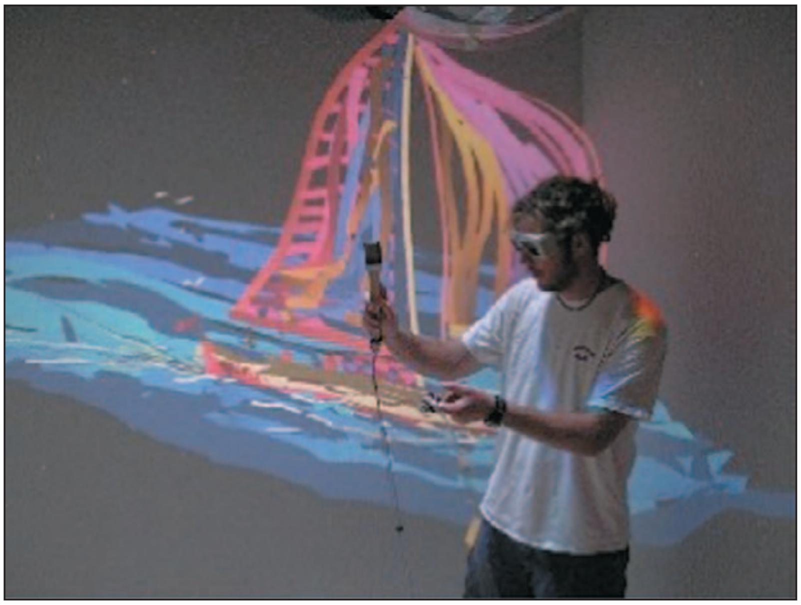
COMPUTER GRAPHICS

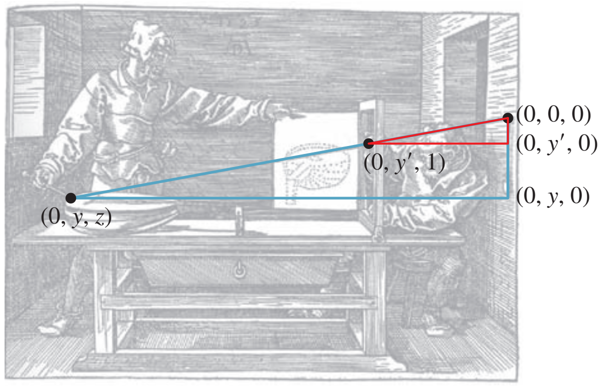
PRINCIPLES AND PRACTICE

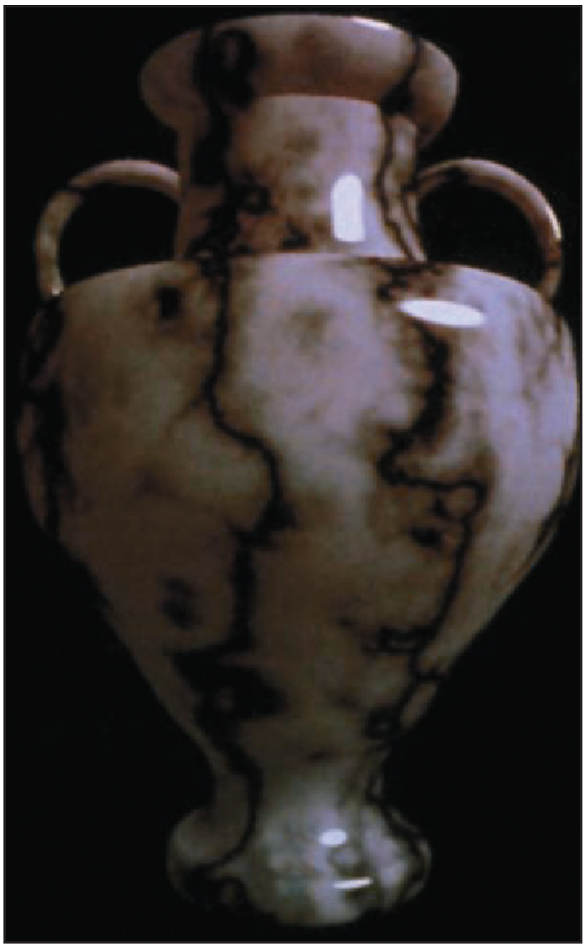
THIRD EDITION

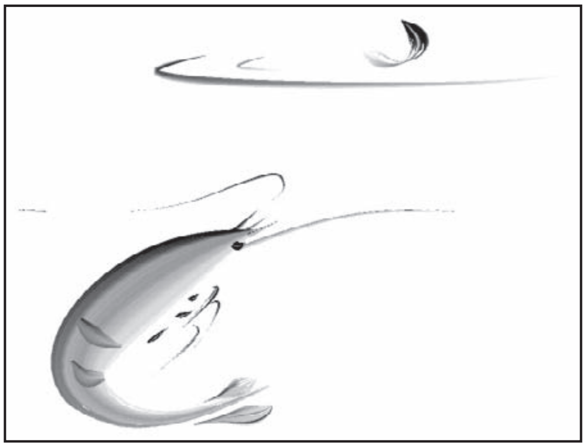
JOHN F. HUGHES · ANDRIES VAN DAM · MORGAN MCGUIRE
DAVID F. SKLAR · JAMES D. FOLEY · STEVEN K. FEINER · KURT AKELEY

Top: Courtesy of Michael Kass, Pixar and Andrew Witkin, © 1991 ACM, Inc. Reprinted by permission. Bottom: Courtesy of Greg Turk, © 1991 ACM, Inc. Reprinted by permission.


Courtesy of Daniel Keefe, University of Minnesota.




Courtesy of Ken Perlin, © 1985 ACM, Inc. Reprinted by permission.


Courtesy of Steve Strassmann. © 1986 ACM, Inc. Reprinted by permission.

The compositing example is one of many cases where a buffer is intended as input for an algorithm rather than for direct display to a human as an image, and α is only one of many common quantities found in buffers that has no direct visible representation. For example, it is common to store "depth" in a buffer that corresponds 1:1 to the color buffer. A **depth buffer** stores some value that maps monotonically to distance from the center of projection to the surface seen at a pixel (we motivate and show how to implement and use a depth buffer in Chapter 15, and evaluate variations on the method and alternatives extensively in Chapter 36 and Section 36.3 in particular).

Another example is a **stencil buffer,** which stores arbitrary bit codes that are frequently used to mask out parts of an image during processing in the way that a physical stencil (see Figure 14.7) does during painting.

Stencil buffers typically use very few bits, so it is common to pack them into some other buffer. For example, Figure 14.8 shows a 3×3 combined depth-and-stencil buffer in the `GL_DEPTH24STENCIL8` format.

A **framebuffer**[1] is an array of buffers with the same dimensions. For example, a framebuffer might contain a `GL_RGBA8` color buffer and a `GL_DEPTH24STENCIL8` depth-and-stencil buffer. The individual buffers act as parallel arrays of fields at each pixel. A program might have multiple framebuffers with many-to-many relationships to the individual buffers.

