viewpoint into an otherwise white texture. This texture can then be projected onto the surfaces that are to receive the shadow. Effectively, each vertex on the receivers has a (u, v) texture coordinate computed for it and has the texture applied to it. These texture coordinates can be computed explicitly by the application. This differs a bit from the ground shadow texture in the previous section, where objects are projected onto a specific physical plane. Here, the image is made as a view from the light, like a frame of film in a projector.

When rendered, the projected shadow texture modifies the receiver surfaces. It can also be combined with other shadow methods, and sometimes is used primarily for helping aid perception of an object's location. For example, in a platform-hopping video game, the main character might always be given a drop shadow directly below it, even when the character is in full shadow [1343]. More elaborate algorithms can give better results. For example, Eisemann and Décoret [411] assume a rectangular overhead light and create a stack of shadow images of horizontal slices of the object, which are then turned into mipmaps or similar. The corresponding area of each slice is accessed proportional to its distance from the receiver by using its mipmap, meaning that more distant slices will cast softer shadows.

There are some serious drawbacks of texture projection methods. First, the application must identify which objects are occluders and which are their receivers. The receiver must be maintained by the program to be further from the light than the occluder, otherwise the shadow is "cast backward." Also, occluding objects cannot shadow themselves. The next two sections present algorithms that generate correct shadows without the need for such intervention or limitations.

Note that a variety of lighting patterns can be obtained by using prebuilt projective textures. A spotlight is simply a square projected texture with a circle inside of it defining the light. A Venetian blinds effect can be created by a projected texture consisting of horizontal lines. This type of texture is called a *light attenuation mask*, *cookie texture*, or *gobo map*. A prebuilt pattern can be combined with a projected texture created on the fly by simply multiplying the two textures together. Such lights are discussed further in Section 6.9.

7.3 Shadow Volumes

Presented by Heidmann in 1991 [701], a method based on Crow's *shadow volumes* [311] can cast shadows onto arbitrary objects by clever use of the stencil buffer. It can be used on any GPU, as the only requirement is a stencil buffer. It is not image based (unlike the shadow map algorithm described next) and so avoids sampling problems, thus producing correct sharp shadows everywhere. This can sometimes be a disadvantage. For example, a character's clothing may have folds that give thin, hard shadows that alias badly. Shadow volumes are rarely used today, due to their unpredictable cost [1599]. We give the algorithm a brief description here, as it illustrates some important principles and research based on these continues.

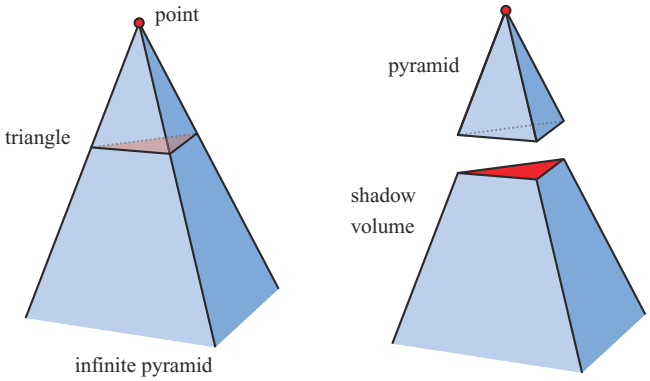

Figure 7.7. Left: the lines from a point light are extended through the vertices of a triangle to form an infinite pyramid. Right: the upper part is a pyramid, and the lower part is an infinite truncated pyramid, also called the shadow volume. All geometry that is inside the shadow volume is in shadow.

To begin, imagine a point and a triangle. Extending the lines from a point through the vertices of a triangle to infinity yields an infinite three-sided pyramid. The part under the triangle, i.e., the part that does not include the point, is a truncated infinite pyramid, and the upper part is simply a pyramid. This is illustrated in Figure 7.7. Now imagine that the point is actually a point light source. Then, any part of an object that is inside the volume of the truncated pyramid (under the triangle) is in shadow. This volume is called a *shadow volume*.

Say we view some scene and follow a ray from the eye through a pixel until the ray hits the object to be displayed on screen. While the ray is on its way to this object, we increment a counter each time it crosses a face of the shadow volume that is frontfacing (i.e., facing toward the viewer). Thus, the counter is incremented each time the ray goes into shadow. In the same manner, we decrement the same counter each time the ray crosses a backfacing face of the truncated pyramid. The ray is then going out of a shadow. We proceed, incrementing and decrementing the counter until the ray hits the object that is to be displayed at that pixel. If the counter is greater than zero, then that pixel is in shadow; otherwise it is not. This principle also works when there is more than one triangle that casts shadows. See Figure 7.8.

Doing this with rays is time consuming. But there is a much smarter solution [701]: A stencil buffer can do the counting for us. First, the stencil buffer is cleared. Second, the whole scene is drawn into the framebuffer with only the color of the unlit material used, to get these shading components in the color buffer and the depth information into the z-buffer. Third, z-buffer updates and writing to the color buffer are turned off (though z-buffer testing is still done), and then the frontfacing triangles of the shadow volumes are drawn. During this process, the stencil operation is set to increment the values in the stencil buffer wherever a triangle is drawn. Fourth, another pass is done with the stencil buffer, this time drawing only the backfacing triangles of the shadow volumes. For this pass, the values in the stencil buffer are decremented when the

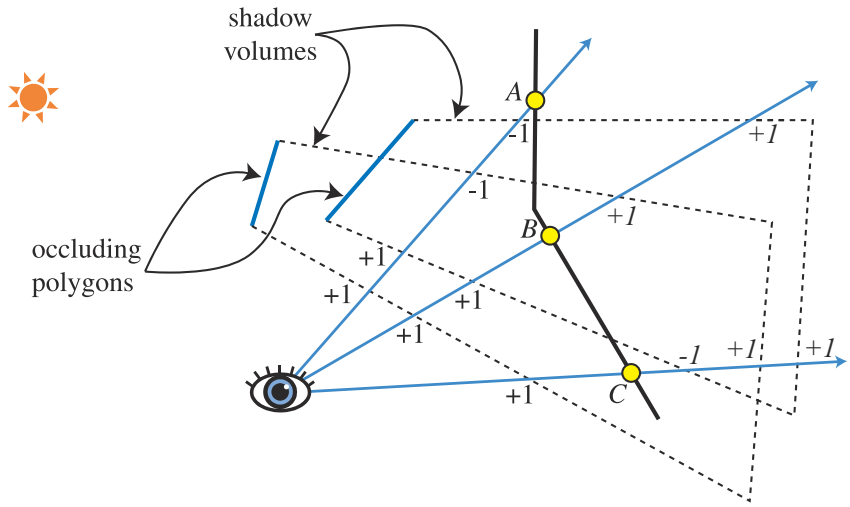

Figure 7.8. A two-dimensional side view of counting shadow-volume crossings using two different counting methods. In z-pass volume counting, the count is incremented as a ray passes through a frontfacing triangle of a shadow volume and decremented on leaving through a backfacing triangle. So, at point A, the ray enters two shadow volumes for $+2$, then leaves two volumes, leaving a net count of zero, so the point is in light. In z-fail volume counting, the count starts beyond the surface (these counts are shown in italics). For the ray at point B, the z-pass method gives a $+2$ count by passing through two frontfacing triangles, and the z-fail gives the same count by passing through two backfacing triangles. Point C shows how z-fail shadow volumes must be capped. The ray starting from point C first hits a frontfacing triangle, giving -1. It then exits two shadow volumes (through their endcaps, necessary for this method to work properly), giving a net count of $+1$. The count is not zero, so the point is in shadow. Both methods always give the same count results for all points on the viewed surfaces.

triangles are drawn. Incrementing and decrementing are done only when the pixels of the rendered shadow-volume face are visible (i.e., not hidden by any real geometry). At this point the stencil buffer holds the state of shadowing for every pixel. Finally, the whole scene is rendered again, this time with only the components of the active materials that are affected by the light, and displayed only where the value in the stencil buffer is 0. A value of 0 indicates that the ray has gone out of shadow as many times as it has gone into a shadow volume—i.e., this location is illuminated by the light.

This counting method is the basic idea behind shadow volumes. An example of shadows generated by the shadow volume algorithm is shown in Figure 7.9. There are efficient ways to implement the algorithm in a single pass [1514]. However, counting problems will occur when an object penetrates the camera's near plane. The solution, called *z-fail*, involves counting the crossings hidden behind visible surfaces instead of in front [450, 775]. A brief summary of this alternative is shown in Figure 7.8.

Creating quadrilaterals for every triangle creates a huge amount of overdraw. That is, each triangle will create three quadrilaterals that must be rendered. A sphere

Figure 7.9. Shadow volumes. On the left, a character casts a shadow. On the right, the extruded triangles of the model are shown. *(Images from Microsoft SDK [1208] sample "ShadowVolume.")*

made of a thousand triangles creates three thousand quadrilaterals, and each of those quadrilaterals may span the screen. One solution is to draw only those quadrilaterals along the silhouette edges of the object, e.g., our sphere may have only fifty silhouette edges, so only fifty quadrilaterals are needed. The geometry shader can be used to automatically generate such silhouette edges [1702]. Culling and clamping techniques can also be used to lower fill costs [1061].

However, the shadow volume algorithm still has a terrible drawback: extreme variability. Imagine a single, small triangle in view. If the camera and the light are in exactly the same position, the shadow-volume cost is minimal. The quadrilaterals formed will not cover any pixels as they are edge-on to the view. Only the triangle itself matters. Say the viewer now orbits around the triangle, keeping it in view. As the camera moves away from the light source, the shadow-volume quadrilaterals will become more visible and cover more of the screen, causing more computation to occur. If the viewer should happen to move into the shadow of the triangle, the shadow volume will entirely fill the screen, costing a considerable amount of time to evaluate compared to our original view. This variability is what makes shadow volumes unusable in interactive applications where a consistent frame rate is important. Viewing toward the light can cause huge, unpredictable jumps in the cost of the algorithm, as can other scenarios.

For these reasons shadow volumes have been for the most part abandoned by applications. However, given the continuing evolution of new and different ways to access data on the GPU, and the clever repurposing of such functionality by researchers, shadow volumes may someday come back into general use. For example, Sintorn et al. [1648] give an overview of shadow-volume algorithms that improve efficiency and propose their own hierarchical acceleration structure.

The next algorithm presented, shadow mapping, has a much more predictable cost and is well suited to the GPU, and so forms the basis for shadow generation in many applications.

7.4 Shadow Maps

In 1978, Williams [1888] proposed that a common z-buffer-based renderer could be used to generate shadows quickly on arbitrary objects. The idea is to render the scene, using the z-buffer, from the position of the light source that is to cast shadows. Whatever the light "sees" is illuminated, the rest is in shadow. When this image is generated, only z-buffering is required. Lighting, texturing, and writing values into the color buffer can be turned off.

Each pixel in the z-buffer now contains the z-depth of the object closest to the light source. We call the entire contents of the z-buffer the *shadow map*, also sometimes known as the *shadow depth map* or *shadow buffer*. To use the shadow map, the scene is rendered a second time, but this time with respect to the viewer. As each drawing primitive is rendered, its location at each pixel is compared to the shadow map. If a rendered point is farther away from the light source than the corresponding value in the shadow map, that point is in shadow, otherwise it is not. This technique is implemented by using texture mapping. See Figure 7.10. Shadow mapping is a popular algorithm because it is relatively predictable. The cost of building the shadow map is roughly linear with the number of rendered primitives, and access time is constant. The shadow map can be generated once and reused each frame for scenes where the light and objects are not moving, such as for computer-aided design.

When a single z-buffer is generated, the light can "look" in only a particular direction, like a camera. For a distant directional light such as the sun, the light's view is set to encompass all objects casting shadows into the viewing volume that the eye sees. The light uses an orthographic projection, and its view needs to be made wide and high enough in x and y to view this set of objects. Local light sources need similar adjustments, as possible. If the local light is far enough away from the shadow-casting objects, a single view frustum may be sufficient to encompass all of these. Alternately, if the local light is a spotlight, it has a natural frustum associated with it, with everything outside its frustum considered not illuminated.

If the local light source is inside a scene and is surrounded by shadow-casters, a typical solution is to use a six-view cube, similar to cubic environment mapping [865]. These are called *omnidirectional shadow maps*. The main challenge for omnidirectional maps is avoiding artifacts along the seams where two separate maps meet. King and Newhall [895] analyze the problems in depth and provide solutions, and Gerasimov [525] provides some implementation details. Forsyth [484, 486] presents a general multi-frustum partitioning scheme for omnidirectional lights that also provides more shadow map resolution where needed. Crytek [1590, 1678, 1679] sets the resolution of each of the six views for a point light based on the screen-space coverage of each view's projected frustum, with all maps stored in a texture atlas.

Not all objects in the scene need to be rendered into the light's view volume. First, only objects that can cast shadows need to be rendered. For example, if it is known that the ground can only receive shadows and not cast one, then it does not have to be rendered into the shadow map.

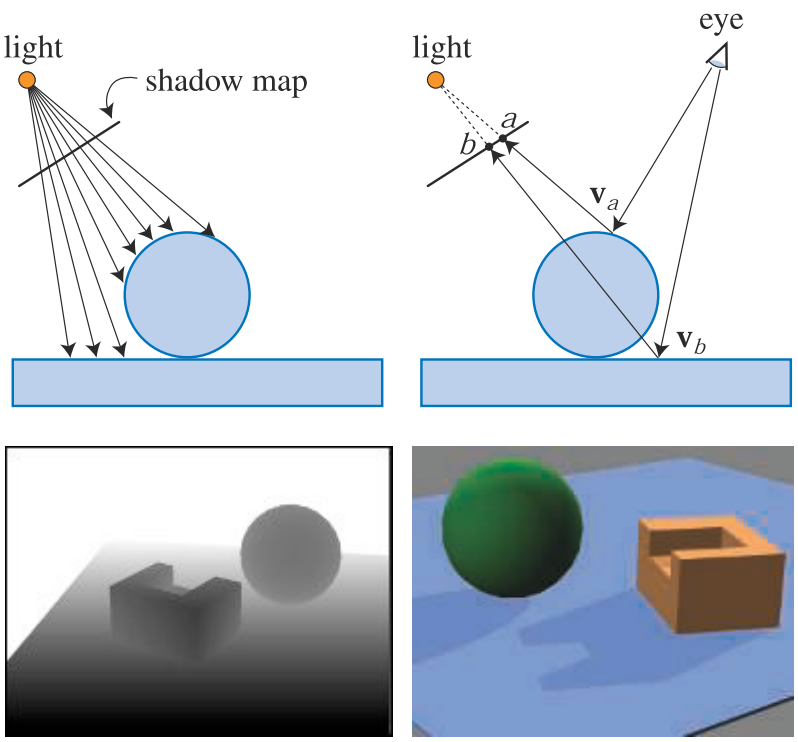

Figure 7.10. Shadow mapping. On the top left, a shadow map is formed by storing the depths to the surfaces in view. On the top right, the eye is shown looking at two locations. The sphere is seen at point $\mathbf{v}_a$, and this point is found to be located at texel a on the shadow map. The depth stored there is not (much) less than point $\mathbf{v}_a$ is from the light, so the point is illuminated. The rectangle hit at point $\mathbf{v}_b$ is (much) farther away from the light than the depth stored at texel b, and so is in shadow. On the bottom left is the view of a scene from the light's perspective, with white being farther away. On the bottom right is the scene rendered with this shadow map.

