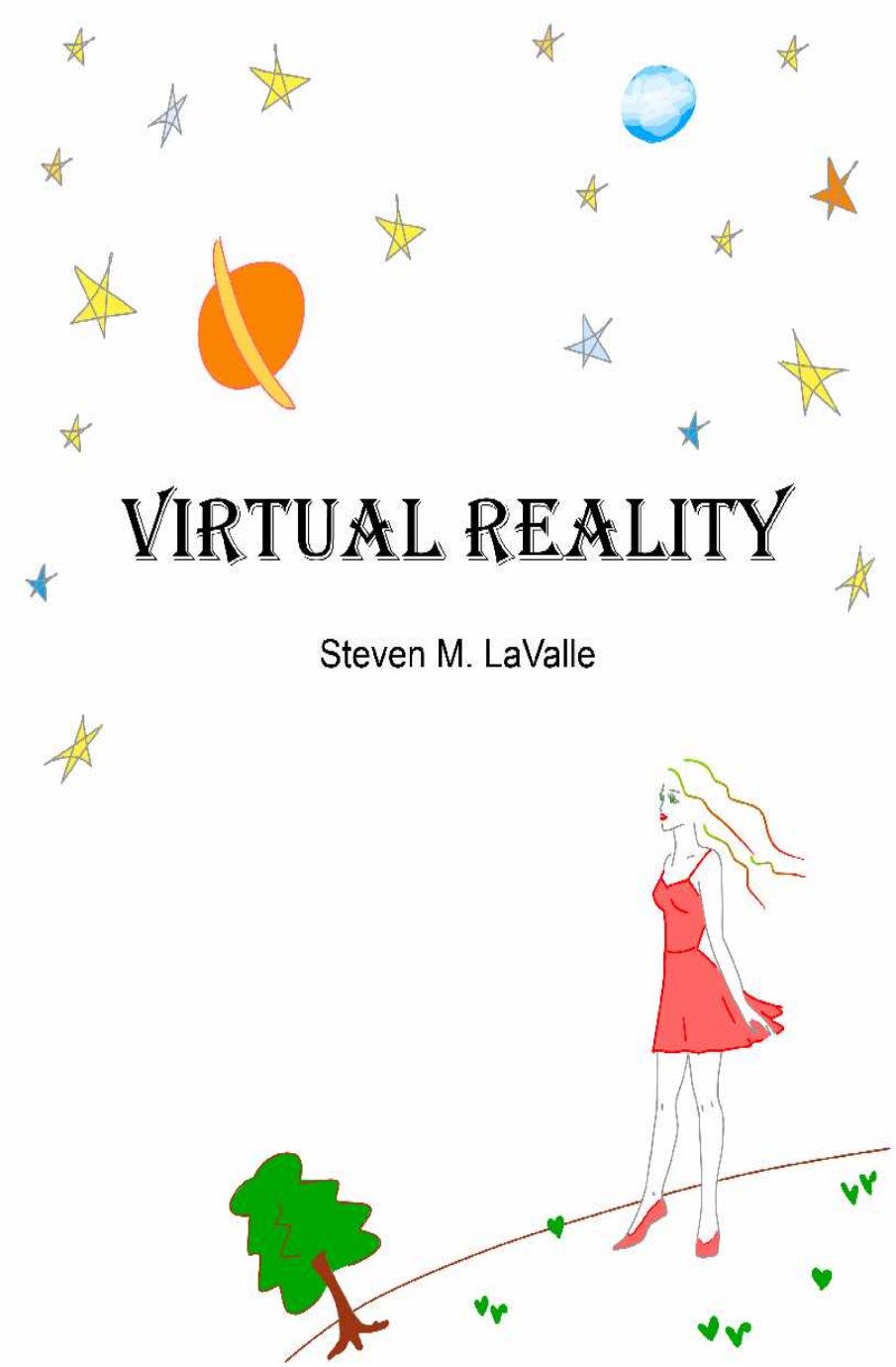


Chapter 7

Visual Rendering

Steven M. LaValle

University of Oulu

Available for downloading at **http://lavalle.pl/vr/**

Chapter 7

Visual Rendering

This chapter addresses visual rendering, which specifies what the visual display should show through an interface to the virtual world generator (VWG). Chapter 3 already provided the mathematical parts, which express *where* the objects in the virtual world should appear on the screen. This was based on geometric models, rigid body transformations, and viewpoint transformations. We next need to determine *how* these objects should appear, based on knowledge about light propagation, visual physiology, and visual perception. These were the topics of Chapters 4, 5, and 6, respectively. Thus, visual rendering is a culmination of everything covered so far.

Sections 7.1 and 7.2 cover the basic concepts; these are considered the core of computer graphics, but VR-specific issues also arise. They mainly address the case of rendering for virtual worlds that are formed synthetically. Section 7.1 explains how to determine the light that should appear at a pixel based on light sources and the reflectance properties of materials that exist purely in the virtual world. Section 7.2 explains rasterization methods, which efficiently solve the rendering problem and are widely used in specialized graphics hardware, called GPUs. Section 7.3 addresses VR-specific problems that arise from imperfections in the optical system. Section 7.4 focuses on latency reduction, which is critical to VR, so that virtual objects appear in the right place at the right time. Otherwise, many side effects could arise, such as VR sickness, fatigue, adaptation to the flaws, or simply having an unconvincing experience. Finally, Section 7.5 explains rendering for captured, rather than synthetic, virtual worlds. This covers VR experiences that are formed from panoramic photos and videos.

7.1 Ray Tracing and Shading Models

Suppose that a virtual world has been defined in terms of triangular primitives. Furthermore, a virtual eye has been placed in the world to view it from some particular position and orientation. Using the full chain of transformations from Chapter 3, the location of every triangle is correctly positioned onto a virtual

screen (this was depicted in Figure 3.13). The next steps are to determine which screen pixels are covered by the transformed triangle and then illuminate them according to the physics of the virtual world.

An important condition must also be checked: For each pixel, is the triangle even *visible* to the eye, or will it be blocked by part of another triangle? This classic *visibility computation* problem dramatically complicates the rendering process. The general problem is to determine for any pair of points in the virtual world, whether the line segment that connects them intersects with any objects (triangles). If an intersection occurs, then the line-of-sight visibility between the two points is blocked. The main difference between the two major families of rendering methods is how visibility is handled.

Object-order versus image-order rendering For rendering, we need to consider all combinations of objects and pixels. This suggests a nested loop. One way to resolve the visibility is to iterate over the list of all triangles and attempt to render each one to the screen. This is called *object-order rendering*, and is the main topic of Section 7.2. For each triangle that falls into the field of view of the screen, the pixels are updated *only if* the corresponding part of the triangle is closer to the eye than any triangles that have been rendered so far. In this case, the outer loop iterates over triangles whereas the inner loop iterates over pixels. The other family of methods is called *image-order rendering*, and it reverses the order of the loops: Iterate over the image pixels and for each one, determine which triangle should influence its RGB values. To accomplish this, the path of light waves that would enter each pixel is traced out through the virtual environment. This method will be covered first, and many of its components apply to object-order rendering as well.

Ray tracing To calculate the RGB values at a pixel, a *viewing ray* is drawn from the focal point through the center of the pixel on a virtual screen that is placed in the virtual world; see Figure 7.1. The process is divided into two phases:

1. *Ray casting*, in which the viewing ray is defined and its nearest point of intersection among all triangles in the virtual world is calculated.

2. *Shading*, in which the pixel RGB values are calculated based on lighting conditions and material properties at the intersection point.

The first step is based entirely on the virtual world geometry. The second step uses simulated physics of the virtual world. Both the material properties of objects and the lighting conditions are artificial, and are chosen to produce the desired effect, whether realism or fantasy. Remember that the ultimate judge is the user, who interprets the image through perceptual processes.

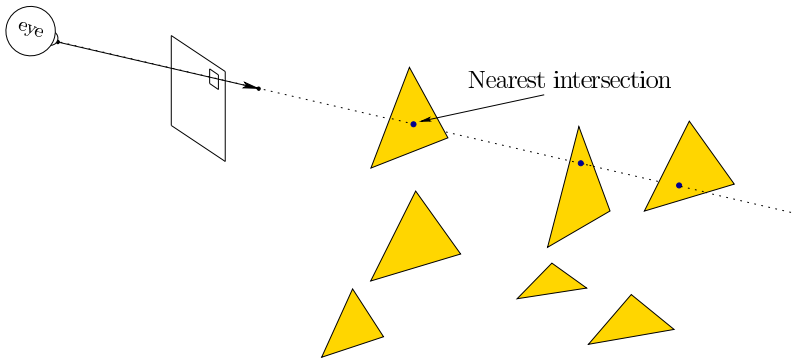


Figure 7.1: The first step in a ray tracing approach is called *ray casting*, which extends a viewing ray that corresponds to a particular pixel on the image. The ray starts at the focal point, which is the origin after the eye transform T_{eye} has been applied. The task is to determine what part of the virtual world model is visible. This is the closest intersection point between the viewing ray and the set of all triangles.

Ray casting Calculating the first triangle hit by the viewing ray after it leaves the image pixel (Figure 7.1) is straightforward if we neglect the computational performance. Starting with the triangle coordinates, focal point, and the ray direction (vector), the closed-form solution involves basic operations from analytic geometry, including dot products, cross products, and the plane equation [26]. For each triangle, it must be determined whether the ray intersects it. If not, then the next triangle is considered. If it does, then the intersection is recorded as the candidate solution only if it is closer than the closest intersection encountered so far. After all triangles have been considered, the closest intersection point will be found. Although this is simple, it is far more efficient to arrange the triangles into a 3D data structure. Such structures are usually hierarchical so that many triangles can be eliminated from consideration by quick coordinate tests. Popular examples include BSP-trees and Bounding Volume Hierarchies [7, 11]. Algorithms that sort geometric information to obtain greater efficiently generally fall under *computational geometry* [9]. In addition to eliminating many triangles from quick tests, many methods of calculating the ray-triangle intersection have been developed to reduce the number of operations. One of the most popular is the *Möller-Trumbore intersection algorithm* [19].

Lambertian shading Now consider lighting each pixel and recall the basic behavior of light from Section 4.1. The virtual world simulates the real-world physics, which includes the spectral power distribution and spectral reflection function. Suppose that a point-sized light source is placed in the virtual world. Using the trichromatic theory from Section 6.3, its spectral power distribution

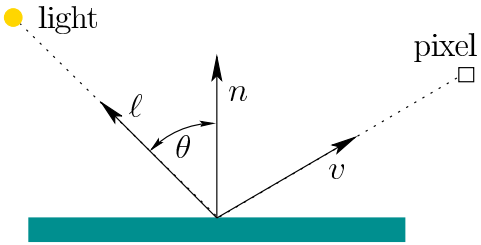


Figure 7.2: In the *Lambertian shading* model, the light reaching the pixel depends on the angle θ between the incoming light and the surface normal, but is independent of the viewing angle.

is sufficiently represented by R, G, and B values. If the viewing ray hits the surface as shown in Figure 7.2, then how should the object appear? Assumptions about the spectral reflection function are taken into account by a *shading model*. The simplest case is *Lambertian shading*, for which the angle that the viewing ray strikes the surface is independent of the resulting pixel R, G, B values. This corresponds to the case of diffuse reflection, which is suitable for a "rough" surface (recall Figure 4.4). All that matters is the angle that the surface makes with respect to the light source.

