Chapter 7

Visual Rendering

Steven M. LaValle
University of Illinois

Copyright Steven M. LaValle 2015

Available for downloading at http://msl.cs.uiuc.edu/vr/
Chapter 7
Visual Rendering

<table>
<thead>
<tr>
<th>Chapter Status</th>
<th>Taken from Virtual Reality, S. M. LaValle</th>
</tr>
</thead>
<tbody>
<tr>
<td>![Warning Icon]</td>
<td>This online chapter is not the final version! Check <a href="http://msl.cs.uiuc.edu/vr/">http://msl.cs.uiuc.edu/vr/</a> for information on the latest draft version.</td>
</tr>
<tr>
<td></td>
<td>This draft was compiled on February 18, 2016.</td>
</tr>
</tbody>
</table>

This chapter explains visual rendering, which specifies what the visual display will 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 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 all exist 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 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 touched by the 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 due to the 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 ?? 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 material properties at the intersection point and the lighting conditions.

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 understands the image through perceptual processes.

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. 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 can be eliminated from consideration by quick coordinate tests. Popular examples include BSP-trees and Bounding Volume Hierarchies. Algorithms that sort geometric information to obtain greater efficiently generally fall under computational geometry. In addition to eliminating many triangles from quick tests, many methods of calculating the ray-triangle intersection has been developed to reduce the number of operations. One of the most popular is the Möller-Trumbore intersection algorithm.

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 is sufficiently represented by R, G, and B values. If the viewing ray the surface 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 \( \theta \) that the surface makes with respect to the light source. Let \( n \) be the outward surface normal and let \( \ell \) be a vector from the surface intersection point to the light source. Assume both \( n \) and \( \ell \) are unit vectors. The dot product \( n \cdot \ell = \sin \theta \) yields the amount of attenuation (between 0 and 1) due to the tilting of the surface relative to the 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

\[
R = d_R I_R \max(0, n \cdot \ell) \\
G = d_G I_G \max(0, n \cdot \ell) \\
B = d_B I_B \max(0, n \cdot \ell),
\]

in which \((d_R, d_G, d_B)\) represents the spectral reflectance property of the material (triangle) and \((I_R, I_G, I_B)\) 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 \(d_R = d_G = d_B\).

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 \(n \cdot \ell < 0\).
7.1. RAY TRACING AND SHADING MODELS

Using vector notation, (7.1) can be compressed into

\[ L = dI \max(0, n \cdot \ell) \]  

(7.2)

in which \( L = (R, G, B) \), \( d = (d_R, d_G, d_B) \), and \( I = (I_R, I_G, I_B) \).

**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 \( \ell \) are 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 \( \ell \). See Figure 7.3. The **bisector** \( b \) is the vector obtained by averaging \( \ell \) and \( v \):

\[ b = \frac{\ell + v}{\| \ell + v \|} \]  

(7.3)

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

\[ L = dI \max(0, n \cdot \ell) + sI \max(0, n \cdot b)^x. \]  

(7.4)

This additively takes into account shading due to both diffuse and specular components. The first term is just the Lambertian shading model. 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.

**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 \( \ell \) and \( v \). If \( n = b \), then ideal reflection is obtained, as in the case of a mirror.

**Ambient shading** Another heuristic is **ambient shading**, which gives 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 light interreflections between objects to illuminate an entire environment. Such propagation has not been taken into account in the shading model so far, thereby requiring a hack to fix it. Adding ambient shading yields

\[ L = L_a + dI \max(0, n \cdot \ell) + sI \max(0, n \cdot b)^x, \]  

(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

\[ L = L_a + \sum_{i=1}^{N} d_I \max(0, n \cdot \ell_i) + sI_i \max(0, n \cdot b_i)^x, \]  

(7.6)

in which \( I_i, \ell_i, \) 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; see Figure 7.4. The \( \theta_l \) and \( \theta_r \) represent the angles of light source and viewing ray, respectively, with respect to the surface. The \( \phi_l \) and \( \phi_r \) parameters range from 0 to 2\( \pi \) 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

\[ f(\theta_l, \phi_l, \theta_r, \phi_r) = \frac{\text{radiance}}{\text{irradiance}}. \]  

(7.7)
in which radiance is the light energy reflected from the surface in directions \( \theta_r \) and \( \phi_r \), and irradiance is the light energy arriving at the surface from directions \( \theta_i \) and \( \phi_i \). 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 outgoing 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 for 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, just like 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 set on the number of bounces to consider. The simple Lambertian and Blinn-Phong models are often used, but more general BRDFs are also common. Increasing levels of realism can be calculated, but with corresponding increases in computation time.