Shadow casters are by definition those inside the light's view frustum. This frustum can be augmented or tightened in several ways, allowing us to safely disregard some shadow casters [896, 1812]. Think of the set of shadow receivers visible to the eye. This set of objects is within some maximum distance along the light's view direction. Anything beyond this distance cannot cast a shadow on the visible receivers. Similarly, the set of visible receivers may well be smaller than the light's original x and y view bounds. See Figure 7.11. Another example is that if the light source is inside the eye's view frustum, no object outside this additional frustum can cast a shadow on a receiver. Rendering only relevant objects not only can save time rendering, but can also reduce the size required for the light's frustum and so increase the effective resolution of the shadow map, thus improving quality. In addition, it helps if the light frustum's near plane is as far away from the light as possible, and if the far plane is

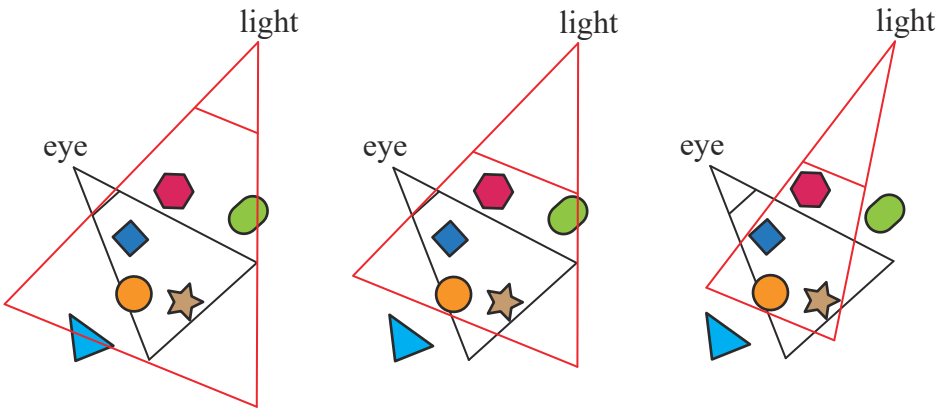

Figure 7.11. On the left, the light's view encompasses the eye's frustum. In the middle, the light's far plane is pulled in to include only visible receivers, so culling the triangle as a caster; the near plane is also adjusted. On the right, the light's frustum sides are made to bound the visible receivers, culling the green capsule.

as close as possible. Doing so increases the effective precision of the z-buffer [1792] (Section 4.7.2).

One disadvantage of shadow mapping is that the quality of the shadows depends on the resolution (in pixels) of the shadow map and on the numerical precision of the z-buffer. Since the shadow map is sampled during the depth comparison, the algorithm is susceptible to aliasing problems, especially close to points of contact between objects. A common problem is *self-shadow aliasing*, often called "surface acne" or "shadow acne," in which a triangle is incorrectly considered to shadow itself. This problem has two sources. One is simply the numerical limits of precision of the processor. The other source is geometric, from the fact that the value of a point sample is being used to represent an area's depth. That is, samples generated for the light are almost never at the same locations as the screen samples (e.g., pixels are often sampled at their centers). When the light's stored depth value is compared to the viewed surface's depth, the light's value may be slightly lower than the surface's, resulting in self-shadowing. The effects of such errors are shown in Figure 7.12.

One common method to help avoid (but not always eliminate) various shadow-map artifacts is to introduce a bias factor. When checking the distance found in the shadow map with the distance of the location being tested, a small bias is subtracted from the receiver's distance. See Figure 7.13. This bias could be a constant value [1022], but doing so can fail when the receiver is not mostly facing the light. A more effective method is to use a bias that is proportional to the angle of the receiver to the light. The more the surface tilts away from the light, the greater the bias grows, to avoid the problem. This type of bias is called the *slope scale bias*. Both biases can be applied by using a command such as OpenGL's `glPolygonOffset`) to shift each polygon away from the light. Note that if a surface directly faces the light, it is not biased backward

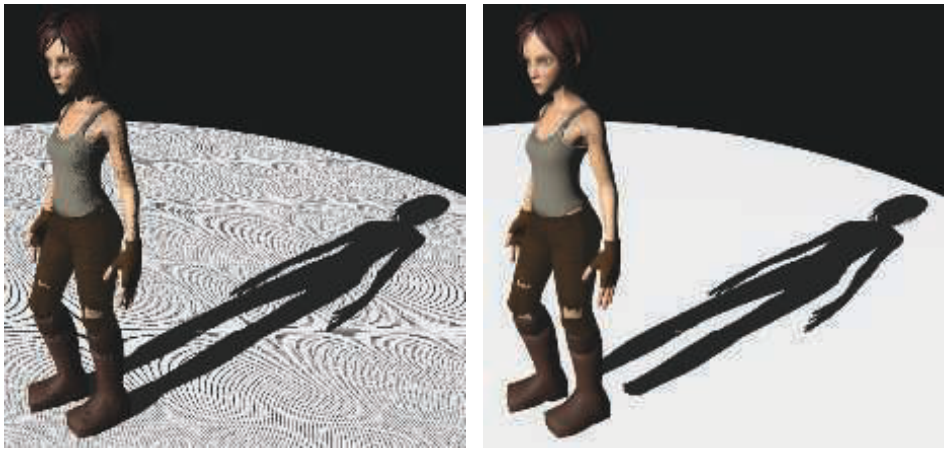

Figure 7.12. Shadow-mapping bias artifacts. On the left, the bias is too low, so self-shadowing occurs. On the right, a high bias causes the shoes to not cast contact shadows. The shadow-map resolution is also too low, giving the shadow a blocky appearance. *(Images generated using a shadow demo by Christoph Peters.)*

at all by slope scale bias. For this reason, a constant bias is used along with slope scale bias to avoid possible precision errors. Slope scale bias is also often clamped at some maximum, since the tangent value can be extremely high when the surface is nearly edge-on when viewed from the light.

Figure 7.13. Shadow bias. The surfaces are rendered into a shadow map for an overhead light, with the vertical lines representing shadow-map pixel centers. Occluder depths are recorded at the × locations. We want to know if the surface is lit at the three samples shown as dots. The closest shadow-map depth value for each is shown with the same color ×. On the left, if no bias is added, the blue and orange samples will be incorrectly determined to be in shadow, since they are farther from the light than their corresponding shadow-map depths. In the middle, a constant depth bias is subtracted from each sample, placing each closer to the light. The blue sample is still considered in shadow because it is not closer to the light than the shadow-map depth it is tested against. On the right, the shadow map is formed by moving each polygon away from the light proportional to its slope. All sample depths are now closer than their shadow-map depths, so all are lit.

Holbert [759, 760] introduced *normal offset bias*, which first shifts the receiver's world-space location a bit along the surface's normal direction, proportional to the sine of the angle between the light's direction and the geometric normal. See Figure 7.24 on page 250. This changes not only the depth but also the x- and y-coordinates where the sample is tested on the shadow map. As the light's angle becomes more shallow to the surface, this offset is increased, in hopes that the sample becomes far enough above the surface to avoid self-shadowing. This method can be visualized as moving the sample to a "virtual surface" above the receiver. This offset is a world-space distance, so Pettineo [1403] recommends scaling it by the depth range of the shadow map. Pesce [1391] suggests the idea of biasing along the camera view direction, which also works by adjusting the shadow-map coordinates. Other bias methods are discussed in Section 7.5, as the shadow method presented there needs to also test several neighboring samples.

Too much bias causes a problem called *light leaks* or *Peter Panning*, in which the object appears to float slightly above the underlying surface. This artifact occurs because the area beneath the object's point of contact, e.g., the ground under a foot, is pushed too far forward and so does not receive a shadow.

One way to avoid self-shadowing problems is to render only the backfaces to the shadow map. Called *second-depth shadow mapping* [1845], this scheme works well for many situations, especially for a rendering system where hand-tweaking a bias is not an option. The problem cases occur when objects are two-sided, thin, or in contact with one another. If an object is a model where both sides of the mesh are visible, e.g., a palm frond or sheet of paper, self-shadowing can occur because the backface and the frontface are in the same location. Similarly, if no biasing is performed, problems can occur near silhouette edges or thin objects, since in these areas backfaces are close to frontfaces. Adding a bias can help avoid surface acne, but the scheme is more susceptible to light leaking, as there is no separation between the receiver and the backfaces of the occluder at the point of contact. See Figure 7.14. Which scheme to choose can be situation dependent. For example, Sousa et al. [1679] found using frontfaces for sun shadows and backfaces for interior lights to work best for their applications.

Note that for shadow mapping, objects must be "watertight" (manifold and closed, i.e., solid; Section 16.3.3), or must have both front- and backfaces rendered to the map, else the object may not fully cast a shadow. Woo [1900] proposes a general method that attempts to, literally, be a happy medium between using just frontfaces or backfaces for shadowing. The idea is to render solid objects to a shadow map and keep track of the two closest surfaces to the light. This process can be performed by depth peeling or other transparency-related techniques. The average depth between the two objects forms an intermediate layer whose depth is used as a shadow map, sometimes called a *dual shadow map* [1865]. If the object is thick enough, self-shadowing and light-leak artifacts are minimized. Bavoil et al. [116] discuss ways to address potential artifacts, along with other implementation details. The main drawbacks are the additional costs

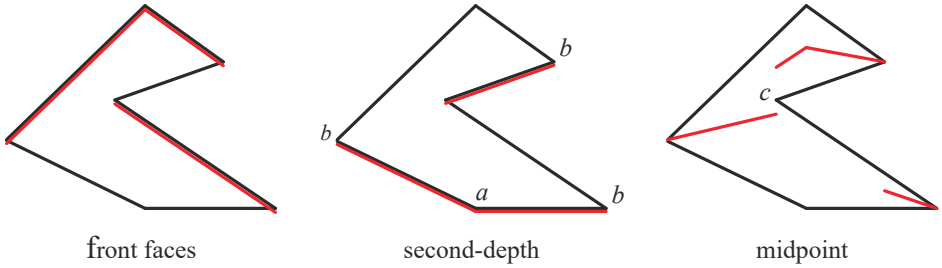

Figure 7.14. Shadow-map surfaces for an overhead light source. On the left, surfaces facing the light, marked in red, are sent to the shadow map. Surfaces may be incorrectly determined to shadow themselves ("acne"), so need to be biased away from the light. In the middle, only the backfacing triangles are rendered into the shadow map. A bias pushing these occluders downward could let light leak onto the ground plane near location a; a bias forward can cause illuminated locations near the silhouette boundaries marked b to be considered in shadow. On the right, an intermediate surface is formed at the midpoints between the closest front- and backfacing triangles found at each location on the shadow map. A light leak can occur near point c (which can also happen with second-depth shadow mapping), as the nearest shadow-map sample may be on the intermediate surface to the left of this location, and so the point would be closer to the light.

associated with using two shadow maps. Myers [1253] discusses an artist-controlled depth layer between the occlude and receiver.

As the viewer moves, the light's view volume often changes size as the set of shadow casters changes. Such changes in turn cause the shadows to shift slightly from frame to frame. This occurs because the light's shadow map is sampling a different set of directions from the light, and these directions are not aligned with the previous set. For directional lights, the solution is to force each succeeding shadow map generated to maintain the same relative texel beam locations in world space [927, 1227, 1792, 1810]. That is, you can think of the shadow map as imposing a two-dimensional gridded frame of reference on the whole world, with each grid cell representing a pixel sample on the map. As you move, the shadow map is generated for a different set of these same grid cells. In other words, the light's view projection is forced to this grid to maintain frame to frame coherence.

7.4.1 Resolution Enhancement

Similar to how textures are used, ideally we want one shadow-map texel to cover about one image pixel. If we have a light source located at the same position as the eye, the shadow map perfectly maps one-to-one with the screen-space pixels (and there are no visible shadows, since the light illuminates exactly what the eye sees). As soon as the light's direction changes, this per-pixel ratio changes, which can cause artifacts. An example is shown in Figure 7.15. The shadow is blocky and poorly defined because a large number of pixels in the foreground are associated with each texel of the shadow map. This mismatch is called *perspective aliasing*. Single shadow-map texels can also

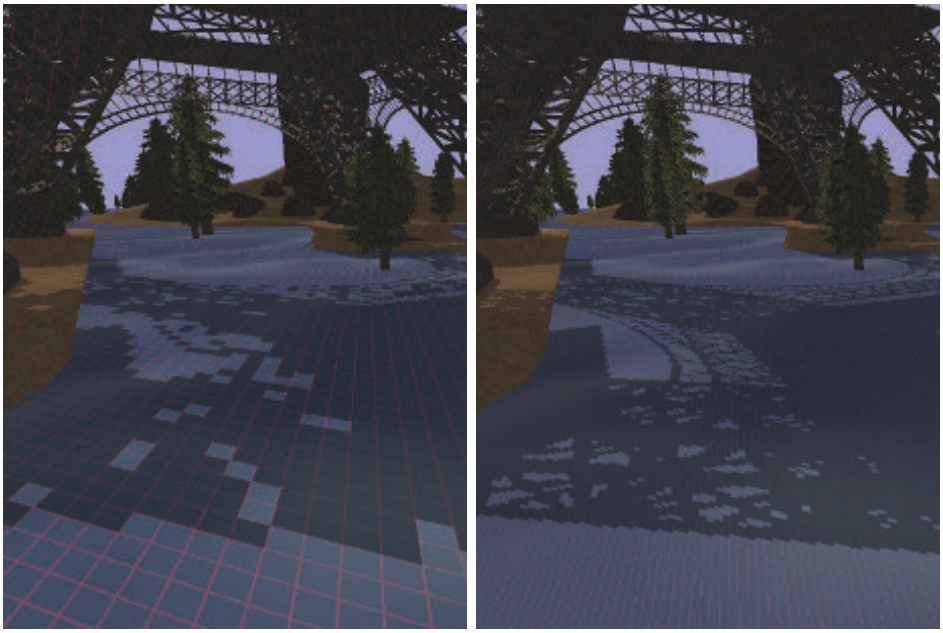

Figure 7.15. The image to the left is created using standard shadow mapping; the image to the right using LiSPSM. The projections of each shadow map's texels are shown. The two shadow maps have the same resolution, the difference being that LiSPSM reforms the light's matrices to provide a higher sampling rate nearer the viewer. *(Images courtesy of Daniel Scherzer, Vienna University of Technology.)*

cover many pixels if a surface is nearly edge-on to the light, but faces the viewer. This problem is known as *projective aliasing* [1792]; see Figure 7.16. Blockiness can be decreased by increasing the shadow-map resolution, but at the cost of additional memory and processing.

There is another approach to creating the light's sampling pattern that makes it more closely resemble the camera's pattern. This is done by changing the way the scene projects toward the light. Normally we think of a view as being symmetric, with the view vector in the center of the frustum. However, the view direction merely defines a view plane, but not which pixels are sampled. The window defining the frustum can be shifted, skewed, or rotated on this plane, creating a quadrilateral that gives a different mapping of world to view space. The quadrilateral is still sampled at regular intervals, as this is the nature of a linear transform matrix and its use by the GPU. The sampling rate can be modified by varying the light's view direction and the view window's bounds. See Figure 7.17.

There are 22 degrees of freedom in mapping the light's view to the eye's [896]. Exploration of this solution space led to several different algorithms that attempt

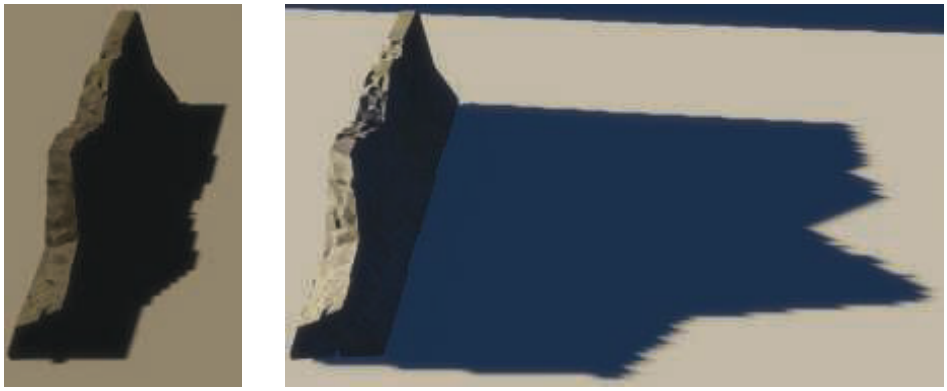

Figure 7.16. On the left the light is nearly overhead. The edge of the shadow is a bit ragged due to a low resolution compared to the eye's view. On the right the light is near the horizon, so each shadow texel covers considerably more screen area horizontally and so gives a more jagged edge. *(Images generated by TheRealMJP's "Shadows" program on Github.)*

to better match the light's sampling rates to the eye's. Methods include *perspective shadow maps* (PSM) [1691], *trapezoidal shadow maps* (TSM) [1132], and *light space perspective shadow maps* (LiSPSM) [1893, 1895]. See Figure 7.15 and Figure 7.26 on page 254 for examples. Techniques in this class are referred to as *perspective warping* methods.