Let n be the outward surface normal and let ℓ be a vector from the surface intersection point to the light source. Assume both n and ℓ are unit vectors, and let θ denote the angle between them. The dot product $n \cdot \ell = \cos\theta$ yields the amount of attenuation (between 0 and 1) due to the tilting of the surface relative to the light source. Think about how the effective area of the triangle is reduced due to its tilt. A pixel under the *Lambertian shading* model is illuminated as

$$\begin{aligned} R &= d_R I_R \max(0, n \cdot \ell) \\ G &= d_G I_G \max(0, n \cdot \ell) \\ B &= d_B I_B \max(0, n \cdot \ell), \end{aligned} \tag{7.1}$$

in which (d_R, d_G, d_B) represents the spectral reflectance property of the material (triangle) and (I_r, I_G, I_R) is represents the spectral power distribution of the light source. Under the typical case of white light, $I_R = I_G = I_B$. For a white or gray material, we would also have $d_R = d_G = d_B$.

Using vector notation, (7.1) can be compressed into

$$L = dI \max(0, n \cdot \ell) \tag{7.2}$$

in which $L = (R, G, B)$, $d = (d_R, d_G, d_B)$, and $I = (I_R, I_G, I_B)$. Each triangle is assumed to be on the surface of an object, rather than the object itself. Therefore, if the light source is behind the triangle, then the triangle should not be illuminated because it is facing away from the light (it cannot be lit from behind). To handle this case, the max function appears in (7.2) to avoid $n \cdot \ell < 0$.

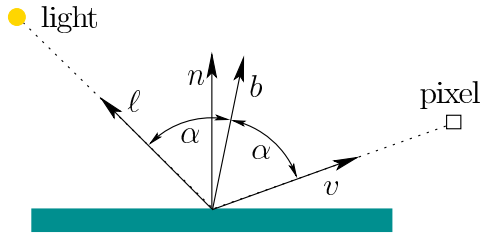


Figure 7.3: In the *Blinn-Phong shading* model, the light reaching the pixel depends on the angle between the normal n and the bisector b of the ℓ and v. If $n = b$, then ideal reflection is obtained, as in the case of a mirror.

Blinn-Phong shading Now suppose that the object is "shiny". If it were a perfect mirror, then all of the light from the source would be reflected to the pixel only if they are perfectly aligned; otherwise, no light would reflect at all. Such full reflection would occur if v and ℓ form the same angle with respect to n. What if the two angles are close, but do not quite match? The *Blinn-Phong shading* model proposes that some amount of light is reflected, depending on the amount of surface shininess and the difference between v and ℓ [3]. See Figure 7.3. The *bisector* b is the vector obtained by averaging ℓ and v:

$$b = \frac{\ell + v}{\|\ell + v\|}. \tag{7.3}$$

Using the compressed vector notation, the *Blinn-Phong shading* model sets the RGB pixel values as

$$L = dI \max(0, n \cdot \ell) + sI \max(0, n \cdot b)^x. \tag{7.4}$$

This additively takes into account shading due to both diffuse and specular components. The first term is just the Lambertian shading model, (7.2). The second component causes increasing amounts of light to be reflected as b becomes closer to n. The exponent x is a material property that expresses the amount of surface shininess. A lower value, such as $x = 100$, results in a mild amount of shininess, whereas $x = 10000$ would make the surface almost like a mirror. This shading model does not correspond directly to the physics of the interaction between light and surfaces. It is merely a convenient and efficient heuristic, but widely used in computer graphics.

Ambient shading Another heuristic is *ambient shading*, which causes an object to glow without being illuminated by a light source. This lights surfaces that fall into the shadows of all lights; otherwise, they would be completely black. In the real world this does not happen because light interreflects between objects to illuminate an entire environment. Such propagation has not been taken into

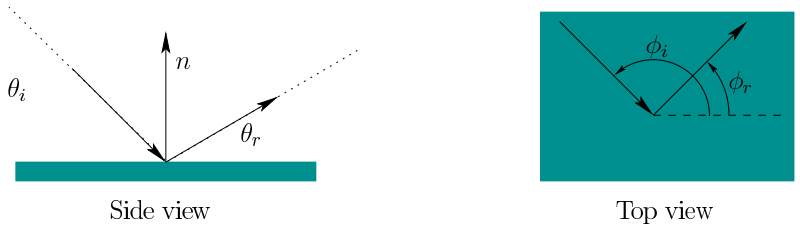


Figure 7.4: A *bidirectional reflectance distribution function* (*BRDF*), meticulously specifies the ratio of incoming and outgoing light energy for all possible perspectives.

account in the shading model so far, thereby requiring a hack to fix it. Adding ambient shading yields

$$L = dI \max(0, n \cdot \ell) + sI \max(0, n \cdot b)^x + L_a, \tag{7.5}$$

in which L_a is the ambient light component.

Multiple light sources Typically, the virtual world contains multiple light sources. In this case, the light from each is combined additively at the pixel. The result for N light sources is

$$L = L_a + \sum_{i=1}^{N} dI_i \max(0, n \cdot \ell_i) + sI_i \max(0, n \cdot b_i)^x, \tag{7.6}$$

in which I_i, ℓ_i, and b_i correspond to each source.

BRDFs The shading models presented so far are in widespread use due to their simplicity and efficiency, even though they neglect most of the physics. To account for shading in a more precise and general way, a *bidirectional reflectance distribution function* (*BRDF*) is constructed [21]; see Figure 7.4. The θ_i and θ_r parameters represent the angles of light source and viewing ray, respectively, with respect to the surface. The ϕ_i and ϕ_r parameters range from 0 to 2π and represent the angles made by the light and viewing vectors when looking straight down on the surface (the vector n would point at your eye).

The BRDF is a function of the form

$$f(\theta_i, \phi_i, \theta_r, \theta_i) = \frac{\text{radiance}}{\text{irradiance}}, \tag{7.7}$$

in which *radiance* is the light energy reflected from the surface in directions θ_r and ϕ_r and *irradiance* is the light energy arriving at the surface from directions θ_i and ϕ_i. These are expressed at a differential level, roughly corresponding to an infinitesimal surface patch. Informally, it is the ratio of the amount of outgoing

light to the amount of incoming light at one point on the surface. The previous shading models can be expressed in terms of a simple BRDF. For Lambertian shading, the BRDF is constant because the surface reflects equally in all directions. The BRDF and its extensions can account for much more complex and physically correct lighting effects, with a wide variety of surface textures. See Chapter 7 of [1] for extensive coverage.

Global illumination Recall that the ambient shading term (7.5) was introduced to prevent surfaces in the shadows of the light source from appearing black. The computationally intensive but proper way to fix this problem is to calculate how light reflects from object to object in the virtual world. In this way, objects are illuminated *indirectly* from the light that reflects from others, as in the real world. Unfortunately, this effectively turns all object surfaces into potential sources of light. This means that ray tracing must account for multiple reflections. This requires considering piecewise linear paths from the light source to the viewpoint, in which each bend corresponds to a reflection. An upper limit is usually set on the number of bounces to consider. The simple Lambertian and Blinn-Phong models are often used, but more general BDRFs are also common. Increasing levels of realism can be calculated, but with corresponding increases in computation time.

VR-specific issues VR inherits all of the common issues from computer graphics, but also contains unique challenges. Chapters 5 and 6 mentioned the increased resolution and frame rate requirements. This provides strong pressure to reduce rendering complexity. Furthermore, many heuristics that worked well for graphics on a screen may be perceptibly wrong in VR. The combination of high field-of-view, resolution, varying viewpoints, and stereo images may bring out new problems. For example, Figure 7.5 illustrates how differing viewpoints from stereopsis could affect the appearance of shiny surfaces. In general, some rendering artifacts could even contribute to VR sickness. Throughout the remainder of this chapter, complications that are unique to VR will be increasingly discussed.

7.2 Rasterization

The ray casting operation quickly becomes a bottleneck. For a 1080p image at 90Hz, it would need to be performed over 180 million times per second, and the ray-triangle intersection test would be performed for every triangle (although data structures such as a BSP would quickly eliminate many from consideration). In most common cases, it is much more efficient to switch from such image-order rendering to object-order rendering. The objects in our case are triangles and the resulting process is called *rasterization*, which is the main function of modern graphical processing units (GPUs). In this case, an image is rendered by iterating over every triangle and attempting to color the pixels where the triangle lands

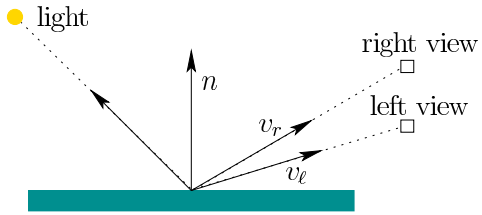


Figure 7.5: Complications emerge with shiny surfaces because the viewpoints are different for the right and left eyes. Using the Blinn-Phong shading model, a specular reflection should have different brightness levels for each eye. It may be difficult to match the effect so that it is consistent with real-world behavior.

on the image. The main problem is that the method must solve the unavoidable problem of determining which part, if any, of the triangle is the closest to the focal point (roughly, the location of the virtual eye).

One way to solve it is to sort the triangles in *depth order* so that the closest triangle is last. This enables the triangles to be drawn on the screen in back-to-front order. If they are properly sorted, then any later triangle to be rendered will rightfully clobber the image of previously rendered triangles at the same pixels. The triangles can be drawn one-by-one while totally neglecting the problem of determining which is nearest. This is known as the *Painter's algorithm*. The main flaw, however, is the potential existence of *depth cycles*, shown in Figure 7.6, in which three or more triangles cannot be rendered correctly in any order by the Painter's algorithm. One possible fix is to detect such cases and split the triangles.

Depth buffer A simple and efficient method to resolve this problem is to manage the depth problem on a pixel-by-pixel basis by maintaining a *depth buffer* (also called *z-buffer*), which for every pixel records the distance of the triangle from the focal point to the intersection point of the ray that intersects the triangle at that pixel. In other words, if this were the ray casting approach, it would be distance along the ray from the focal point to the intersection point. Using this method, the triangles can be rendered in arbitrary order. The method is also commonly applied to compute the effect of shadows by determining depth order from a light source, rather than the viewpoint. Objects that are closer to the light cast a shadow on further objects.