Why create the framebuffer level of abstraction at all? In the previous example, instead of two buffers, one storing four channels and one with two, why not simply store a single six-channel buffer? One reason for framebuffers is the many-to-many relationship. Consider a 3D modeling program that shows two views of the same object with a common camera but different rendering styles. The left view is wireframe with hidden lines removed, which allows the artist to see the tessellation of the meshes involved. The right view has full, realistic shading. These images can be rendered with two framebuffers. The framebuffers share a single depth buffer but have different color buffers.

Another reason for framebuffers is that the semantic model of channels of specific-bit widths might not match the true implementation, even though it was motivated by implementation details. For example, depth buffers are highly amenable to lossless spatial compression because of how they are computed from continuous surfaces and the spatial-coherence characteristics of typically rendered scenes. Thus, a compressed representation of the depth buffer might take significantly less space (and correspondingly take less time to access because doing so consumes less memory bandwidth) than a naive representation. Yet the compressed representation in this case still maintains the full precision required by the semantic buffer format requested through an API. Unsurprisingly given these observations, it is common practice to store depth buffers in compressed form but present them with the semantics of uncompressed buffers [HAM06]. Taking advantage of this compressibility, especially using dedicated circuitry in a hardware renderer, requires storing the depth values separately from the

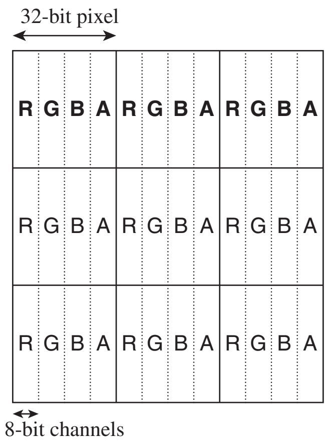


Figure 14.6: The `GL_RGBA8` buffer format packs three 8-bit normalized fixed-point values representing red, green, blue, and coverage values, each on [0, 1], into every 32-bit pixel. This format allows efficient, word-aligned access to an entire pixel for a memory system with 32-bit words. A 64-bit system might fetch two pixels at once and mask off the unneeded bits—although if processing multiple pixels of an image in parallel, both pixels likely need to be read anyway.

