

Surface Shading

To make objects appear to have more volume, it can help to use *shading*, i.e., the surface is “painted” with light. This chapter presents the most common heuristic shading methods. The first two, diffuse and Phong shading, were developed in the 1970s and are available in most graphics libraries. The last, artistic shading, uses artistic conventions to assign color to objects. This creates images reminiscent of technical drawings, which is desirable in many applications.

10.1 Diffuse Shading

Many objects in the world have a surface appearance loosely described as “matte,” indicating that the object is not at all shiny. Examples include paper, unfinished wood, and dry unpolished stones. To a large degree, such objects do not have a color change with a change in viewpoint. For example, if you stare at a particular point on a piece of paper and move while keeping your gaze fixed on that point, the color at that point will stay relatively constant. Such matte objects can be considered as behaving as *Lambertian* objects. This section discusses how to implement the shading of such objects. A key point is that all formulas in this chapter should be evaluated in world coordinates and not in the warped coordinates after the perspective transform is applied. Otherwise, the angles between normals are changed and the shading will be inaccurate.

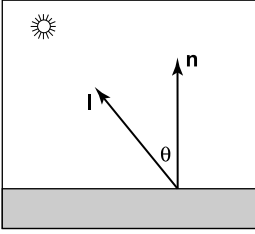


Figure 10.1. The geometry for Lambert's Law. Both \mathbf{n} and \mathbf{l} are unit vectors.

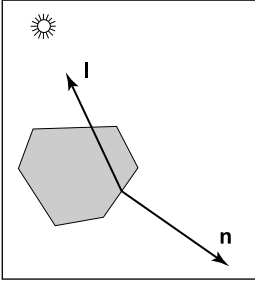


Figure 10.2. When a surface points away from the light, it should receive no light. This case can be verified by checking whether the dot product of \mathbf{l} and \mathbf{n} is negative.

10.1.1 Lambertian Shading Model

A Lambertian object obeys *Lambert's cosine law*, which states that the color c of a surface is proportional to the cosine of the angle between the surface normal and the direction to the light source (Gouraud, 1971):

$$c \propto \cos \theta,$$

or in vector form,

$$c \propto \mathbf{n} \cdot \mathbf{l},$$

where \mathbf{n} and \mathbf{l} are shown in Figure 10.1. Thus, the color on the surface will vary according to the cosine of the angle between the surface normal and the light direction. Note that the vector \mathbf{l} is typically assumed not to depend on the location of the object. That assumption is equivalent to assuming the light is “distant” relative to object size. Such a “distant” light is often called a *directional light*, because its position is specified only by a direction.

A surface can be made lighter or darker by changing the intensity of the light source or the reflectance of the surface. The diffuse reflectance c_r is the fraction of light reflected by the surface. This fraction will be different for different color components. For example, a surface is red if it reflects a higher fraction of red incident light than blue incident light. If we assume surface color is proportional to the light reflected from a surface, then the diffuse reflectance c_r —an RGB color—must also be included:

$$c \propto c_r \mathbf{n} \cdot \mathbf{l}. \quad (10.1)$$

The right-hand side of Equation (10.1) is an RGB color with all RGB components in the range $[0, 1]$. We would like to add the effects of light intensity while keeping the RGB components in the range $[0, 1]$. This suggests adding an RGB intensity term c_l which itself has components in the range $[0, 1]$:

$$c = c_r c_l \mathbf{n} \cdot \mathbf{l}. \quad (10.2)$$

This is a very convenient form, but it can produce RGB components for c that are outside the range $[0, 1]$, because the dot product can be negative. The dot product is negative when the surface is pointing away from the light as shown in Figure 10.2.

The “max” function can be added to Equation (10.2) to test for that case:

$$c = c_r c_l \max(0, \mathbf{n} \cdot \mathbf{l}). \quad (10.3)$$

Another way to deal with the “negative” light is to use an absolute value:

$$c = c_r c_l |\mathbf{n} \cdot \mathbf{l}|. \quad (10.4)$$



While Equation (10.4) may seem physically implausible, it actually corresponds to Equation (10.3) with two lights in opposite directions. For this reason it is often called *two-sided* lighting (Figure 10.3).

10.1.2 Ambient Shading

One problem with the diffuse shading of Equation (10.3) is that any point whose normal faces away from the light will be black. In real life, light is reflected all over, and some light is incident from every direction. In addition, there is often skylight giving “ambient” lighting. One way to handle this is to use several light sources. A common trick is to always put a dim source at the eye so that all visible points will receive some light. Another way is to use two-sided lighting as described by Equation (10.4). A more common approach is to add an ambient term (Gouraud, 1971). This is just a constant color term added to Equation (10.3):

$$c = c_r (c_a + c_l \max(0, \mathbf{n} \cdot \mathbf{l})) .$$

Intuitively, you can think of the ambient color c_a as the average color of all surfaces in the scene. If you want to ensure that the computed RGB color stays in the range $[0, 1]^3$, then $c_a + c_l \leq (1, 1, 1)$. Otherwise your code should “clamp” RGB values above one to have the value one.

10.1.3 Vertex-Based Diffuse Shading

If we apply Equation (10.1) to an object made up of triangles, it will typically have a faceted appearance. Often, the triangles are an approximation to a smooth surface. To avoid the faceted appearance, we can place surface normal vectors at the vertices of the triangles (Phong, 1975), and apply Equation (10.3) at each of the vertices using the normal vectors at the vertices (see Figure 10.4). This will give a color at each triangle vertex, and this color can be interpolated using the barycentric interpolation described in Section 8.1.2.

One problem with shading at triangle vertices is that we need to get the normals from somewhere. Many models will come with normals supplied. If you tessellate your own smooth model, you can create normals when you create the triangles. If you are presented with a polygonal model that does not have normals at vertices and you want to shade it smoothly, you can compute normals by a variety of heuristic methods. The simplest is to just average the normals of the triangles that share each vertex and use this average normal at the vertex. This

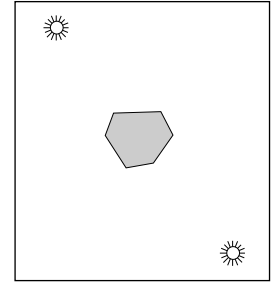


Figure 10.3. Using Equation (10.4), the two-sided lighting formula, is equivalent to assuming two opposing light sources of the same color.

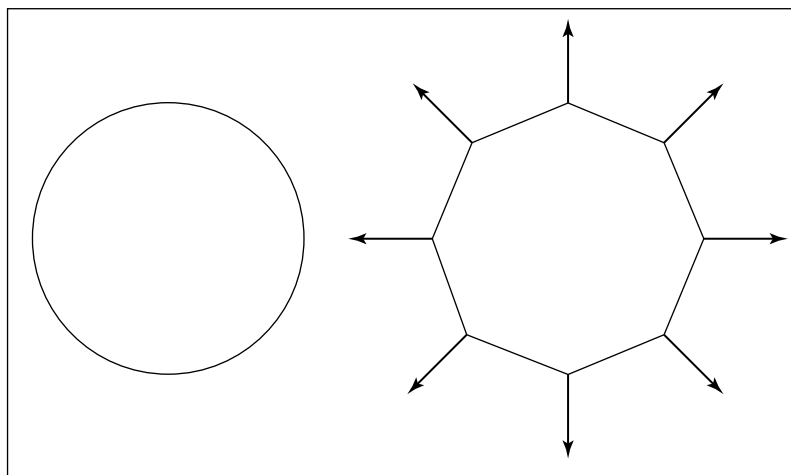


Figure 10.4. A circle (left) is approximated by an octagon (right). Vertex normals record the surface normal of the original curve.

average normal will not automatically be of unit length, so you should convert it to a unit vector before using it for shading.