The depth buffer stores a positive real number (floating point number in practice) at every pixel location. Before any triangles have been rendered, a maximum value (floating-point infinity) is stored at every location to reflect that no surface has yet been encountered at each pixel. At any time in the rendering process, each value in the depth buffer records the distance of the point on the most recently rendered triangle to the focal point, for the corresponding pixel in the image. Initially, all depths are at maximum to reflect that no triangles were rendered yet.

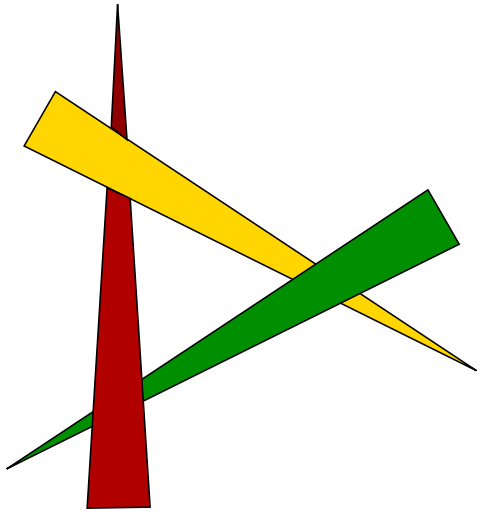


Figure 7.6: Due to the possibility of *depth cycles*, objects cannot be sorted in three dimensions with respect to distance from the observer. Each object is partially in front of one and partially behind another.

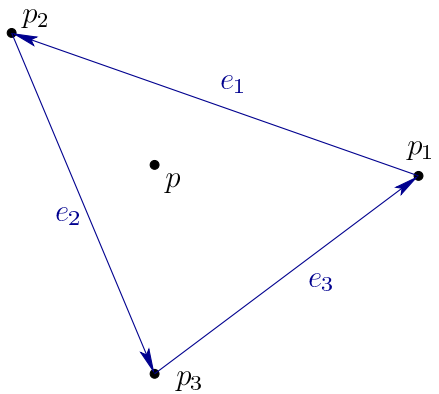


Figure 7.7: If p is inside of the triangle, then it must be to the right of each of the edge vectors, e_1, e_2 and e_3. Barycentric coordinates specify the location of every point p in a triangle as a weighted average of its vertices p_1, p_2, and p_3.

Each triangle is rendered by calculating a rectangular part of the image that fully contains it. This is called a *bounding box*. The box is quickly determined by transforming all three of the triangle vertices to determine the minimum and maximum values for i and j (the row and column indices). An iteration is then performed over all pixels inside of the bounding box to determine which ones lie in inside the triangle and should therefore be rendered. This can be quickly determined by forming the three edge vectors shown in Figure 7.7 as

$$\begin{aligned} e_1 &= p_2 - p_1 \\ e_2 &= p_3 - p_2 \\ e_3 &= p_1 - p_3. \end{aligned} \tag{7.8}$$

The point p lies inside of the triangle if and only if

$$(p - p_1) \times e_1 < 0 \ , \ (p - p_2) \times e_2 < 0 \ , \ (p - p_3) \times e_3 < 0, \tag{7.9}$$

in which $\times$ denotes the standard vector cross product. These three conditions ensure that p is "to the left" of each edge vector.

Barycentric coordinates As each triangle is rendered, information from it is mapped from the virtual world onto the screen. This is usually accomplished using *barycentric coordinates* (see Figure 7.7), which expresses each point in the triangle interior as a weighted average of the three vertices:

$$p = \alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3 \tag{7.10}$$

for which $0 \leq \alpha_1, \alpha_2, \alpha_3 \leq 1$ and $\alpha_1 + \alpha_2 + \alpha_3 = 1$. The closer p is to a vertex p_i, the larger the weight α_i. If p is at the centroid of the triangle, then $\alpha_1 = \alpha_2 = \alpha_3 = 1/3$. If p lies on an edge, then the opposing vertex weight is zero. For example, if p lies on the edge between p_1 and p_2, then $\alpha_3 = 0$. If p lies on a vertex, p_i, then $\alpha_i = 1$, and the other two barycentric coordinates are zero.

The coordinates are calculated using Cramer's rule to solve a resulting linear system of equations. In particular, let $d_{ij} = e_i \cdot e_j$ for all combinations of i and j. Furthermore, let

$$s = 1/(d_{11}d_{22} - d_{12}d_{12}). \tag{7.11}$$

The coordinates are then given by

$$\begin{aligned} \alpha_1 &= s(d_{22}d_{31} - d_{12}d_{32}) \\ \alpha_2 &= s(d_{11}d_{32} - d_{12}d_{31}) \\ \alpha_3 &= 1 - \alpha_1 - \alpha_2. \end{aligned} \tag{7.12}$$

The same barycentric coordinates may be applied to the points on the model in $\mathbb{R}^3$, or on the resulting 2D projected points (with i and j coordinates) in the image plane. In other words, α_1, α_2, and α_3 refer to the same point on the model both before, during, and after the entire chain of transformations from Section 3.5.

Furthermore, given the barycentric coordinates, the test in (7.9) can be replaced by simply determining whether $\alpha_1 \geq 0$, $\alpha_2 \geq 0$, and $\alpha_3 \geq 0$. If any barycentric coordinate is less than zero, then p must lie outside of the triangle.

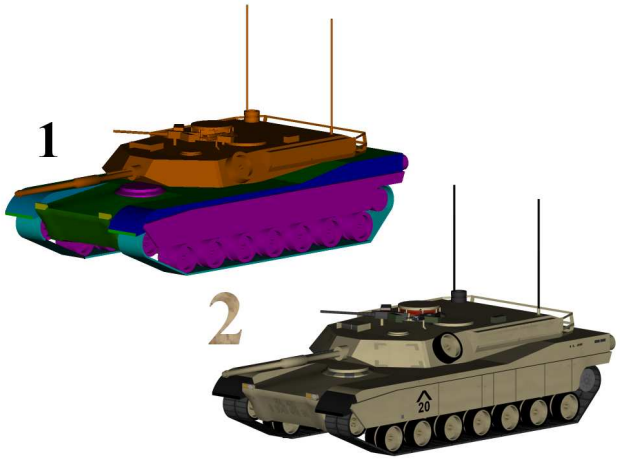


Figure 7.8: Texture mapping: A simple pattern or an entire image can be mapped across the triangles and then rendered in the image to provide much more detail than provided by the triangles in the model. (Figure from Wikipedia.)

Mapping the surface Barycentric coordinates provide a simple and efficient method for linearly interpolating values across a triangle. The simplest case is the propagation of RGB values. Suppose RGB values are calculated at the three triangle vertices using the shading methods of Section 7.1. This results in values (R_i, G_i, B_i) for each i from 1 to 3. For a point p in the triangle with barycentric coordinates $(\alpha_1, \alpha_2, \alpha_3)$, the RGB values for the interior points are calculated as

$$\begin{aligned} R &= \alpha_1 R_1 + \alpha_2 R_2 + \alpha_3 R_3 \\ G &= \alpha_1 G_1 + \alpha_2 G_2 + \alpha_3 G_3 \\ B &= \alpha_1 B_1 + \alpha_2 B_2 + \alpha_3 B_3. \end{aligned} \tag{7.13}$$

The object need not maintain the same properties over an entire triangular patch. With *texture mapping*, a repeating pattern, such as tiles or stripes can be propagated over the surface [6]; see Figure 7.8. More generally, any digital picture can be mapped onto the patch. The barycentric coordinates reference a point inside of the image to be used to influence a pixel. The picture or "texture" is treated as if it were painted onto the triangle; the lighting and reflectance properties are additionally taken into account for shading the object.

Another possibility is *normal mapping*, which alters the shading process by allowing the surface normal to be artificially varied over the triangle, even though geometrically it is impossible. Recall from Section 7.1 that the normal is used in the shading models. By allowing it to vary, simulated curvature can be given to an object. An important case of mapping the normals is called *bump mapping*, which makes a flat surface look rough by irregularly perturbing the normals. If

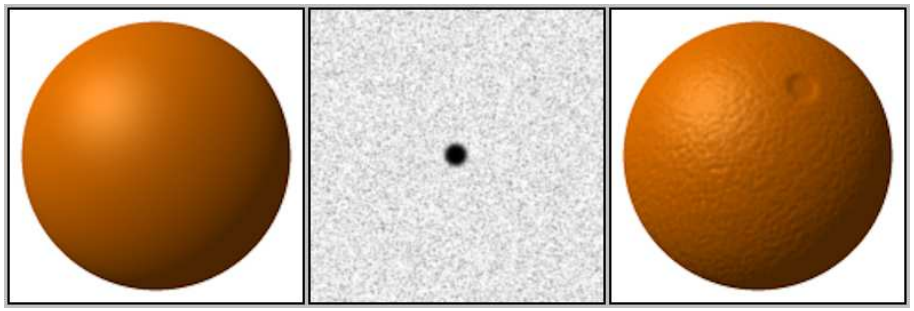


Figure 7.9: Bump mapping: By artificially altering the surface normals, the shading algorithms produce an effect that looks like a rough surface. (Figure by Brian Vibber.)

the normals appear to have texture, then the surface will look rough after shading is computed.

Aliasing Several artifacts arise due to discretization. Aliasing problems were mentioned in Section 5.4, which result in perceptible staircases in the place of straight lines, due to insufficient pixel density. Figure 7.10(a) shows the pixels selected inside of a small triangle by using (7.9). The point p usually corresponds to the center of the pixel, as shown in Figure 7.10(b). Note that the point may be inside of the triangle while the entire pixel is not. Likewise, part of the pixel might be inside of the triangle while the center is not. You may notice that Figure 7.10 is not entirely accurate due to the subpixel mosaics used in displays (recall Figure 4.36). To be more precise, aliasing analysis should take this into account as well.

By deciding to fully include or exclude the triangle based on the coordinates of p alone, the staircasing effect is unavoidable. A better way is to render the pixel according to the fraction of the pixel region that is covered by the triangle. This way its values could be blended from multiple triangles that are visible within the pixel region. Unfortunately, this requires *supersampling*, which means casting rays at a much higher density than the pixel density so that the triangle coverage fraction can be estimated. This dramatically increases cost. Commonly, a compromise is reached in a method called *multisample anti-aliasing* (or *MSAA*), in which only some values are calculated at the higher density. Typically, depth values are calculated for each sample, but shading is not.

