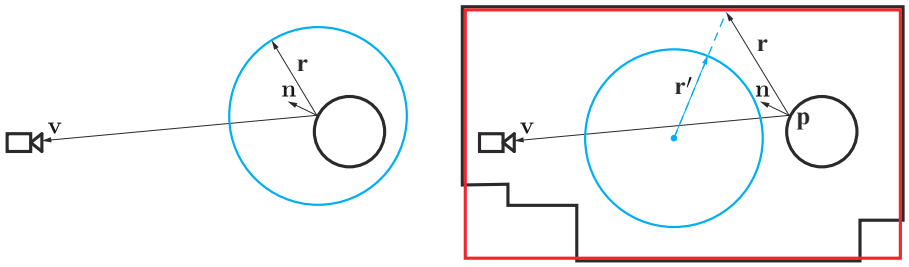

Figure 11.37. The effect of using reflection proxies to spatially localize an environment map (EM). In both cases we want to render a reflection of the environment on the surface of the black circle. On the left is regular environment mapping, represented by the blue circle (but which could be of any representation, e.g., a cube map). Its effect is determined for a point on the black circle by accessing the environment map using the reflected view direction $\mathbf{r}$. By using just this direction, the blue circle EM is treated as if it is infinitely large and far away. For any point on the black circle, it is as if the EM is centered there. On the right, we want the EM to represent the surrounding black room as being local, not infinitely far away. The blue circle EM is generated from the center of the room. To access this EM as if it were a room, a reflection ray from position $\mathbf{p}$ is traced along the reflected view direction and intersected in the shader with a simple proxy object, the red box around the room. This intersection point and the center of the EM are then used to form direction $\mathbf{r}'$, which is used to access the EM as usual, by just a direction. By finding $\mathbf{r}'$, this process treats the EM as if it has a physical shape, the red box. This proxy box assumption will break down for this room in the two lower corners, since the proxy shape does not match the actual room's geometry.

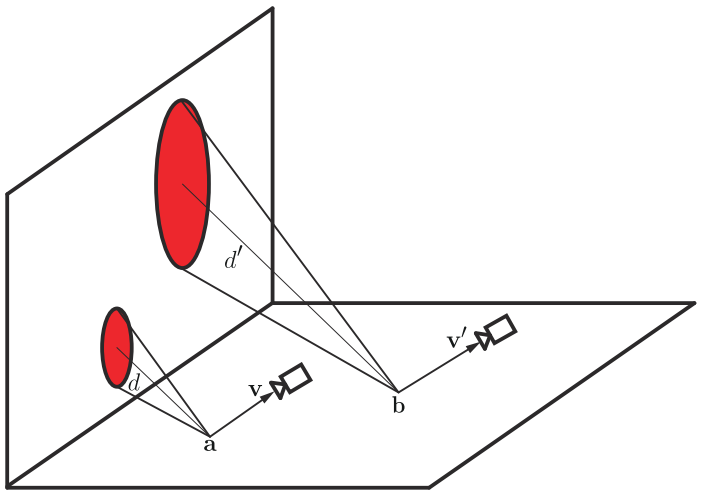

Figure 11.38. The BRDFs at points $\mathbf{a}$ and $\mathbf{b}$ are the same, and the view vectors $\mathbf{v}$ and $\mathbf{v}'$ are equal. Because the distance d to the reflection proxy from point $\mathbf{a}$ is shorter than the distance d' from $\mathbf{b}$, the footprint of the BRDF lobe on the side of the reflection proxy (marked in red) is smaller. When sampling a prefiltered environment map, this distance can be used along with the roughness at the reflection point to affect the mip level.

ronment map have their BRDFs evaluated from the map's location. Surface locations accessing the environment map will not have the exact same view of these objects, so the texture's stored results are not perfectly correct.

Proxies also cause (sometimes severe) light leaks. Often, the lookup will return values from bright areas of the environment map, because the simplified ray cast misses the local geometry that should cause occlusion. This problem is sometimes mitigated by using directional occlusion methods (Section 11.4). Another popular strategy for mitigating this issue is using the precomputed diffuse lighting, which is usually stored with a higher resolution. The values in the environment map are first divided by the average diffuse lighting at the position from which it was rendered. Doing so effectively removes the smooth, diffuse contribution from the environment map, leaving only higher-frequency components. When the shading is performed, the reflections are multiplied by the diffuse lighting at the shaded location [384, 999]. Doing so can partially alleviate the lack of spatial precision of the reflection probes.

Solutions have been developed that use more sophisticated representation of the geometry captured by the reflection probe. Szirmay-Kalos et al. [1730] store a depth map for each reflection probe and perform a ray trace against it on lookup. This can produce more accurate results, but at an extra cost. McGuire et al. [1184] propose a more efficient way of tracing rays against the probes' depth buffer. Their system stores multiple probes. If the initially chosen probe does not contain enough information to reliably determine the hit location, a fallback probe is selected and the trace continues using the new depth data.

When using a glossy BRDF, the environment map is usually prefiltered, and each mipmap stores the incident radiance convolved with a progressively larger kernel. The prefiltering step assumes that this kernel is radially symmetric (Section 10.5). However, when using parallax correction, the footprint of the BRDF lobe on the shape of the reflection proxy changes depending on the location of the shaded point. Doing so makes the prefiltering slightly incorrect. Pesce and Iwanicki analyze different aspects of this problem and discuss potential solutions [807, 1395].

Reflection proxies do not have to be closed, convex shapes. Simple, planar rectangles can also be used, either instead of or to augment the box or sphere proxies with high-quality details [1228, 1640].

11.6.2 Dynamic Update of Environment Maps

Using localized reflection probes requires that each environment map needs to be rendered and filtered. This work is often done offline, but there are cases where it might be necessary to do it at runtime. In case of an open-world game with a changing time of day, or when the world geometry is generated dynamically, processing all these maps offline might take too long and impact productivity. In extreme cases, when many variants are needed, it might even be impossible to store them all on disk.

In practice, some games render the reflection probes at runtime. This type of system needs to be carefully tuned not to affect the performance in a significant way.

Except for trivial cases, it is not possible to re-render all the visible probes every frame, as a typical frame from a modern game can use tens or even hundreds of them. Fortunately, this is not needed. We rarely require the reflection probes to accurately depict all the geometry around them at all times. Most often we do want them to properly react to changes in the time of day, but we can approximate reflections of dynamic geometry by some other means, such as the screen-space methods described later (Section 11.6.5). These assumptions allow us to render a few probes at load time and the rest progressively, one at a time, as they come into view.

Even when we do want dynamic geometry to be rendered in the reflection probes, we almost certainly can afford to update the probes at a lower frame rate. We can define how much frame time we want to spend rendering reflection probes and update just some fixed number of them every frame. Heuristics based on each probe's distance to the camera, time since the last update, and similar factors can determine the update order. In cases where the time budget is particularly small, we can even split the rendering of a single environment map over multiple frames. For instance, we could render just a single face of a cube map each frame.

High-quality filtering is usually used when performing convolution offline. Such filtering involves sampling the input texture many times, which is impossible to afford at high frame rates. Colbert and Křivánek [279] developed a method to achieve comparable filtering quality with relatively low sample counts (in the order of 64), using importance sampling. To eliminate the majority of the noise, they sample from a cube map with a full mip chain, and use heuristics to determine which mip level should be read by each sample. Their method is a popular choice for fast, runtime prefiltering of environment maps [960, 1154]. Manson and Sloan [1120] construct the desired filtering kernel out of basis functions. The exact coefficients for constructing a particular kernel must be obtained during an optimization process, but it happens only once for a given shape. The convolution is performed in two stages. First, the environment map is downsampled and simultaneously filtered with a simple kernel. Next, samples from the resulting mip chain are combined to construct the final environment map.

To limit the bandwidth used in the lighting passes, as well as the memory usage, it is beneficial to compress the resulting textures. Narkowicz [1259] describes an efficient method for compressing high dynamic range reflection probes to BC6H format (Section 6.2.6), which is capable of storing half-precision floating point values.

Rendering complex scenes, even one cube map face at a time, might be too expensive for the CPU. One solution is to prepare G-buffers for the environment maps offline and calculate only the (much less CPU-demanding) lighting and convolution [384, 1154]. If needed, we can even render dynamic geometry on top of the pregenerated G-buffers.

11.6.3 Voxel-Based Methods

In the most performance-restricted scenarios, localized environment maps are an excellent solution. However, their quality can often be somewhat unsatisfactory. In

practice, workarounds have to be used to mask problems resulting from insufficient spatial density of the probes or from proxies being too crude an approximation of the actual geometry. More elaborate methods can be used when more time is available per frame.