10.2 Phong Shading

Some surfaces are essentially like matte surfaces, but they have *highlights*. Examples of such surfaces include polished tile floors, gloss paint, and whiteboards. Highlights move across a surface as the viewpoint moves. This means that we must add a unit vector \mathbf{e} toward the eye into our equations. If you look carefully at highlights, you will see that they are really reflections of the light; sometimes these reflections are blurred. The color of these highlights is the color of the light—the surface color seems to have little effect. This is because the reflection occurs at the object’s surface, and the light that penetrates the surface and picks up the object’s color is scattered diffusely.

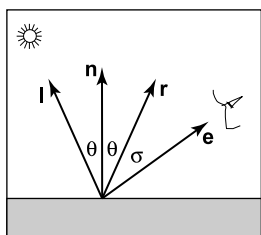


Figure 10.5. The geometry for the Phong illumination model. The eye should see a highlight if σ is small.

10.2.1 Phong Lighting Model

We want to add a fuzzy “spot” the same color as the light source in the right place. The center of the dot should be drawn where the direction \mathbf{e} to the eye “lines” up with the natural direction of reflection \mathbf{r} as shown in Figure 10.5. Here “lines up” is mathematically equivalent to “where σ is zero.” We would like to have the



highlight have some non-zero area, so that the eye sees some highlight wherever σ is small.

Given \mathbf{r} , we'd like a heuristic function that is bright when $\mathbf{e} = \mathbf{r}$ and falls off gradually when \mathbf{e} moves away from \mathbf{r} . An obvious candidate is the cosine of the angle between them:

$$c = c_l(\mathbf{e} \cdot \mathbf{r}),$$

There are two problems with using this equation. The first is that the dot product can be negative. This can be solved computationally with an “if” statement that sets the color to zero when the dot product is negative. The more serious problem is that the highlight produced by this equation is much wider than that seen in real

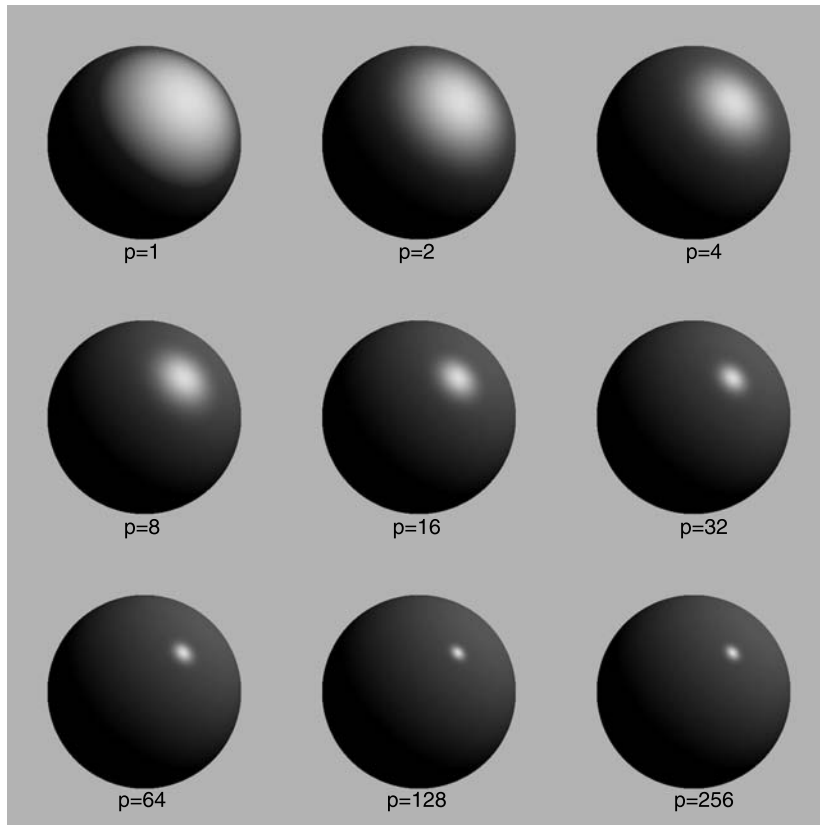


Figure 10.6. The effect of the Phong exponent on highlight characteristics. This uses Equation (10.5) for the highlight. There is also a diffuse component, giving the objects a shiny but non-metallic appearance. *Image courtesy Nate Robins.*

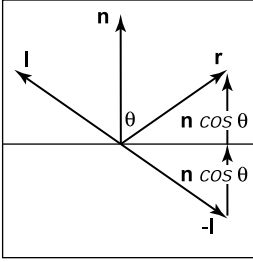


Figure 10.7. The geometry for calculating the vector \mathbf{r} .

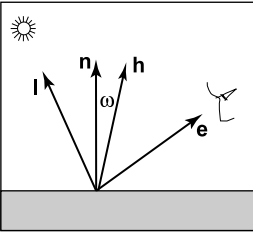


Figure 10.8. The unit vector \mathbf{h} is halfway between \mathbf{l} and \mathbf{e} .

life. The maximum is in the right place and it is the right color, but it is just too big. We can narrow it without reducing its maximum color by raising to a power:

$$c = c_l \max(0, \mathbf{e} \cdot \mathbf{r})^p. \quad (10.5)$$

Here p is called the *Phong exponent*; it is a positive real number (Phong, 1975). The effect that changing the Phong exponent has on the highlight can be seen in Figure 10.6.

To implement Equation (10.5), we first need to compute the unit vector \mathbf{r} . Given unit vectors \mathbf{l} and \mathbf{n} , \mathbf{r} is the vector \mathbf{l} reflected about \mathbf{n} . Figure 10.7 shows that this vector can be computed as

$$\mathbf{r} = -\mathbf{l} + 2(\mathbf{l} \cdot \mathbf{n})\mathbf{n}, \quad (10.6)$$

where the dot product is used to compute $\cos \theta$.

An alternative heuristic model based on Equation (10.5) eliminates the need to check for negative values of the number used as a base for exponentiation (Warn, 1983). Instead of \mathbf{r} , we compute \mathbf{h} , the unit vector halfway between \mathbf{l} and \mathbf{e} (Figure 10.8):

$$\mathbf{h} = \frac{\mathbf{e} + \mathbf{l}}{\|\mathbf{e} + \mathbf{l}\|}.$$

The highlight occurs when \mathbf{h} is near \mathbf{n} , i.e., when $\cos \omega = \mathbf{h} \cdot \mathbf{n}$ is near 1. This suggests the rule:

$$c = c_l (\mathbf{h} \cdot \mathbf{n})^p. \quad (10.7)$$

The exponent p here will have analogous control behavior to the exponent in Equation (10.5), but the angle between \mathbf{h} and \mathbf{n} is half the size of the angle between \mathbf{e} and \mathbf{r} , so the details will be slightly different. The advantage of using the cosine between \mathbf{n} and \mathbf{h} is that it is always positive for eye and light above the plane. The disadvantage is that a square root and divide is needed to compute \mathbf{h} .