A *spatial aliasing* problem results from texture mapping. The viewing transformation may dramatically reduce the size and aspect ratio of the original texture as it is mapped from the virtual world onto the screen. This may leave insufficient resolution to properly represent a repeating pattern in the texture; see Figure 7.12. This problem is often addressed in practice by pre-calculating and storing

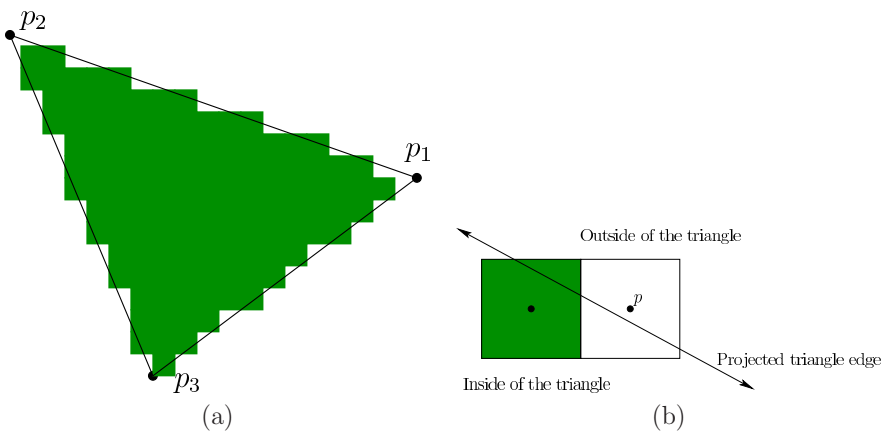


Figure 7.10: (a) The rasterization stage results in aliasing; straight edges appear to be staircases. (b) Pixels are selected for inclusion based on whether their center point p lies inside of the triangle.

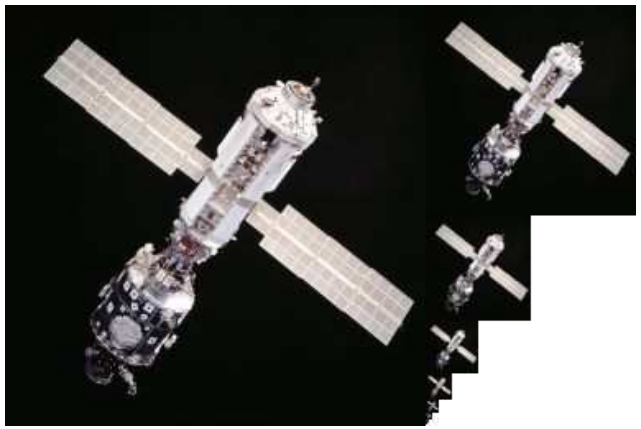


Figure 7.11: A mipmap stores the texture at multiple resolutions so that it can be appropriately scaled without causing signficant aliasing. The overhead for storing the extra image is typically only 1/3 the size of the original (largest) image. (The image is from NASA and the mipmap was created by Mike Hicks.)

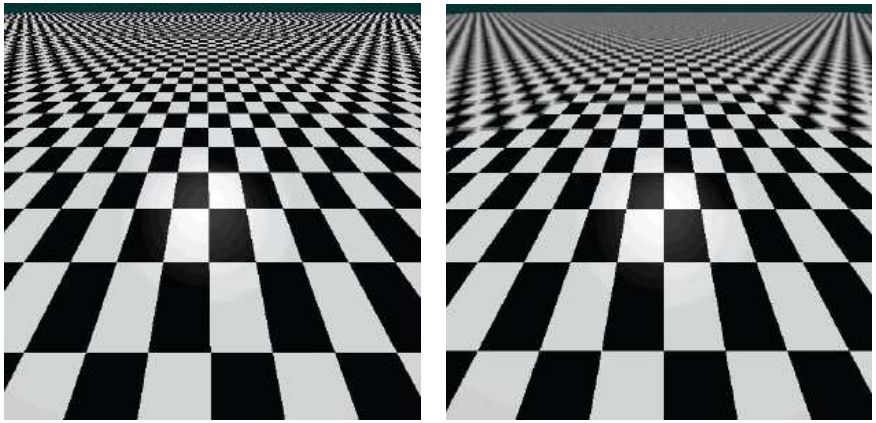


Figure 7.12: (a) Due to the perspective transformation, the tiled texture suffers from *spatial aliasing* as the depth increases. (b) The problem can be fixed by performing supersampling.

a *mipmap* for each texture; see Figure 7.11. The texture is calculated at various resolutions by performing high-density sampling and storing the rasterized result in images. Based on the size and viewpoint of the triangle on the screen, the appropriately scaled texture image is selected and mapped onto the triangle to reduce the aliasing artifacts.

Culling In practice, many triangles can be quickly eliminated before attempting to render them. This results in a preprocessing phase of the rendering approach called *culling*, which dramatically improves performance and enables faster frame rates. The efficiency of this operation depends heavily on the data structure used to represent the triangles. Thousands of triangles could be eliminated with a single comparison of coordinates if they are all arranged in a hierarchical structure. The most basic form of culling is called *view volume culling*, which eliminates all triangles that are wholly outside of the viewing frustum (recall Figure 3.18). For a VR headset, the frustum may have a curved cross section due to the limits of the optical system (see Figure 7.13). In this case, the frustum must be replaced with a region that has the appropriate shape. In the case of a truncated cone, a simple geometric test can quickly eliminate all objects outside of the view. For example, if

$$\frac{\sqrt{x^2 + y^2}}{-z} > \tan\theta, \tag{7.14}$$

in which 2θ is the angular field of view, then the point (x, y, z) is outside of the cone. Alternatively, the *stencil buffer* can be used in a GPU to mark all pixels that

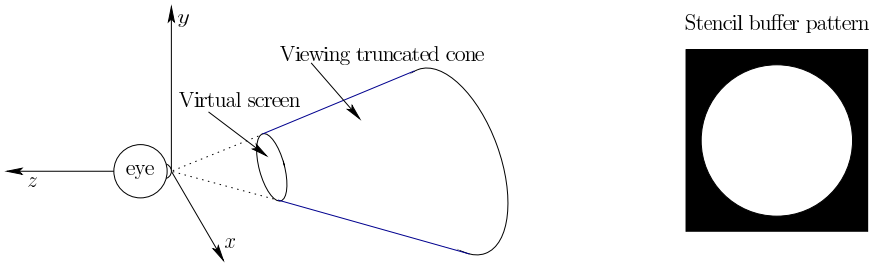


Figure 7.13: Due to the optical system in front of the screen, the viewing frustum is replaced by a truncated cone in the case of a circularly symmetric view. Other cross-sectional shapes may be possible to account for the asymmetry of each eye view (for example, the nose is obstructing part of the view).

would be outside of the lens view. These are quickly eliminated from consideration by a simple test as each frame is rendered.

Another form is called *backface culling*, which removes triangles that have outward surface normals that point away from the focal point. These should not be rendered "from behind" if the model is consistently formed. Additionally, *occlusion culling* may be used to eliminate parts of the model that might be hidden from view by a closer object. This can get complicated because it once again considers the depth ordering problem. For complete details, see [1].

VR-specific rasterization problems The staircasing problem due to aliasing is expected to be worse for VR because current resolutions are well below the required retina display limit calculated in Section 5.4. The problem is made significantly worse by the continuously changing viewpoint due to head motion. Even as the user attempts to stare at an edge, the "stairs" appear to be more like an "escalator" because the exact choice of pixels to include in a triangle depends on subtle variations in the viewpoint. As part of our normal perceptual processes, our eyes are drawn toward this distracting motion. With stereo viewpoints, the situation is worse: The "escalators" from the right and left images will usually not match. As the brain attempts to fuse the two images into one coherent view, the aliasing artifacts provide a strong, moving mismatch. Reducing contrast at edges and using anti-aliasing techniques help alleviate the problem, but aliasing is likely to remain a significant problem until displays reach the required retina display density for VR.

A more serious difficulty is caused by the enhanced depth perception afforded by a VR system. Both head motions and stereo views enable users to perceive small differences in depth across surfaces. This should be a positive outcome; however, many tricks developed in computer graphics over the decades rely on the fact that people cannot perceive these differences when a virtual world is rendered onto a fixed screen that is viewed from a significant distance. The result

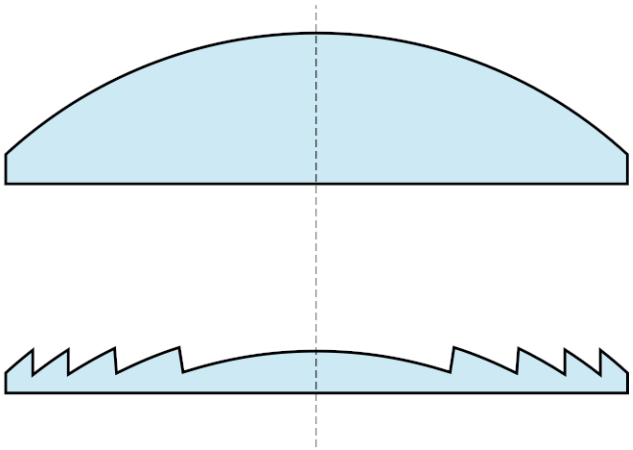


Figure 7.14: A Fresnel lens (pronounced like "frenelle") simulates a simple lens by making a corrugated surface. The convex surface on the top lens is implemented in the Fresnel lens shown on the bottom. (Figure by Piotr Kożurno.)

for VR is that texture maps may look fake. For example, texture mapping a picture of a carpet onto the floor might inadvertently cause the floor to look as if it were simply painted. In the real world we would certainly be able to distinguish painted carpet from real carpet. The same problem occurs with normal mapping. A surface that might look rough in a single static image due to bump mapping could look completely flat in VR as both eyes converge onto the surface. Thus, as the quality of VR systems improves, we should expect the rendering quality requirements to increase, causing many old tricks to be modified or abandoned.

7.3 Correcting Optical Distortions

Recall from Section 4.3 that barrel and pincushion distortions are common for an optical system with a high field of view (Figure 4.20). When looking through the lens of a VR headset, a pincushion distortion typically results. If the images are drawn on the screen without any correction, then the virtual world appears to be incorrectly warped. If the user yaws his head back and forth, then fixed lines in the world, such as walls, appear to dynamically change their curvature because the distortion in the periphery is much stronger than in the center. If it is not corrected, then the perception of stationarity will fail because static objects should not appear to be warping dynamically. Furthermore, contributions may be made to VR sickness because incorrect accelerations are being visually perceived near the periphery.

How can this problem be solved? Significant research is being done in this

area, and the possible solutions involve different optical systems and display technologies. For example, *digital light processing* (*DLP*) technology directly projects light into the eye without using lenses. Another way to greatly reduce this problem is to use a *Fresnel lens* (see Figure 7.14), which more accurately controls the bending of light rays by using a corrugated or sawtooth surface over a larger area; an aspheric design can be implemented as well. A Fresnel lens is used, for example, in the HTC Vive VR headset. One unfortunate side effect of Fresnel lenses is that glaring can be frequently observed as light scatters across the ridges along the surface.