Voxel cone tracing—both in the sparse octree [307] as well as the cascaded version [1190] (Section 11.5.7)—can be used for the specular component as well. The method performs cone tracing against a representation of the scene stored in a sparse voxel octree. A single cone trace provides just one value, representing the average radiance coming from the solid angle subtended by the cone. For diffuse lighting, we need to trace multiple cones, as using just a single cone is inaccurate.

It is significantly more efficient to use cone tracing for glossy materials. In the case of specular lighting, the BRDF lobe is narrow, and only radiance coming from a small solid angle needs to be considered. We no longer need to trace multiple cones; in many cases just one is enough. Only specular effects on rougher materials might require tracing multiple cones, but because such reflections are blurry, it is often sufficient to fall back to localized reflection probes for these cases and not trace cones at all.

On the opposite end of the spectrum are highly polished materials. The specular reflection off of these is almost mirror-like. This makes the cone thin, resembling a single ray. With such a precise trace, the voxel nature of the underlying scene representation might be noticeable in the reflection. Instead of polygonal geometry it will show the cubes resulting from the voxelization process. This artifact is rarely a problem in practice, as the reflection is almost never seen directly. Its contribution is modified by textures, which often mask any imperfections. When perfect mirror reflections are needed, other methods can be used that provide them at lower runtime cost.

11.6.4 Planar Reflections

Another alternative is to reuse the regular representation of the scene and re-render it to create a reflected image. If there is a limited number of reflective surfaces, and they are planar, we can use the regular GPU rendering pipeline to create an image of the scene reflected off such surfaces. These images can not only provide accurate mirror reflections, but also render plausible glossy effects with some extra processing of each image.

An ideal reflector follows the *law of reflection*, which states that the angle of incidence is equal to the angle of reflection. That is, the angle between the incident ray and the normal is equal to the angle between the reflected ray and the normal. See Figure 11.39. This figure also shows an "image" of the reflected object. Due to the law of reflection, the reflected image of the object is simply the object itself, physically reflected through the plane. That is, instead of following the reflected ray, we could follow the incident ray through the reflector and hit the same point, but on the reflected object.

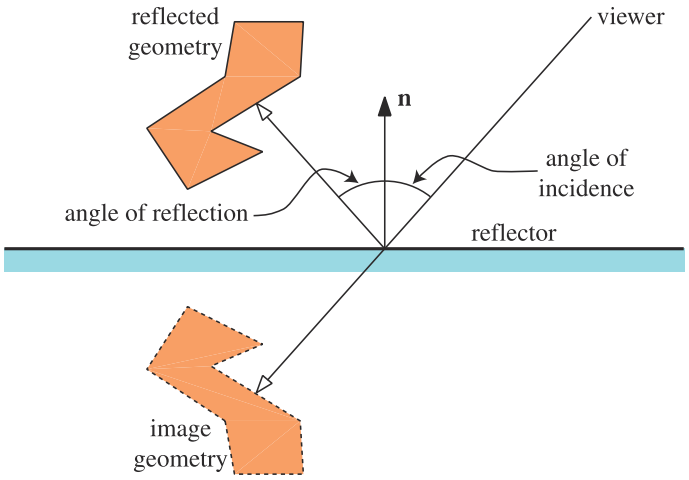

Figure 11.39. Reflection in a plane, showing angle of incidence and reflection, the reflected geometry, and the reflector.

This leads us to the principle that a reflection can be rendered by creating a copy of the object, transforming it into the reflected position, and rendering it from there. To achieve correct lighting, light sources have to be reflected in the plane as well, with respect to both position and direction [1314]. An equivalent method is to instead reflect the viewer's position and orientation through the mirror to the opposite side of the reflector. This reflection can be achieved by simple modification to the projection matrix.

Objects that are on the far side of (i.e., behind) the reflector plane should not be reflected. This problem can be solved by using the reflector's plane equation. The simplest method is to define a clipping plane in the pixel shader. Place the clipping plane so that it coincides with the plane of the reflector [654]. Using this clipping plane when rendering the reflected scene will clip away all reflected geometry that is on the same side as the viewpoint, i.e., all objects that were originally behind the mirror.

11.6.5 Screen-Space Methods

Just as with ambient occlusion and diffuse global illumination, some specular effects can be calculated solely in screen space. Doing so is slightly more precise than in the diffuse case, because of the sharpness of the specular lobe. Information about the radiance is needed from only a limited solid angle around the reflected view vector, not from the full hemisphere, so the screen data are much more likely to contain it. This type of method was first presented by Sousa et al. [1678] and was simultaneously discovered by other developers. The whole family of methods is called *screen-space reflections* (SSR).

Given the position of the point being shaded, the view vector, and the normal, we can trace a ray along the view vector reflected across the normal, testing for intersections with the depth buffer. This testing is done by iteratively moving along the ray, projecting the position to screen space, and retrieving the z-buffer depth from that location. If the point on the ray is further from the camera than the geometry represented by the depth buffer, it means that the ray is inside the geometry and a hit is detected. A corresponding value from the color buffer can then be read to obtain the value of the radiance incident from the traced direction. This method assumes that the surface hit by the ray is Lambertian, but this condition is an approximation common to many methods and is rarely a constraint in practice. The ray can be traced in uniform steps in world space. This method is fairly coarse, so when a hit is detected, a refinement pass can be performed. Over a limited distance, a binary search can be used to accurately locate the intersection position.

McGuire and Mara [1179] note that, because of perspective projection, stepping in uniform world-space intervals creates uneven distributions of sampling points along the ray in screen space. Sections of the rays close to the camera are undersampled, so some hit events might be missed. Those farther away are oversampled, so the same depth buffer pixels are read multiple times, generating unnecessary memory traffic and redundant computations. They suggest performing the ray march in screen space instead, using a *digital differential analyzer* (DDA), a method that can be used for rasterizing lines.

First, both the start and end points of the ray to be traced are projected to screen space. Pixels along this line are each examined in turn, guaranteeing uniform precision. One consequence of this approach is that the intersection test does not require full reconstruction of the view-space depth for every pixel. The reciprocal of the view-space depth, which is the value stored in the z-buffer in the case of a typical perspective projection, changes linearly in screen space. This means that we can compute its derivatives with respect to screen-space x- and y-coordinates before the actual trace, then use simple linear interpolation to get the value anywhere along the screen-space segment. The computed value can be directly compared with the data from the depth buffer.

The basic form of screen-space reflections traces just a single ray, and can provide only mirror reflections. However, perfectly specular surfaces are fairly rare. In modern, physically based rendering pipelines, glossy reflections are needed more often, and SSR can also be used to render these.

In simple, ad hoc approaches [1589, 1812], the reflections are still traced with a single ray, along the reflected direction. The results are stored in an offscreen buffer that is processed in a subsequent step. A series of filtering kernels is applied, often combined with downsampling the buffer to create a set of reflection buffers, each blurred to a different degree. When computing the lighting, the width of the BRDF lobe determines which reflection buffer is sampled. Even though the shape of the filter is often chosen to match the shape of the BRDF lobe, doing so is still only a crude approximation, as screen-space filtering is performed without considering discontinuities, surface

orientation, and other factors crucial to the precision of the result. Custom heuristics are added at the end to make the glossy screen-space reflections visually match specular contributions from other sources. Even though it is an approximation, the results are convincing.

Stachowiak [1684] approaches the problem in a more principled way. Computing screen-space reflections is a form of ray tracing, and just like ray tracing it can be used to perform proper Monte Carlo integration. Instead of just using the reflected view direction, he uses importance sampling of the BRDF and shoots rays stochastically. Because of performance constraints, the tracing is done at half resolution and a small number of rays are traced per pixel (between one and four). This is too few rays to produce a noise-free image, so the intersection results are shared between neighboring pixels. It is assumed that the local visibility can be considered the same for pixels within some range. If a ray shot from point $\mathbf{p}_0$ in direction $\mathbf{d}_0$ intersects the scene in point $\mathbf{i}_0$, we can assume that if we shoot a ray from point $\mathbf{p}_1$, in a direction $\mathbf{d}_1$ such that it also passes through $\mathbf{i}_0$, it will also hit the geometry in $\mathbf{i}_0$ and there will not be any intersections before it. This lets us use the ray, without actually tracing it, just by appropriately modifying its contribution to the neighbor's integral. Formally speaking, the direction of a ray shot from a neighboring pixel will have a different probability when computed with respect to the probability distribution function of the BRDF of the current pixel.

To further increase the effective number of rays, the results are filtered temporally. The variance of the final integral is also reduced by performing the scene-independent part of the integration offline and storing it in a lookup table indexed by BRDF parameters. In situations where all the required information for the reflected rays is available in screen space, these strategies allow us to achieve precise, noise-free results, close to path-traced ground-truth images (Figure 11.40).