An advantage of these matrix-warping algorithms is that no additional work is needed beyond modifying the light's matrices. Each method has its own strengths and weaknesses [484], as each can help match sampling rates for some geometry and lighting situations, while worsening these rates for others. Lloyd et al. [1062, 1063] analyze the equivalences between PSM, TSM, and LiSPSM, giving an excellent overview of the sampling and aliasing issues with these approaches. These schemes work best when the light's direction is perpendicular to the view's direction (e.g., overhead), as the perspective transform can then be shifted to put more samples closer to the eye.

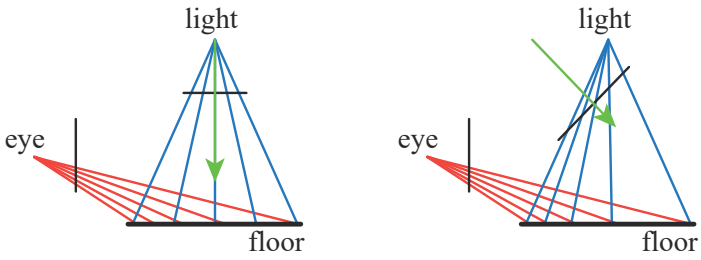

Figure 7.17. For an overhead light, on the left the sampling on the floor does not match the eye's rate. By changing the light's view direction and projection window on the right, the sampling rate is biased toward having a higher density of texels nearer the eye.

One lighting situation where matrix-warping techniques fail to help is when a light is in front of the camera and pointing at it. This situation is known as *dueling frusta*, or more colloquially as "deer in the headlights." More shadow-map samples are needed nearer the eye, but linear warping can only make the situation worse [1555]. This and other problems, such as sudden changes in quality [430] and a "nervous," unstable quality to the shadows produced during camera movement [484, 1227], have made these approaches fall out of favor.

The idea of adding more samples where the viewer is located is a good one, leading to algorithms that generate several shadow maps for a given view. This idea first made a noticeable impact when Carmack described it at his keynote at Quakecon 2004. Blow independently implemented such a system [174]. The idea is simple: Generate a fixed set of shadow maps (possibly at different resolutions), covering different areas of the scene. In Blow's scheme, four shadow maps are nested around the viewer. In this way, a high-resolution map is available for nearby objects, with the resolution dropping for those objects far away. Forsyth [483, 486] presents a related idea, generating different shadow maps for different visible sets of objects. The problem of how to handle the transition for objects spanning the border between two shadow maps is avoided in his setup, since each object has one and only one shadow map associated with it. Flagship Studios developed a system that blended these two ideas. One shadow map is for nearby dynamic objects, another is for a grid section of the static objects near the viewer, and a third is for the static objects in the scene as a whole. The first shadow map is generated each frame. The other two could be generated just once, since the light source and geometry are static. While all these particular systems are now quite old, the ideas of multiple maps for different objects and situations, some precomputed and some dynamic, is a common theme among algorithms that have been developed since.

In 2006 Engel [430], Lloyd et al. [1062, 1063], and Zhang et al. [1962, 1963] independently researched the same basic idea.[1] The idea is to divide the view frustum's volume into a few pieces by slicing it parallel to the view direction. See Figure 7.18. As depth increases, each successive volume has about two to three times the depth range of the previous volume [430, 1962]. For each view volume, the light source can make a frustum that tightly bounds it and then generate a shadow map. By using texture atlases or arrays, the different shadow maps can be treated as one large texture object, thus minimizing cache access delays. A comparison of the quality improvement obtained is shown in Figure 7.19. Engel's name for this algorithm, *cascaded shadow maps* (CSM), is more commonly used than Zhang's term, *parallel-split shadow maps*, but both appear in the literature and are effectively the same [1964].

This type of algorithm is straightforward to implement, can cover huge scene areas with reasonable results, and is robust. The dueling frusta problem can be addressed by sampling at a higher rate closer to the eye, and there are no serious worst-case

[1] Tadamura et al. [1735] introduced the idea seven years earlier, but it did not have an impact until other researchers explored its usefulness.

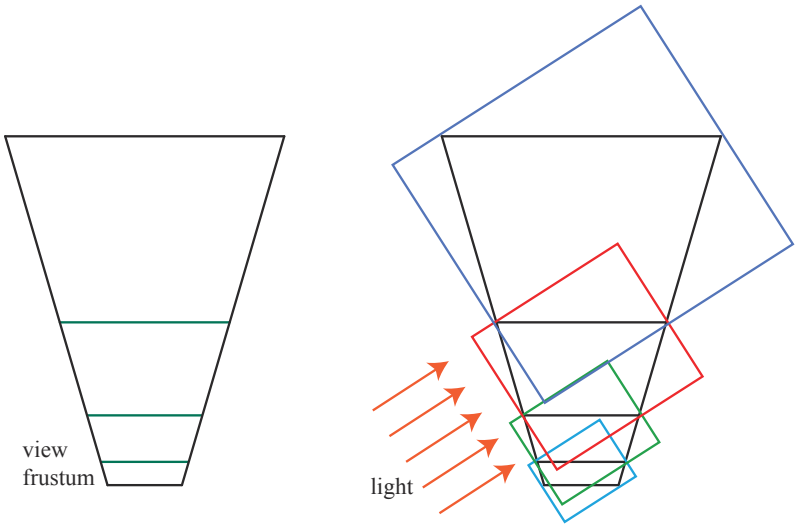

Figure 7.18. On the left, the view frustum from the eye is split into four volumes. On the right, bounding boxes are created for the volumes, which determine the volume rendered by each of the four shadow maps for the directional light. *(After Engel [430].)*

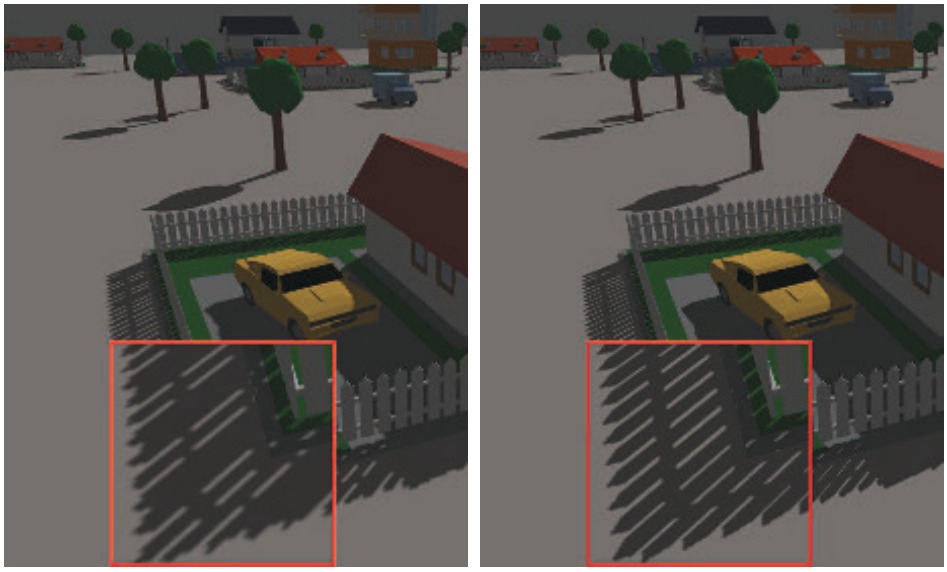

Figure 7.19. On the left, the scene's wide viewable area causes a single shadow map at a 2048×2048 resolution to exhibit perspective aliasing. On the right, four 1024×1024 shadow maps placed along the view axis improve quality considerably [1963]. A zoom of the front corner of the fence is shown in the inset red boxes. *(Images courtesy of Fan Zhang, The Chinese University of Hong Kong.)*

Figure 7.20. Shadow cascade visualization. Purple, green, yellow, and red represent the nearest through farthest cascades. *(Image courtesy of Unity Technologies.)*

problems. Because of these strengths, cascaded shadow mapping is used in many applications.

While it is possible to use perspective warping to pack more samples into subdivided areas of a single shadow map [1783], the norm is to use a separate shadow map for each cascade. As Figure 7.18 implies, and Figure 7.20 shows from the viewer's perspective, the area covered by each map can vary. Smaller view volumes for the closer shadow maps provide more samples where they are needed. Determining how the range of z-depths is split among the maps—a task called *z-partitioning*—can be quite simple or involved [412, 991, 1791]. One method is logarithmic partitioning [1062], where the ratio of far to near plane distances is made the same for each cascade map:

$$r = \sqrt[c]{\frac{f}{n}}, \tag{7.5}$$

where n and f are the near and far planes of the whole scene, c is the number of maps, and r is the resulting ratio. For example, if the scene's closest object is 1 meter away, the maximum distance is 1000 meters, and we have three cascaded maps, then $r = \sqrt[3]{1000/1} = 10$. The near and far plane distances for the closest view would be 1 and 10, the next interval is 10 to 100 to maintain this ratio, and the last is 100 to 1000 meters. The initial near depth has a large effect on this partitioning. If the near depth was only 0.1 meters, then the cube root of 10000 is 21.54, a considerably higher ratio, e.g., 0.1 to 2.154 to 46.42 to 1000. This would mean that each shadow map generated must cover a larger area, lowering its precision. In practice such a partitioning gives considerable resolution to the area close to the near plane, which is wasted if there are no objects in this area. One way to avoid this mismatch is to set the partition distances as a weighted blend of logarithmic and equidistant distributions [1962, 1963], but it would be better still if we could determine tight view bounds for the scene.

The challenge is in setting the near plane. If set too far from the eye, objects may be clipped by this plane, an extremely bad artifact. For a cut scene, an artist can set

this value precisely in advance [1590], but for an interactive environment the problem is more challenging. Lauritzen et al. [991, 1403] present *sample distribution shadow maps* (SDSM), which use the z-depth values from the previous frame to determine a better partitioning by one of two methods.

The first method is to look through the z-depths for the minimum and maximum values and use these to set the near and far planes. This is performed using what is called a *reduce* operation on the GPU, in which a series of ever-smaller buffers are analyzed by a compute or other shader, with the output buffer fed back as input, until a 1×1 buffer is left. Normally, the values are pushed out a bit to adjust for the speed of movement of objects in the scene. Unless corrective action is taken, nearby objects entering from the edge of the screen may still cause problems for a frame, though will quickly be corrected in the next.

The second method also analyzes the depth buffer's values, making a graph called a *histogram* that records the distribution of the z-depths along the range. In addition to finding tight near and far planes, the graph may have gaps in it where there are no objects at all. Any partition plane normally added to such an area can be snapped to where objects actually exist, giving more z-depth precision to the set of cascade maps.

In practice, the first method is general, is quick (typically in the 1 ms range per frame), and gives good results, so it has been adopted in several applications [1405, 1811]. See Figure 7.21.

As with a single shadow map, shimmering artifacts due to light samples moving frame to frame are a problem, and can be even worse as objects move between cascades. A variety of methods are used to maintain stable sample points in world space, each with their own advantages [41, 865, 1381, 1403, 1678, 1679, 1810]. A sudden change in a shadow's quality can occur when an object spans the boundary between two shadow maps. One solution is to have the view volumes slightly overlap. Samples taken in these overlap zones gather results from both adjoining shadow maps and are blended [1791]. Alternately, a single sample can be taken in such zone by using dithering [1381].

Due to its popularity, considerable effort has been put into improving efficiency and quality [1791, 1964]. If nothing changes within a shadow map's frustum, that shadow map does not need to be recomputed. For each light, the list of shadow casters can be precomputed by finding which objects are visible to the light, and of these, which can cast shadows on receivers [1405]. Since it is fairly difficult to perceive whether a shadow is correct, some shortcuts can be taken that are applicable to cascades and other algorithms. One technique is to use a low level of detail model as a proxy to actually cast the shadow [652, 1812]. Another is to remove tiny occluders from consideration [1381, 1811]. The more distant shadow maps may be updated less frequently than once a frame, on the theory that such shadows are less important. This idea risks artifacts caused by large moving objects, so needs to be used with care [865, 1389, 1391, 1678, 1679]. Day [329] presents the idea of "scrolling" distant maps from frame to frame, the idea being that most of each static shadow map is

Figure 7.21. Effect of depth bounds. On the left, no special processing is used to adjust the near and far planes. On the right, SDSM is used to find tighter bounds. Note the window frame near the left edge of each image, the area beneath the flower box on the second floor, and the window on the first floor, where undersampling due to loose view bounds causes artifacts. Exponential shadow maps are used to render these particular images, but the idea of improving depth precision is useful for all shadow map techniques. *(Image courtesy of Ready at Dawn Studios, copyright Sony Interactive Entertainment.)*

reusable frame to frame, and only the fringes may change and so need rendering. Games such as *DOOM* (2016) maintain a large atlas of shadow maps, regenerating only those where objects have moved [294]. The farther cascaded maps could be set to ignore dynamic objects entirely, since such shadows may contribute little to the scene. With some environments, a high-resolution static shadow map can be used in place of these farther cascades, which can significantly reduce the workload [415, 1590]. A sparse texture system (Section 19.10.1) can be employed for worlds where a single static shadow map would be enormous [241, 625, 1253]. Cascaded shadow mapping can be combined with baked-in light-map textures or other shadow techniques that are more appropriate for particular situations [652]. Valient's presentation [1811] is noteworthy in that it describes different shadow system customizations and techniques for a wide range of video games. Section 11.5.1 discusses precomputed light and shadow algorithms in detail.

Creating several separate shadow maps means a run through some set of geometry for each. A number of approaches to improve efficiency have been built on the idea of rendering occluders to a set of shadow maps in a single pass. The geometry shader can be used to replicate object data and send it to multiple views [41]. Instanced geometry shaders allow an object to be output into up to 32 depth textures [1456].

Multiple-viewport extensions can perform operations such as rendering an object to a specific texture array slice [41, 154, 530]. Section 21.3.1 discusses these in more detail, in the context of their use for virtual reality. A possible drawback of viewport-sharing techniques is that the occluders for all the shadow maps generated must be sent down the pipeline, versus the set found to be relevant to each shadow map [1791, 1810].

You yourself are currently in the shadows of billions of light sources around the world. Light reaches you from only a few of these. In real-time rendering, large scenes with multiple lights can become swamped with computation if all lights are active at all times. If a volume of space is inside the view frustum but not visible to the eye, objects that occlude this receiver volume do not need to be evaluated [625, 1137]. Bittner et al. [152] use occlusion culling (Section 19.7) from the eye to find all visible shadow receivers, and then render all potential shadow receivers to a stencil buffer mask from the light's point of view. This mask encodes which visible shadow receivers are seen from the light. To generate the shadow map, they render the objects from the light using occlusion culling and use the mask to cull objects where no receivers are located. Various culling strategies can also work for lights. Since irradiance falls off with the square of the distance, a common technique is to cull light sources after a certain threshold distance. For example, the portal culling technique in Section 19.5 can find which lights affect which cells. This is an active area of research, since the performance benefits can be considerable [1330, 1604].