Whether small or large, the distortion can also be corrected in software. One assumption is that the distortion is circularly symmetric. This means that the amount of distortion depends only on the distance from the lens center, and not the particular direction from the center. Even if the lens distortion is perfectly circularly symmetric, it must also be placed so that it is centered over the eye. Some headsets offer IPD adjustment, which allows the distance between the lenses to be adjusted so that they are matched to the user's eyes. If the eye is not centered on the lens, then asymmetric distortion arises. The situation is not perfect because as the eye rotates, the pupil moves along a spherical arc. As the position of the pupil over the lens changes laterally, the distortion varies and becomes asymmetric. This motivates making the lens as large as possible so that this problem is reduced. Another factor is that the distortion will change as the distance between the lens and the screen is altered. This adjustment may be useful to accommodate users with nearsightedness or farsightedness, as done in the Samsung Gear VR headset. The adjustment is also common in binoculars and binoculars, which explains why many people do not need their glasses to use them. To handle distortion correctly, the headset should ideally sense the adjustment setting and take it into account.

To fix radially symmetric distortion, suppose that the transformation chain $T_{can}T_{eye}T_{rb}$ has been applied to the geometry, resulting in the canonical view volume, as covered in Section 3.5. All points that were inside of the viewing frustum now have x and y coordinates ranging from -1 to 1. Consider referring to these points using polar coordinates (r, θ):

$$\begin{aligned} r &= \sqrt{x^2 + y^2} \\ \theta &= \text{atan2}(y, x), \end{aligned} \tag{7.15}$$

in which atan2 represents the inverse tangent of y/x. This function is commonly used in programming languages to return an angle θ over the entire range from 0 to 2π. (The arctangent alone cannot do this because the quadrant that (x, y) came from is needed.)

We now express the lens distortion in terms of transforming the radius r, without affecting the direction θ (because of symmetry). Let f denote a function that applies to positive real numbers and distorts the radius. Let r_u denote the undistorted radius, and let r_d denote the distorted radius. Both pincushion and barrel distortion are commonly approximated using polynomials with odd powers,

resulting in f being defined as

$$r_d = f(r_u) = r_u + c_1 r_u^3 + c_2 r_u^5, \tag{7.16}$$

in which c_1 and c_2 are suitably chosen constants. If $c_1 < 0$, then barrel distortion occurs. If $c_1 > 0$, then pincushion distortion results. Higher-order polynomials could also be used, such as adding a term $c_3 r_u^7$ on the right above; however, in practice this is often considered unnecessary.

Correcting the distortion involves two phases:

1. Determine the radial distortion function f for a particular headset, which involves a particular lens placed at a fixed distance from the screen. This is a regression or curve-fitting problem that involves an experimental setup that measures the distortion of many points and selects the coefficients c_1, c_2, and so on, that provide the best fit.

2. Determine the inverse of f so that it be applied to the rendered image before the lens causes its distortion. The composition of the inverse with f should cancel out the distortion function.

Unfortunately, polynomial functions generally do not have inverses that can be determined or even expressed in a closed form. Therefore, approximations are used. One commonly used approximation is [13]:

$$f^{-1}(r_d) \approx \frac{c_1 r_d^2 + c_2 r_d^4 + c_1^2 r_d^4 + c_2^2 r_d^8 + 2 c_1 c_2 r_d^6}{1 + 4 c_1 r_d^2 + 6 c_2 r_d^4}. \tag{7.17}$$

Alternatively, the inverse can be calculated very accurately off-line and then stored in an array for fast access. It needs to be done only once per headset design. Linear interpolation can be used for improved accuracy. The inverse values can be accurately calculated using Newton's method, with initial guesses provided by simply plotting $f(r_u)$ against r_u and swapping the axes.

The transformation f^{-1} could be worked directly into the perspective transformation, thereby replacing T_p and T_{can} with a nonlinear operation. By leveraging the existing graphics rendering pipeline, it is instead handled as a post-processing step. The process of transforming the image is sometimes called *distortion shading* because it can be implemented as a shading operation in the GPU; it has nothing to do with "shading" as defined in Section 7.1. The rasterized image that was calculated using methods in Section 7.2 can be converted into a transformed image using (7.17), or another representation of f^{-1}, on a pixel-by-pixel basis. If compensating for a pincushion distortion, the resulting image will appear to have a barrel distortion; see Figure 7.15. To improve VR performance, *multiresolution shading* is used in Nvidia GTX 1080 GPUs. One problem is that the resolution is effectively dropped near the periphery because of the transformed image (Figure 7.15). This results in wasted shading calculations in the original image. Instead, the image can be rendered before the transformation by taking into account the

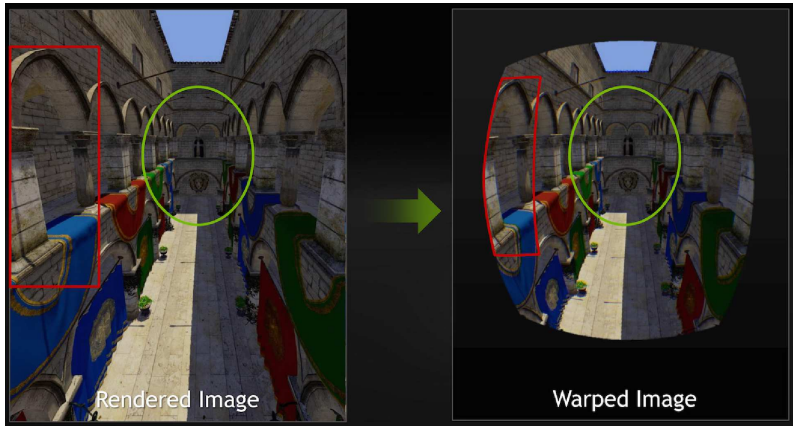


Figure 7.15: The rendered image appears to have a barrel distortion. Note that the resolution is effectively dropped near the periphery. (Figure by Nvidia.)

final resulting resolutions after the transformation. A lower-resolution image is rendered in a region that will become compressed by the transformation.

The methods described in this section may also be used for other optical distortions that are radially symmetric. For example, chromatic aberration can be partially corrected by transforming the red, green, and blue subpixels differently. Each color is displaced radially by a different amount to compensate for the radial distortion that occurs based on its wavelength. If chromatic aberration correction is being used, then if the lenses are removed from the VR headset, it would become clear that the colors are not perfectly aligned in the images being rendered to the display. The rendering system must create a distortion of pixel placements on the basis of color so that they will be moved closer to the correct places after they pass through the lens.

7.4 Improving Latency and Frame Rates

The *motion-to-photons* latency in a VR headset is the amount of time it takes to update the display in response to a change in head orientation and position. For example, suppose the user is fixating on a stationary feature in the virtual world. As the head yaws to the right, the image of the feature on the display must immediately shift to the left. Otherwise, the feature will appear to move if the eyes remain fixated on it. This breaks the perception of stationarity.

A simple example Consider the following example to get a feeling for the latency problem. Let d be the density of the display in pixels per degree. Let ω be the angular velocity of the head in degrees per second. Let ℓ be the latency

in seconds. Due to latency ℓ and angular velocity ω, the image is shifted by $d\omega\ell$ pixels. For example, if $d = 40$ pixels per degree, $\omega = 50$ degrees per second, and $\ell = 0.02$ seconds, then the image is incorrectly displaced by $d\omega\ell = 4$ pixels. An extremely fast head turn might be at 300 degrees per second, which would result in a 24-pixel error.

The perfect system As a thought experiment, imagine the perfect VR system. As the head moves, the viewpoint must accordingly change for visual rendering. A magic oracle perfectly indicates the head position and orientation at any time. The VWG continuously maintains the positions and orientations of all objects in the virtual world. The visual rendering system maintains all perspective and viewport transformations, and the entire rasterization process continuously sets the RGB values on the display according to the shading models. Progressing with this fantasy, the display itself continuously updates, taking no time to switch the pixels. The display has retina-level resolution, as described in Section 5.4, and a dynamic range of light output over seven orders of magnitude to match human perception. In this case, visual stimulation provided by the virtual world should match what would occur in a similar physical world in terms of the geometry. There would be no errors in time and space (although the physics might not match anyway due to assumptions about lighting, shading, material properties, color spaces, and so on).

Historical problems In practice, the perfect system is not realizable. All of these operations require time to propagate information and perform computations. In early VR systems, the total motion-to-photons latency was often over 100ms. In the 1990s, 60ms was considered an acceptable amount. Latency has been stated as one of the greatest causes of VR sickness, and therefore one of the main obstructions to widespread adoption over the past decades. People generally adapt to a fixed latency, which somewhat mitigates the problem, but then causes problems when they have to readjust to the real world. Variable latencies are even worse due to the inability to adapt [10]. Fortunately, latency is no longer the main problem in most VR systems because of the latest-generation tracking, GPU, and display technology. The latency may be around 15 to 25ms, which is even compensated for by predictive methods in the tracking system. The result is that the *effective* latency is very close to zero. Thus, other factors are now contributing more strongly to VR sickness and fatigue, such as vection and optical aberrations.

Overview of latency reduction methods The following strategies are used together to both reduce the latency and to minimize the side effects of any remaining latency:

1. Lower the complexity of the virtual world.

2. Improve rendering pipeline performance.

Figure 7.16: A variety of mesh simplification algorithms can be used to reduce the model complexity while retaining the most important structures. Shown here is a simplification of a hand model made by the open-source library CGAL. (Figure by Fernando Cacciola.)

3. Remove delays along the path from the rendered image to switching pixels.

4. Use prediction to estimate future viewpoints and world states.

5. Shift or distort the rendered image to compensate for last-moment viewpoint errors and missing frames.

Each of these will be described in succession.

Simplifying the virtual world Recall from Section 3.1 that the virtual world is composed of geometric primitives, which are usually 3D triangles arranged in a mesh. The chain of transformations and rasterization process must be applied for each triangle, resulting in a computational cost that is directly proportional to the number of triangles. Thus, a model that contains tens of millions of triangles will take orders of magnitude longer to render than one made of a few thousand. In many cases, we obtain models that are much larger than necessary. They can often be made much smaller (fewer triangles) with no perceptible difference, much in the same way that image, video, and audio compression works. Why are they too big in the first place? If the model was captured from a 3D scan of the real world, then it is likely to contain highly dense data. Capture systems such as the FARO Focus3D X Series capture large worlds while facing outside. Others, such as the

Matter and Form MFSV1, capture a small object by rotating it on a turntable. As with cameras, systems that construct 3D models automatically are focused on producing highly accurate and dense representations, which maximize the model size. Even in the case of purely synthetic worlds, a modeling tool such as Maya or Blender will automatically construct a highly accurate mesh of triangles over a curved surface. Without taking specific care of later rendering burdens, the model could quickly become unwieldy. Fortunately, it is possible to reduce the model size by using *mesh simplification* algorithms; see Figure 7.16. In this case, one must be careful to make sure that the simplified model will have sufficient quality from all viewpoints that might arise in the targeted VR system. In some systems, such as Unity 3D, reducing the number of different material properties across the model will also improve performance.