Figure 14.7: A real "stencil" is a piece of paper with a shape cut out of it. The stencil is placed against a surface and then painted over. When the stencil is removed, the surface is only painted where the holes were. A computer graphics stencil is a buffer of data that provides similar functionality.

1. The framebuffer is an abstraction of an older idea called the "frame buffer," which was a buffer that held the pixels of the frame. The modern parallel-rendering term is "framebuffer" as a nod to history, but note that it is no longer an actual buffer. It stores the other buffers (depth, color, stencil, etc.). Old "frame buffers" stored multiple "planes" or kinds of values at each pixel, but they often stored these values in the pixel, using an array-of-structs model. Parallel processors don't work as well with an array of structs, so a struct of arrays became preferred for the modern "framebuffer."

other channels. Thus, the framebuffer/color buffer distinction steers the high-level system toward an efficient low-level implementation while abstracting the details of that implementation.

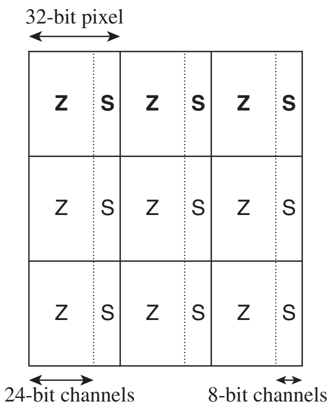


Figure 14.8: The `GL_DEPTH24 STENCIL8` buffer format encodes a 24-bit normalized fixed point "depth" value with eight stencil bits used for arbitrary masking operations.

14.4 Building Blocks of Ray Optics

In the real world, light sources emit photons. These scatter through the world and interact with matter. Some scatter from matter, through an aperture, and then onto a sensor. The aperture may be the iris of a human observer and the sensor that person's retina. Alternatively, the aperture may be at the lens of a camera and the sensor the film or CCD that captures the image. Photorealistic rendering models these systems, from emitter to sensor. It depends on five other categories of models:

1. Light
2. Light emitters
3. Light transport
4. Matter
5. Sensors and their imaging apertures and optics (e.g., cameras and eyes)

We now explore the concepts of each category and some high-level aspects that can be abstracted to conserve space, time, and implementation complexity. Later in the chapter we return to specific common models within each category. We must defer that until later because the models interact, so it is important to understand all before refining any.

Although the first few sections of this chapter have covered a great many details, there is a high-level message as well, one that we summarize in a principle we apply throughout the remainder of the chapter:

✓ **THE HIGH-LEVEL DESIGN PRINCIPLE:** Start from the broadest possible view. Elements of a graphics system don't separate as cleanly as we might like; you can't design the ideal representation for an emitter without considering its impact on light transport. Investing time at the high level lets us avoid the drawbacks of committing, even if it defers gratification.

14.4.1 Light

14.4.1.1 The Visible Spectrum

The energy of real light is transported by photons. Each photon is a quantized amount of energy, so a powerful beam of light contains more photons than a weak beam with the same spectrum, not more powerful photons. The exact amount of energy per photon determines the frequency of the corresponding electromagnetic wave; we perceive it as color. Low-frequency photons appear red to us and high-frequency ones appear blue, with the entire rainbow spectrum in between (see Figure 14.9). "Low" and "high" here are used relative to the visible spectrum. There are photons whose frequencies are outside the visible spectrum, but those can't directly affect rendering, so they are almost always ignored.