7.5 Percentage-Closer Filtering

A simple extension of the shadow-map technique can provide pseudo-soft shadows. This method can also help ameliorate resolution problems that cause shadows to look blocky when a single light-sample cell covers many screen pixels. The solution is similar to texture magnification (Section 6.2.1). Instead of a single sample being taken off the shadow map, the four nearest samples are retrieved. The technique does not interpolate between the depths themselves, but rather the results of their comparisons with the surface's depth. That is, the surface's depth is compared separately to the four texel depths, and the point is then determined to be in light or shadow for each shadow-map sample. These results, i.e., 0 for shadow and 1 for light, are then bilinearly interpolated to calculate how much the light actually contributes to the surface location. This filtering results in an artificially soft shadow. These penumbrae change, depending on the shadow map's resolution, camera location, and other factors. For example, a higher resolution makes for a narrower softening of the edges. Still, a little penumbra and smoothing is better than none at all.

This idea of retrieving multiple samples from a shadow map and blending the results is called *percentage-closer filtering* (PCF) [1475]. Area lights produce soft shadows. The amount of light reaching a location on a surface is a function of what proportion of the light's area is visible from the location. PCF attempts to approximate a soft shadow for a punctual (or directional) light by reversing the process.

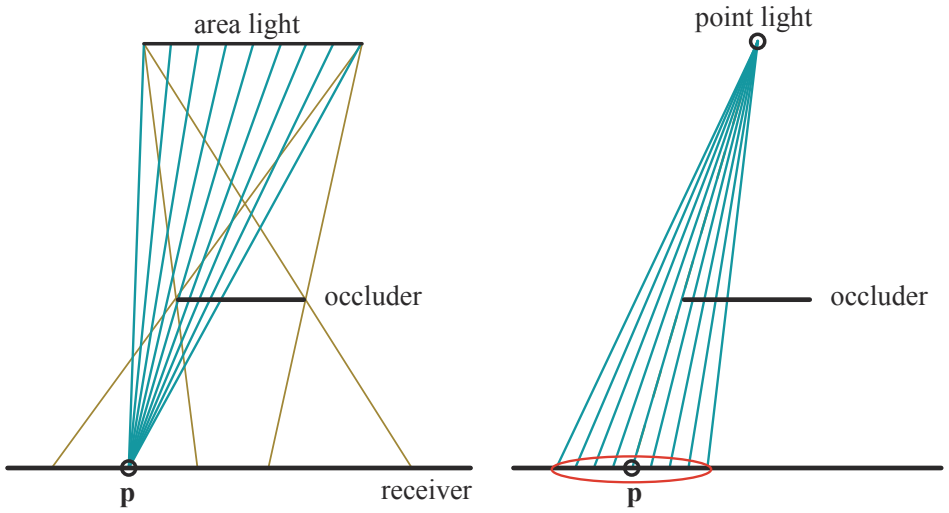

Figure 7.22. On the left, the brown lines from the area light source show where penumbrae are formed. For a single point **p** on the receiver, the amount of illumination received could be computed by testing a set of points on the area light's surface and finding which are not blocked by any occluders. On the right, a point light does not cast a penumbra. PCF approximates the effect of an area light by reversing the process: At a given location, it samples over a comparable area on the shadow map to derive a percentage of how many samples are illuminated. The red ellipse shows the area sampled on the shadow map. Ideally, the width of this disk is proportional to the distance between the receiver and occluder.

Instead of finding the light's visible area from a surface location, it finds the visibility of the punctual light from a set of surface locations near the original location. See Figure 7.22. The name "percentage-closer filtering" refers to the ultimate goal, to find the percentage of the samples taken that are visible to the light. This percentage is how much light then is used to shade the surface.

In PCF, locations are generated nearby to a surface location, at about the same depth, but at different texel locations on the shadow map. Each location's visibility is checked, and these resulting boolean values, lit or unlit, are then blended to get a soft shadow. Note that this process is non-physical: Instead of sampling the light source directly, this process relies on the idea of sampling over the surface itself. The distance to the occluder does not affect the result, so shadows have similar-sized penumbrae. Nonetheless this method provides a reasonable approximation in many situations.

Once the width of the area to be sampled is determined, it is important to sample in a way that avoids aliasing artifacts. There are numerous variations of how to sample and filter nearby shadow-map locations. Variables include how wide of an area to sample, how many samples to use, the sampling pattern, and how to weight the results. With less-capable APIs, the sampling process could be accelerated by a special texture sampling mode similar to bilinear interpolation, which accesses the four neighboring

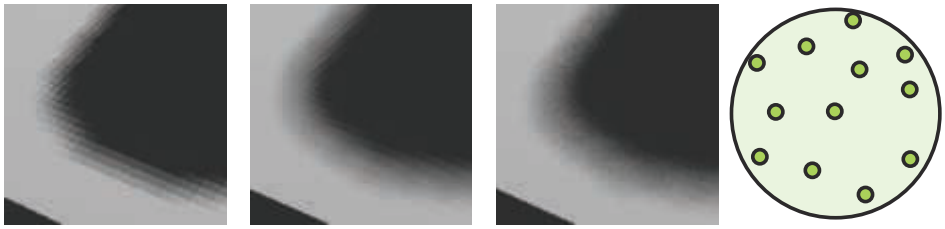

Figure 7.23. The far left shows PCF sampling in a 4×4 grid pattern, using nearest neighbor sampling. The far right shows a 12-tap Poisson sampling pattern on a disk. Using this pattern to sample the shadow map gives the improved result in the middle left, though artifacts are still visible. In the middle right, the sampling pattern is rotated randomly around its center from pixel to pixel. The structured shadow artifacts turn into (much less objectionable) noise. *(Images courtesy of John Isidoro, ATI Research, Inc.)*

locations. Instead of blending the results, each of the four samples is compared to a given value, and the ratio that passes the test is returned [175]. However, performing nearest neighbor sampling in a regular grid pattern can create noticeable artifacts. Using a joint bilateral filter that blurs the result but respects object edges can improve quality while avoiding shadows leaking onto other surfaces [1343]. See Section 12.1.1 for more on this filtering technique.

DirectX 10 introduced single-instruction bilinear filtering support for PCF, giving a smoother result [53, 412, 1709, 1790]. This offers considerable visual improvement over nearest neighbor sampling, but artifacts from regular sampling remain a problem. One solution to minimize grid patterns is to sample an area using a precomputed Poisson distribution pattern, as illustrated in Figure 7.23. This distribution spreads samples out so that they are neither near each other nor in a regular pattern. It is well known that using the same sampling locations for each pixel, regardless of the distribution, can result in patterns [288]. Such artifacts can be avoided by randomly rotating the sample distribution around its center, which turns aliasing into noise. Castaño [235] found that the noise produced by Poisson sampling was particularly noticeable for their smooth, stylized content. He presents an efficient Gaussian-weighted sampling scheme based on bilinear sampling.

Self-shadowing problems and light leaks, i.e., acne and Peter Panning, can become worse with PCF. Slope scale bias pushes the surface away from the light based purely on its angle to the light, with the assumption that a sample is no more than a texel away on the shadow map. By sampling in a wider area from a single location on a surface, some of the test samples may get blocked by the true surface.

A few different additional bias factors have been invented and used with some success to reduce the risk of self-shadowing. Burley [212] describes the *bias cone*, where each sample is moved toward the light proportional to its distance from the original sample. Burley recommends a slope of 2.0, along with a small constant bias. See Figure 7.24.

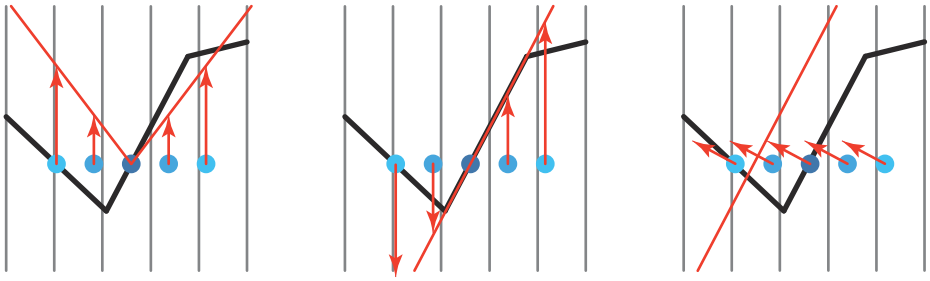

Figure 7.24. Additional shadow bias methods. For PCF, several samples are taken surrounding the original sample location, the center of the five dots. All these samples should be lit. In the left figure, a bias cone is formed and the samples are moved up to it. The cone's steepness could be increased to pull the samples on the right close enough to be lit, at the risk of increasing light leaks from other samples elsewhere (not shown) that truly are shadowed. In the middle figure, all samples are adjusted to lie on the receiver's plane. This works well for convex surfaces but can be counterproductive at concavities, as seen on the left side. In the right figure, normal offset bias moves the samples along the surface's normal direction, proportional to the sine of the angle between the normal and the light. For the center sample, this can be thought of as moving to an imaginary surface above the original surface. This bias not only affects the depth but also changes the texture coordinates used to test the shadow map.

Schüler [1585], Isidoro [804], and Tuft [1790] present techniques based on the observation that the slope of the receiver itself should be used to adjust the depths of the rest of the samples. Of the three, Tuft's formulation [1790] is most easily applied to cascaded shadow maps. Dou et al. [373] further refine and extend this concept, accounting for how the z-depth varies in a nonlinear fashion. These approaches assume that nearby sample locations are on the same plane formed by the triangle. Referred to as *receiver plane depth bias* or other similar terms, this technique can be quite precise in many cases, as locations on this imaginary plane are indeed on the surface, or in front of it if the model is convex. As shown in Figure 7.24, samples near concavities can become hidden. Combinations of constant, slope scale, receiver plane, view bias, and normal offset biasing have been used to combat the problem of self-shadowing, though hand-tweaking for each environment can still be necessary [235, 1391, 1403].

One problem with PCF is that because the sampling area's width remains constant, shadows will appear uniformly soft, all with the same penumbra width. This may be acceptable under some circumstances, but appears incorrect where there is ground contact between the occluder and receiver. See Figure 7.25.

7.6 Percentage-Closer Soft Shadows

In 2005 Fernando [212, 467, 1252] published an influential approach called *percentage-closer soft shadows* (PCSS). It attempts a solution by searching the nearby area on

Figure 7.25. Percentage-closer filtering and percentage-closer soft shadows. On the left, hard shadows with a little PCF filtering. In the middle, constant width soft shadows. On the right, variable-width soft shadows with proper hardness where objects are in contact with the ground. *(Images courtesy of NVIDIA Corporation.)*

the shadow map to find all possible occluders. The average distance of these occluders from the location is used to determine the sample area width:

$$w_{\text{sample}} = w_{\text{light}} \frac{d_r - d_o}{d_r}, \tag{7.6}$$

where d_r is the distance of the receiver from the light and d_o the average occluder distance. In other words, the width of the surface area to the sample grows as the average occluder gets farther from the receiver and closer to the light. Examine Figure 7.22 and think about the effect of moving the occluder to see how this occurs. Figures 7.2 (page 224), 7.25, and 7.26 show examples.

If there are no occluders found, the location is fully lit and no further processing is necessary. Similarly, if the location is fully occluded, processing can end. Otherwise, then the area of interest is sampled and the light's approximate contribution is computed. To save on processing costs, the width of the sample area can be used to vary the number of samples taken. Other techniques can be implemented, e.g., using lower sampling rates for distant soft shadows that are less likely to be important.

A drawback of this method is that it needs to sample a fair-sized area of the shadow map to find the occluders. Using a rotated Poisson disk pattern can help hide undersampling artifacts [865, 1590]. Jimenez [832] notes that Poisson sampling can be unstable under motion and finds that a spiral pattern formed by using a function halfway between dithering and random gives a better result frame to frame.

Sikachev et al. [1641] discuss in detail a faster implementation of PCSS using features in SM 5.0, introduced by AMD and often called by their name for it, *contact hardening shadows* (CHS). This new version also addresses another problem with basic PCSS: The penumbra's size is affected by the shadow map's resolution. See Figure 7.25. This problem is minimized by first generating mipmaps of the shadow map, then choosing the mip level closest to a user-defined world-space kernel size. An 8×8 area is sampled to find the average blocker depth, needing only 16 `GatherRed()` texture calls. Once a penumbra estimate is found, the higher-resolution mip levels are

used for the sharp area of the shadow, while the lower-resolution mip levels are used for softer areas.

CHS has been used in a large number of video games [1351, 1590, 1641, 1678, 1679], and research continues. For example, Buades et al. [206] present *separable soft shadow mapping* (SSSM), where the PCSS process of sampling a grid is split into separable parts and elements are reused as possible from pixel to pixel.

One concept that has proven helpful for accelerating algorithms that need multiple samples per pixel is the hierarchical *min/max shadow map*. While shadow map depths normally cannot be averaged, the minimum and maximum values at each mipmap level can be useful. That is, two mipmaps can be formed, one saving the largest z-depth found in each area (sometimes called *HiZ*), and one the smallest. Given a texel location, depth, and area to be sampled, the mipmaps can be used to rapidly determine fully lit and fully shadowed conditions. For example, if the texel's z-depth is greater than the maximum z-depth stored for the corresponding area of the mipmap, then the texel must be in shadow—no further samples are needed. This type of shadow map makes the task of determining light visibility much more efficient [357, 415, 610, 680, 1064, 1811].

Methods such as PCF work by sampling the nearby receiver locations. PCSS works by finding an average depth of nearby occluders. These algorithms do not directly take into account the area of the light source, but rather sample the nearby surfaces, and are affected by the resolution of the shadow map. A major assumption behind PCSS is that the average blocker is a reasonable estimate of the penumbra size. When two occluders, say a street lamp and a distant mountain, partially occlude the same surface at a pixel, this assumption is broken and can lead to artifacts. Ideally, we want to determine how much of the area light source is visible from a single receiver location. Several researchers have explored *backprojection* using the GPU. The idea is to treat each receiver's location as a viewpoint and the area light source as part of a view plane, and to project occluders onto this plane. Both Schwarz and Stamminger [1593] and Guennebaud et al. [617] summarize previous work and offer their own improvements. Bavoil et al. [116] take a different approach, using depth peeling to create a multi-layer shadow map. Backprojection algorithms can give excellent results, but the high cost per pixel has (so far) meant they have not seen adoption in interactive applications.

7.7 Filtered Shadow Maps

One algorithm that allows filtering of the shadow maps generated is Donnelly and Lauritzen's *variance shadow map* (VSM) [368]. The algorithm stores the depth in one map and the depth squared in another map. MSAA or other antialiasing schemes can be used when generating the maps. These maps can be blurred, mipmapped, put in summed area tables [988], or any other method. The ability to treat these maps as filterable textures is a huge advantage, as the entire array of sampling and filtering techniques can be brought to bear when retrieving data from them.

We will describe VSM in some depth here, to give a sense of how this process works; also, the same type of testing is used for all methods in this class of algorithm. Readers interested in learning more about this area should access the relevant references, and we also recommend the book by Eisemann et al. [412], which gives the topic considerably more space.

To begin, for VSM the depth map is sampled (just once) at the receiver's location to return an average depth of the closest light occluder. When this average depth M_1, called the *first moment*, is greater than the depth on the shadow receiver t, the receiver is considered fully in light. When the average depth is less than the receiver's depth, the following equation is used:

$$p_{\max}(t) = \frac{\sigma^2}{\sigma^2 + (t - M_1)^2}, \tag{7.7}$$

where $p_{\max}$ is the maximum percentage of samples in light, σ^2 is the variance, t is the receiver depth, and M_1 is the average expected depth in the shadow map. The depth-squared shadow map's sample M_2, called the *second moment*, is used to compute the variance:

$$\sigma^2 = M_2 - M_1^2. \tag{7.8}$$

The value $p_{\max}$ is an upper bound on the visibility percentage of the receiver. The actual illumination percentage p cannot be larger than this value. This upper bound is from the one-sided variant of Chebyshev's inequality. The equation attempts to estimate, using probability theory, how much of the distribution of occluders at the surface location is beyond the surface's distance from the light. Donnelly and Lauritzen show that for a planar occluder and planar receiver at fixed depths, $p = p_{\max}$, so Equation 7.7 can be used as a good approximation of many real shadowing situations.