In addition to reducing the rendering time, a simplified model will also lower computational demands on the Virtual World Generator (VWG). For a *static world*, the VWG does not need to perform any updates after initialization. The user simply views the fixed world. For *dynamic worlds*, the VWG maintains a simulation of the virtual world that moves all geometric bodies while satisfying physical laws that mimic the real world. It must handle the motions of any avatars, falling objects, moving vehicles, swaying trees, and so on. Collision detection methods are needed to make bodies react appropriately when in contact. Differential equations that model motion laws may be integrated to place bodies correctly over time. These issues will be explained in Chapter 8, but for now it is sufficient to understand that the VWG must maintain a coherent snapshot of the virtual world each time a rendering request is made. Thus, the VWG has a frame rate in the same way as a display or visual rendering system. Each VWG frame corresponds to the placement of all geometric bodies for a common time instant. How many times per second can the VWG be updated? Can a high, constant rate of VWG frames be maintained? What happens when a rendering request is made while the VWG is in the middle of updating the world? If the rendering module does not wait for the VWG update to be completed, then some objects could be incorrectly placed because some are updated while others are not. Thus, the system should ideally wait until a complete VWG frame is finished before rendering. This suggests that the VWG update should be at least as fast as the rendering process, and the two should be carefully synchronized so that a complete, fresh VWG frame is always ready for rendering.

Improving rendering performance Any techniques that improve rendering performance in the broad field of computer graphics apply here; however, one must avoid cases in which side effects that were imperceptible on a computer display become noticeable in VR. It was already mentioned in Section 7.2 that texture and normal mapping methods are less effective in VR for this reason; many more discrepancies are likely to be revealed in coming years. Regarding improvements that are unique to VR, it was mentioned in Sections 7.2 and 7.3 that the stencil

Figure 7.17: If a new frame is written to the video memory while a display scanout occurs, then *tearing* arises, in which parts of two or more frames become visible at the same time. (Figure from http://www.overclock.net/ user Forceman.)

buffer and multiresolution shading can be used to improve rendering performance by exploiting the shape and distortion due to the lens in a VR headset. A further improvement is to perform rasterization for the left and right eyes in parallel in the GPU, using one processor for each. The two processes are completely independent. This represents an important first step, among many that are likely to come, in design of GPUs that are targeted specifically for VR.

From rendered image to switching pixels The problem of waiting for coherent VWG frames also arises in the process of rendering frames to the display: When it is time to scan out the rendered image to the display, it might not be finished yet. Recall from Section 5.4 that most displays have a rolling scanout that draws the rows of the rasterized image, which sits in the *video memory*, onto the screen one-by-one. This was motivated by the motion of the electron beam that lit phosphors on analog TV screens. The motion is left to right, and top to bottom, much in the same way we would write out a page of English text with a pencil and paper. Due to inductive inertia in the magnetic coils that bent the beam, there was a period of several milliseconds called VBLANK (*vertical blanking interval*) in which the beam moves from the lower right back to the upper left of the screen to start the next frame. During this time, the beam was turned off to avoid drawing a diagonal streak across the frame, hence, the name "blanking". Short blanking intervals also occurred as each horizontal line to bring the beam back from the right to the left.

In the era of digital displays, the scanning process in unnecessary, but it nevertheless persists and causes some trouble. Suppose that a display runs at 100

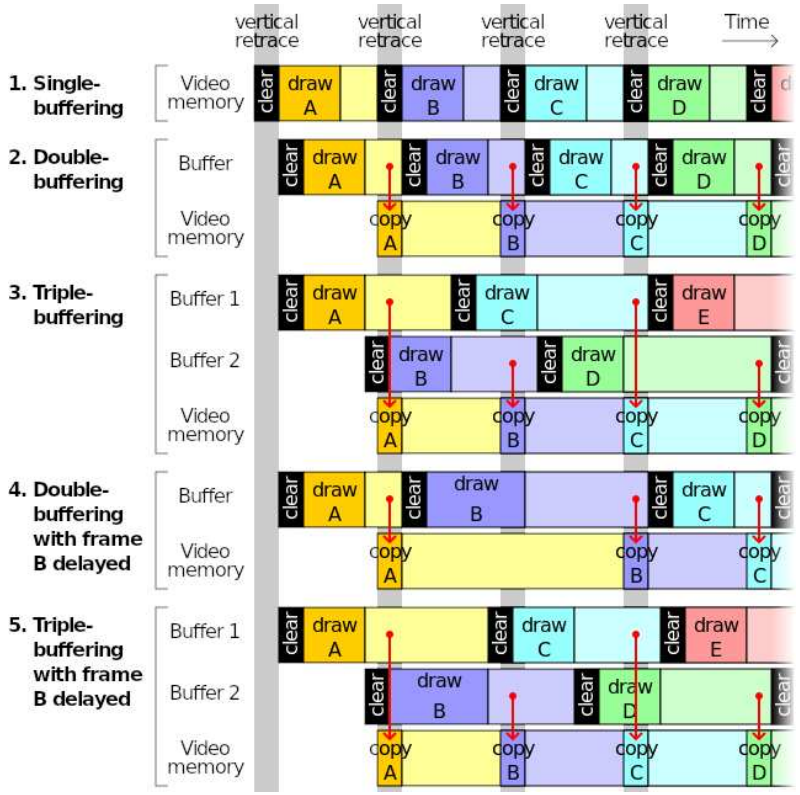


Figure 7.18: Buffering is commonly used in visual rendering pipelines to avoid tearing and lost frames; however, it introduces more latency, which is detrimental to VR. (Figure by Wikipedia user Cmglee.)

FPS. In this case, a request to draw a new rendered image is made every 10ms. Suppose that VBLANK occurs for 2ms and the remaining 8ms is spent drawing lines on the display. If the new rasterized image is written to the video memory during the 2ms of VBLANK, then it will be correctly drawn in the remaining 8ms. It is also possible to earn extra time through beam racing [4, 18]. However, if a new image is being written and passes where the beam is scanning it out, then tearing occurs because it appears as if is screen is torn into pieces; see Figure 7.17. If the VWG and rendering system produce frames at 300 FPS, then parts of 3 or 4 images could appear on the display because the image changes several times while the lines are being scanned out. One solution to this problem to use VSYNC (pronounced "vee sink"), which is a flag that prevents the video memory from being written outside of the VBLANK interval.

Another strategy to avoid tearing is *buffering*, which is shown in Figure 7.18.

The approach is simple for programmers because it allows the frames to be written in memory that is not being scanned for output to the display. The unfortunate side effect is that it increases the latency. For *double buffering*, a new frame is first drawn into the buffer and then transferred to the video memory during VBLANK. It is often difficult to control the rate at which frames are produced because the operating system may temporarily interrupt the process or alter its priority. In this case, *triple buffering* is an improvement that allows more time to render each frame. For avoiding tearing and providing smooth video game performance, buffering has been useful; however, it is detrimental to VR because of the increased latency.

Ideally, the displays should have a global scanout, in which all pixels are switched at the same time. This allows a much longer interval to write to the video memory and avoids tearing. It would also reduce the latency in the time it takes to scan the first pixel to the last pixel. In our example, this was an 8ms interval. Finally, displays should reduce the pixel switching time as much as possible. In a smartphone LCD screen, it could take up to 20ms to switch pixels; however, OLED pixels can be switched in under 0.1ms.

The power of prediction For the rest of this section, we consider how to live with whatever latency remains. As another thought experiment, imagine that a fortune teller is able to accurately predict the future. With such a device, it should be possible to eliminate all latency problems. We would want to ask the fortune teller the following:

1. At what future time will the pixels be switching?
2. What will be the positions and orientations of all virtual world models at that time?
3. Where will the user be looking at that time?

Let t_s be answer to the first question. We need to ask the VWG to produce a frame for time t_s and then perform visual rendering for the user's viewpoint at time t_s. When the pixels are switched at time t_s, then the stimulus will be presented to the user at the exact time and place it is expected. In this case, there is *zero effective latency*.

Now consider what happens in practice. First note that using information from all three questions above implies significant time synchronization across the VR system: All operations must have access to a common clock. For the first question above, determining t_s should be feasible if the computer is powerful enough and the VR system has enough control from the operating system to ensure that VWG frames will be consistently produced and rendered at the frame rate. The second question is easy for the case of a static virtual world. In the case of a dynamic world, it might be straightforward for all bodies that move according to predictable physical laws. However, it is difficult to predict what humans will

Perturbation	Image effect
$\Delta\alpha$ (yaw)	Horizontal shift
$\Delta\beta$ (pitch)	Vertical shift
$\Delta\gamma$ (roll)	Rotation about image center
Δx	Horizontal shift
Δy	Vertical shift
Δz	Contraction or expansion

Figure 7.19: Six cases of post-rendering image warp based on the degrees of freedom for a change in viewpoint. The first three correspond to an orientation change. The remaining three correspond to a position change. These operations can be visualized by turning on a digital camera and observing how the image changes under each of these perturbations.

do in the virtual world. This complicates the answers to both the second and third questions. Fortunately, the latency is so small that *momentum* and *inertia* play a significant role; see Chapter 8. Bodies in the matched zone are following physical laws of motion from the real world. These motions are sensed and tracked according to methods covered in Chapter 9. Although it might be hard to predict where you will be looking in 5 seconds, it is possible to predict with very high accuracy where your head will be positioned and oriented in 20ms. You have no free will on the scale of 20ms! Instead, momentum dominates and the head motion can be accurately predicted. Some body parts, especially fingers, have much less inertia, and therefore become more difficult to predict; however, these are not as important as predicting head motion. The viewpoint depends only on the head motion, and latency reduction is most critical in this case to avoid perceptual problems that lead to fatigue and VR sickness.

Post-rendering image warp Due to both latency and imperfections in the prediction process, a last-moment adjustment might be needed before the frame is scanned out to the display. This is called *post-rendering image warp* [16] (it has also been rediscovered and called *time warp* and *asynchronous reprojection* in the recent VR industry). At this stage, there is no time to perform complicated shading operations; therefore, a simple transformation is made to the image.

Suppose that an image has been rasterized for a particular viewpoint, expressed by position (x, y, z) and orientation given by yaw, pitch, and roll (α, β, γ). What would be different about the image if it were rasterized for a nearby viewpoint? Based on the degrees of freedom for viewpoints, there are six types of adjustments; see Figure 7.19. Each one of these has a direction that is not specified in the figure. For example, if $\Delta\alpha$ is positive, which corresponds to a small, counterclockwise yaw of the viewpoint, then the image is shifted horizontally *to the right*.