Tracing rays in screen space is generally expensive. It consists of repeatedly sampling the depth buffer, possibly multiple times, and performing some operations on the lookup results. Because the reads are fairly incoherent, the cache utilization can be poor, leading to long stalls during shader execution from waiting on memory transactions to finish. Much care needs to be put into making the implementation as fast as possible. Screen-space reflections are most often calculated at a reduced resolution [1684, 1812], and temporal filtering is used to make up for the decreased quality.

Uludag [1798] describes an optimization that uses a hierarchical depth buffer (Section 19.7.2) to accelerate tracing. First, a hierarchy is created. The depth buffer is progressively downsampled, by a factor of two in each direction for each step. A pixel on the higher level stores the minimum depth value between the four corresponding pixels at the lower level. Next, the trace is performed through the hierarchy. If in a given step the ray does not hit the geometry stored in the cell it passes through, it is advanced to the cell's boundary, and in the next step a lower-resolution buffer is used. If the ray encounters a hit in the current cell, it is advanced to the hit location, and in the next step a higher-resolution buffer is used. The trace terminates when a hit on the highest-resolution buffer is registered (Figure 11.41).

Figure 11.40. All the specular effects in this image were rendered using a stochastic screen-space reflection algorithm [1684]. Notice the vertical stretching, characteristic of reflections from microfacet models. *(Image courtesy of Tomasz Stachowiak. Scene modeled and textured by Joacim Lunde.)*

The scheme is particularly good for long traces, as it ensures no features will be missed and at the same time allows the ray to advance in large increments. It also accesses the caches well, as the depth buffers are not read in random, distant locations, but rather in a local neighborhood. Many practical tips on implementing this method are presented by Grenier [599].

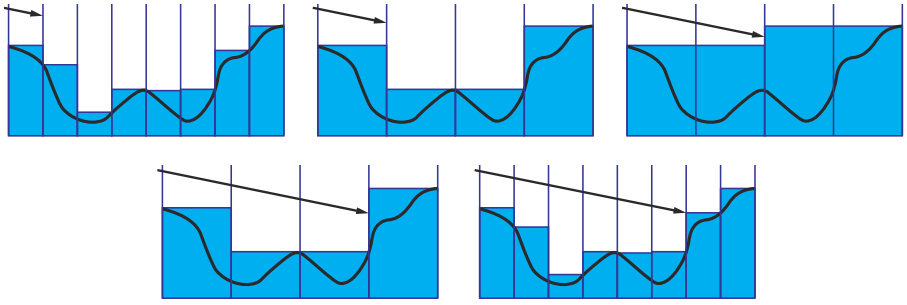

Figure 11.41. Tracing a ray through a hierarchical depth buffer. If the ray does not hit geometry when passing through a pixel, the next step uses a coarser resolution. If a hit is registered, the subsequent step uses finer resolution. This process allows the ray to traverse empty areas in large steps, providing higher performance.

Others avoid tracing the rays entirely. Drobot [384] reuses the location of the intersection with the reflection proxies and looks up the screen-space radiance from there. Cichocki [266] assumes planar reflectors and, instead of tracing the rays, reverses the process and runs a full-screen pass in which every pixel writes its value into the location where it should be reflected.

Just as with other screen-space approaches, reflections can also suffer from artifacts caused by the limited data available. It is common for the reflected rays to leave the screen area before registering a hit, or to hit the backfaces of the geometry, for which no lighting information is available. Such situations need to be handled gracefully, as the validity of the traces is often different even for neighboring pixels. Spatial filters can be used to partially fill the gaps in the traced buffer [1812, 1913].

Another problem with SSR is the lack of information about the thickness of the objects in the depth buffer. Because only a single value is stored, there is no way to tell if the ray hit anything when it goes behind a surface described by depth data. Cupisz [315] discusses various low-cost ways to mitigate the artifacts arising from not knowing the thickness of the objects in the depth buffer. Mara et al. [1123] describe the deep G-buffer, which stores multiple layers of data and so has more information about the surface and environment.

Screen-space reflection is a great tool to provide a specific set of effects, such as local reflections of nearby objects on mostly flat surfaces. They substantially improve the quality of real-time specular lighting, but they do not provide a complete solution. Different methods described in this chapter are often stacked on top of each other, to deliver a complete and robust system. Screen-space reflection serves as a first layer. If it cannot provide accurate results, localized reflection probes are used as a fallback. If none of the probes are applied in a given area, a global, default probe is used [1812]. This type of setup provides a consistent and robust way of obtaining a plausible indirect specular contribution, which is especially important for a believable look.

11.7 Unified Approaches

The methods presented so far can be combined into a coherent system capable of rendering beautiful images. However, they lack the elegance and conceptual simplicity of path tracing. Every aspect of the rendering equation is handled in a different way, each making various compromises. Even though the final image can look realistic, there are many situations when these methods fail and the illusion breaks. For these reasons, real-time path tracing has been the focus of significant research efforts.

The amount of computation needed for rendering images of acceptable quality with path tracing far exceeds the capabilities of even fast CPUs, so GPUs are used instead. Their extreme speed and the flexibility of the compute units make them good candidates for this task. Applications of real-time path tracing include architectural

walkthroughs and previsualization for movie rendering. Lower and varying frame rates can be acceptable for these use cases. Techniques such as progressive refinement (Section 13.2) can be used to improve the image quality when the camera is still. High-end systems can expect to have use of multiple GPUs.

In contrast, games need to render frames in final quality, and they need to do it consistently within the time budget. The GPU may also need to perform tasks other than rendering itself. For example, systems such as particle simulations are often offloaded to the GPU to free up some CPU processing power. All these elements combined make path tracing impractical for rendering games today.

There is a saying in the graphics community: "Ray tracing is the technology of the future and it always will be!" This quip implies that the problem is so complex, that even with all the advances in both hardware speed and algorithms, there will always be more efficient ways of handling particular parts of the rendering pipeline. Paying the extra cost and using only ray casting, including for primary visibility, may be hard to justify. There is currently considerable truth to it, because GPUs were never designed to perform efficient ray tracing. Their main goal has always been rasterizing triangles, and they have become extremely good at this task. While ray tracing can be mapped to the GPU, current solutions do not have any direct support from fixed-function hardware. It is difficult at best to always beat hardware rasterization with what is effectively a software solution running on the GPU's compute units.

The more reasonable, less purist approach is to use path-tracing methods for effects that are difficult to handle within the rasterization framework. Rasterize triangles visible from the camera, but instead of relying on approximate reflection proxies, or incomplete screen-space information, trace paths to compute reflections. Instead of trying to simulate area light shadows with ad hoc blurs, trace rays toward the source and compute the correct occlusion. Play to the GPU's strengths and use the more general solution for elements that cannot be handled efficiently in hardware. Such a system would still be a bit of a patchwork, and would lack the simplicity of path tracing, but real-time rendering has always been about compromises. If some elegance has to be given up for a few extra milliseconds, it is the right choice—frame rate is nonnegotiable.

While we perhaps will never be able to call real-time rendering a "solved problem," more use of path tracing would help bring theory and practice closer together. With GPUs getting faster every day, such hybrid solutions should be applicable to even the most demanding applications in the near future. Initial examples of systems built on these principles are already starting to appear [1548].

A ray tracing system relies on an acceleration scheme such as using a *bounding volume hierarchy* (BVH) to accelerate visibility testing. See Section 19.1.1 for more information about this topic. A naive implementation of a BVH does not map well to the GPU. As explained in Chapter 3, GPUs natively execute groups of threads, called warps or wavefronts. A warp is processed in lock-step, with every thread performing the same operation. If some of the threads do not execute particular parts of the

Figure 11.42. Spatiotemporal variance-guided filtering can be used to denoise a one-sample-per-pixel, path-traced image (left) to creates a smooth artifact-free image (center). The quality is comparable to a reference rendered with 2048 samples per pixel (right). *(Image courtesy of NVIDIA Corporation.)*

code, they are temporarily disabled. For this reason, GPU code should be written in a way that minimizes divergent flow control between the threads within the same wavefront. Say each thread processes a single ray. This scheme usually leads to large divergence between threads. Different rays will execute diverging branches of the traversal code, intersecting different bounding volumes along the way. Some rays will finish tree traversal earlier than others. This behavior takes us away from the ideal, where all threads in a warp are using the GPU's compute capabilities. To eliminate these inefficiencies, traversal methods have been developed that minimize divergence and reuse threads that finished early [15, 16, 1947].

Hundreds or thousands of rays may need to be traced per pixel to generate high-quality images. Even with an optimal BVH, efficient tree traversal algorithms, and fast GPUs, doing so is not possible in real time today for any but the simplest scenes. The images that we can generate within the available performance constraints are extremely noisy and are not suitable for display. However, they can be treated with denoising algorithms, to produce mostly noise-free images. See Figure 11.42, as well as Figure 24.2 on page 1044. Impressive advances have been made in the field recently, and algorithms have been developed that can create images visually close to high-quality, path-traced references from input generated by tracing even just a single path per pixel [95, 200, 247, 1124, 1563].