Myers [1251] builds up an intuition as to why this method works. The variance over an area increases at shadow edges. The greater the difference in depths, the greater the variance. The $(t - M_1)^2$ term is then a significant determinant in the visibility percentage. If this value is just slightly above zero, this means the average occluder depth is slightly closer to the light than the receiver, and $p_{\max}$ is then near 1 (fully lit). This would happen along the fully lit edge of the penumbra. Moving into the penumbra, the average occluder depth gets closer to the light, so this term becomes larger and $p_{\max}$ drops. At the same time the variance itself is changing within the penumbra, going from nearly zero along the edges to the largest variance where the occluders differ in depth and equally share the area. These terms balance out to give a linearly varying shadow across the penumbra. See Figure 7.26 for a comparison with other algorithms.

One significant feature of variance shadow mapping is that it can deal with the problem of surface bias problems due to geometry in an elegant fashion. Lauritzen [988] gives a derivation of how the surface's slope is used to modify the value of the second moment. Bias and other problems from numerical stability can be a problem for variance mapping. For example, Equation 7.8 subtracts one large value

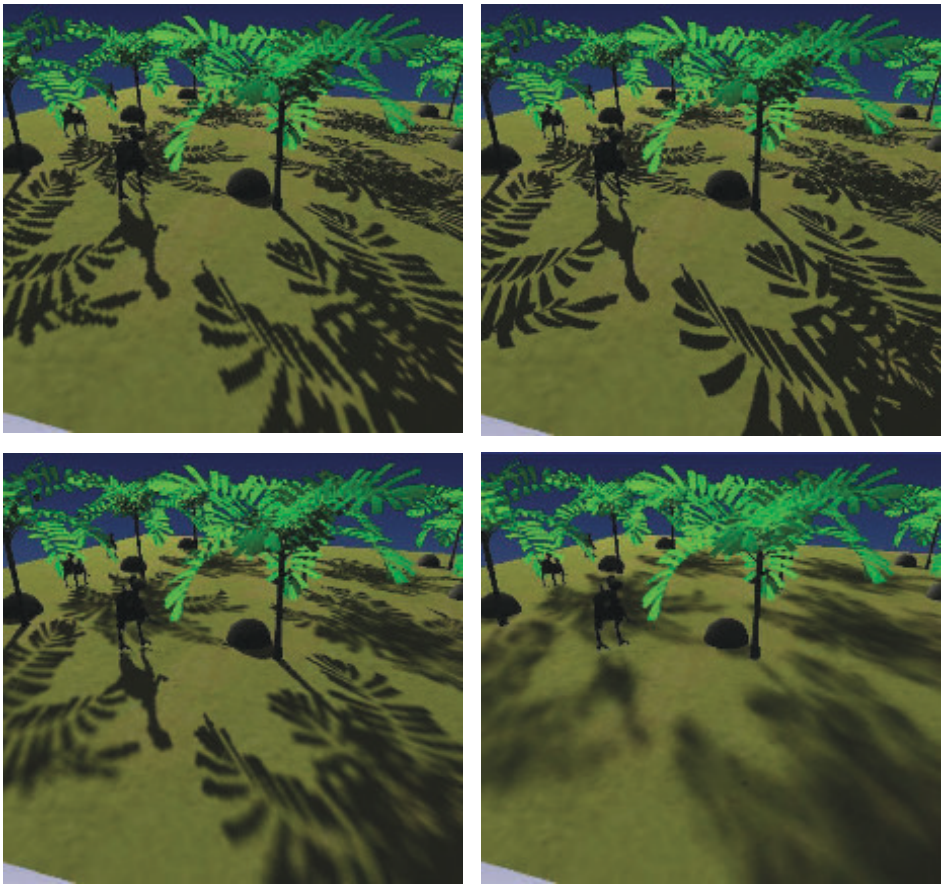

Figure 7.26. In the upper left, standard shadow mapping. Upper right, perspective shadow mapping, increasing the density of shadow-map texel density near the viewer. Lower left, percentage-closer soft shadows, softening the shadows as the occluder's distance from the receiver increases. Lower right, variance shadow mapping with a constant soft shadow width, each pixel shaded with a single variance map sample. *(Images courtesy of Nico Hempe, Yvonne Jung, and Johannes Behr.)*

from another similar value. This type of computation tends to magnify the lack of accuracy of the underlying numerical representation. Using floating point textures helps avoid this problem.

Overall VSM gives a noticeable increase in quality for the amount of time spent processing, since the GPU's optimized texture capabilities are used efficiently. While PCF needs more samples, and hence more time, to avoid noise when generating softer shadows, VSM can work with just a single, high-quality sample to determine the entire area's effect and produce a smooth penumbra. This ability means that shadows can be made arbitrarily soft at no additional cost, within the limitations of the algorithm.

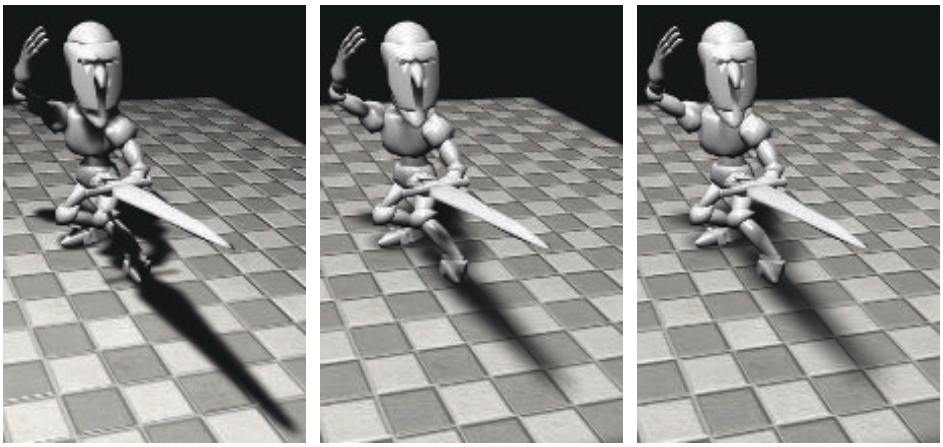

Figure 7.27. Variance shadow mapping, where the distance to the light source increases from left to right. *(Images from the NVIDIA SDK 10 [1300] samples, courtesy of NVIDIA Corporation.)*

As with PCF, the width of the filtering kernel determines the width of the penumbra. By finding the distance between the receiver and the closest occluder, the kernel width can be varied, so giving convincing soft shadows. Mipmapped samples are poor estimators of coverage for a penumbra with a slowly increasing width, creating boxy artifacts. Lauritzen [988] details how to use summed-area tables to give considerably better shadows. An example is shown in Figure 7.27.

One place variance shadow mapping breaks down is along the penumbrae areas when two or more occluders cover a receiver and one occluder is close to the receiver. The Chebyshev inequality from probability theory will produce a maximum light value that is not related to the correct light percentage. The closest occluder, by only partially hiding the light, throws off the equation's approximation. This results in *light bleeding* (a.k.a. light leaks), where areas that are fully occluded still receive light. See Figure 7.28. By taking more samples over smaller areas, this problem can be resolved, turning variance shadow mapping into a form of PCF. As with PCF, speed and performance trade off, but for scenes with low shadow depth complexity, variance mapping works well. Lauritzen [988] gives one artist-controlled method to ameliorate the problem, which is to treat low percentages as fully shadowed and to remap the rest of the percentage range to 0% to 100%. This approach darkens light bleeds, at the cost of narrowing penumbrae overall. While light bleeding is a serious limitation, VSM is good for producing shadows from terrain, since such shadows rarely involve multiple occluders [1227].

The promise of being able to use filtering techniques to rapidly produce smooth shadows generated much interest in filtered shadow mapping; the main challenge is solving the various bleeding problems. Annen et al. [55] introduced the *convolution shadow map*. Extending the idea behind Soler and Sillion's algorithm for planar

Figure 7.28. On the left, variance shadow mapping applied to a teapot. On the right, a triangle (not shown) casts a shadow on the teapot, causing objectionable artifacts in the shadow on the ground. *(Images courtesy of Marco Salvi.)*

receivers [1673], the idea is to encode the shadow depth in a Fourier expansion. As with variance shadow mapping, such maps can be filtered. The method converges to the correct answer, so the light leak problem is lessened.

A drawback of convolution shadow mapping is that several terms need to be computed and accessed, considerably increasing both execution and storage costs [56, 117]. Salvi [1529, 1530] and Annen et al. [56] concurrently and independently came upon the idea of using a single term based on an exponential function. Called an *exponential shadow map* (ESM) or *exponential variance shadow map* (EVSM), this method saves the exponential of the depth along with its second moment into two buffers. An exponential function more closely approximates the step function that a shadow map performs (i.e., in light or not), so this works to significantly reduce bleeding artifacts. It avoids another problem that convolution shadow mapping has, called *ringing*, where minor light leaks can happen at particular depths just past the original occluder's depth.

A limitation with storing exponential values is that the second moment values can become extremely large and so run out of range using floating point numbers. To improve precision, and to allow the exponential function to drop off more steeply, z-depths can be generated so that they are linear [117, 258].

Due to its improved quality over VSM, and its lower storage and better performance compared to convolution maps, the exponential shadow map approach has sparked the most interest of the three filtered approaches. Pettineo [1405] notes several other improvements, such as the ability to use MSAA to improve results and to obtain some limited transparency, and describes how filtering performance can be improved with compute shaders.

More recently, *moment shadow mapping* was introduced by Peters and Klein [1398]. It offers better quality, though at the expense of using four or more moments, increasing storage costs. This cost can be decreased by the use of 16-bit integers to store the moments. Pettineo [1404] implements and compares this new approach with ESM, providing a code base that explores many variants.

Cascaded shadow-map techniques can be applied to filtered maps to improve precision [989]. An advantage of cascaded ESM over standard cascaded maps is that a single bias factor can be set for all cascades [1405]. Chen and Tatarchuk [258] go into detail about various light leak problems and other artifacts encountered with cascaded ESM, and present a few solutions.

Filtered maps can be thought of as an inexpensive form of PCF, one that needs few samples. Like PCF, such shadows have a constant width. These filtered approaches can all be used in conjunction with PCSS to provide variable-width penumbrae [57, 1620, 1943]. An extension of moment shadow mapping also includes the ability to provide light scattering and transparency effects [1399].

7.8 Volumetric Shadow Techniques

Transparent objects will attenuate and change the color of light. For some sets of transparent objects, techniques similar to those discussed in Section 5.5 can be used to simulate such effects. For example, in some situations a second type of shadow map can be generated. The transparent objects are rendered to it, and the closest depth and color or alpha coverage is stored. If the receiver is not blocked by the opaque shadow map, the transparency depth map is then tested and, if occluded, the color or coverage is retrieved as needed [471, 1678, 1679]. This idea is reminiscent of shadow and light projection in Section 7.2, with the stored depths avoiding projection onto receivers between the transparent objects and the light. Such techniques cannot be applied to the transparent objects themselves.

Self-shadowing is critical for realistic rendering of objects such as hair and clouds, where objects are either small or semitransparent. Single-depth shadow maps will not work for these situations. Lokovic and Veach [1066] first presented the concept of *deep shadow maps*, in which each shadow-map texel stores a function of how light drops off with depth. This function is typically approximated by a series of samples at different depths, with each sample having an opacity value. The two samples in the map that bracket a given position's depth are used to find the shadow's effect. The challenge on the GPU is in generating and evaluating such functions efficiently. These algorithms use similar approaches and run into similar challenges found for some order-independent transparency algorithms (Section 5.5), such as compact storage of the data needed to faithfully represent each function.

Kim and Neumann [894] were the first to present a GPU-based method, which they call *opacity shadow maps*. Maps storing just the opacities are generated at a fixed set of depths. Nguyen and Donnelly [1274] give an updated version of this approach, producing images such as Figure 17.2 on page 719. However, the depth slices are all parallel and uniform, so many slices are needed to hide in-between slice opacity artifacts due to linear interpolation. Yuksel and Keyser [1953] improve efficiency and quality by creating opacity maps that more closely follow the shape of the model.

Figure 7.29. Hair and smoke rendering with adaptive volumetric shadow maps [1531]. *(Reprinted by permission of Marco Salvi and Intel Corporation, copyright Intel Corporation, 2010.)*

Doing so allows them to reduce the number of layers needed, as evaluation of each layer is more significant to the final image.

To avoid having to rely on fixed slice setups, more adaptive techniques have been proposed. Salvi et al. [1531] introduce *adaptive volumetric shadow maps*, in which each shadow-map texel stores both the opacities and layer depths. Pixel shaders operations are used to lossily compress the stream of data (surface opacities) as it is rasterized. This avoids needing an unbounded amount of memory to gather all samples and process them in a set. The technique is similar to deep shadow maps [1066], but with the compression step done on the fly in the pixel shader. Limiting the function representation to a small, fixed number of stored opacity/depth pairs makes both compression and retrieval on the GPU more efficient [1531]. The cost is higher than simple blending because the curve needs to be read, updated, and written back, and it depends on the number of points used to represent a curve. In this case, this technique also requires recent hardware that supports UAV and ROV functionality (end of Section 3.8). See Figure 7.29 for an example.

The adaptive volumetric shadow mapping method was used for realistic smoke rendering in the game *GRID2*, with the average cost being below 2 ms/frame [886]. Fürst et al. [509] describe and provide code for their implementation of deep shadow maps for a video game. They use linked lists to store depths and alphas, and use exponential shadow mapping to provide a soft transition between lit and shadowed regions.

Exploration of shadow algorithms continues, with a synthesis of a variety of algorithms and techniques becoming more common. For example, Selgrad et al. [1603] research storing multiple transparent samples with linked lists and using compute shaders with scattered writes to build the map. Their work uses deep shadow-map concepts, along with filtered maps and other elements, which give a more general solution for providing high-quality soft shadows.

7.9 Irregular *Z*-Buffer Shadows

Shadow-map approaches of various sorts are popular for several reasons. Their costs are predictable and scale well to increasing scene sizes, at worst linear with the number of primitives. They map nicely onto the GPU, as they rely on rasterization to regularly sample the light's view of the world. However, due to this discrete sampling, problems arise because the locations the eye sees do not map one-to-one with those the light sees. Various aliasing problems arise when the light samples a surface less frequently than the eye. Even when sampling rates are comparable, there are biasing problems because the surface is sampled in locations slightly different than those the eye sees.

Shadow volumes provide an exact, analytical solution, as the light's interactions with surfaces result in sets of triangles defining whether any given location is lit or in shadow. The unpredictable cost of the algorithm when implemented on the GPU is a serious drawback. The improvements explored in recent years [1648] are tantalizing, but have not yet had an "existence proof" of being adopted in commercial applications.

Another analytical shadow-testing method may have potential in the longer term: ray tracing. Described in detail in Section 11.2.2, the basic idea is simple enough, especially for shadowing. A ray is shot from the receiver location to the light. If any object is found that blocks the ray, the receiver is in shadow. Much of a fast ray tracer's code is dedicated to generating and using hierarchical data structures to minimize the number of object tests needed per ray. Building and updating these structures each frame for a dynamic scene is a decades-old topic and a continuing area of research.