Figure 7.20 shows some examples of the image warp. Most cases require the

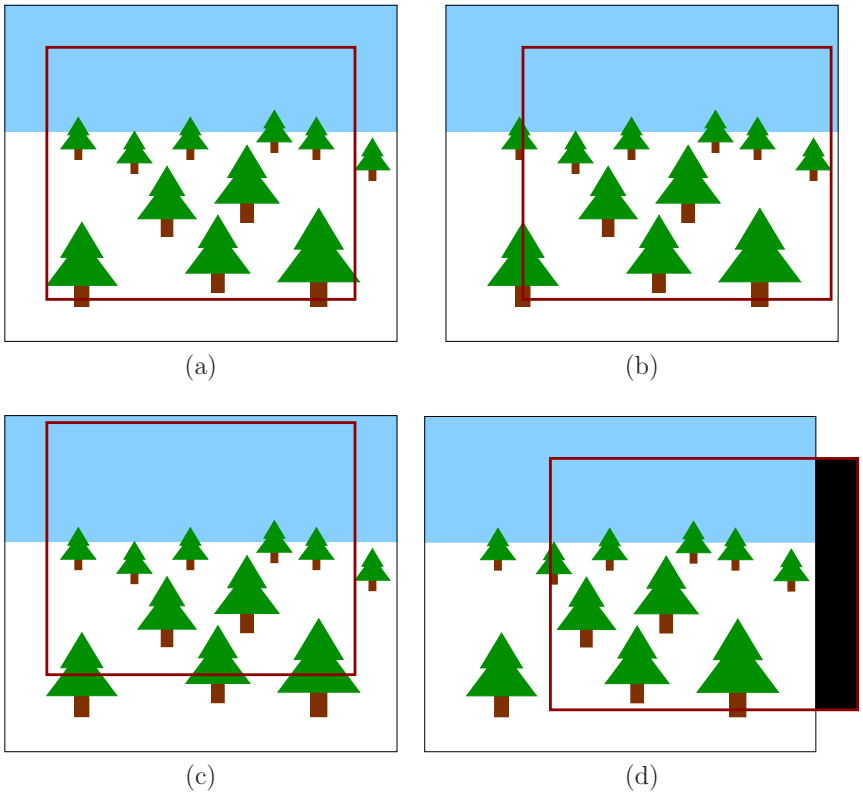


Figure 7.20: Several examples of post-rendering image warp: (a) Before warping, a larger image is rasterized. The red box shows the part that is intended to be sent to the display based on the viewpoint that was used at the time of rasterization; (b) A negative yaw (turning the head to the right) causes the red box to shift to the right. The image appears to shift to the left; (c) A positive pitch (looking upward) causes the box to shift upward; (d) In this case, the yaw is too large and there is no rasterized data to use for part of the image (this region is shown as a black rectangle).

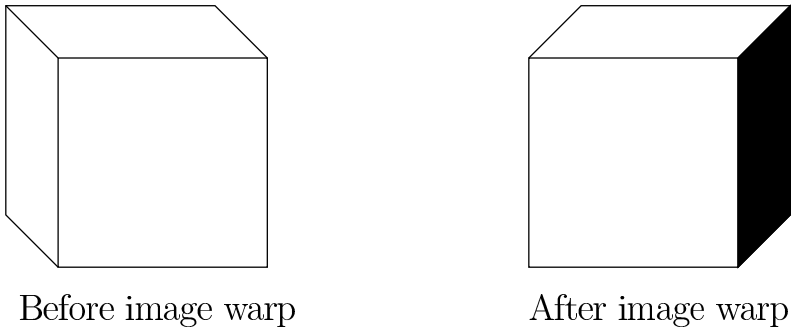


Figure 7.21: If the viewing position changes, then a *visibility event* might be encountered. This means that part of the object might suddenly become visible from the new perspective. In this sample, a horizontal shift in the viewpoint reveals a side of the cube that was originally hidden. Furthermore, the top of the cube changes its shape.

rendered image to be larger than the targeted display; otherwise, there will be no data to shift into the warped image; see Figure 7.20(d). If this ever happens, then it is perhaps best to repeat pixels from the rendered image edge, rather than turning them black [16].

Flaws in the warped image Image warping due to orientation changes produces a correct image in the sense that it should be exactly what would have been rendered from scratch for that orientation (without taking aliasing issues into account). However, positional changes are incorrect! Perturbations in x and y do not account for motion parallax (recall from Section 6.1), which would require knowing the depths of the objects. Changes in z produce similarly incorrect images because nearby objects should expand or contract by a larger amount than further ones. To make matters worse, changes in viewpoint position might lead to a *visibility event*, in which part of an object may become visible only in the new viewpoint; see Figure 7.21. Data structures such as an *aspect graph* [22] and *visibility complex* [23] are designed to maintain such events, but are usually not included in the rendering process. As latencies become shorter and prediction becomes better, the amount of perturbation is reduced. Careful perceptual studies are needed to evaluate conditions under which image warping errors are perceptible or cause discomfort. An alternative to image warping is to use parallel processing to sample several future viewpoints and render images for all of them. The most correct image can then be selected, to greatly reduce the image warping artifacts.

Increasing the frame rate Post-rendering image warp can also be used to artificially increase the frame rate. For example, suppose that only one rasterized

(a) (b)

Figure 7.22: (a) As of 2015, Netflix offers online movie streaming onto a large virtual TV screen while the user appears to sit in a living room. (b) The movies are texture-mapped onto the TV screen, frame by frame. Furthermore, the gaze pointer allows the user to look in a particular direction to select content.

image is produced every 100 milliseconds by a weak computer or GPU. This would result in poor performance at 10 FPS. Suppose we would like to increase this to 100 FPS. In this case, a single rasterized image can be warped to produce frames every 10ms until the next rasterized image is computed. In this case, 9 warped frames are inserted for every rasterized image that is properly rendered. This process is called *inbetweening* or *tweening*, and has been used for over a century (one of the earliest examples is the making of Fantasmagorie, which was depicted in Figure 1.26(a)).

7.5 Immersive Photos and Videos

Up until now, this chapter has focused on rendering a virtual world that was constructed synthetically from geometric models. The methods developed over decades of computer graphics research have targeted this case. The trend has recently changed, though, toward capturing real-world images and video, which are then easily embedded into VR experiences. This change is mostly due to the smartphone industry, which has led to hundreds of millions of people carrying high resolution cameras with them everywhere. Furthermore, 3D camera technology continues to advance, which provides distance information in addition to color and light intensity. All of this technology is quickly converging to the case of *panoramas*, which contained captured image data from all possible viewing directions. A current challenge is to also capture data within a region of all possible viewing *positions and orientations*.

Texture mapping onto a virtual screen Putting a photo or video into a virtual world is an extension of texture mapping. Figure 7.22 shows a commercial use in which Netflix offers online movie streaming through the Samsung Gear VR headset. The virtual screen is a single rectangle, which may be viewed as a simple mesh consisting of two triangles. A photo can be mapped across any triangular mesh in the virtual world. In the case of a movie, each frame is treated as a photo that is texture-mapped to the mesh. The movie frame rate is usually much lower than that of the VR headset (recall Figure 6.17). As an example, suppose the movie was recorded at 24 FPS and the headset runs at 96 FPS. In this case, each movie frame is rendered for four frames on the headset display. Most often, the frame rates are not perfectly divisible, which causes the number of repeated frames to alternate in a pattern. An old example of this is called *3:2 pull down*, in which 24 FPS movies were converted to NTSC TV format at 30 FPS. Interestingly, a 3D movie (stereoscopic) experience can even be simulated. For the left eye on the headset display, the left-eye movie frame is rendered to the virtual screen. Likewise, the right-eye movie frame is rendered to the right-eyed portion of the headset display. The result is that the user perceives it as a 3D movie, without wearing the special glasses! Of course, she would be wearing a VR headset.

Capturing a wider field of view Mapping onto a rectangle makes it easy to bring pictures or movies that were captured with ordinary cameras into VR; however, the VR medium itself allows great opportunities to expand the experience. Unlike life in the real world, the size of the virtual screen can be expanded without any significant cost. To fill the field of view of the user, it makes sense to curve the virtual screen and put the user at the center. Such curving already exists in the real world; examples are the 1950s Cinerama experience, which was shown in Figure 1.29(d), and modern curved displays. In the limiting case, we obtain a panoramic photo, sometimes called a *photosphere*. Displaying many photospheres per second leads to a panoramic movie, which we may call a *moviesphere*.

Recalling the way cameras work from Section 4.5, it is impossible to capture a photosphere from a single camera in a single instant of time. Two obvious choices exist:

1. Take multiple images with one camera by pointing it in different directions each time, until the entire sphere of all viewing directions is covered.

2. Use multiple cameras, pointing in various viewing directions, so that all directions are covered by taking synchronized pictures.

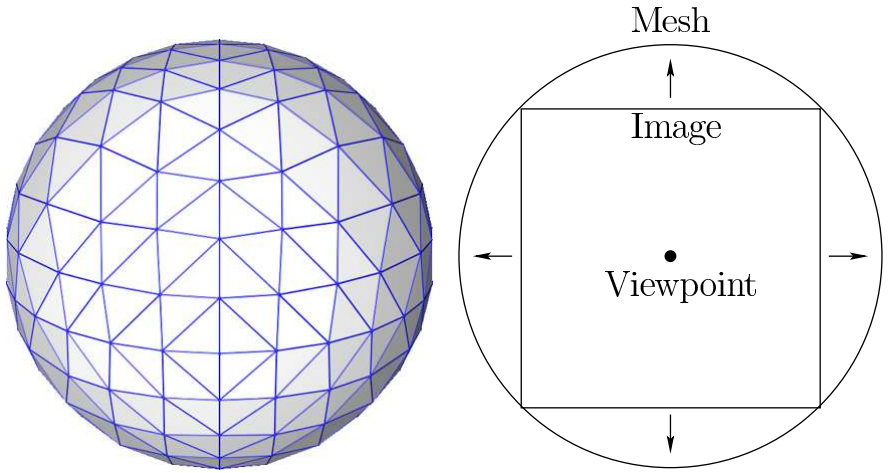
The first case leads to a well-studied problem in computer vision and computational photography called *image stitching*. A hard version of the problem can be made by stitching together an arbitrary collection of images, from various cameras and times. This might be appropriate, for example, to build a photosphere of a popular tourist site from online photo collections. More commonly, a smartphone

(a) (b)

Figure 7.23: (a) The 360Heros Pro10 HD is a rig that mounts ten GoPro cameras in opposing directions to capture panoramic images. (b) The Ricoh Theta S captures panoramic photos and videos using only two cameras, each with a lens that provides a field of view larger than 180 degrees.

user may capture a photosphere by pointing the outward-facing camera in enough directions. In this case, a software app builds the photosphere dynamically while images are taken in rapid succession. For the hard version, a difficult optimization problem arises in which features need to be identified and matched across overlapping parts of multiple images while unknown, intrinsic camera parameters are taken into account. Differences in perspective, optical aberrations, lighting conditions, exposure time, and changes in the scene over different times must be taken into account. In the case of using a smartphone app, the same camera is being used and the relative time between images is short; therefore, the task is much easier. Furthermore, by taking rapid images in succession and using internal smartphone sensors, it is much easier to match the overlapping image parts. Most flaws in such hand-generated photospheres are due to the user inadvertently changing the position of the camera while pointing it in various directions.