In practice, we want most materials to have a diffuse appearance in addition to a highlight. We can combine Equations (10.3) and (10.7) to get

$$c = c_r (c_a + c_l \max(0, \mathbf{n} \cdot \mathbf{l})) + c_l (\mathbf{h} \cdot \mathbf{n})^p. \quad (10.8)$$

If we want to allow the user to dim the highlight, we can add a control term c_p :

$$c = c_r (c_a + c_l \max(0, \mathbf{n} \cdot \mathbf{l})) + c_l c_p (\mathbf{h} \cdot \mathbf{n})^p. \quad (10.9)$$

The term c_p is a RGB color, which allows us to change highlight colors. This is useful for metals where $c_p = c_r$, because highlights on metal take on a metallic color. In addition, it is often useful to make c_p a neutral value less than one, so that colors stay below one. For example, setting $c_p = 1 - M$ where M is the maximum component of c_r will keep colors below one for one light source and no ambient term.



10.2.2 Surface Normal Vector Interpolation

Smooth surfaces with highlights tend to change color quickly compared to Lambertian surfaces with the same geometry. Thus, shading at the normal vectors can generate disturbing artifacts.

These problems can be reduced by interpolating the normal vectors across the polygon and then applying Phong shading at each pixel. This allows you to get good images without making the size of the triangles extremely small. Recall from Chapter 3, that when rasterizing a triangle, we compute barycentric coordinates (α, β, γ) to interpolate the vertex colors c_0, c_1, c_2 :

$$c = \alpha c_0 + \beta c_1 + \gamma c_2. \quad (10.10)$$

We can use the same equation to interpolate surface normals $\mathbf{n}_0, \mathbf{n}_1$, and \mathbf{n}_2 :

$$\mathbf{n} = \alpha \mathbf{n}_0 + \beta \mathbf{n}_1 + \gamma \mathbf{n}_2. \quad (10.11)$$

And Equation (10.9) can then be evaluated for the \mathbf{n} computed at each pixel. Note that the \mathbf{n} resulting from Equation (10.11) is usually not a unit normal. Better visual results will be achieved if it is converted to a unit vector before it is used in shading computations. This type of normal interpolation is often called *Phong normal interpolation* (Phong, 1975).

10.3 Artistic Shading

The Lambertian and Phong shading methods are based on heuristics designed to imitate the appearance of objects in the real world. Artistic shading is designed to mimic drawings made by human artists (Yessios, 1979; Dooley & Cohen, 1990; Saito & Takahashi, 1990; L. Williams, 1991). Such shading seems to have advantages in many applications. For example, auto manufacturers hire artists to draw diagrams for car owners' manuals. This is more expensive than using much more "realistic" photographs, so there is probably some intrinsic advantage to the techniques of artists when certain types of communication are needed. In this section, we show how to make subtly shaded line drawings reminiscent of human-drawn images. Creating such images is often called *non-photorealistic rendering*, but we will avoid that term because many non-photorealistic techniques are used for efficiency that are not related to any artistic practice.

10.3.1 Line Drawing

The most obvious thing we see in human drawings that we don't see in real life is *silhouettes*. When we have a set of triangles with shared edges, we should draw

an edge as a silhouette when one of the two triangles sharing an edge faces toward the viewer, and the other triangle faces away from the viewer. This condition can be tested for two normals \mathbf{n}_0 and \mathbf{n}_1 by

$$\text{draw silhouette if } (\mathbf{e} \cdot \mathbf{n}_0)(\mathbf{e} \cdot \mathbf{n}_1) \leq 0.$$

Here \mathbf{e} is a vector from the edge to the eye. This can be any point on the edge or either of the triangles. Alternatively, if $f_i(\mathbf{p}) = 0$ are the implicit plane equations for the two triangles, the test can be written

$$\text{draw silhouette if } f_0(\mathbf{e})f_1(\mathbf{e}) \leq 0.$$

We would also like to draw visible edges of a polygonal model. To do this, we can use either of the hidden surface methods of Chapter 12 for drawing in the background color and then draw the outlines of each triangle in black. This, in fact, will also capture the silhouettes. Unfortunately, if the polygons represent a smooth surface, we really don't want to draw most of those edges. However, we might want to draw all *creases* where there really is a corner in the geometry. We can test for creases by using a heuristic threshold:

$$\text{draw crease if } (\mathbf{n}_0 \cdot \mathbf{n}_1) \leq \text{threshold}.$$

This combined with the silhouette test will give nice-looking line drawings.

10.3.2 Cool-to-Warm Shading

When artists shade line drawings, they often use low intensity shading to give some impression of curve to the surface and to give colors to objects (Gooch et al., 1998). Surfaces facing in one direction are shaded with a cool color, such as a blue, and surfaces facing in the opposite direction are shaded with a warm color, such as orange. Typically these colors are not very saturated and are also not dark. That way, black silhouettes show up nicely. Overall this gives a cartoon-like effect. This can be achieved by setting up a direction to a “warm” light \mathbf{l} and using the cosine to modulate color, where the warmth constant k_w is defined on $[0, 1]$:

$$k_w = \frac{1 + \mathbf{n} \cdot \mathbf{l}}{2}.$$

The color c is then just a linear blend of the cool color c_c and the warm color c_w :

$$c = k_w c_w + (1 - k_w) c_c.$$

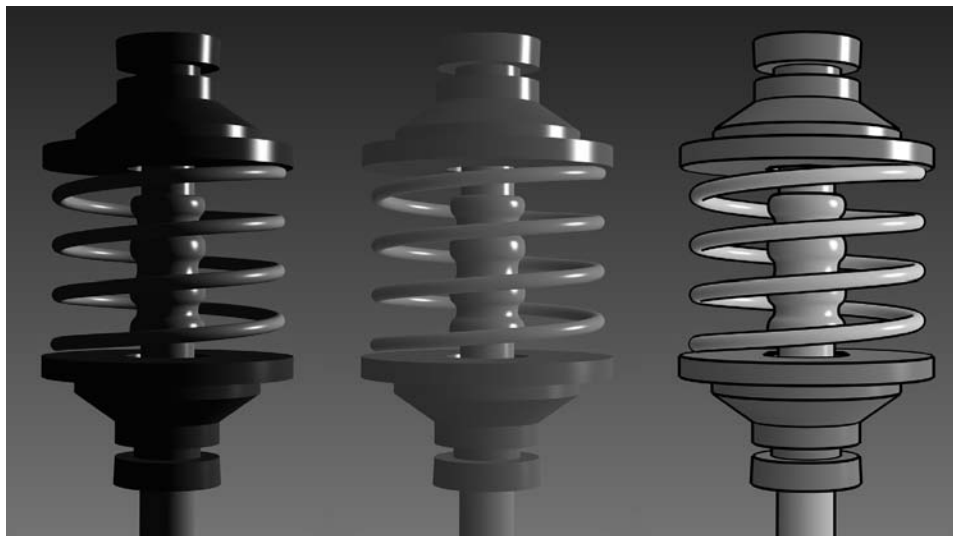


Figure 10.9. Left: a Phong-illuminated image. Middle: cool-to-warm shading is not useful without silhouettes. Right: cool-to-warm shading plus silhouettes. *Image courtesy Amy Gooch.* (See also Plate III.)