Another approach is to use the GPU's rasterization hardware to view the scene, but instead of just z-depths, additional information is stored about the edges of the occluders in each grid cell of the light [1003, 1607]. For example, imagine storing at each shadow-map texel a list of triangles that overlap the grid cell. Such a list can be generated by *conservative rasterization*, in which a triangle generates a fragment if any part of the triangle overlaps a pixel, not just the pixel's center (Section 23.1.2). One problem with such schemes is that the amount of data per texel normally needs to be limited, which in turn can lead to inaccuracies in determining the status of every receiver location. Given modern linked-list principles for GPUs [1943], it is certainly possible to store more data per pixel. However, aside from physical memory limits, a problem with storing a variable amount of data in a list per texel is that GPU processing can become extremely inefficient, as a single warp can have a few fragment

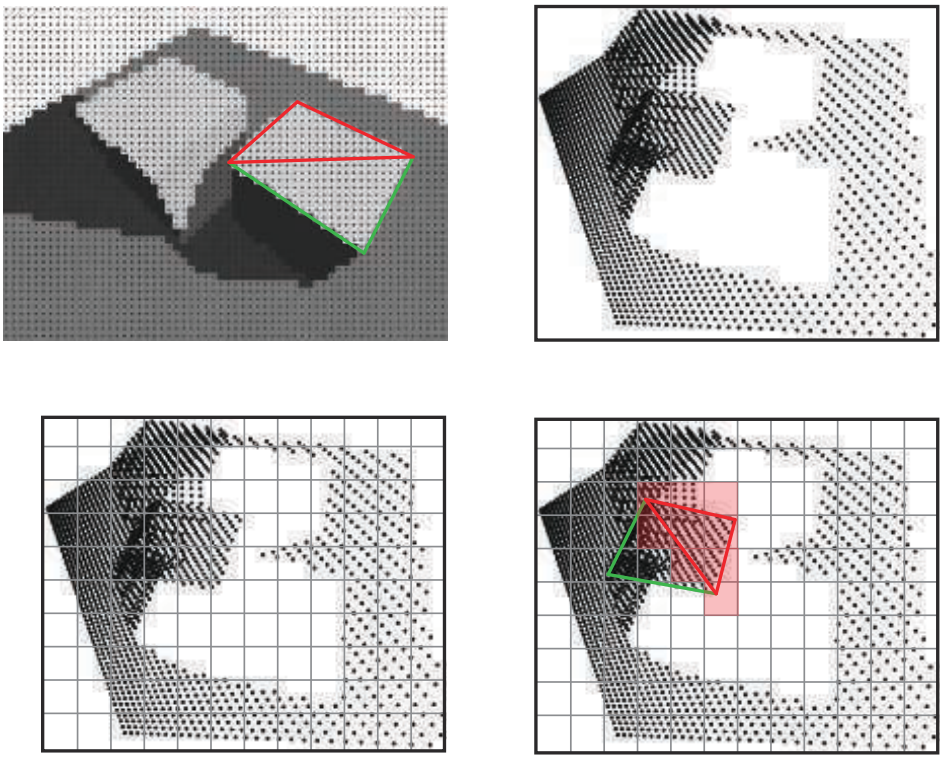

Figure 7.30. Irregular z-buffer. In the upper left, the view from the eye generates a set of dots at the pixel centers. Two triangles forming a cube face are shown. In the upper right, these dots are shown from the light's view. In the lower left, a shadow-map grid is imposed. For each texel a list of all dots inside its grid cell is generated. In the lower right, shadow testing is performed for the red triangle by conservatively rasterizing it. At each texel touched, shown in light red, all dots in its list are tested against the triangle for visibility by the light. *(Underlying raster images courtesy of Timo Aila and Samuli Laine [14].)*

threads that need to retrieve and process many items, while the rest of the threads are idle, having no work to do. Structuring a shader to avoid thread divergence due to dynamic "if" statements and loops is critical for performance.

An alternative to storing triangles or other data in the shadow map and testing receiver locations against them is to flip the problem, storing receiver locations and then testing triangles against each. This concept of saving the receiver locations, first explored by Johnson et al. [839] and Aila and Laine [14], is called the *irregular z-buffer* (IZB). The name is slightly misleading, in that the buffer itself has a normal, regular shape for a shadow map. Rather, the buffer's contents are irregular, as each shadow-map texel will have one or more receiver locations stored in it, or possibly none at all. See Figure 7.30.

Using the method presented by Sintorn et al. [1645] and Wyman et al. [1930, 1932], a multi-pass algorithm creates the IZB and tests its contents for visibility from the light. First, the scene is rendered from the eye, to find the z-depths of the surfaces seen from the eye. These points are transformed to the light's view of the scene, and tight bounds are formed from this set for the light's frustum. The points are then deposited in the light's IZB, each placed into a list at its corresponding texel. Note that some lists may be empty, a volume of space that the light views but that has no surfaces seen by the eye. Occluders are conservatively rasterized to the light's IZB to determine whether any points are hidden, and so in shadow. Conservative rasterization ensures that, even if a triangle does not cover the center of a light texel, it will be tested against points it may overlap nonetheless.

Visibility testing occurs in the pixel shader. The test itself can be visualized as a form of ray tracing. A ray is generated from an image point's location to the light. If a point is inside the triangle and more distant than the triangle's plane, it is hidden. Once all occluders are rasterized, the light-visibility results are used to shade the surface. This testing is also called *frustum tracing*, as the triangle can be thought of as defining a view frustum that checks points for inclusion in its volume.

Careful coding is critical in making this approach work well with the GPU. Wyman et al. [1930, 1932] note that their final version was two orders of magnitude faster than the initial prototypes. Part of this performance increase was straightforward algorithm improvements, such as culling image points where the surface normal was facing away from the light (and so always unlit) and avoiding having fragments generated for empty texels. Other performance gains were from improving data structures for the GPU, and from minimizing thread divergence by working to have short, similar-length lists of points in each texel. Figure 7.30 shows a low-resolution shadow map with long lists for illustrative purposes. The ideal is one image point per list. A higher resolution gives shorter lists, but also increases the number of fragments generated by occluders for evaluation.

As can be seen in the lower left image in Figure 7.30, the density of visible points on the ground plane is considerably higher on the left side than the right, due to the perspective effect. Using cascaded shadow maps helps lower list sizes in these areas by focusing more light-map resolution closer to the eye.

This approach avoids the sampling and bias issues of other approaches and provides perfectly sharp shadows. For aesthetic and perceptual reasons, soft shadows are often desired, but can have bias problems with nearby occluders, such as Peter Panning. Story and Wyman [1711, 1712] explore hybrid shadow techniques. The core idea is to use the occluder distance to blend IZB and PCSS shadows, using the hard shadow result when the occluder is close and soft when more distant. See Figure 7.31. Shadow quality is often most important for nearby objects, so IZB costs can be reduced by using this technique on only a selected subset. This solution has successfully been used in video games. This chapter started with such an image, shown in Figure 7.2 on page 224.

Figure 7.31. On the left, PCF gives uniformly softened shadows for all objects. In the middle, PCSS softens the shadow with distance to the occluder, but the tree branch shadow overlapping the left corner of the crate creates artifacts. On the right, sharp shadows from IZB blended with soft from PCSS give an improved result [1711]. *(Images from "Tom Clancy's The Division," courtesy of Ubisoft.)*

7.10 Other Applications

Treating the shadow map as defining a volume of space, separating light from dark, can also help in determining what parts of objects to shadow. Gollent [555] describes how CD Projekt's terrain shadowing system computes for each area a maximum height that is still occluded, which can then be used to shadow not only terrain but also trees and other elements in the scene. To find each height, a shadow map of the visible area is rendered for the sun. Each terrain heightfield location is then checked for visibility from the sun. If in shadow, the height where the sun is first visible is estimated by increasing the world height by a fixed step size until the sun comes into view and then performing a binary search. In other words, we march along a vertical line and iterate to narrow down the location where it intersects the shadow map's surface that separates light from dark. Neighboring heights are interpolated to find this occlusion height at any location. An example of this technique used for soft shadowing of a terrain heightfield can be seen in Figure 7.32. We will see more use of ray marching through areas of light and dark in Chapter 14.

One last method worth a mention is rendering *screen-space shadows*. Shadow maps often fail to produce accurate occlusions on small features, because of their limited resolution. This is especially problematic when rendering human faces, because we are particularly prone to noticing any visual artifacts on them. For example, rendering glowing nostrils (when not intended) looks jarring. While using higher-resolution shadow maps or a separate shadow map targeting only the area of interest can help, another possibility is to leverage the already-existing data. In most modern rendering engines the depth buffer from the camera perspective, coming from an earlier prepass, is available during rendering. The data stored in it can be treated as a heightfield. By iteratively sampling this depth buffer, we can perform a ray-marching process (Section 6.8.1) and check if the direction toward the light is unoccluded. While costly, as it involves repeatedly sampling the depth buffer, doing so can provide high-quality results for closeups in cut scenes, where spending extra milliseconds is often justified. The method was proposed by Sousa at al. [1678] and is commonly used in many game engines today [384, 1802].

Figure 7.32. Terrain lit with the height where the sun is first seen computed for each heightfield location. Note how trees along the shadow's edge are properly shadowed [555]. *(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.)*

To summarize this whole chapter, shadow mapping in some form is by far the most common algorithm used for shadows cast onto arbitrary surface shapes. Cascaded shadow maps improve sampling quality when shadows are cast in a large area, such as an outdoor scene. Finding a good maximum distance for the near plane via SDSM can further improve precision. Percentage-closer filtering (PCF) gives some softness to the shadows, percentage-closer soft shadows (PCSS) and its variants give contact hardening, and the irregular z-buffer can provide precise hard shadows. Filtered shadow maps provide rapid soft-shadow computation and work particularly well when the occluder is far from the receiver, as with terrain. Finally, screen-space techniques can be used for additional precision, though at a noticeable cost.

In this chapter, we have focused on key concepts and techniques currently used in applications. Each has its own strengths, and choices depend on world size, composition (static content versus animated), material types (opaque, transparent, hair, or smoke), and number and type of lights (static or dynamic; local or distant; point, spot, or area), as well as factors such as how well the underlying textures can hide any artifacts. GPU capabilities evolve and improve, so we expect to continue seeing new algorithms that map well to the hardware appear in the years ahead. For example, the sparse-texture technique described in Section 19.10.1 has been applied to shadow-map storage to improve resolution [241, 625, 1253]. In an inventive approach, Sintorn,

Figure 7.33. At the top is an image generated with a basic soft-shadows approximation. At the bottom is voxel-based area light shadowing using cone tracing, on a voxelization of the scene. Note the considerably more diffuse shadows for the cars. Lighting also differs due to a change in the time of day. *(Images courtesy of Crytek [865].)*

Kämpe, and others [850, 1647] explore the idea of converting a two-dimensional shadow map for a light into a three-dimensional set of voxels (small boxes; see Section 13.10). An advantage of using a voxel is that it can be categorized as lit or in shadow, thus needing minimal storage. A highly compressed sparse voxel octree representation stores shadows for a huge number of lights and static occluders. Scandolo et al. [1546] combine their compression technique with an interval-based scheme using dual shadow maps, giving still higher compression rates. Kasyan [865] uses voxel cone tracing (Section 13.10) to generate soft shadows from area lights. See Figure 7.33 for an example. More cone-traced shadows are shown in Figure 13.33 on page 585.

Further Reading and Resources

Our focus in this chapter is on basic principles and what qualities a shadow algorithm needs—predictable quality and performance—to be useful for interactive rendering. We have avoiding an exhaustive categorization of the research done in this area of rendering, as two texts tackle the subject. The book *Real-Time Shadows* by Eisemann et al. [412] focuses directly on interactive rendering techniques, discussing a wide range of algorithms along with their strengths and costs. A SIGGRAPH 2012 course provides an excerpt of this book, while also adding references to newer work [413]. Presentations from their SIGGRAPH 2013 course are available at their website, www.realtimeshadows.com. Woo and Poulin's book *Shadow Algorithms Data Miner* [1902] provides an overview of a wide range of shadow algorithms for interactive and batch rendering. Both books supply references to hundreds of research articles in the field.

Tuft's pair of articles [1791, 1792] are an excellent overview of commonly used shadow-mapping techniques and the issues involved. Bjørge [154] presents a range of popular shadow algorithms suitable for mobile devices, along with images comparing various algorithms. Lilley's presentation [1046] gives a solid and extensive overview of practical shadow algorithms, with a focus on terrain rendering for GIS systems. Blog articles by Pettineo [1403, 1404] and Castaño [235] are particularly valuable for their practical tips and solutions, as well as a demo code base. See Scherzer et al. [1558] for a shorter summary of work specifically focused on hard shadows. The survey of algorithms for soft shadows by Hasenfratz et al. [675] is dated, but covers a wide range of early work in some depth.

Chapter 8
Light and Color

"Unweave a rainbow, as it erewhile made
The tender-person'd Lamia melt into a shade."
—John Keats

Many of the RGB color values discussed in previous chapters represent intensities and shades of light. In this chapter we will learn about the various physical light quantities measured by these values, laying the groundwork for subsequent chapters, which discuss rendering from a more physically based perspective. We will also learn more about the often-neglected "second half" of the rendering process: the transformation of colors that represent scene linear light quantities into final display colors.

8.1 Light Quantities

The first step in any physically based approach to rendering is to quantify light in a precise manner. Radiometry is presented first, as this is the core field concerned with the physical transmission of light. We follow with a discussion of photometry, which deals with light values that are weighted by the sensitivity of the human eye. Our perception of color is a *psychophysical* phenomenon: the psychological perception of physical stimuli. Color perception is discussed in the section on colorimetry. Finally, we discuss the validity of rendering with RGB color values.

8.1.1 Radiometry

Radiometry deals with the measurement of electromagnetic radiation. As will be discussed in more detail in Section 9.1, this radiation propagates as waves. Electromagnetic waves with different *wavelengths*—the distance between two adjacent points with the same phase, e.g., two adjacent peaks—tend to have different properties. In nature, electromagnetic waves exist across a huge range of wavelengths, from gamma waves less than a hundredth of a nanometer in length to extreme low frequency (ELF) radio waves tens of thousands of kilometers long. The waves that humans can see

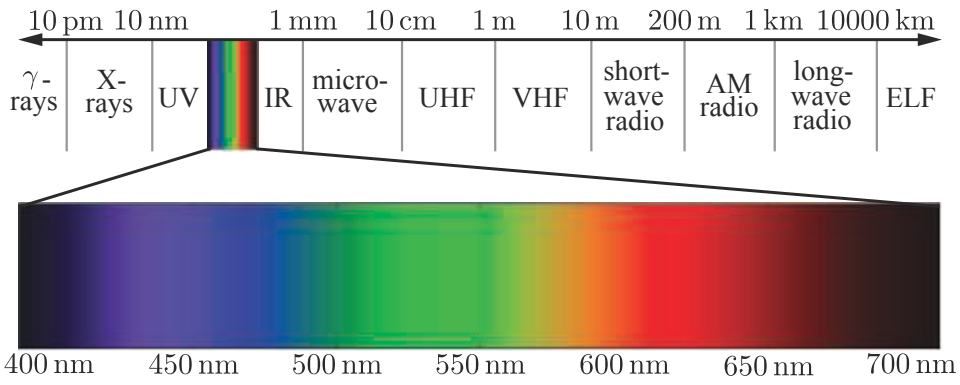

Figure 8.1. The range of wavelengths for visible light, shown in context within the full electromagnetic spectrum.

comprise a tiny subset of that range, extending from about 400 nanometers for violet light to a bit over 700 nanometers for red light. See Figure 8.1.

Radiometric quantities exist for measuring various aspects of electromagnetic radiation: overall energy, power (energy over time), and power density with respect to area, direction, or both. These quantities are summarized in Table 8.1.

In radiometry, the basic unit is *radiant flux*, Φ. Radiant flux is the flow of radiant energy over time—power—measured in *watts* (W).

Irradiance is the density of radiant flux with respect to area, i.e., $d\Phi/dA$. Irradiance is defined with respect to an area, which may be an imaginary area in space, but is most often the surface of an object. It is measured in watts per square meter.

Before we get to the next quantity, we need to first introduce the concept of a *solid angle*, which is a three-dimensional extension of the concept of an angle. An angle can be thought of as a measure of the size of a continuous set of directions in a plane, with a value in radians equal to the length of the arc this set of directions intersects on an enclosing circle with radius 1. Similarly, a solid angle measures the size of a continuous set of directions in three-dimensional space, measured in *steradians* (abbreviated "sr"), which are defined by the area of the intersection patch on an enclosing sphere with radius 1 [544]. Solid angle is represented by the symbol ω.

Name	**Symbol**	**Units**
radiant flux	Φ	watt (W)
irradiance	E	W/m^2
radiant intensity	I	W/sr
radiance	L	W/(m^{2}sr)

Table 8.1. Radiometric quantities and units.