The human visual system perceives light containing a mixture of photons of different frequencies as a color somewhere between those created by the

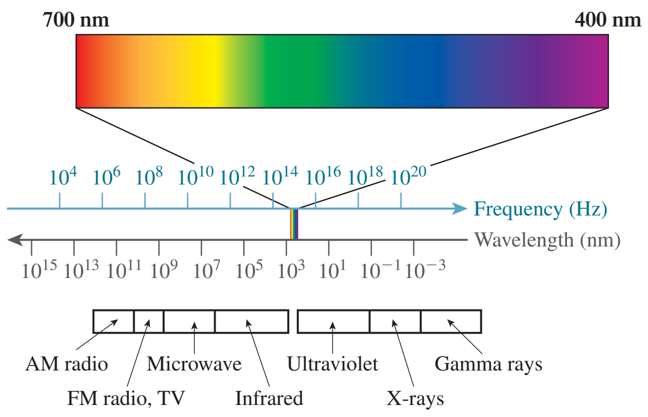


Figure 14.9: The visible spectrum is part of the full electromagnetic spectrum. The color of light that we perceive from an electromagnetic wave is determined by its frequency. The relationship between frequency and wavelength is determined by the medium through which the wave is propagating. (Courtesy of Leonard McMillan)

individual photons. For example, a mixture of "red" and "green" photons appears yellow, and is mostly indistinguishable from pure "yellow" photons. This **aliasing** (i.e., the substitutability of one item for another) is fortunate. It allows displays to create the appearance of many colors using only three relatively narrow frequency bands. Digital cameras also rely on this principle—because the image will be displayed using three frequencies, they only need to measure three.[2] Most significantly for our purposes, almost all 3D rendering treats photons as belonging to three distinct frequencies (or bands of frequencies), corresponding to red, green, and blue. This includes film and games; some niche predictive rendering does simulate more spectral samples. We'll informally refer to rendering with three "frequencies," when what we really mean is "rendering with three frequency bands," Using only three frequencies in simulation minimizes both the space and time cost of rendering algorithms. It creates two limitations. The first is that certain phenomena are impossible to simulate with only three frequencies. For example, the colors of clothing often appear different under fluorescent light and sunlight, even though these light sources may themselves appear fairly similar. This is partly because fluorescent bulbs produce white light by mixing a set of narrow frequency bands, while photons from the sun span the entire visible spectrum. The second limitation of using only three frequencies is that renderers, cameras, and displays rarely use the *same* three frequencies. Each system is able to create the perception of a slightly different space of colors, called a **gamut.** Some colors may simply be outside the gamut of a particular device and lost during capture or display. This also means that the input and output image data for a renderer must be adjusted based on the color profile of the device. Today most devices automatically convert to and from a standard color profile, called sRGB, so color shifts are minimized on such devices but gamut remains a problem.

2. This is not strictly true; Chapter 28 explains why.

The different appearance of cloth under fluorescent light and sunlight is the first example of the Noncommutativity principle—the idea that order matters in some operations we perform in graphics, but that this is often ignored for the sake of speed or simplicity. In this case, the computation of the spectrum of reflected light *should* be carried out with a full representation of the spectrum, and the light should be represented by three samples only when it comes time to store an image preparatory to it being shown on a three-color display. Instead, we've sampled both the spectrum of the emitted light from the source and the reflectance characteristics of the cloth, and multiplied samples rather than spectra. This often produces good-enough results, but can lead to errors.

✓ **THE NONCOMMUTATIVITY PRINCIPLE:** The order of operations often matters in graphics. Swapping the order of operations can introduce both efficiencies in computations and errors in results. You should be sure that you know when you're doing so.

14.4.1.2 Propagation

The speed of propagation of a photon is determined by a material. In a vacuum, it is about $c = 3 \times 10^8$ m/s, which is therefore called the speed of light. The **index of refraction** of a material is the ratio of the speed of light in a vacuum to the rate s of propagation in that material:

$$\eta = \frac{c}{s}. \tag{14.1}$$

For everyday materials, $s < c$, so $\eta \geq 1$ (e.g., household glass has $\eta \approx 1.5$). The exact propagation speed and index of refraction depend on the wavelength of the photon, but the variation is small within the visible spectrum, so it is common to use a single constant for all wavelengths. The primary limitation of this approximation is that the angle of refraction at the interface to a transmissive material is constant for all wavelengths, when it should in fact vary slightly. Effects like rainbows and the spectrum seen from a prism cannot be rendered under this approximation—but when simulating only three wavelengths, rainbows would have only three colors anyway.

Beware that it is common in graphics to refer to the **wavelength** λ of a photon, which is related to temporal frequency[3] f by

$$\lambda = \frac{s}{f}. \tag{14.2}$$

Because the speed of propagation changes when a stream of photons enters a different medium, the wavelength also changes. Yet in graphics we assume that each of our spectral samples is fixed independent of the speed of propagation, so frequency is really what is meant in most cases.