There are many possible c_w and c_b that will produce reasonable looking results. A good starting place for a guess is

$$c_c = (0.4, 0.4, 0.7),$$

$$c_c = (0.8, 0.6, 0.6).$$

Figure 10.9 shows a comparison between traditional Phong lighting and this type of artistic shading.

Frequently Asked Questions

- All of the shading in this chapter seems like enormous hacks. Is that true?

Yes. However, they are carefully designed hacks that have proven useful in practice. In the long run, we will probably have better-motivated algorithms that include physics, psychology, and tone-mapping. However, the improvements in image quality will probably be incremental.

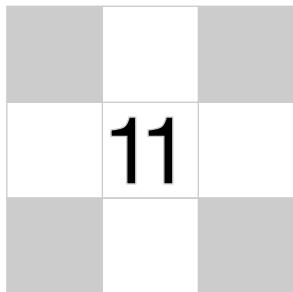


- I hate calling `pow()`. Is there a way to avoid it when doing Phong lighting?

A simple way is to only have exponents that are themselves a power of two, i.e., 2, 4, 8, 16, In practice, this is not a problematic restriction for most applications. A look-up table is also possible, but will often not give a large speed-up.

Exercises

1. The moon is poorly approximated by diffuse or Phong shading. What observations tell you that this is true?
2. Velvet is poorly approximated by diffuse or Phong shading. What observations tell you that this is true?
3. Why do most highlights on plastic objects look white, while those on gold metal look gold?



Texture Mapping

The shading models presented in Chapter 10 assume that a diffuse surface has uniform reflectance c_r . This is fine for surfaces such as blank paper or painted walls, but it is inefficient for objects such as a printed sheet of paper. Such objects have an appearance whose complexity arises from variation in reflectance properties. While we could use such small triangles that the variation is captured by varying the reflectance properties of the triangles, this would be inefficient.

The common technique to handle variations of reflectance is to store the reflectance as a function or a pixel-based image and “map” it onto a surface (Catmull, 1975). The function or image is called a *texture map*, and the process of controlling reflectance properties is called *texture mapping*. This is not hard to implement once you understand the coordinate systems involved. Texture mapping can be classified by several different properties:

1. the dimensionality of the texture function,
2. the correspondences defined between points on the surface and points in the texture function, and
3. whether the texture function is primarily procedural or primarily a table look-up.

These items are usually closely related, so we will somewhat arbitrarily classify textures by their dimension. We first cover 3D textures, often called *solid* textures or *volume* textures. We will then cover 2D textures, sometimes called *image*

textures. When graphics programmers talk about textures without specifying dimension, they usually mean 2D textures. However, we begin with 3D textures because, in many ways, they are easier to understand and implement. At the end of the chapter we discuss bump mapping and displacement mapping which use textures to change surface normals and position, respectively. Although those methods modify properties other than reflectance, the images/functions they use are still called textured. This is consistent with common usage where any image used to modify object appearance is called a texture.

11.1 3D Texture Mapping

In previous chapters we used c_r as the diffuse reflectance at a point on an object. For an object that does not have a solid color, we can replace this with a function $c_r(\mathbf{p})$ which maps 3D points to RGB colors (Peachey, 1985; Perlin, 1985). This function might just return the reflectance of the object that contains \mathbf{p} . But for objects with *texture*, we should expect $c_r(\mathbf{p})$ to vary as \mathbf{p} moves across a surface. One way to do this is to create a 3D texture that defines an RGB value at every point in 3D space. We will only call it for points \mathbf{p} on the surface, but it is usually easier to define it for all 3D points than a potentially strange 2D subset of points that are on an arbitrary surface. Such a strategy is clearly suitable for surfaces that are “carved” from a solid medium, such as a marble sculpture.

Note that in a ray-tracing program, we have immediate access to the point \mathbf{p} seen through a pixel. However, for a z-buffer or BSP-tree program, we only know the point after projection into device coordinates. We will show how to resolve this problem in Section 11.3.1.

11.1.1 3D Stripe Textures

There are a surprising number of ways to make a striped texture. Let’s assume we have two colors c_0 and c_1 that we want to use to make the stripe color. We need some oscillating function to switch between the two colors. An easy one is a sine:

```

RGB stripe( point  $\mathbf{p}$  )
if (sin( $x_p$ ) > 0) then
    return  $c_0$ 
else
    return  $c_1$ 
```



We can also make the stripe's width w controllable:

```

RGB stripe( point p, real  $w$  )
if ( $\sin(\pi x_p/w) > 0$ ) then
    return  $c_0$ 
else
    return  $c_1$ 

```

If we want to interpolate smoothly between the stripe colors, we can use a parameter t to vary the color linearly:

```

RGB stripe( point p, real  $w$  )
 $t = (1 + \sin(\pi x_p/w))/2$ 
return  $(1 - t)c_0 + tc_1$ 

```

These three possibilities are shown in Figure 11.1.

11.1.2 Texture Arrays

Another way we can specify texture in space is to store a 3D array of color values and to associate a spatial position to each of these values. We first discuss this for 2D arrays in 2D space. Such textures can be applied in 3D by using two of the dimensions, e.g. x and y , to determine what texture values are used. We then extend those 2D results to 3D.

We will assume the two dimensions to be mapped are called u and v . We also assume we have an n_x by n_y image that we use as the texture. Somehow we need every (u, v) to have an associated color found from the image. A fairly standard way to make texturing work for (u, v) is to first remove the integer portion of (u, v) so that it lies in the unit square. This has the effect of “tiling” the entire uv plane with copies of the now-square texture (Figure 11.2). We then use one of three interpolation strategies to compute the image color for that coordinate. The simplest strategy is to treat each image pixel as a constant colored rectangular tile (Figure 11.3 (a)). To compute the colors, we apply $c(u, v) = c_{ij}$, where $c(u, v)$ is the texture color at (u, v) and c_{ij} is the pixel color for pixel indices:

$$\begin{aligned} i &= \lfloor un_x \rfloor, \\ j &= \lfloor vn_y \rfloor; \end{aligned} \tag{11.1}$$

$\lfloor x \rfloor$ is the floor of x , (n_x, n_y) is the size of the image being textured, and the indices start at $(i, j) = (0, 0)$. This method for a simple image is shown in Figure 11.3 (b).

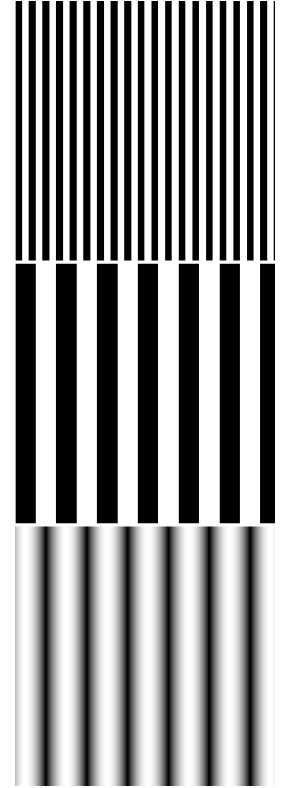


Figure 11.1. Various stripe textures result from drawing a regular array of xy points while keeping z constant.