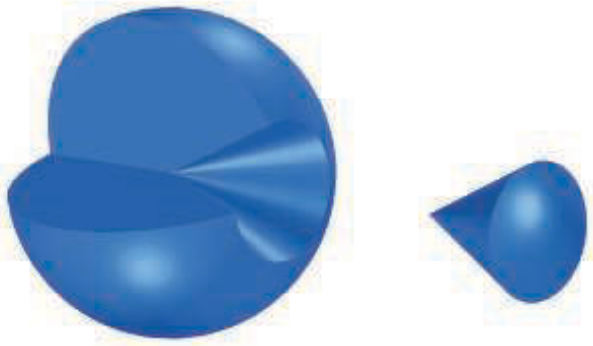

Figure 8.2. A cone with a solid angle of one steradian removed from a cutaway view of a sphere. The shape itself is irrelevant to the measurement. The coverage on the sphere's surface is the key.

In two dimensions, an angle of 2π radians covers the whole unit circle. Extending this to three dimensions, a solid angle of 4π steradians would cover the whole area of the unit sphere. The size of a solid angle of one steradian can be seen in Figure 8.2.

Now we can introduce *radiant intensity*, I, which is flux density with respect to direction—more precisely, solid angle ($d\Phi/d\omega$). It is measured in watts per steradian.

Finally, *radiance*, L, is a measure of electromagnetic radiation in a single ray. More precisely, it is defined as the density of radiant flux with respect to both area and solid angle ($d^2\Phi/dAd\omega$). This area is measured in a plane perpendicular to the ray. If radiance is applied to a surface at some other orientation, then a cosine correction factor must be used. You may encounter definitions of radiance using the term "projected area" in reference to this correction factor.

Radiance is what sensors, such as eyes or cameras, measure (see Section 9.2 for more details), so it is of prime importance for rendering. The purpose of evaluating a shading equation is to compute the radiance along a given ray, from the shaded surface point to the camera. The value of L along that ray is the physically based equivalent of the quantity $\mathbf{c}_{\text{shaded}}$ in Chapter 5. The metric units of radiance are watts per square meter per steradian.

The radiance in an environment can be thought of as a function of five variables (or six, including wavelength), called the *radiance distribution* [400]. Three of the variables specify a location, the other two a direction. This function describes all light traveling anywhere in space. One way to think of the rendering process is that the eye and screen define a point and a set of directions (e.g., a ray going through each pixel), and this function is evaluated at the eye for each direction. Image-based rendering, discussed in Section 13.4, uses a related concept, called the *light field*.

In shading equations, radiance often appears in the form $L_o(\mathbf{x}, \mathbf{d})$ or $L_i(\mathbf{x}, \mathbf{d})$, which mean radiance going out from the point $\mathbf{x}$ or entering into it, respectively. The direction vector $\mathbf{d}$ indicates the ray's direction, which by convention always points away from $\mathbf{x}$. While this convention may be somewhat confusing in the case of L_i,

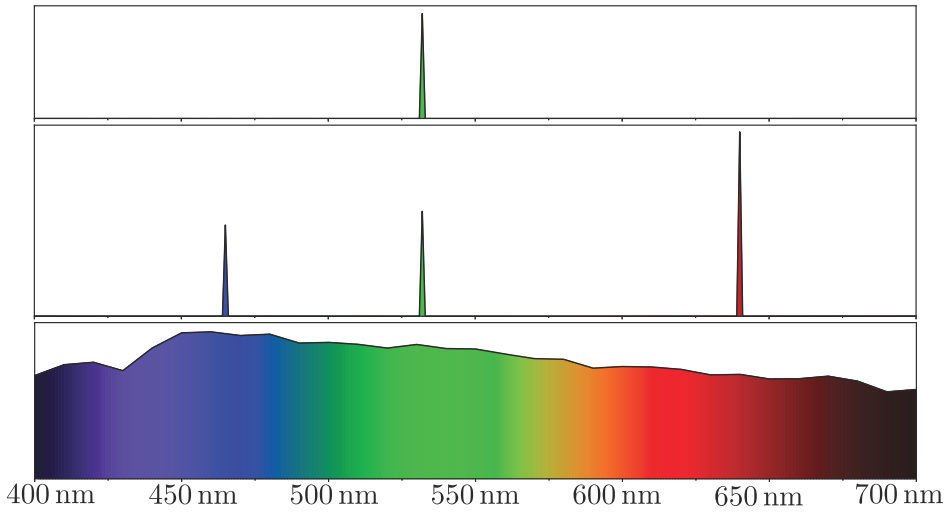

Figure 8.3. SPDs (spectral power distributions) for three different light waves. The top SPD is for a green laser, which has an extremely narrow spectral distribution. Its waveform is similar to the simple sine wave in Figure 9.1 on page 294. The middle SPD is for light comprised of the same green laser plus two additional lasers, one red and one blue. The wavelengths and relative intensities of these lasers correspond to an RGB laser projection display showing a neutral white color. The bottom SPD is for the standard D65 illuminant, which is a typical neutral white reference intended to represent outdoor lighting. Such SPDs, with energy continuously spread across the visible spectrum, are typical for natural lighting.

since $\mathbf{d}$ points in the opposite direction to the light propagation, it is convenient for calculations such as dot products.

An important property of radiance is that it is not affected by distance, ignoring atmospheric effects such as fog. In other words, a surface will have the same radiance regardless of its distance from the viewer. The surface covers fewer pixels when more distant, but the radiance from the surface at each pixel is constant.

Most light waves contain a mixture of many different wavelengths. This is typically visualized as a *spectral power distribution* (SPD), which is a plot showing how the light's energy is distributed across different wavelengths. Figure 8.3 shows three examples. Notably, despite the dramatic differences between the middle and bottom SPDs in Figure 8.3, they are perceived as the same color. Clearly, human eyes make for poor spectrometers. We will discuss color vision in detail in Section 8.1.3.

All radiometric quantities have spectral distributions. Since these distributions are densities over wavelength, their units are those of the original quantity divided by nanometers. For example, the spectral distribution of irradiance has units of watts per square meter per nanometer.

Since full SPDs are unwieldy to use for rendering, especially at interactive rates, in practice radiometric quantities are represented as RGB triples. In Section 8.1.3 we will explain how these triples relate to spectral distributions.

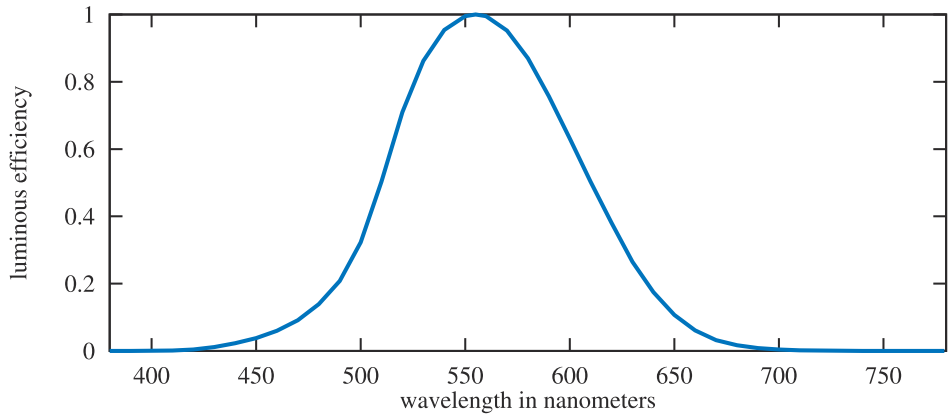

Figure 8.4. The photometric curve.

8.1.2 Photometry

Radiometry deals purely with physical quantities, without taking account of human perception. A related field, *photometry*, is like radiometry, except that it weights everything by the sensitivity of the human eye. The results of radiometric computations are converted to photometric units by multiplying by the *CIE photometric curve*,[1] a bell-shaped curve centered around 555 nm that represents the eye's response to various wavelengths of light [76, 544]. See Figure 8.4.

The conversion curve and the units of measurement are the only difference between the theory of photometry and the theory of radiometry. Each radiometric quantity has an equivalent metric photometric quantity. Table 8.2 shows the names and units of each. The units all have the expected relationships (e.g., lux is lumens per square meter). Although logically the lumen should be the basic unit, historically the candela was defined as a basic unit and the other units were derived from it. In North America, lighting designers measure illuminance using the deprecated Imperial unit of measurement, called the foot-candle (fc), instead of lux. In either case, illuminance is what most light meters measure, and it is important in illumination engineering.

Luminance is often used to describe the brightness of flat surfaces. For example, high dynamic range (HDR) television screens' peak brightness typically ranges from about 500 to 1000 nits. In comparison, clear sky has a luminance of about 8000 nits, a 60-watt bulb about 120,000 nits, and the sun at the horizon 600,000 nits [1413].

[1] The full and more accurate name is the "CIE photopic spectral luminous efficiency curve." The word "photopic" refers to lighting conditions brighter than 3.4 candelas per square meter—twilight or brighter. Under these conditions the eye's cone cells are active. There is a corresponding "scotopic" CIE curve, centered around 507 nm, that is for when the eye has become dark-adapted to below 0.034 candelas per square meter—a moonless night or darker. The rod cells are active under these conditions.

Radiometric Quantity: Units	Photometric Quantity: Units
radiant flux: watt (W)	luminous flux: lumen (lm)
irradiance: W/m^2	illuminance: lux (lx)
radiant intensity: W/sr	luminous intensity: candela (cd)
radiance: $W/(m^2 sr)$	luminance: cd/m^2 = nit

Table 8.2. Radiometric and photometric quantities and units.

8.1.3 Colorimetry

In Section 8.1.1 we have seen that our perception of the color of light is strongly connected to the light's SPD (spectral power distribution). We also saw that this is not a simple one-to-one correspondence. The bottom and middle SPDs in Figure 8.3 are completely different yet are perceived as the exact same color. *Colorimetry* deals with the relationship between spectral power distributions and the perception of color.

Humans can distinguish about 10 million different colors. For color perception, the eye works by having three different types of cone receptors in the retina, with each type of receptor responding differently to various wavelengths. Other animals have varying numbers of color receptors, in some cases as many as fifteen [260]. So, for a given SPD, our brain receives only three different signals from these receptors. This is why just three numbers can be used to precisely represent any color stimulus [1707].

But what three numbers? A set of standard conditions for measuring color was proposed by the CIE (*Commission Internationale d'Eclairage*), and color-matching experiments were performed using them. In color matching, three colored lights are projected on a white screen so that their colors add together and form a patch. A test color to match is projected next to this patch. The test color patch is of a single wavelength. The observer can then change the three colored lights using knobs calibrated to a range weighted $[-1, 1]$ until the test color is matched. A negative weight is needed to match some test colors, and such a weight means that the corresponding light is added instead to the wavelength's test color patch. One set of test results for three lights, called r, g, and b, is shown in Figure 8.5. The lights were almost monochromatic, with the energy distribution of each narrowly clustered around one of the following wavelengths: 645 nm for r, 526 nm for g, and 444 nm for b. The functions relating each set of matching weights to the test patch wavelengths are called *color-matching functions*.

What these functions give is a way to convert a spectral power distribution to three values. Given a single wavelength of light, the three colored light settings can be read off the graph, the knobs set, and lighting conditions created that will give an identical sensation from both patches of light on the screen. For an arbitrary spectral distribution, the color-matching functions can be multiplied by the distribution and the area under each resulting curve (i.e., the integral) gives the relative amounts

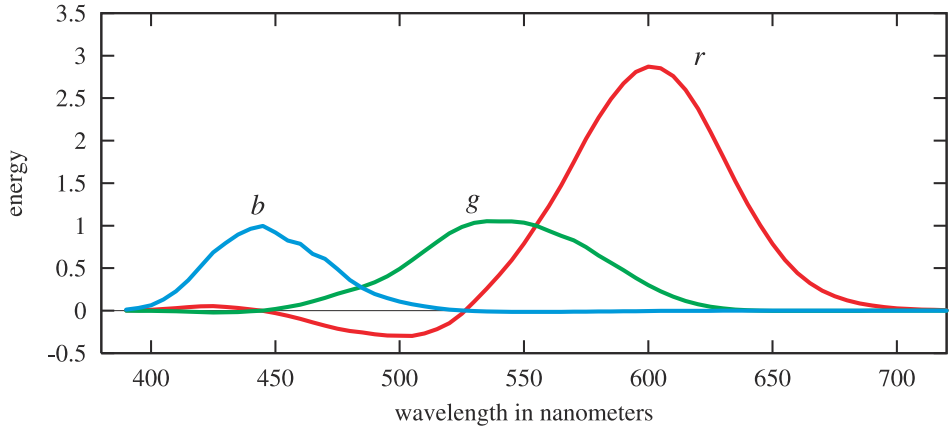

Figure 8.5. The r, g, and b 2-degree color-matching curves, from Stiles and Burch [1703]. These color-matching curves are not to be confused with the spectral distributions of the light sources used in the color-matching experiment, which are pure wavelengths.

to set the colored lights to match the perceived color produced by the spectrum. Considerably different spectral distributions can resolve to the same three weights, i.e., they look the same to an observer. Spectral distributions that give matching weights are called *metamers*.

The three weighted r, g, and b lights cannot directly represent all visible colors, as their color-matching functions have negative weights for various wavelengths. The CIE proposed three different hypothetical light sources with color-matching functions that are positive for all visible wavelengths. These curves are linear combinations of the original r, g, and b color-matching functions. This requires their spectral power distributions of the light sources to be negative at some wavelengths, so these lights are unrealizable mathematical abstractions. Their color-matching functions are denoted $\overline{x}(\lambda)$, $\overline{y}(\lambda)$, and $\overline{z}(\lambda)$, and are shown in Figure 8.6. The color-matching function $\overline{y}(\lambda)$ is the same as the photometric curve (Figure 8.4), as radiance is converted to luminance with this curve.

As with the previous set of color-matching functions, $\overline{x}(\lambda)$, $\overline{y}(\lambda)$, and $\overline{z}(\lambda)$ are used to reduce any SPD $s(\lambda)$ to three numbers via multiplication and integration:

$$X = \int_{380}^{780} s(\lambda)\overline{x}(\lambda)d\lambda, \quad Y = \int_{380}^{780} s(\lambda)\overline{y}(\lambda)d\lambda, \quad Z = \int_{380}^{780} s(\lambda)\overline{z}(\lambda)d\lambda. \tag{8.1}$$

These X, Y, and Z *tristimulus values* are weights that define a color in CIE XYZ space. It is often convenient to separate colors into luminance (brightness) and *chromaticity*. Chromaticity is the character of a color independent of its brightness. For example, two shades of blue, one dark and one light, can have the same chromaticity despite differing in luminance.

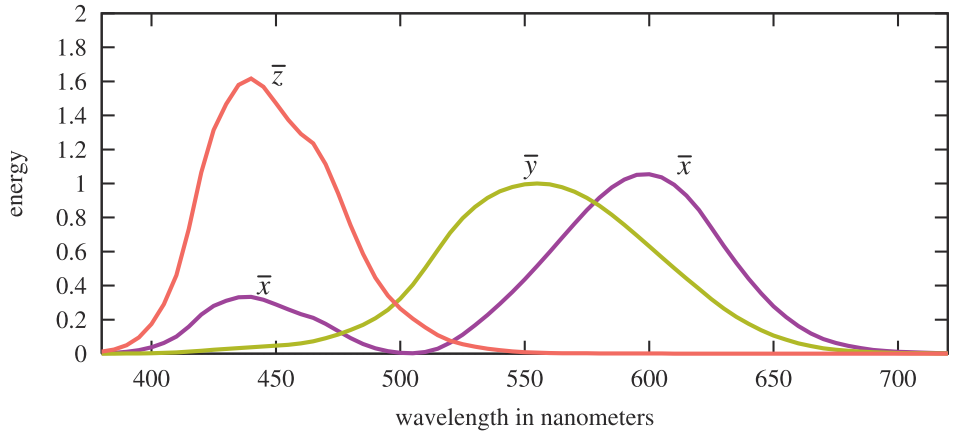

Figure 8.6. The Judd-Vos-modified CIE (1978) 2-degree color-matching functions. Note that the two $\bar{x}$'s are part of the same curve.