In 2014 PowerVR announced their Wizard GPU [1158]. In addition to typical functionality, it contains units that construct and traverse acceleration structures in hardware (Section 23.11). This system proves that there are both interest and the ability to tailor fixed-function units to accelerate ray casting. It will be exciting to see what the future holds!

Further Reading and Resources

Pharr et al.'s book *Physically Based Rendering* [1413] is an excellent guide to non-interactive global illumination algorithms. What is particularly valuable about their work is that they describe in depth what they found works. Glassner's (now free) *Principles of Digital Image Synthesis* [543, 544] discusses the physical aspects of the interaction of light and matter. *Advanced Global Illumination* by Dutré et al. [400] provides a foundation in radiometry and on (primarily offline) methods of solving Kajiya's rendering equation. McGuire's *Graphics Codex* [1188] is an electronic reference that holds a huge range of equations and algorithms pertaining to computer graphics. Dutré's *Global Illumination Compendium* [399] reference work is quite old, but free. Shirley's series of short books [1628] are an inexpensive and quick way to learn about ray tracing.

Chapter 12
Image-Space Effects

"The world was not wheeling anymore. It was just very clear and bright and inclined to blur at the edges."
—Ernest Hemingway

There is more involved in making an image than simply portraying objects. Part of making an image appear photorealistic is having it look like a photo. Just as a photographer adjusts their final result, we may also wish to modify, say, the color balance. Adding film grain, vignetting, and other subtle changes to the rendered image can make a rendering look more convincing. Alternately, more dramatic effects such as lens flares and bloom can convey a sense of drama. Portraying depth of field and motion blur can increase realism and be used for artistic effect.

The GPU can be used to efficiently sample and manipulate images. In this chapter we first discuss modifying a rendered image with *image processing* techniques. Additional data, such as depths and normals, can be used to enhance these operations, for example, by allowing a noisy area to be smoothed while still retaining sharp edges. Reprojection methods can be used to save on shading computations, or rapidly create missing frames. We conclude by presenting a variety of sample-based techniques to produce lens flares, bloom, depth of field, motion blur, and other effects.

12.1 Image Processing

Graphics accelerators have generally been concerned with creating artificial scenes from geometry and shading descriptions. Image processing is different, where we take an input image and modify it in various ways. The combination of programmable shaders and the ability to use an output image as an input texture opened the way to using the GPU for a wide variety of image processing effects. Such effects can be combined with image synthesis. Typically, an image is generated and then one or more image processing operations is performed on it. Modifying the image after rendering

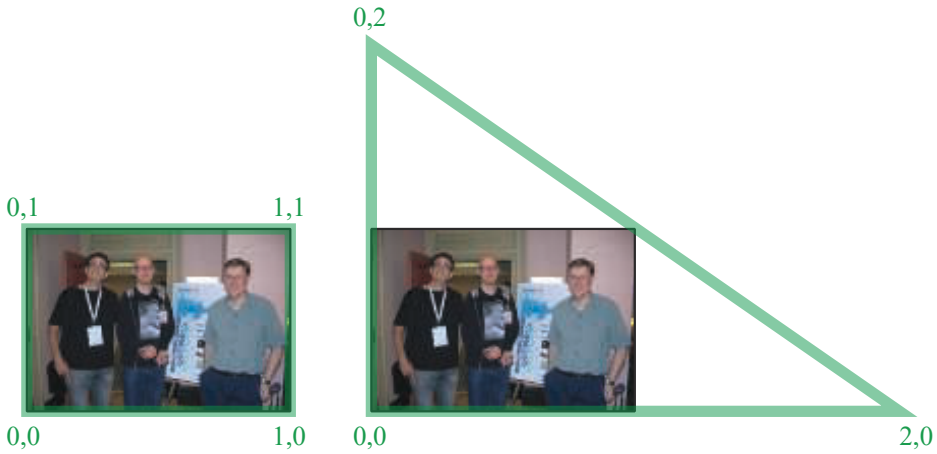

Figure 12.1. On the left, a screen-filling quadrilateral, with (u, v) texture coordinates shown. On the right, a single triangle fills the screen, with its texture coordinates adjusted appropriately to give the same mapping.

is called *post-processing*. A large number of passes, accessing image, depth, and other buffers, can be performed while rendering just a single frame [46, 1918]. For example, the game *Battlefield 4* has over fifty different types of rendering passes [1313], though not all are used in a single frame.

There are a few key techniques for post-processing using the GPU. A scene is rendered in some form to an offscreen buffer, such as a color image, z-depth buffer, or both. This resulting image is then treated as a texture. This texture is applied to a screen-filling quadrilateral. Post-processing is performed by rendering this quadrilateral, as the pixel shader program will be invoked for every pixel. Most image processing effects rely on retrieving each image texel's information at the corresponding pixel. Depending on system limitations and algorithm, this can be done by retrieving the pixel location from the GPU or by assigning texture coordinates in the range $[0, 1]$ to the quadrilateral and scaling by the incoming image size.

In practice a screen-filling triangle can be more efficient than a quadrilateral. For example, image processing proceeds almost 10% faster on the AMD GCN architecture when a single triangle is used instead of a quadrilateral formed from two triangles, due to better cache coherency [381]. The triangle is made large enough to fill the screen [146]. See Figure 12.1. Whatever primitive object is used, the intent is the same: to have the pixel shader be evaluated for every pixel on the screen. This type of rendering is called a *full screen pass*. If available, you can also use compute shaders to perform image processing operations. Doing so has several advantages, described later.

Using the traditional pipeline, the stage is now set for the pixel shader to access the image data. All relevant neighboring samples are retrieved and operations are applied to them. The contribution of the neighbor is weighted by a value depending on its relative location from the pixel being evaluated. Some operations, such as edge detection, have a fixed-size neighborhood (for example, 3×3 pixels) with different weights (sometimes negative) for each neighbor and the pixel's original value itself. Each texel's value is multiplied by its corresponding weight and the results are summed, thus producing the final result.

As discussed in Section 5.4.1, various filter kernels can be used for reconstruction of a signal. In a similar fashion, filter kernels can be used to blur the image. A *rotation-invariant filter kernel* is one that has no dependency on radial angle for the weight assigned to each contributing texel. That is, such filter kernels are described entirely by a texel's distance from the central pixel for the filtering operation. The sinc filter, shown in Equation 5.22 on page 135, is a simple example. The Gaussian filter, the shape of the well-known bell curve, is a commonly used kernel:

$$\text{Gaussian}(x) = \left(\frac{1}{\sigma\sqrt{2\pi}}\right) e^{-\frac{r^2}{2\sigma^2}}, \tag{12.1}$$

where r is the distance from the texel's center and σ is the standard deviation; σ^2 is called the variance. A larger standard deviation makes a wider bell curve. A rough rule of thumb is to make the *support*, the filter size, 3σ pixels wide or greater, as a start [1795]. A wider support gives more blur, at the cost of more memory accesses.

The term in front of the e keeps the area under the continuous curve equal to one. However, this term is irrelevant when forming a discrete filter kernel. The values computed per texel are summed together over the area, and all values are then divided by this sum, so that the final weights sum to one. Because of this normalization process, the constant term serves no purpose, and so often is not shown in filter kernel descriptions. The Gaussian two-dimensional and one-dimensional filters shown in Figure 12.2 are formed this way.

A problem with the sinc and Gaussian filters is that the functions go on forever. One expedient is to clamp such filters to a specific diameter or square area and simply treat anything beyond as having a value of zero. Other filtering kernels are designed for various properties, such as ease of control, smoothness, or simplicity of evaluation. Bjorke [156] and Mitchell et al. [1218], provide some common rotation-invariant filters and other information on image processing on the GPU.

Any full-screen filtering operation will attempt to sample pixels from outside the bounds of the display. For example, if you gather 3×3 samples for, say, the upper left pixel on the screen, you are attempting to retrieve texels that do not exist. One basic solution is to set the texture sampler to clamp to the edge. When an offscreen, nonexistent texel is requested, instead the nearest edge texel is retrieved. This leads to filtering errors at the edges of the image, but these are often not noticeable. Another solution is to generate the image to be filtered at a slightly higher resolution than the display area, so that these offscreen texels exist.