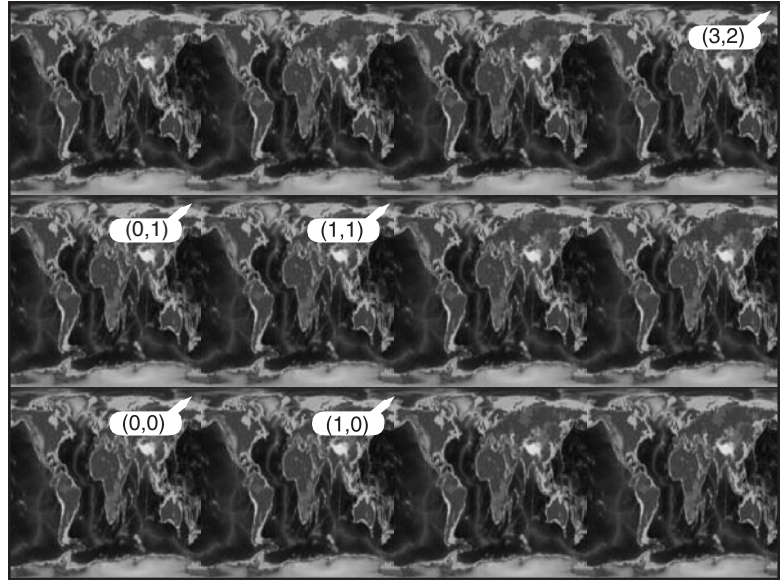


Figure 11.2. The tiling of an image onto the (u, v) plane. Note that the input image is rectangular, and that this rectangle is mapped to a unit square on the (u, v) plane.

For a smoother texture, a bilinear interpolation can be used as shown in Figure 11.3 (c). Here we use the formula

$$\begin{aligned} c(u, v) = & (1 - u')(1 - v')c_{ij} \\ & + u'(1 - v')c_{(i+1)j} \\ & + (1 - u')v'c_{i(j+1)} \\ & + u'v'c_{(i+1)(j+1)} \end{aligned}$$

where

$$\begin{aligned} u' &= n_x u - \lfloor n_x u \rfloor, \\ v' &= n_y v - \lfloor n_y v \rfloor. \end{aligned}$$

The discontinuities in the derivative in intensity can cause visible mach bands, so hermite smoothing can be used:

$$\begin{aligned} c(u, v) = & (1 - u'')(1 - v'')c_{ij} + \\ & + u''(1 - v'')c_{(i+1)j} \\ & + (1 - u'')v''c_{i(j+1)} \\ & + u''v''c_{(i+1)(j+1)}, \end{aligned}$$

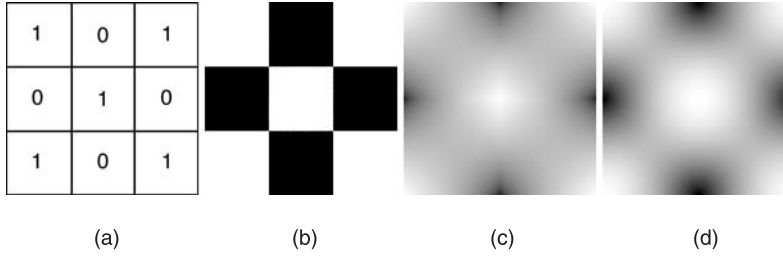


Figure 11.3. (a) The image on the left has nine pixels that are all either black or white. The three interpolation strategies are (b) nearest-neighbor, (c) bilinear, and (d) hermite.

where

$$\begin{aligned} u'' &= 3(u')^2 - 2(u')^3, \\ v'' &= 3(v')^2 - 2(v')^3, \end{aligned}$$

which results in Figure 11.3 (d).

In 3D, we have a 3D array of values. All of the ideas from 2D extend naturally. As an example, let's assume that we will do *trilinear* interpolation between values. First, we compute the texture coordinates (u', v', w') and the lower indices (i, j, k) of the array element to be interpolated:

$$\begin{aligned} c(u, v, w) &= (1 - u')(1 - v')(1 - w')c_{ijk} \\ &\quad + u'(1 - v')(1 - w')c_{(i+1)jk} \\ &\quad + (1 - u')v'(1 - w')c_{i(j+1)k} \\ &\quad + (1 - u')(1 - v')w'c_{ij(k+1)} \\ &\quad + u'v'(1 - w')c_{(i+1)(j+1)k} \\ &\quad + u'(1 - v')w'c_{(i+1)j(k+1)} \\ &\quad + (1 - u')v'w'c_{i(j+1)(k+1)} \\ &\quad + u'v'w'c_{(i+1)(j+1)(k+1)}, \end{aligned} \tag{11.2}$$

where

$$\begin{aligned} u' &= n_x u - \lfloor n_x u \rfloor, \\ v' &= n_y v - \lfloor n_y v \rfloor, \\ w' &= n_z w - \lfloor n_z w \rfloor. \end{aligned} \tag{11.3}$$

11.1.3 Solid Noise

Although regular textures such as stripes are often useful, we would like to be able to make “mottled” textures such as we see on birds’ eggs. This is usually done

by using a sort of “solid noise,” usually called *Perlin noise* after its inventor, who received a technical Academy Award for its impact in the film industry (Perlin, 1985).

Getting a noisy appearance by calling a random number for every point would not be appropriate, because it would just be like “white noise” in TV static. We would like to make it smoother without losing the random quality. One possibility is to blur white noise, but there is no practical implementation of this. Another possibility is to make a large lattice with a random number at every lattice point, and then interpolate these random points for new points between lattice nodes; this is just a 3D texture array as described in the last section with random numbers in the array. This technique makes the lattice too obvious. Perlin used a variety of tricks to improve this basic lattice technique so the lattice was not so obvious. This results in a rather baroque-looking set of steps, but essentially there are just three changes from linearly interpolating a 3D array of random values. The first change is to use Hermite interpolation to avoid mach bands, just as can be done with regular textures. The second change is the use of random vectors rather than values, with a dot product to derive a random number; this makes the underlying grid structure less visually obvious by moving the local minima and maxima off the grid vertices. The third change is to use a 1D array and hashing to create a virtual 3D array of random vectors. This adds computation to lower memory use. Here is his basic method:

$$n(x, y, z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor+1} \sum_{j=\lfloor y \rfloor}^{\lfloor y \rfloor+1} \sum_{k=\lfloor z \rfloor}^{\lfloor z \rfloor+1} \Omega_{ijk}(x-i, y-j, z-k),$$

where (x, y, z) are the Cartesian coordinates of \mathbf{x} , and

$$\Omega_{ijk}(u, v, w) = \omega(u)\omega(v)\omega(w) (\Gamma_{ijk} \cdot (u, v, w)),$$

and $\omega(t)$ is the cubic weighting function:

$$\omega(t) = \begin{cases} 2|t|^3 - 3|t|^2 + 1 & \text{if } |t| < 1, \\ 0 & \text{otherwise.} \end{cases}$$

The final piece is that Γ_{ijk} is a random unit vector for the lattice point $(x, y, z) = (i, j, k)$. Since we want any potential ijk , we use a pseudorandom table:

$$\Gamma_{ijk} = \mathbf{G}(\phi(i + \phi(j + \phi(k)))),$$

where \mathbf{G} is a precomputed array of n random unit vectors, and $\phi(i) = P[i \bmod n]$ where P is an array of length n containing a permutation of the