3. Waves have a *temporal* frequency measured in 1/s (i.e., Hz) and a spatial frequency measured in 1/m. The *spatial* frequency of a photon is necessarily $1/\lambda$ and is rarely used in graphics because it varies with the speed of propagation.

Photons propagate along rays within a volume that has a uniform index of refraction, even if the material in that volume is chemically or structurally inhomogeneous. Photons are also selectively absorbed, which is why the world looks darker when seen through a thick pane of glass. At the boundary between volumes with different indices of refraction, light **scatters** by reflecting and refracting in complex ways determined by the microscopic geometry and chemistry of the material. Chapter 26 describes the physics and measurement of light in detail, and Chapter 27 discusses scattering.

14.4.1.3 Units

Photons transport **energy,** which is measured in joules. They move immensely fast compared to a human timescale, so renderers simulate the steady state observed under continuous streams of photons. The **power** of a stream of photons is the rate of energy delivery per unit time, measured in watts. You are familiar with appliance labels that measure the *consumption* in watts and kilowatts. Common household lighting solutions today convert 4% to 10% of the power they consume into visible light, so a typical "100 W" incandescent lightbulb emits at best 10 W of visible light, with 4 W being a more typical value.

In addition to measuring power in watts, there are two other measurements of light that appear frequently in rendering. The first is the power per unit area entering or leaving a surface, in units of W/m^2. This is called **irradiance** or **radiosity** and is especially useful for measuring the light transported between matte surfaces like painted walls. The second is the power per unit area per unit solid angle, measured[4] in $W/(m^2\ sr)$, which is called **radiance.** It is conserved along a ray in a homogeneous medium. It is the quantity transported between two points on different surfaces, and from a point on a surface to a sample location on the image plane.

14.4.1.4 Implementation

It is common practice to represent all of these quantities using a generic 3-vector class (e.g., as done in the GLSL and HLSL APIs), although in general-purpose languages it is frequently considered better practice to at least name the fields based on their frequency, as shown in Listing 14.1.

Listing 14.1: A general class for recording quantities sampled at three visible frequencies.

```
class Color3 {
public:
    /** Magnitude near 650 THz ("red"), either at a single
        frequency or representing a broad range centered at
        650 THz, depending on the usage context. 650 THz
        photons have a wavelength of about 450 nm in air.*/
    float r;

    /** Near 550 THz ("green"); about 500 nm in air. */
    float g;

    /** Near 450 THz ("blue"); about 650 nm in air. */
    float b;
```

4. The unit "sr" is "steradians," a measure of the size of a region on the unit sphere, described in more detail in Section 14.11.1.

```

    Color3() : r(0), g(0), b(0) {}
    Color3(float r, float g, float b) : r(r), g(g), b(b);
    Color3 operator*(float s) const {
      return Color3(s * r, s * g, s * b);
    }
    ...
};
```

One could use the type system to help track units by creating distinct classes for power, radiance, etc. However, it is often convenient to reduce the complexity of the types in a program by simply aliasing these to the common "color"[5] class, as shown, for example, in Listing 14.2.

Listing 14.2: Aliases of `Color3` with unit semantics.

```
typedef Color3 Power3;
typedef Color3 Radiosity3;
typedef Color3 Radiance3;
typedef Color3 Biradiance3;
```

Because bandwidth and total storage space are often limited resources, it is common to employ the fewest bits practical for your needs for each frequency-varying quantity. One implementation strategy is to parameterize the class, as shown in Listing 14.3.

Listing 14.3: A templated `Color` class and instantiations.

```
template<class T>
class Color3 {
public:
    T r, g, b;

    Color3() : r(0), g(0), b(0) {}
    ...
};

/** Matches GL_RGB8 format */
typedef Color3<unint8> Color3un8;

/** Matches GL_RGB32F format */
typedef Color3<float> Color3f32;

/** Matches GL_RGB16I format */
typedef Color3<unsigned short> Color3ui16;
```

14.4.2 Emitters

Emitters are fairly straightforward to model accurately. They create and cast photons into the scene. The photons have locations, propagation directions, and frequencies (i.e., "colors"), and are emitted at some rate. Given probability distributions for those parameters, we can generate many representative photons and

5. We discuss why color is not a quantifiable phenomenon in Chapter 28; here we use the term in a nontechnical fashion that is casual jargon in the field.