(a)

0.0030	0.0133	0.0219	0.0133	0.0030
0.0133	0.0596	0.0983	0.0596	0.0133
0.0219	0.0983	0.1621	0.0983	0.0219
0.0133	0.0596	0.0983	0.0596	0.0133
0.0030	0.0133	0.0219	0.0133	0.0030

(b)

0.0545	0.2442	0.4026	0.2442	0.0545
0.0545	0.2442	0.4026	0.2442	0.0545
0.0545	0.2442	0.4026	0.2442	0.0545
0.0545	0.2442	0.4026	0.2442	0.0545
0.0545	0.2442	0.4026	0.2442	0.0545

(c)

0.0545	0.0545	0.0545	0.0545	0.0545
0.2442	0.2442	0.2442	0.2442	0.2442
0.4026	0.4026	0.4026	0.4026	0.4026
0.2442	0.2442	0.2442	0.2442	0.2442
0.0545	0.0545	0.0545	0.0545	0.0545

Figure 12.2. One way to perform a Gaussian blur is to sample a 5×5 area, weighting each contribution and summing them. Part (a) of the figure shows these weights for a blur kernel with $\sigma = 1$. A second way is to use separable filters. Two one-dimensional Gaussian blurs, (b) and (c), are performed in series, with the same net result. The first pass, shown in (b) for 5 separate rows, blurs each pixel horizontally using 5 samples in its row. The second pass, (c), applies a 5-sample vertical blur filter to the resulting image from (b) to give the final result. Multiplying the weights in (b) by those in (c) gives the same weights as in (a), showing that this filter is equivalent and therefore separable. Instead of needing 25 samples, as in (a), each of (b) and (c) effectively each use 5 per pixel, for a total of 10 samples.

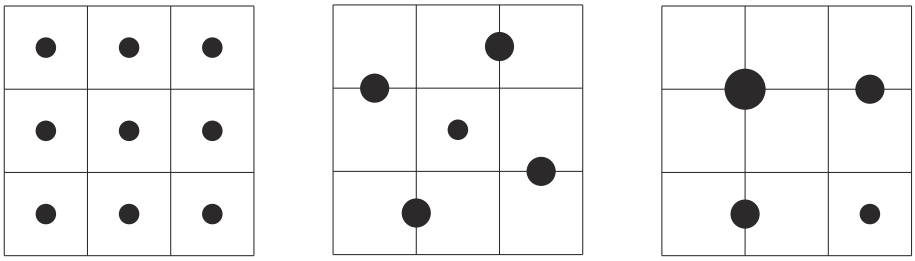

Figure 12.3. On the left, a box filter is applied by performing nine texture samples and averaging the contributions together. In the middle, a symmetric pattern of five samples is used, with the outside samples each representing two texels, and so each is given twice the weight of the center sample. This type of pattern can be useful for other filter kernels, where by moving the outside samples between their two texels, the relative contribution of each texel can be changed. On the right, a more efficient four-sample pattern is used instead. The sample on the upper left interpolates between four texels' values. Those on the upper right and lower left each interpolate the values of two texels. Each sample is given a weight proportional to the number of texels it represents.

One advantage of using the GPU is that built-in interpolation and mipmapping hardware can be used to help minimize the number of texels accessed. For example, say the goal is to use a box filter, i.e., to take the average of the nine texels forming a 3×3 grid around a given texel and display this blurred result. These nine texture samples would then be weighted and summed together by the pixel shader, which would then output the blurred result to the pixel.

Nine explicit sample operations are unnecessary, however. By using bilinear interpolation with a texture, a single texture access can retrieve the weighted sum of up to four neighboring texels [1638]. Using this idea, the 3×3 grid could be sampled with just four texture accesses. See Figure 12.3. For a box filter, where the weights are equal, a single sample could be placed midway among four texels, obtaining the average of the four. For a filter such as the Gaussian, where the weights differ in such a way that bilinear interpolation between four samples can be inaccurate, each sample can still be placed between two texels, but offset closer to one than the other. For instance, imagine one texel's weight was 0.01 and its neighbor was 0.04. The sample could be put so that it was a distance of 0.8 from the first, and so 0.2 from the neighbor, giving each texel its proper proportion. This single sample's weight would be the sum of the two texels' weights, 0.05. Alternatively, the Gaussian could be approximated by using a bilinear interpolated sample for every four texels, finding the offset that gives the closest approximation to the ideal weights.

Some filtering kernels are *separable*. Two examples are the Gaussian and box filters. This means that they can be applied in two separate one-dimensional blurs. Doing so results in considerably less texel access being needed overall. The cost goes from d^2 to $2d$, where d is the kernel diameter or support [815, 1218, 1289]. For example, say the box filter is to be applied in a 5×5 area at each pixel in an image. First the image could be filtered horizontally: The two neighboring texels to the left and

two to the right of each pixel, along with the pixel's value itself, are equally weighted by 0.2 and summed together. The resulting image is then blurred vertically, with the two neighboring texels above and below averaged with the central pixel. For example, instead of having to access 25 texels in a single pass, a total of 10 texels are accessed in two passes. See Figure 12.2. Wider filter kernels benefit even more.

Circular disk filters, useful for bokeh effects (Section 12.4), are normally expensive to compute, since they are not separable in the domain of real numbers. However, using complex numbers opens up a wide family of functions that are. Wronski [1923] discusses implementation details for this type of separable filter.

Compute shaders are good for filtering, and the larger the kernel, the better the performance compared to pixel shaders [1102, 1710]. For example, thread group memory can be used to share image accesses among different pixels' filter computations, reducing bandwidth [1971]. A box filter of any radius can be performed for a constant cost by using scattered writes with a compute shader. For the horizontal and vertical passes, the kernel for the first pixel in a row or column is computed. Each successive pixel's result is determined by adding in the next sample at the leading edge of the kernel and subtracting the sample at the far end that was left behind. This "moving average" technique can be used to approximate a Gaussian blur of any size in constant time [531, 588, 817].

Downsampling is another GPU-related technique commonly used when blurring. The idea is to make a smaller version of the image to be manipulated, e.g., halving the resolution along both axes to make a quarter-screen image. Depending on the input data and the algorithm's requirements, the original image may be filtered down in size or simply created at this lower resolution. When this image is accessed to blend into the final, full resolution image, magnification of the texture will use bilinear interpolation to blend between samples. This gives a further blurring effect. Performing manipulations on a smaller version of the original image considerably decreases the overall number of texels accessed. Also, any filters applied to this smaller image have the net effect of increasing the relative size of the filter kernel. For example, applying a kernel with a width of five (i.e., two texels to each side of the central pixel) to the smaller image is similar in effect to applying a kernel with a width of nine to the original image. Quality will be lower, but for blurring large areas of similar color, a common case for many glare effects and other phenomena, most artifacts will be minimal [815]. Reducing the number of bits per pixel is another method that can lower memory access costs. Downsampling can be used for other slowly varying phenomena, e.g., many particle systems can be rendered at half resolution [1391]. This idea of downsampling can be extended to creating a mipmap of an image and sampling from multiple layers to increase the speed of the blurring process [937, 1120].

12.1.1 Bilateral Filtering

Upsampling results and other image processing operations can be improved by using some form of *bilateral filter* [378, 1355]. The idea is to discard or lower the influence

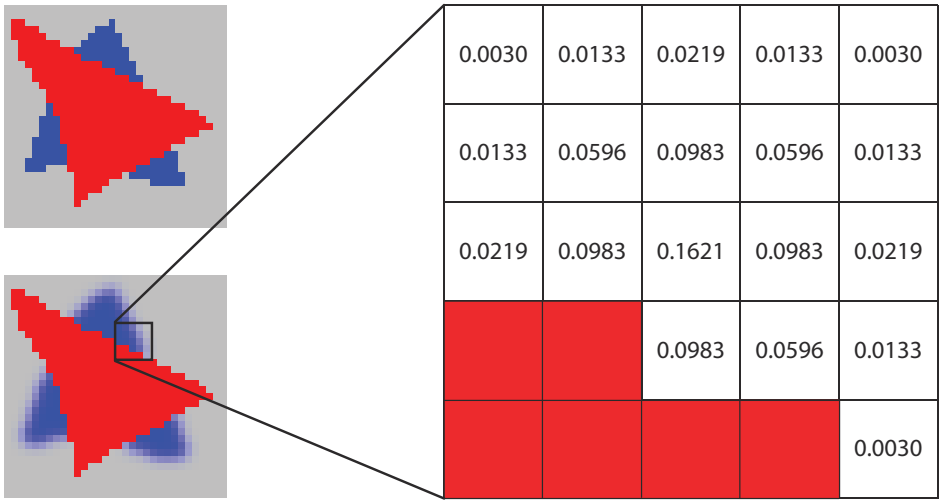

Figure 12.4. Bilateral filter. In the upper left is the original image. In the lower left, we blur and use samples from only those pixels that are not red. On the right, one pixel's filter kernel is shown. The red pixels are ignored when computing the Gaussian blur. The rest of the pixels' colors are multiplied by the corresponding filter weights and summed, and the sum of these weights is also computed. For this case the weights sum to 0.8755, so the color computed is divided by this value.

of samples that appear unrelated to the surface at our center sample. This filter is used to preserve edges. Imagine that you focus a camera on a red object in front of a distant blue object, against a gray background. The blue object should be blurry and the red sharp. A simple bilateral filter would examine the color of the pixel. If red, no blurring would occur—the object stays sharp. Otherwise, the pixel is blurred. All samples that are not red will be used to blur the pixel. See Figure 12.4.