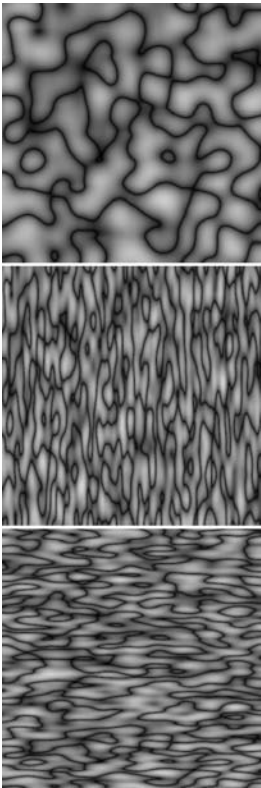


Figure 11.4. Absolute value of solid noise, and noise for scaled x and y values.



integers 0 through $n - 1$. In practice, Perlin reports $n = 256$ works well. To choose a random unit vector (v_x, v_y, v_z) first set

$$\begin{aligned}v_x &= 2\xi - 1, \\v_y &= 2\xi' - 1, \\v_z &= 2\xi'' - 1,\end{aligned}$$

where ξ, ξ', ξ'' are canonical random numbers (uniform in the interval $[0, 1)$). Then, if $(v_x^2 + v_y^2 + v_z^2) < 1$, make the vector a unit vector. Otherwise keep setting it randomly until its length is less than one, and then make it a unit vector. This is an example of a *rejection method*, which will be discussed more in Chapter 14. Essentially, the “less than” test gets a random point in the unit sphere, and the vector for the origin to that point is uniformly random. That would not be true of random points in the cube, so we “get rid” of the corners with the test.

Because solid noise can be positive or negative, it must be transformed before being converted to a color. The absolute value of noise over a ten by ten square is shown in Figure 11.4, along with stretched versions. There versions are stretched by scaling the points input to the noise function.

The dark curves are where the original noise function changed from positive to negative. Since noise varies from -1 to 1 , a smoother image can be achieved by using $(\text{noise} + 1)/2$ for color. However, since noise values close to 1 or -1 are rare, this will be a fairly smooth image. Larger scaling can increase the contrast (Figure 11.5).

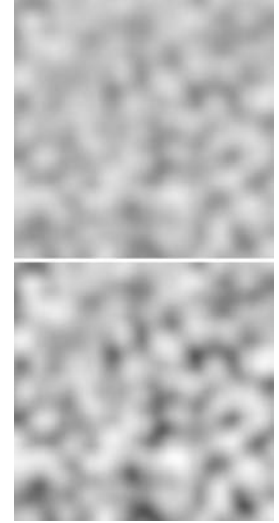


Figure 11.5. Using $0.5(\text{noise}+1)$ (top) and $0.8(\text{noise}+1)$ (bottom) for intensity.

11.1.4 Turbulence

Many natural textures contain a variety of feature sizes in the same texture. Perlin uses a pseudofractal “turbulence” function:

$$n_t(\mathbf{x}) = \sum_i \frac{|n(2^i \mathbf{x})|}{2^i}$$

This effectively repeatedly adds scaled copies of the noise function on top of itself as shown in Figure 11.6.

The turbulence can be used to distort the stripe function:

```
RGB turbstripe( point p, double w )
double t = (1 + sin( $k_1 z_p + \text{turbulence}(k_2 \mathbf{p})$ ))/w)/2
return t * s0 + (1 - t) * s1
```

Various values for k_1 and k_2 were used to generate Figure 11.7.

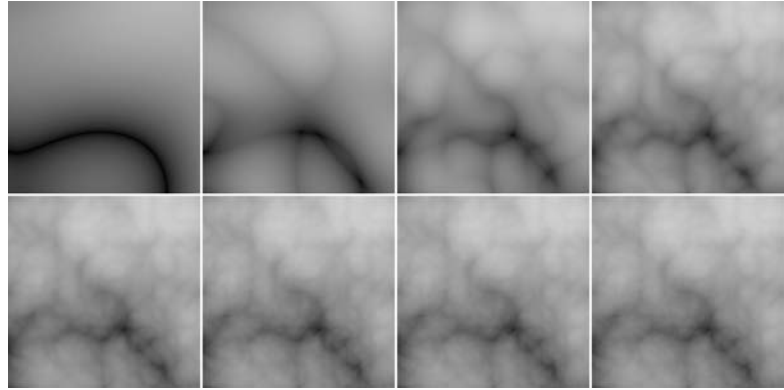


Figure 11.6. Turbulence function with (from top left to bottom right) one through eight terms in the summation.

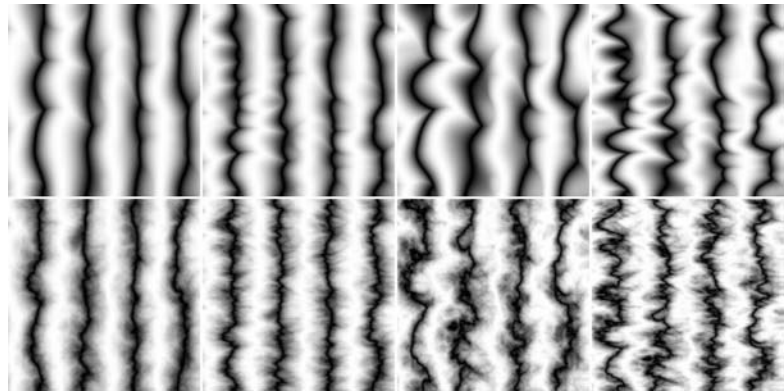


Figure 11.7. Various turbulent stripe textures with different k_1, k_2 . The top row has only the first term of the turbulence series.

11.2 2D Texture Mapping

For 2D texture mapping, we use a 2D coordinate, often called uv , which is used to create a reflectance $R(u, v)$. The key is to take an image and associate a (u, v) coordinate system on it so that it can, in turn, be associated with points on a 3D surface. For example, if the latitudes and longitudes on the world map are associated with a polar coordinate system on the sphere, we get a globe (Figure 11.8).

It is crucial that the coordinates on the image and the object match in “just the right way.” As a convention, the coordinate system on the image is set to be the unit square $(u, v) \in [0, 1]^2$. For (u, v) outside of this square, only the fractional parts of the coordinates are used resulting in a tiling of the plane (Figure 11.2).



Figure 11.8. A Miller cylindrical projection map world map and its placement on the sphere. The distortions in the texture map (i.e., Greenland being so large) exactly correspond to the shrinking that occurs when the map is applied to the sphere.

Note that the image has a different number of pixels horizontally and vertically, so the image pixels have a non-uniform aspect ratio in (u, v) space.

To map this $(u, v) \in [0, 1]^2$ image onto a sphere, we first compute the polar coordinates. Recall the spherical coordinate system described by Equation (2.25). For a sphere of radius R with center (c_x, c_y, c_z) , the parametric equation of the sphere is

$$\begin{aligned}x &= x_c + R \cos \phi \sin \theta, \\y &= y_c + R \sin \phi \sin \theta, \\z &= z_c + R \cos \theta.\end{aligned}$$

We can find (θ, ϕ) :

$$\begin{aligned}\theta &= \arccos\left(\frac{z - z_c}{R}\right), \\ \phi &= \arctan2(y - y_c, x - x_c),\end{aligned}$$

where $\arctan2(a, b)$ is the *atan2* of most math libraries which returns the arctangent of a/b . Because $(\theta, \phi) \in [0, \pi] \times [-\pi, \pi]$, we convert to (u, v) as follows, after first adding 2π to ϕ if it is negative:

$$\begin{aligned}u &= \frac{\phi}{2\pi}, \\ v &= \frac{\pi - \theta}{\pi}.\end{aligned}$$

This mapping is shown in Figure 11.8. There is a similar, although likely more complicated way, to generate coordinates for most 3D shapes.