For this purpose, the CIE defined a two-dimensional chromaticity space by projecting colors onto the $X+Y+Z=1$ plane. See Figure 8.7. Coordinates in this space are called x and y, and are computed as follows:

$$\begin{aligned} x &= \frac{X}{X+Y+Z}, \\ y &= \frac{Y}{X+Y+Z}, \\ z &= \frac{Z}{X+Y+Z} = 1-x-y. \end{aligned} \tag{8.2}$$

The z value gives no additional information, so it is normally omitted. The plot of the *chromaticity coordinates* x and y values is known as the *CIE 1931 chromaticity diagram*. See Figure 8.8. The curved outline in the diagram shows where the colors of the visible spectrum lie, and the straight line connecting the ends of the spectrum is called the *purple line*. The black dot shows the chromaticity of illuminant D65, which is a frequently used *white point*—a chromaticity used to define the white or *achromatic* (colorless) stimulus.

To summarize, we began with an experiment that used three single-wavelength lights and measured how much of each was needed to match the appearance of some other wavelength of light. Sometimes these pure lights had to be added to the sample being viewed in order to match. This gave one set of color-matching functions, which were combined to create a new set without negative values. With this non-negative set of color-matching functions in hand, we can convert any spectral distribution to an XYZ coordinate that defines a color's chromaticity and luminance, which can be reduced to xy to describe just the chromaticity, keeping luminance constant.

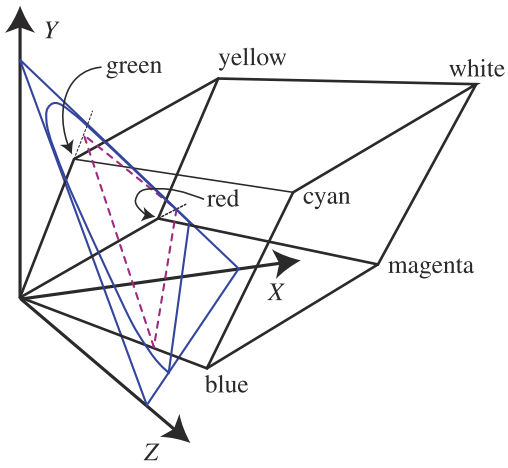

Figure 8.7. The RGB color cube for the CIE RGB primaries is shown in XYZ space, along with its projection (in violet) onto the $X + Y + Z = 1$ plane. The blue outline encloses the space of possible chromaticity values. Each line radiating from the origin has a constant chromaticity value, varying only in luminance.

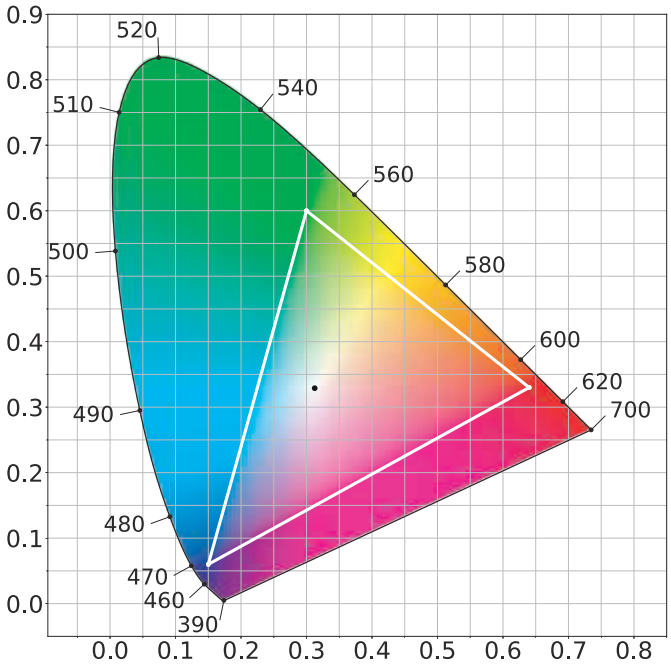

Figure 8.8. The CIE 1931 chromaticity diagram. The curve is labeled with the wavelengths of the corresponding pure colors. The white triangle and black dot show the gamut and white point, respectively, used for the sRGB and Rec. 709 color spaces.

Given a color point (x, y), draw a line from the white point through this point to the boundary (spectral or purple line). The relative distance of the color point compared to the distance to the edge of the region is the *excitation purity* of the color. The point on the region edge defines the *dominant wavelength*. These colorimetric terms are rarely encountered in graphics. Instead, we use *saturation* and *hue*, which correlate loosely with excitation purity and dominant wavelength, respectively. More precise definitions of saturation and hue can be found in books by Stone [1706] and others [456, 789, 1934].

The chromaticity diagram describes a plane. The third dimension needed to fully describe a color is the Y value, luminance. These then define what is called the xyY-coordinate system. The chromaticity diagram is important in understanding how color is used in rendering, and the limits of the rendering system. A television or computer monitor presents colors by using some settings of R, G, and B color values. Each color channel controls a *display primary* that emits light with a particular spectral power distribution. Each of the three primaries is scaled by its respective color value, and these are added together to create a single spectral power distribution that the viewer perceives.

The triangle in the chromaticity diagram represents the *gamut* of a typical television or computer monitor. The three corners of the triangle are the primaries, which are the most saturated red, green, and blue colors the screen can display. An important property of the chromaticity diagram is that these limiting colors can be joined by straight lines to show the limits of the display system as a whole. The straight lines represent the limits of colors that can be displayed by mixing these three primaries. The white point represents the chromaticity that is produced by the display system when the R, G, and B color values are equal to each other. It is important to note that the full gamut of a display system is a three-dimensional volume. The chromaticity diagram shows only the projection of this volume onto a two-dimensional plane. See Stone's book [1706] for more information.

There are several RGB spaces of interest in rendering, each defined by R, G, and B primaries and a white point. To compare them we will use a different type of chromaticity diagram, called the *CIE 1976 UCS* (uniform chromaticity scale) diagram. This diagram is part of the CIELUV color space, which was adopted by the CIE (along with another color space, CIELAB) with the intention of providing more perceptually uniform alternatives to the XYZ space [1707]. Color pairs that are perceptibly different by the same amount can be up to 20 times different in distance in CIE XYZ space. CIELUV improves upon this, bringing the ratio down to a maximum of four times. This increased perceptual uniformity makes the 1976 diagram much better than the 1931 one for the purpose of comparing the gamuts of RGB spaces. Continued research into perceptually uniform color spaces has recently resulted in the $\mathrm{IC_TC_P}$ [364] and $\mathrm{J_za_zb_z}$ [1527] spaces. These color spaces are more perceptually uniform than CIELUV, especially for the high luminance and saturated colors typical of modern displays. However, chromaticity diagrams based on these color spaces have not yet been widely adopted, so we use the CIE 1976 UCS diagrams in this chapter, for example in the case of Figure 8.9.

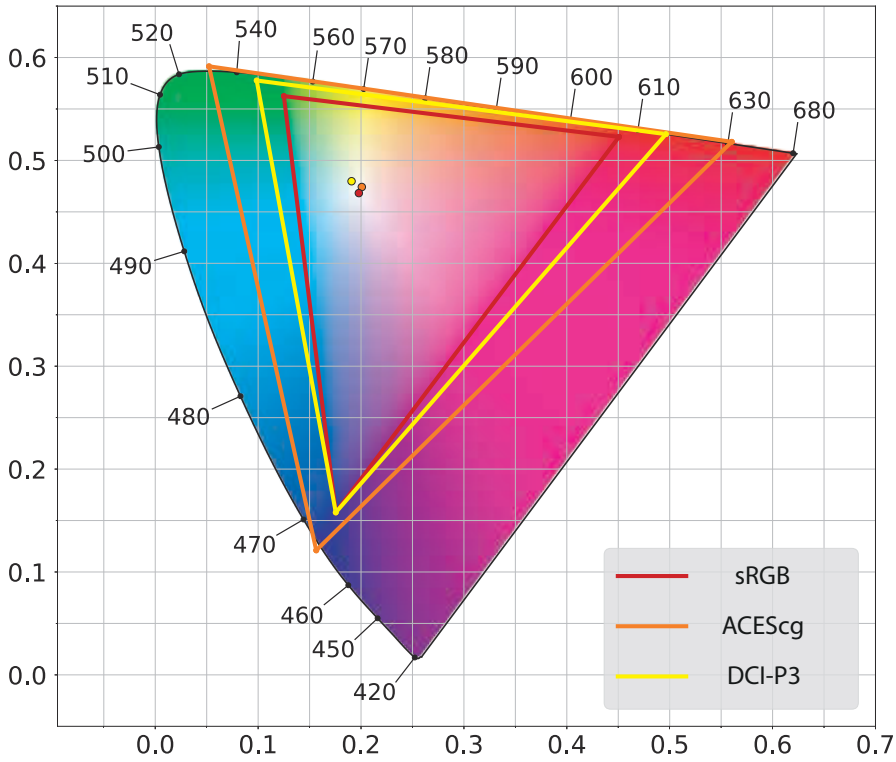

Figure 8.9. A CIE 1976 UCS diagram showing the primaries and white points of three RGB color spaces: sRGB, DCI-P3, and ACEScg. The sRGB plot can be used for Rec. 709 as well, since the two color spaces have the same primaries and white point.

Of the three RGB spaces shown in Figure 8.9, sRGB is by far the most commonly used in real-time rendering. It is important to note that in this section we use "sRGB color space" to refer to a linear color space that has the sRGB primaries and white point, and not to the nonlinear sRGB color encoding that was discussed in Section 5.6. Most computer monitors are designed for the sRGB color space, and the same primaries and white point apply to the Rec. 709 color space as well, which is used for HDTV displays and thus is important for game consoles. However, more displays are being made with wider gamuts. Some computer monitors intended for photo editing use the Adobe 1998 color space (not shown). The DCI-P3 color space—initially developed for feature film production—is seeing broader use. Apple has adopted this color space across their product line from iPhones to Macs, and other manufacturers have been following suit. Although ultra-high definition (UHD) content and displays are specified to use the extremely-wide-gamut Rec. 2020 color space, in many cases DCI-P3 is used as a de facto color space for UHD as well. Rec. 2020 is not shown in

Figure 8.9, but its gamut is quite close to that of the third color space in the figure, ACEScg. The ACEScg color space was developed by the Academy of Motion Picture Arts and Sciences (AMPAS) for feature film computer graphics rendering. It is not intended for use as a display color space, but rather as a *working color space* for rendering, with colors converted to the appropriate display color space after rendering.

While currently the sRGB color space is ubiquitous in real-time rendering, the use of wider color spaces is likely to increase. The most immediate benefit is for applications targeting wide-gamut displays [672], but there are advantages even for applications targeting sRGB or Rec. 709 displays. Routine rendering operations such as multiplication give different results when performed in different color spaces [672, 1117], and there is evidence that performing these operations in the DCI-P3 or ACEScg space produces more accurate results than performing them in linear sRGB space [660, 975, 1118].

Conversion from an RGB space to XYZ space is linear and can be done with a matrix derived from the RGB space's primaries and white point [1048]. Via matrix inversion and concatenation, matrices can be derived to convert from XYZ to any RGB space, or between two different RGB spaces. Note that after such a conversion the RGB values may be negative or greater than one. These are colors that are out of gamut, i.e., not reproducible in the target RGB space. Various methods can be used to map such colors into the target RGB gamut [785, 1241].

One often-used conversion is to transform an RGB color to a grayscale luminance value. Since luminance is the same as the Y coefficient, this operation is just the "Y part" of the RGB-to-XYZ conversion. In other words, it is a dot product between the RGB coefficients and the middle row of the RGB-to-XYZ matrix. In the case of the sRGB and Rec. 709 spaces, the equation is [1704]

$$Y = 0.2126R + 0.7152G + 0.0722B. \tag{8.3}$$

This brings us again to the photometric curve, shown in Figure 8.4 on page 271. This curve, representing how a standard observer's eye responds to light of various wavelengths, is multiplied by the spectral power distributions of the three primaries, and each resulting curve is integrated. The three resulting weights are what form the luminance equation above. The reason that a grayscale intensity value is not equal parts red, green, and blue is because the eye has a different sensitivity to various wavelengths of light.

Colorimetry can tell us whether two color stimuli match, but it cannot predict their appearance. The appearance of a given XYZ color stimulus depends heavily on factors such as the lighting, surrounding colors, and previous conditions. *Color appearance models* (CAM) such as CIECAM02 attempt to deal with these issues and predict the final color appearance [456].

Color appearance modeling is part of the wider field of visual perception, which includes effects such as *masking* [468]. This is where a high-frequency, high-contrast pattern laid on an object tends to hide flaws. In other words, a texture such as a

Persian rug will help disguise color banding and other shading artifacts, meaning that less rendering effort needs to be expended for such surfaces.

8.1.4 Rendering with RGB Colors

Strictly speaking, RGB values represent perceptual rather than physical quantities. Using them for physically based rendering is technically a category error. The correct method would be to perform all rendering computations on spectral quantities, represented either via dense sampling or projection onto a suitable basis, and to convert to RGB colors only at the end.

For example, one of the most common rendering operations is calculating the light reflected from an object. The object's surface typically will reflect light of some wavelengths more than others, as described by its *spectral reflectance* curve. The strictly correct way to compute the color of the reflected light is to multiply the SPD of the incident light by the spectral reflectance at each wavelength, yielding the SPD of the reflected light that would then be converted to an RGB color. Instead, in an RGB renderer the RGB colors of the lights and surface are multiplied together to give the RGB color of the reflected light. In the general case, this does not give the correct result. To illustrate, we will look at a somewhat extreme example, shown in Figure 8.10.

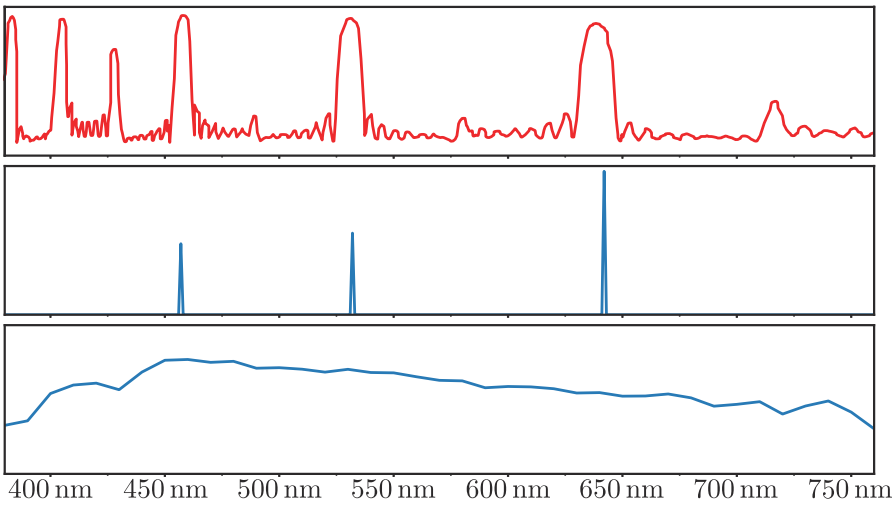

Figure 8.10. The top plot shows the spectral reflectance of a material designed for use in projection screens. The lower two plots show the spectral power distributions of two illuminants with the same RGB colors: an RGB laser projector in the middle plot and the D65 standard illuminant in the bottom plot. The screen material would reflect about 80% of the light from the laser projector because it has reflectance peaks that line up with the projectors primaries. However, it will reflect less than 20% of the light from the D65 illuminant since most of the illuminant's energy is outside the screen's reflectance peaks. An RGB rendering of this scene would predict that the screen would reflect the same intensity for both lights.