For this example we could determine which pixels to ignore by examining their colors. The *joint*, or *cross*, bilateral filter uses additional information, such as depths, normals, identification values, velocities, or other data to determine whether to use neighboring samples. For example, Ownby et al. [1343] show how patterns can occur when using just a few samples for shadow mapping. Blurring these results looks considerably better. However, a shadow on one object should not affect another unrelated model, and a blur will make shadows bleed outside the edges of the object. They use a bilateral filter, discarding samples on different surfaces by comparing the depth of a given pixel to that of its neighbors. Reducing variability in an area in this way is called *denoising*, and is commonly used in screen-space ambient occlusion algorithms (Section 11.3.6), for example [1971].

Using only the distance from the camera to find edges is often insufficient. For example, a soft shadow crossing the edge formed between two cube faces may fall on

only one face, the other facing away from the light. Using just the depth could cause the shadow to bleed from one face to the other when blurred, since this edge would not be detected. We can solve this problem by using only those neighbors where both the depth and surface normal are similar to those of the center sample. Doing so limits samples crossing shared edges, and so such bilateral filters are also called *edge-preserving filters*. Determining whether and how much to weaken or ignore the influence of a neighboring sample is up to the developer, and is dependent on such factors as the models, rendering algorithms, and viewing conditions.

Beyond the additional time spent examining neighbors and summing weights, bilateral filtering has other performance costs. Filtering optimizations such as two-pass separable filtering and bilinear interpolated weighted sampling are more difficult to use. We do not know in advance which samples should be disregarded or weakened in influence, so we cannot use techniques in which the GPU gathers multiple image texels in a single "tap." That said, the speed advantage of a separable two-pass filter has led to approximation methods [1396, 1971].

Paris et al. [1355] discuss many other applications of bilateral filters. Bilateral filters get applied wherever edges must be preserved but samples could be reused to reduce noise. They are also used to decouple shading frequency from the frequency at which geometry is rendered. For example, Yang et al. [1944] perform shading at a lower resolution, then using normals and depths, perform bilateral filtering during upsampling to form the final frame. An alternative is nearest-depth filtering, where the four samples in the low-resolution image are retrieved and the one whose depth is closest to the high-resolution image's depth is used [816]. Hennessy [717] and Pesce [1396] contrast and compare these and other methods for upsampling images. A problem with low-resolution rendering is that fine details can then be lost. Herzog et al. [733] further improve quality by exploiting temporal coherence and reprojection. Note that a bilateral filter is not separable, since the number of samples per pixel can vary. Green [589] notes that the artifacts from treating it as separable can be hidden by other shading effects.

A common way to implement a post-processing pipeline is to use *ping-pong buffers* [1303]. This is simply the idea of applying operations between two offscreen buffers, each used to hold intermediate or final results. For the first pass, the first buffer is the input texture and the second buffer is where output is sent. In the next pass the roles are reversed, with the second now acting as the input texture and the first getting reused for output. In this second pass the first buffer's original contents are overwritten—it is transient, being used as temporary storage for a processing pass. Managing and reusing transient resources is a critical element in designing a modern rendering system [1313]. Making each separate pass perform a particular effect is convenient from an architectural perspective. However, for efficiency, it is best to combine as many effects as possible in a single pass [1918].

In previous chapters, pixel shaders that access their neighbors were used for morphological antialiasing, soft shadows, screen-space ambient occlusion, and other techniques. Post-processing effects are generally run on the final image, and can imitate

Figure 12.5. Image processing using pixel shaders. The original image in the upper left is processed in various ways. The upper right shows a Gaussian difference operation, the lower left edge detection, and lower right a composite of the edge detection blended with the original image. *(Images courtesy of NVIDIA Corporation.)*

thermal imaging [734], reproduce film grain [1273] and chromatic aberration [539], perform edge detection [156, 506, 1218], generate heat shimmer [1273] and ripples [58], posterize an image [58], help render clouds [90], and perform a huge number of other operations [156, 539, 814, 1216, 1217, 1289]. Section 15.2.3 presents a few image processing techniques used for non-photorealistic rendering. See Figure 12.5 for but a few examples. These each use a color image as the only input.

Rather than continue with an exhaustive (and exhausting) litany of all possible algorithms, we end this chapter with some effects achieved using various billboard and image processing techniques.

12.2 Reprojection Techniques

Reprojection is based on the idea of reusing samples computed in previous frames. As its name implies, these samples are reused as possible from a new viewing location and orientation. One goal of reprojection methods is to amortize rendering cost over several frames, i.e., to exploit temporal coherence. Hence, this is also related to temporal antialiasing, covered in Section 5.4.2. Another goal is to form an approximate result if the application fails to finish rendering the current frame in time. This approach is particularly important in virtual reality applications, to avoid simulator sickness (Section 21.4.1).

Reprojection methods are divided into *reverse* reprojection and *forward* reprojection. The basic idea of reverse reprojection [1264, 1556] is shown in Figure 12.6. When a triangle is rendered at time t, the vertex positions are computed for both the current frame (t) and the previous ($t-1$). Using z and w from vertex shading, the pixel shader can compute an interpolated value z/w for both t and $t-1$, and if they are sufficiently close, a bilinear lookup at $\mathbf{p}_i^{t-1}$ can be done in the previous color buffer and that shaded value can be used instead of computing a new shaded value. For areas that previously were occluded, which then become visible (e.g., the dark green area in Figure 12.6), there are no shaded pixels available. This is called a *cache miss*. On such events, we compute new pixel shading to fill these holes. Since reuse

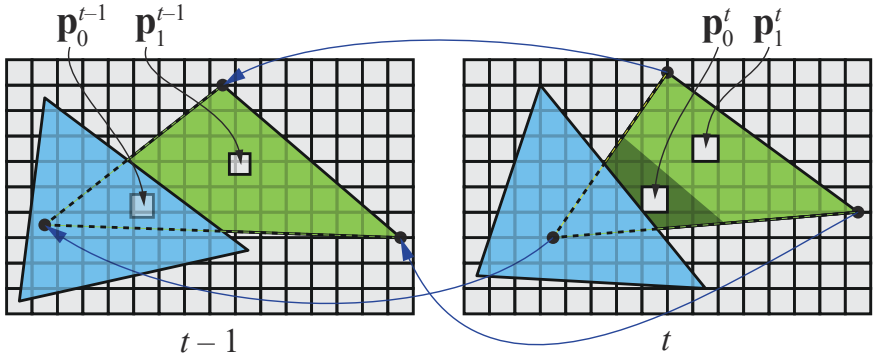

Figure 12.6. A green and a blue triangle at time $t-1$ and the frame after, at time t. The three-dimensional points $\mathbf{p}_i^t$ on the green triangle at the center of two pixels, together with pixel area, are reverse-reprojected to points $\mathbf{p}_i^{t-1}$. As can be seen, $\mathbf{p}_0^{t-1}$ is occluded while $\mathbf{p}_0^t$ is visible, in which case no shaded results can be reused. However, $\mathbf{p}_1$ is visible at both $t-1$ and t, and so shading can potentially be reused for that point. *(Illustration after Nehab et al. [1264].)*

of shaded values assumes that they are independent of any type of motion (objects, camera, light source), it is wise to not reuse shaded values over too many frames. Nehab et al. [1264] suggest that an automatic refresh should always happen after several frames of reuse. One way to do this is to divide the screen into n groups, where each group is a pseudo-random selection of 2×2 pixel regions. Each frame, a single group is updated, which avoids reusing pixel values for too long. Another variant of reverse reprojection is to store a velocity buffer and perform all testing in screen space, which avoids the double transform of vertices.

For better quality, one may also use a *running-average filter* [1264, 1556], which gradually phases out older values. These are particularly recommended for spatial antialiasing, soft shadows, and global illumination. The filter is described by

$$\mathbf{c}_f(\mathbf{p}^t) = \alpha \mathbf{c}(\mathbf{p}^t) + (1-\alpha)\mathbf{c}(\mathbf{p}^{t-1}), \tag{12.2}$$

where $\mathbf{c}(\mathbf{p}^t)$ is the newly shaded pixel value at $\mathbf{p}^t$, $\mathbf{c}(\mathbf{p}^{t-1})$ is the reverse-reprojected color from the previous frame, and $\mathbf{c}_f(\mathbf{p}^t)$ is the final color after applying the filter. Nehab et al. use $\alpha = 3/5$ for some use cases, but recommend that different values should be tried depending on what is rendered.