11.3 Texture Mapping for Rasterized Triangles

For surfaces represented by triangle meshes, texture coordinates are defined by storing (u, v) texture coordinates at each vertex of the mesh (see Section 12.1). So, if a triangle is intersected at barycentric coordinates (β, γ) , you interpolate the (u, v) coordinates the same way you interpolate points. Recall that the point at barycentric coordinate (β, γ) is

$$\mathbf{p}(\beta, \gamma) = \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}).$$

A similar equation applies for (u, v) :

$$\begin{aligned} u(\beta, \gamma) &= u_a + \beta(u_b - u_a) + \gamma(u_c - u_a), \\ v(\beta, \gamma) &= v_a + \beta(v_b - v_a) + \gamma(v_c - v_a). \end{aligned}$$

Several ways a texture can be applied by changing the (u, v) at triangle vertices are shown in Figure 11.10. This sort of calibration texture map makes it easier to understand the texture coordinates of your objects during debugging (Figure 11.9).

We would like to get the same texture images whether we use a ray tracing program or a rasterization method, such as a z-buffer. There are some subtleties in achieving this with correct-looking perspective, but we can address this at the rasterization stage. The reason things are not straightforward is that just interpolating

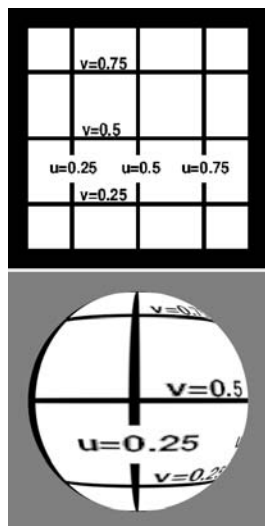


Figure 11.9. Top: a calibration texture map. Bottom: the sphere viewed along the y -axis.

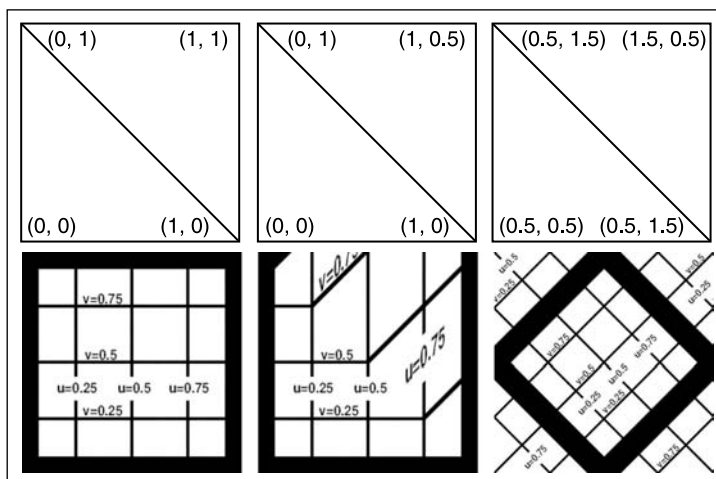


Figure 11.10. Various mesh textures obtained by changing (u, v) coordinates stored at vertices.



texture coordinates in screen space results in incorrect images, as shown for the grid texture shown in Figure 11.11. Because things in perspective get smaller as the distance to the viewer increases, the lines that are evenly spaced in 3D should compress in 2D image space. More careful interpolation of texture coordinates is needed to accomplish this.

11.3.1 Perspective Correct Textures

We can implement texture mapping on triangles by interpolating the (u, v) coordinates, modifying the rasterization method of Section 8.1.2, but this results in the problem shown at the right of Figure 11.11. A similar problem occurs for triangles if screen space barycentric coordinates are used as in the following rasterization code:

```

for all  $x$  do
  for all  $y$  do
    compute  $(\alpha, \beta, \gamma)$  for  $(x, y)$ 
    if  $\alpha \in (0, 1)$  and  $\beta \in (0, 1)$  and  $\gamma \in (0, 1)$  then
       $\mathbf{t} = \alpha \mathbf{t}_0 + \beta \mathbf{t}_1 + \gamma \mathbf{t}_2$ 
      drawpixel  $(x, y)$  with color texture( $\mathbf{t}$ ) for a solid texture
      or with texture( $\beta, \gamma$ ) for a 2D texture.

```

This code will generate images, but there is a problem. To unravel the basic problem, let's consider the progression from world space \mathbf{q} to homogeneous point \mathbf{r} to homogenized point \mathbf{s} :

$$\begin{bmatrix} x_q \\ y_q \\ z_q \\ 1 \end{bmatrix} \xrightarrow{\text{transform}} \begin{bmatrix} x_r \\ y_r \\ z_r \\ h_r \end{bmatrix} \xrightarrow{\text{homogenize}} \begin{bmatrix} x_r/h_r \\ y_r/h_r \\ z_r/h_r \\ 1 \end{bmatrix} \equiv \begin{bmatrix} x_s \\ y_s \\ z_s \\ 1 \end{bmatrix}.$$

If we use screen space, we are interpolating in \mathbf{s} . However, we would like to be interpolating in space \mathbf{q} or \mathbf{r} , where the homogeneous division has not yet non-linearly distorted the barycentric coordinates of the triangle.

The key observation is that $1/h_r$ is interpolated with no distortion. Likewise, so is u/h_r and v/h_r . In fact, so is k/h_r , where k is any quantity that varies linearly across the triangle. Recall from Section 7.4 that if we transform all points along the line segment between points \mathbf{q} and \mathbf{Q} and homogenize, we have

$$\mathbf{s} + \frac{h_R t}{h_r + t(h_R - h_r)} (\mathbf{S} - \mathbf{s}),$$

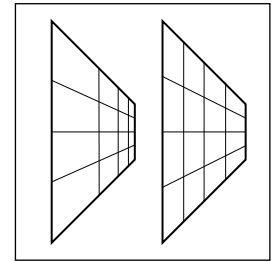


Figure 11.11. Left: correct perspective. Right: interpolation in screen space.

but if we linearly interpolate in the homogenized space we have

$$\mathbf{s} + a(\mathbf{S} - \mathbf{s}).$$

Although those lines sweep out the same points, typically $a \neq t$ for the same points on the line segment. However, if we interpolate $1/h$, we *do* get the same answer regardless of which space we interpolate in. To see this is true, confirm (Exercise 2):

$$\frac{1}{h_r} + \frac{h_R t}{h_r + t(h_R - h_r)} \left(\frac{1}{h_R} - \frac{1}{h_r} \right) = \frac{1}{h_r} + t \left(\frac{1}{h_R} - \frac{1}{h_r} \right) \quad (11.4)$$

This ability to interpolate $1/h$ linearly with no error in the transformed space allows us to correctly texture triangles. Perhaps the least confusing way to deal with this distortion is to compute the world space barycentric coordinates of the triangle (β_w, γ_w) in terms of screen space coordinates (β, γ) . We note that β_s/h and γ_s/h can be interpolated linearly in screen space. For example, at the screen space position associated with screen space barycentric coordinates (β, γ) , we can interpolate β_w/h without distortion. Because $\beta_w = 0$ at vertex 0 and vertex 2, and $\beta_w = 1$ at vertex 1, we have

$$\frac{\beta_s}{h} = \frac{0}{h_0} + \beta \left(\frac{1}{h_1} - \frac{0}{h_0} \right) + \gamma \left(\frac{0}{h_2} - \frac{0}{h_0} \right). \quad (11.5)$$