For the second case, a rig of identical cameras can be carefully designed so that all viewing directions are covered; see Figure 7.23(a). Once the rig is calibrated so that the relative positions and orientations of the cameras are precisely known, stitching the images together becomes straightforward. Corrections may nevertheless be applied to account for variations in lighting or calibration; otherwise, the seams in the stitching may become perceptible. A tradeoff exists in terms of the number of cameras. By using many cameras, very high resolution captures can be made with relatively little optical distortion because each camera contributes a narrow field-of-view image to the photosphere. At the other extreme, as few as two cameras are sufficient, as in the case of the Ricoh Theta S (Figure 7.23(b)).



(a) (b)

Figure 7.24: (a) The photophere is texture-mapped onto the interior of a sphere that is modeled as a triangular mesh. (b) A photosphere stored as a cube of six images can be quickly mapped to the sphere with relatively small loss of resolution; a cross section is shown here.

The cameras are pointed 180 degrees apart and a fish-eyed lens is able to capture a view that is larger than 180 degrees. This design dramatically reduces costs, but requires significant unwarping of the two captured images.

Mapping onto a sphere The well-known *map projection* problem from cartography would be confronted to map the photosphere onto a screen; however, this does not arise when rendering a photosphere in VR because it is mapped directly onto a sphere in the virtual world. The sphere of all possible viewing directions maps to the virtual-world sphere without distortions. To directly use texture mapping techniques, the virtual-world sphere can be approximated by uniform triangles, as shown in Figure 7.24(a). The photosphere itself should be stored in a way that does not degrade its resolution in some places. We cannot simply use latitude and longitude coordinates to index the pixels because the difference in resolution between the poles and the equator would be too large. We could use coordinates that are similar to the way quaternions cover the sphere by using indices (a, b, c) and requiring that $a^2 + b^2 + c^2 = 1$; however, the structure of neighboring pixels (up, down, left, and right) is not clear. A simple and efficient compromise is to represent the photosphere as six square images, each corresponding to the face of a cube. This is like a virtual version of a six-sided CAVE projection system. Each image can then be easily mapped onto the mesh with little loss in resolution, as shown in Figure 7.24(b).

Once the photosphere (or moviesphere) is rendered onto the virtual sphere, the rendering process is very similar to post-rendering image warp. The image presented to the user is shifted for the rotational cases that were described in Figure 7.19. In fact, the entire rasterization process could be performed only once, for the entire sphere, while the image rendered to the display is adjusted based on the viewing direction. Further optimizations could be made by even bypassing the mesh and directly forming the rasterized image from the captured images.

Perceptual issues Does the virtual world appear to be "3D" when viewing a photosphere or moviesphere? Recall from Section 6.1 that there are many more monocular depth cues than stereo cues. Due to the high field-of-view of modern VR headsets and monocular depth cues, a surprisingly immersive experience is obtained. Thus, it may feel more "3D" than people expect, even if the same part of the panoramic image is presented to both eyes. Many interesting questions remain for future research regarding the perception of panoramas. If different viewpoints are presented to the left and right eyes, then what should the radius of the virtual sphere be for comfortable and realistic viewing? Continuing further, suppose positional head tracking is used. This might improve viewing comfort, but the virtual world will appear more flat because parallax is not functioning. For example, closer objects will not move more quickly as the head moves from side to side. Can simple transformations be performed to the images so that depth perception is enhanced? Can limited depth data, which could even be extracted automatically from the images, greatly improve parallax and depth perception? Another issue is designing interfaces inside of photospheres. Suppose we would like to have a shared experience with other users inside of the sphere. In this case, how do we perceive virtual objects inserted into the sphere, such as menus or avatars? How well would a virtual laser pointer work to select objects?

Panoramic light fields Panoramic images are simple to construct, but are clearly flawed because they do not account how the surround world would appear from any viewpoint that could be obtained by user movement. To accurately determine this, the ideal situation would be to capture the entire *light field* of energy inside of whatever viewing volume that user is allowed to move. A light field provides both the spectral power and direction of light propagation at every point in space. If the user is able to walk around in the physical world while wearing a VR headset, then this seems to be an impossible task. How can a rig of cameras capture the light energy in all possible locations at the same instant in an entire room? If the user is constrained to a small area, then the light field can be approximately captured by a rig of cameras arranged on a sphere; a prototype is shown in Figure 7.25. In this case, dozens of cameras may be necessary, and image warping techniques are used to approximate viewpoints between the cameras or from the interior the spherical rig. To further improve the experience,

Figure 7.25: The Pantopticam prototype from Figure Digital uses dozens of cameras to improve the ability to approximate more viewpoints so that stereo viewing and parallax from position changes can be simulated.

light-field cameras (also called *plenoptic cameras*) offer the ability to capture both the intensity of light rays and the direction that they are traveling through space. This offers many advantages, such as refocusing images to different depths, after the light field has already been captured.

Further Reading

Close connections exist between VR and computer graphics because both are required to push visual information onto a display; however, many subtle differences arise and VR is much less developed. For basic computer graphics, many texts provide additional coverage on the topics from this chapter; see, for example [17]. For much more detail on high-performance, high-resolution rendering for computer graphics, see [1]. Comprehensive coverage of BRDFs appears in [2], in addition to [1].

Ray tracing paradigms may need to be redesigned for VR. Useful algorithmic background from a computational geometry perspective can be found in [29, 7]. For optical distortion and correction background, see [8, 12, 14, 15, 27, 28]. Chromatic aberration correction appears in [20]. Automatic stitching of panoramas is covered in [5, 24, 25].

Bibliography

[1] T. Akenine-Möller, E. Haines, and N. Hoffman. *Real-Time Rendering*. CRC Press, Boca Raton, FL, 2008.

[2] J. Birn. *Digital Lighting and Rendering, 3rd Ed.* New Riders, San Francisco, CA, 2013.

[3] J. F. Blinn. Models of light reflection for computer synthesized pictures. In *Proceedings Annual Conference on Computer Graphics and Interactive Techniques*, 1977.

[4] I. Bogost and N. Monfort. *Racing the Beam: The Atari Video Computer System*. MIT Press, Cambridge, MA, 2009.

[5] M. Brown and D. G. Lowe. Automatic panoramic image stitching using invariant features. *International Journal of Computer Vision*, 74(1):59–73, 2007.

[6] E. Catmull. *A subdivision algorithm for computer display of curved surfaces*. PhD thesis, University of Utah, 1974.

[7] A. Y. Chang. A survey of geometric data structures for ray tracing. Technical Report TR-CIS-2001-06, Brooklyn Polytechnic University, 2001.

[8] D. Claus and A. W. Fitzgibbon. A rational function lens distortion model for general cameras. In *Proc. Computer Vision and Pattern Recognition*, pages 213–219, 2005.

[9] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications, 2nd Ed.* Springer-Verlag, Berlin, 2000.

[10] S. R. Ellis, M. J. Young, B. D. Adelstein, and S. M. Ehrlich. Discrimination of changes in latency during head movement. In *Proceedings Computer Human Interfaces*, pages 1129–1133, 1999.

[11] H. Fuchs, Z. M. Kedem, and B. F. Naylor. On visible surface generation by a priori tree structures. In *Proceedings ACM SIGGRAPH*, pages 124–133, 1980.

[12] J. Heikkilä. Geometric camera calibration using circular control points. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 22(10):1066–1077, 2000.

[13] J. Heikkilä and O. Silvén. A four-step camera calibration procedure with implicit image correction. In *Proc. Computer Vision and Pattern Recognition*, pages 1106–1112, 1997.

[14] W. Hugemann. Correcting lens distortions in digital photographs. In *European Association for Accident Research and Analysis (EVU) Conference*, 2010.

[15] K. Mallon and P. F. Whelan. Precise radial un-distortion of images. In *Proc. Computer Vision and Pattern Recognition*, pages 18–21, 2004.

[16] W. R. Mark, L. McMillan, and G. Bishop. Post-rendering 3D warping. In *Proceedings of the Symposium on Interactive 3D Graphics*, pages 7–16, 1997.

[17] S. Marschner and P. Shirley. *Fundamentals of Computer Graphics, 4th Ed.* CRC Press, Boca Raton, FL, 2015.

[18] M. Mine and G. Bishop. Just-in-time pixels. Technical Report TR93-005, University of North Carolina, Chapel Hill, NC, 1993.

[19] T. Möller and N. Trumbore. Fast, minimum storage ray/triangle intersection. *Journal of Graphics Tools*, 2(1):21–28, 1997.

[20] Y. M. H. Ng and C. P. Kwong. Correcting the chromatic aberration in barrel distortion of endoscopic images. *Journal of Systemics, Cybernetics, and Informatics*, 2003.

[21] F. Nicodemus. Directional reflectance and emissivity of an opaque surface. *Applied Optics*, 4(7):767–775, 1965.

[22] S. Petitjean, D. Kriegman, and J. Ponce. Computing exact aspect graphs of curved objects: algebraic surfaces. *International Journal of Computer Vision*, 9:231–255, December 1992.

[23] M. Pocchiola and G. Vegter. The visibility complex. *International Journal Computational Geometry & Applications*, 6(3):279–308, 1996.

[24] N. Snavely, S. M. Seitz, and R. Szeliski. Photo tourism: exploring photo collections in 3D. *ACM Transactions on Graphics*, 25(3):835–846, 2006.

[25] R. Szeliski. Image alignment and stitching: A tutorial. Technical Report MSR-TR-2004-92, Microsoft Research, 2004. Available at http://research.microsoft.com/.

[26] Thomas and Finney. *Calculus and Analytic Geomtry, 9th Ed.* Addison-Wesley, Boston, MA, 1995.

[27] R. Y. Tsai. A versatile camera calibration technique for high-accuracy 3D machine vision metrology using off-the-shelf TV cameras and lenses. *IEEE Journal of Robotics and Automation*, 3(4):323–344, 1987.

[28] G. Vass and T. Perlaki. Applying and removing lens distortion in post production. Technical report, Colorfont, Ltd., Budapest, 2003.

[29] C. Wächter and A. Keller. Instant ray tracing: The bounding interval hierarchy. In T. Akenine-Möller and W. Heidrich, editors, *Eurographics Symposium on Rendering*, pages 139–149. 2006.