Forward reprojection instead works from the pixels of frame $t-1$ and projects them into frame t, which thus does not require double vertex shading. This means that the pixels from frame $t-1$ are *scattered* onto frame t, while reverse reprojection methods *gather* pixel values from frame $t-1$ to frame t. These methods also need to handle occluded areas that become visible, and this is often done with different heuristic hole-filling approaches, i.e., the values in the missing regions are inferred from the surrounding pixels. Yu et al. [1952] use forward reprojection to compute depth-of-field effects in an inexpensive manner. Instead of classic hole filling, Didyk et al. [350] avoid the holes by adaptively generating a grid over frame $t-1$ based on motion vectors. This grid is rendered with depth testing, projected into frame t, which means that occlusions and fold-overs are handled as part of rasterizing the adaptive grid triangles with depth testing. Didyk et al. use their method to reproject from the left eye to the right eye in order to generate a stereo pair for virtual reality, where the coherence normally is high between the two images. Later, Didyk et al. [351] present a perceptually motivated method to perform temporal upsampling, e.g., increasing frame rate from 40 Hz to 120 Hz.

Yang and Bowles [185, 1945, 1946] present methods that project from two frames at t and $t+1$ into a frame at $t+\delta t$ in between these two frames, i.e., $\delta t \in [0,1]$. These approaches have a greater chance of handling occlusion situations better because they use two frames instead of just one. Such methods are used in games to increase frame rate from 30 FPS to 60 FPS, which is possible since their method runs in less than 1 ms. We recommend their course notes [1946] and the wide-ranging survey on temporal coherence methods by Scherzer et al. [1559]. Valient [1812] also uses reprojection to speed up rendering in *Killzone: Shadow Fall*. See the numerous references near the end of Section 5.4.2 for implementation details in using reprojection for temporal antialiasing.

12.3 Lens Flare and Bloom

Lens flare is a phenomenon caused by light that travels through a lens system or the eye by indirect reflection or other unintended paths. Flare can by classified by several phenomena, most significantly a halo and a ciliary corona. The halo is caused by the radial fibers of the crystalline structure of the lens. It looks like a ring around the light, with its outside edge tinged with red, and its inside with violet. The halo has a constant apparent size, regardless of the distance of the source. The ciliary corona comes from density fluctuations in the lens, and appears as rays radiating from a point, which may extend beyond the halo [1683].

Camera lenses can also create secondary effects when parts of the lens reflect or refract light internally. For example, polygonal patterns can appear due to a camera's aperture blades. Streaks of light can also be seen to smear across a windshield, due to small grooves in the glass [1303]. Bloom is caused by scattering in the lens and other parts of the eye, creating a glow around the light and dimming contrast elsewhere in the scene. A video camera captures an image by converting photons to charge using a *charge-coupled device* (CCD). Bloom occurs in a video camera when a charge site in the CCD gets saturated and overflows into neighboring sites. As a class, halos, coronae, and bloom are called *glare effects*.

In reality, most such artifacts are seen less and less as camera technology improves. Better designs, lens hoods, and anti-reflective coatings can reduce or eliminate these stray ghosting artifacts [598, 786]. However, these effects are now routinely added digitally to real photos. Because there are limits to the light intensity produced by a computer monitor, we can give the impression of increased brightness in a scene or from objects by adding such effects to our images [1951]. The bloom effect and lens flare are almost clichés in photos, films, and interactive computer graphics, due to their common use. Nonetheless, when skillfully employed, such effects can give strong visual cues to the viewer.

To provide a convincing effect, the lens flare should change with the position of the light source. King [899] creates a set of squares with different textures to represent the lens flare. These are then oriented on a line going from the light source position on screen through the screen's center. When the light is far from the center of the screen, these squares are small and more transparent, becoming larger and more opaque as the light moves inward. Maughan [1140] varies the brightness of a lens flare by using the GPU to compute the occlusion of an onscreen area light source. He generates a single-pixel intensity texture that is then used to attenuate the brightness of the effect. Sekulic [1600] renders the light source as a single polygon, using occlusion query hardware to give a pixel count of the area visible (Section 19.7.1). To avoid GPU stalls from waiting for the query to return a value to the CPU, the result is used in the next frame to determine the amount of attenuation. Since the intensity is likely to vary in a fairly continuous and predictable fashion, a single frame of delay causes little perceptual confusion. Gjøl and Svendsen [539] first generate a depth buffer (which they use for other effects as well) and sample it 32 times in a spiral

Figure 12.7. Lens flare, star glare, and bloom effects, along with depth of field and motion blur [1208]. Note the strobing artifacts on some moving balls due to accumulating separate images. *(Image from "Rthdribl," by Masaki Kawase.)*

pattern in the area where the lens flare will appear, using the result to attenuate the flare texture. The visibility sampling is done in the vertex shader while rendering the flare geometry, thus avoiding the delay caused by hardware occlusion queries.

Streaks from bright objects or lights in a scene can be performed in a similar fashion by either drawing semitransparent billboards or performing post-processing filtering on the bright pixels themselves. Games such as *Grand Theft Auto V* use a set of textures applied to billboard for these and other effects [293].

Oat [1303] discusses using a *steerable filter* to produce the streak effect. Instead of filtering symmetrically over an area, this type of filter is given a direction. Texel values along this direction are summed together, which produces a streak effect. Using an image downsampled to one quarter of the width and height, and two passes using ping-pong buffers, gives a convincing streak effect. Figure 12.7 shows an example of this technique.

Figure 12.8. The process of generating sun flare in the game *The Witcher 3*. First, a high-contrast correction curve is applied to the input image to isolate the unoccluded parts of the sun. Next, radial blurs, centered on the sun, are applied to the image. Shown on the left, the blurs are performed in a series, each operating on the output of the previous one. Doing so creates a smooth, high-quality blur, while using a limited number of samples in each pass to improve efficiency. All blurs are performed at half resolution to decrease the runtime cost. The final image of the flare is combined additively with the original scene rendering. *(CD PROJEKT®, The Witcher® are registered trademarks of CD PROJEKT Capital Group. The Witcher game© CD PROJEKT S.A. Developed by CD PROJEKT S.A. All rights reserved. The Witcher game is based on the prose of Andrzej Sapkowski. All other copyrights and trademarks are the property of their respective owners.)*

Many other variations and techniques exist, moving well beyond billboarding. Mittring [1229] uses image processing to isolate bright parts, downsample them, and blur them in several textures. They are then composited again over the final image by duplicating, scaling, mirroring, and tinting. Using this approach, it is not possible for artists to control the look of each flare source independently: The same process is applied to each of the flares. However, any bright parts of the image can generate lens flares, such as specular reflections or emissive parts of a surface, or bright spark particles. Wronski [1919] describes anamorphic lens flares, a byproduct of cinematography equipment used in the 1950s. Hullin et al. [598, 786] provide a physical model for various ghosting artifacts, tracing bundles of rays to compute effects. It gives plausible results that are based on the design of the lens system, along with trade-offs between accuracy and performance. Lee and Eisemann [1012] build on this work, using a linear model that avoids expensive preprocesses. Hennessy [716] gives implementation details. Figure 12.8 shows a typical lens flare system used in production.

The bloom effect, where an extremely bright area spills over onto adjoining pixels, is performed by combining several techniques already presented. The main idea is to

create a bloom image consisting only of the bright objects that are to be "overexposed," blur this, then composite it back into the normal image. The blur used is typically a Gaussian [832], though recent matching to reference shots shows that the distribution has more of a spike shape [512]. A common method for making this image is to *bright-pass filter*: Any bright pixels are retained, and all dim pixels are made black, often with some blend or scaling at the transition point [1616, 1674]. For bloom on just a few small objects, a screen bounding box can be computed to limit the extent of the post-processing blur and composite passes [1859].

This bloom image can be rendered at a low resolution, e.g., anywhere from one half to one eighth of the width and height of the original. Doing so saves time and can help increase the effect of filtering. This lower-resolution image is blurred and combined with the original. This reduction in resolution is used in many post-processing effects, as is the technique of compressing or otherwise reducing color resolution [1877]. The bloom image can be downsampled several times and resampled from the set of images produced, giving a wider blur effect while minimizing sampling costs [832, 1391, 1918]. For example, a single bright pixel moving across the screen may cause flicker, as it may not be sampled in some frames.