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, and stereo images may bring out 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). It 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 and is the main function of modern graphical processing units (GPUs). In this case, an image is rendering by iterating over every triangle and attempting to color the pixels where the triangle lands 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 first. 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. They 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. It 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.

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

\[
\begin{align*}
    e_1 &= p_2 - p_1 \\
    e_2 &= p_3 - p_2 \\
    e_3 &= p_1 - p_3.
\end{align*}
\]

The point \( p \) lies inside of the triangle if and only if

\[
p \times e_1 < 0, \quad p \times e_2 < 0, \quad p \times e_3 < 0,
\]

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:

\[
p = \alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3
\]

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 \( \alpha_i \). 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. More particularly, let \( d_{ij} = e_i \cdot e_j \) for all combinations of \( i \) and \( j \).

\[
s = 1/(d_{i1}d_{2j} - d_{1j}d_{2i}).
\]

The coordinates are then given by

\[
\begin{align*}
    \alpha_1 &= s \cdot (d_{2j}d_{31} - d_{1j}d_{32}) \\
    \alpha_2 &= s \cdot (d_{1j}d_{32} - d_{12}d_{31}) \\
    \alpha_3 &= 1 - \alpha_1 - \alpha_2.
\end{align*}
\]

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, \( \alpha_1, \alpha_2, \) and \( \alpha_3 \) 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.
Mapping the surface  Barycentric coordinates provide a simple and efficient method for linearly interpolating values across a triangle. The simple 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{align*}
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{align*}

(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; 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, in affects 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, artificial curvature can be given to an object. An important case of normal mapping is called bump mapping, which makes a flat surface look rough by irregularly perturbing the normals. If 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 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 5.22). 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 a mipmap for each texture; see Figure 7.11. The texture is calculated at various

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.)

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.)
7.2. RASTERIZATION

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 significant 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 Wikipedia user Mulad.)

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

resolutions by performing high-density sampling and stored in images. Based on the size and viewpoint of the triangle on the screen, the appropriate 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

\[
\frac{\sqrt{x^2 + y^2}}{z} > \tan \theta,
\]

in which \( \theta \) 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 would be outside of the lens view. These are quickly eliminated from consideration by a simple test as each frame is rendered.
7.2. RASTERIZATION

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 “escalator” 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 caused by the enhanced depth perception afforded by a VR system. Both head motions and stereo views enable us 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 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 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


7.4 Improving Latency and Frame Rates

Get rid of buffering.
- Reduce polygons: simplification. Reduce materials.
- Post-Rendering Image Warp (also called time warp, by Carmack)
- Beam racing (briefly)

7.5 Panoramic Photos and Videos

Like a virtual IMAX theater. VR cinema is related; it is like an extreme case. One can morph from panoramas to a theater.
- Texture mapping onto a sphere (which could be pixel-by-pixel vr mesh, like above)
- What should the radius be? Pros and cons of 6 DOF tracking.
- Mixed reality in this space (bring in avatars).
- Getting bits of parallax from simple tricks.
- Viewpoint and multiple user issues.

Further Reading

Close connections exist between VR and computer graphics because both are required to push visual information onto a display. For more, consult basic textbooks [3]. For high performance rendering, see [1].

Data structures for ray tracing:
- Eurographics Symposium on Rendering (2006), Tomas Akenine-Möller and Wolfgang Heidrich (Editors), Instant Ray Tracing: The Bounding Interval Hierarchy, Carsten Wächter and Alexander Keller


Bibliography