Because the goal is an image that looks overexposed where it is bright, this image's colors are scaled as desired and added to the original image. Additive blending saturates a color and then goes to white, which is usually just what is desired. An example is shown in Figure 12.9. Alpha blending could be used for more artistic control [1859]. Instead of thresholding, high dynamic range imagery can be filtered for a better result [512, 832]. Low and high dynamic range blooms can be computed separately and composited, to capture different phenomena in a more convincing manner [539]. Other variants are possible, e.g., the previous frame's results can also be added to the current frame, giving animated objects a streaking glow [815].

12.4 Depth of Field

For a camera lens at a given setting, there is a range where objects are in focus, its *depth of field*. Objects outside of this range are blurry—the further outside, the blurrier. In photography, this blurriness is related to aperture size and focal length. Reducing the aperture size increases the depth of field, i.e., a wider range of depths are in focus, but decreases the amount of light forming the image (Section 9.2). A photo taken in an outdoor daytime scene typically has a large depth of field because the amount of light is sufficient to allow a small aperture size, ideally a pinhole camera. Depth of field narrows considerably inside a poorly lit room. So, one way to control a depth-of-field effect is to have it tied to tone mapping, making out-of-focus objects blurrier as the light level decreases. Another is to permit manual artistic control, changing focus and increasing depth of field for dramatic effect as desired. See an example in Figure 12.10.

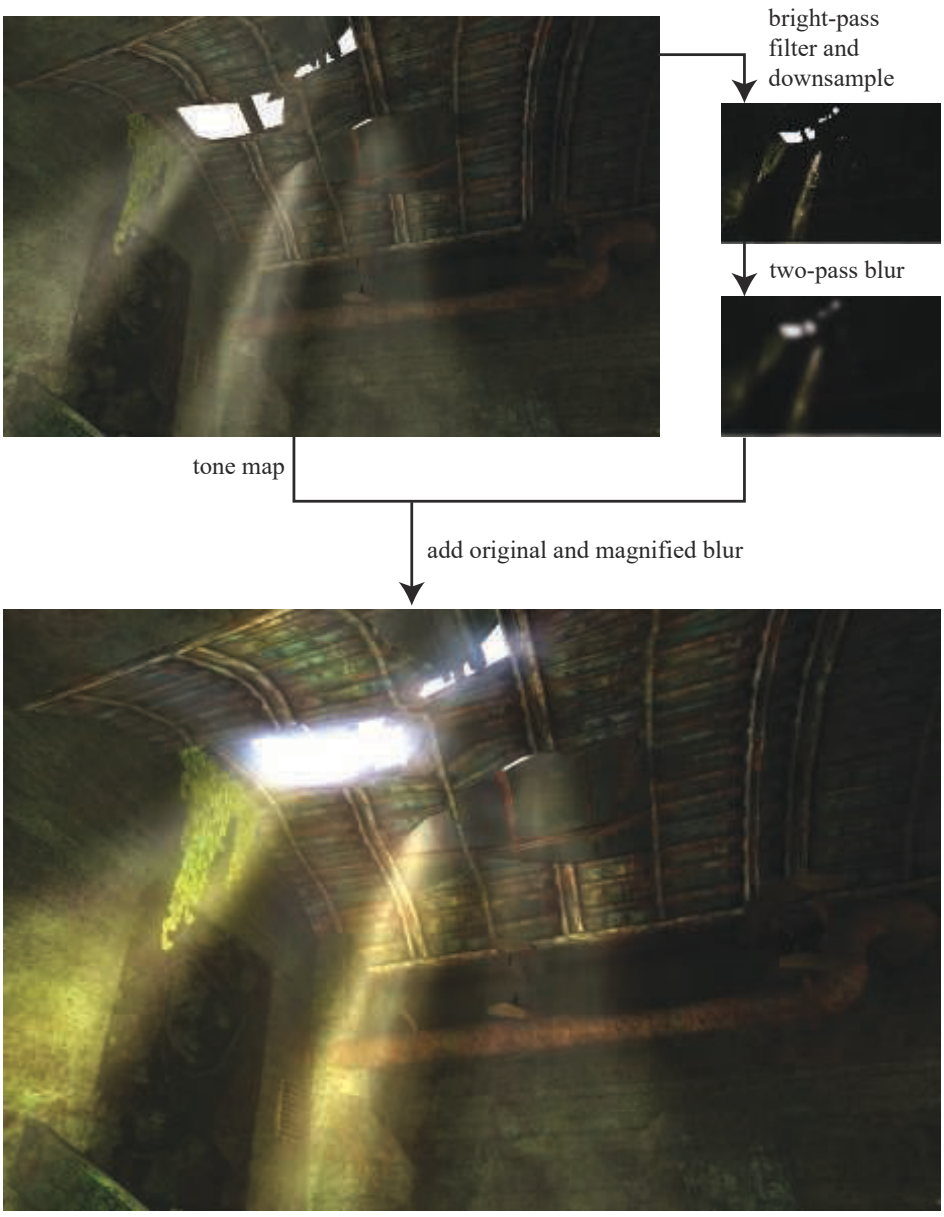

Figure 12.9. High dynamic range tone mapping and bloom. The lower image is produced by using tone mapping on, and adding a post-process bloom to, the original image [1869]. *(Image from "Far Cry," courtesy of Ubisoft.)*

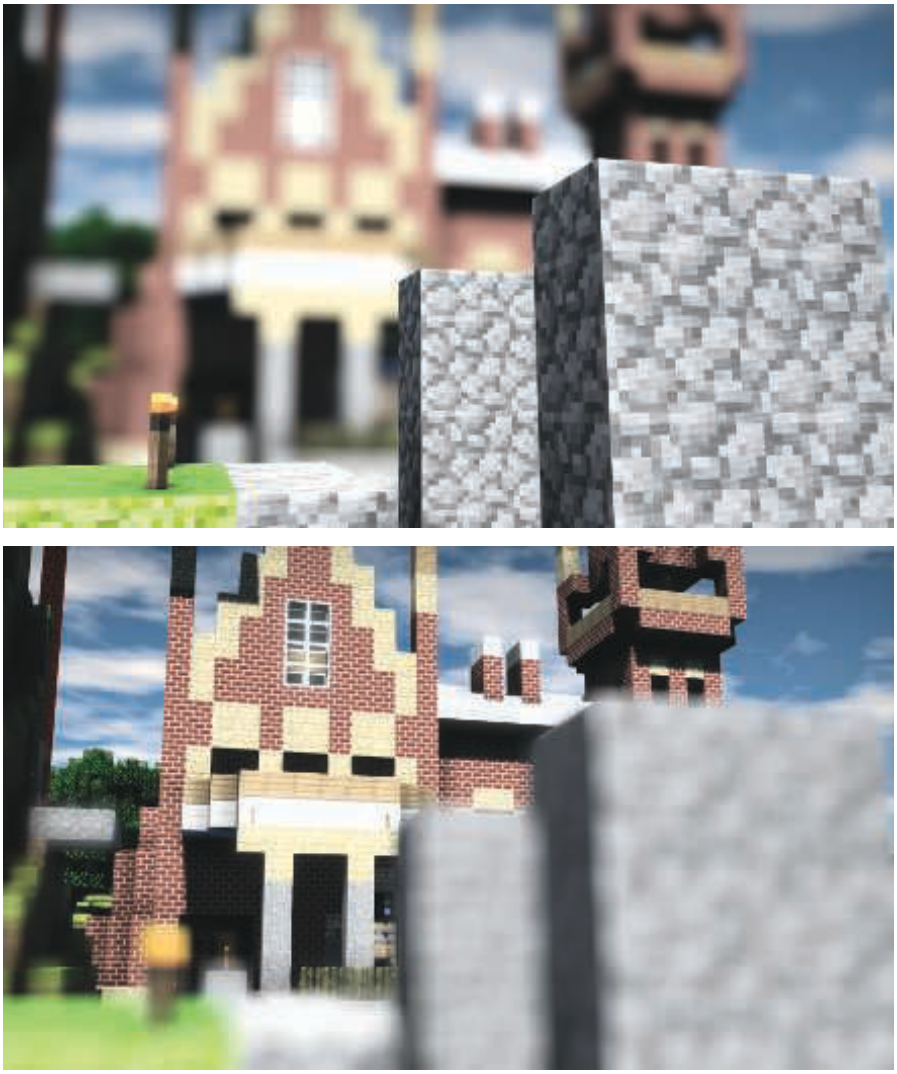

Figure 12.10. Depth of field depends on the camera's focus. *(Images rendered in G3D, courtesy of Morgan McGuire [209, 1178].)*

An accumulation buffer can be used to simulate depth of field [637]. See Figure 12.11. By varying the view position on the lens and keeping the point of focus fixed, objects will be rendered blurrier relative to their distance from this focal point. However, as with other accumulation effects, this method comes at a high cost of multiple renderings per image. That said, it does converge to the correct ground-truth