Because of all the zero terms, Equation (11.5) is fairly simple. However, to get β_w from it, we must know h . Because we know $1/h$ is linear in screen space, we have

$$\frac{1}{h} = \frac{1}{h_0} + \beta \left(\frac{1}{h_1} - \frac{1}{h_0} \right) + \gamma \left(\frac{1}{h_2} - \frac{1}{h_0} \right). \quad (11.6)$$

Dividing Equation (11.5) by Equation (11.6) gives

$$\beta_w = \frac{\frac{\beta}{h_1}}{\frac{1}{h_0} + \beta \left(\frac{1}{h_1} - \frac{1}{h_0} \right) + \gamma \left(\frac{1}{h_2} - \frac{1}{h_0} \right)}.$$

Multiplying numerator and denominator by $h_0 h_1 h_2$ and doing a similar set of manipulations for the analogous equations in γ_w gives

$$\begin{aligned} \beta_w &= \frac{h_0 h_2 \beta}{h_1 h_2 + h_2 \beta (h_0 - h_1) + h_1 \gamma (h_0 - h_2)}, \\ \gamma_w &= \frac{h_0 h_1 \gamma}{h_1 h_2 + h_2 \beta (h_0 - h_1) + h_1 \gamma (h_0 - h_2)}. \end{aligned} \quad (11.7)$$

Note that the two denominators are the same.



For triangles that use the perspective matrix from Chapter 7, recall that $w = z/n$ where z is the distance from the viewer perpendicular to the screen. Thus, for that matrix $1/z$ also varies linearly. We can use this fact to modify our scan-conversion code for three points $\mathbf{t}_i = (x_i, y_i, z_i, h_i)$ that have been passed through the viewing matrices, but have not been homogenized:

```

Compute bounds for  $x = x_i/h_i$  and  $y = y_i/h_i$ 
for all  $x$  do
  for all  $y$  do
    compute  $(\alpha, \beta, \gamma)$  for  $(x, y)$ 
    if  $(\alpha \in [0, 1] \text{ and } \beta \in [0, 1] \text{ and } \gamma \in [0, 1])$  then
       $d = h_1 h_2 + h_2 \beta (h_0 - h_1) + h_1 \gamma (h_0 - h_2)$ 
       $\beta_w = h_0 h_2 \beta / d$ 
       $\gamma_w = h_0 h_1 \gamma / d$ 
       $\alpha_w = 1 - \beta_w - \gamma_w$ 
       $u = \alpha_w u_0 + \beta_w u_1 + \gamma_w u_2$ 
       $v = \alpha_w v_0 + \beta_w v_1 + \gamma_w v_2$ 
      drawpixel  $(x, y)$  with color texture $(u, v)$ 

```

For solid textures, just recall that by the definition of barycentric coordinates

$$\mathbf{p} = (1 - \beta_w - \gamma_w)\mathbf{p}_0 + \beta_w\mathbf{p}_1 + \gamma_w\mathbf{p}_2,$$

where \mathbf{p}_i are the world space vertices. Then, just call a solid texture routine for point \mathbf{p} .

11.4 Bump Textures

Although we have only discussed changing reflectance using texture, you can also change the surface normal to give an illusion of fine-scale geometry on the surface. We can apply a *bump map* that perturbs the surface normal (J. F. Blinn, 1978).

One way to do this is:

```

vector3  $n$  = surfaceNormal( $x$ )
 $n$  +=  $k_1 * \text{vectorTurbulence}(k_2 * x)$ 
return  $t * s_0 + (1 - t) * s_1$ 

```

This is shown in Figure 11.12.

To implement *vectorTurbulence*, we first need *vectorNoise* which produces a simple spatially-varying 3D vector:

$$n_v(x, y, z) = \sum_{i=\lfloor x \rfloor}^{\lfloor x \rfloor+1} \sum_{j=\lfloor y \rfloor}^{\lfloor y \rfloor+1} \sum_{k=\lfloor z \rfloor}^{\lfloor z \rfloor+1} \Gamma_{ijk} \omega(x) \omega(y) \omega(z).$$

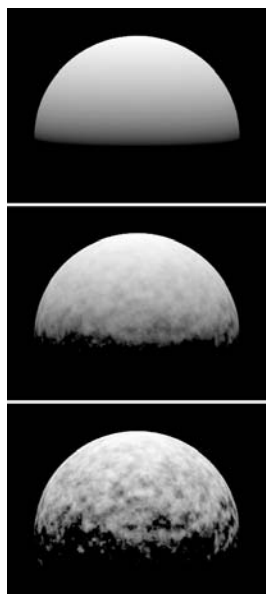


Figure 11.12. Vector turbulence on a sphere of radius 1.6. Lighting directly from above. Top: $k_1 = 0$. Middle: $k_1 = 0.08$, $k_2 = 8$. Bottom: $k_1 = 0.24$, $k_2 = 8$.

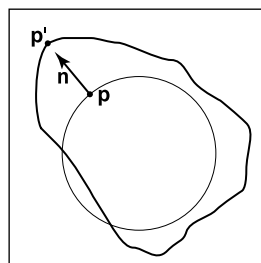


Figure 11.13. The points \mathbf{p} on the circle are each displaced in the direction of \mathbf{n} by the function $f(\mathbf{p})$. If f is continuous, then the resulting points \mathbf{p}' form a continuous surface.

Then, *vectorTurbulence* is a direct analog of turbulence: sum a series of scaled versions of *vectorNoise*.

11.5 Displacement Mapping

One problem with Figure 11.12 is that the bumps neither cast shadows nor affect the silhouette of the object. These limitations occur because we are not really changing any geometry. If we want more realism, we can apply a *displacement map* (Cook et al., 1987). A displacement map actually changes the geometry using a texture. A common simplification is that the displacement will be in the direction of the surface normal.

If we take all points \mathbf{p} on a surface, with associated surface normal vectors \mathbf{n} , then we can make a new surface using a 3D texture $d(\mathbf{p})$:

$$\mathbf{p}' = \mathbf{p} + f(\mathbf{p})\mathbf{n}.$$

This concept is shown in Figure 11.13.

Displacement mapping is straightforward to implement in a z-buffer code by storing the surface to be displaced as a fine mesh of many triangles. Each vertex in the mesh can then be displaced along the normal vector direction. This results in large models, but it is quite robust.

11.6 Environment Maps

Often we would like to have a texture-mapped background and for objects to have specular reflections of that background. This can be accomplished using *environment maps* (J. F. Blinn, 1976). An environment map can be implemented as a background function that takes in a viewing direction \mathbf{b} and returns a RGB color from a texture map. There are many ways to store environment maps. For example, we can use a spherical table indexed by spherical coordinates. In this section, we will instead describe a cube-based table with six square texture maps, often called a *cube map*.

The basic idea of a cube map is that we have an infinitely large cube with a texture on each face. Because the cube is large, the origin of a ray does not change what the ray “sees.” This is equivalent to an arbitrarily-sized cube that is queried by a ray whose origin is at the Cartesian origin. As an example of how a given direction \mathbf{b} is converted to (u, v) coordinates, consider the right face of

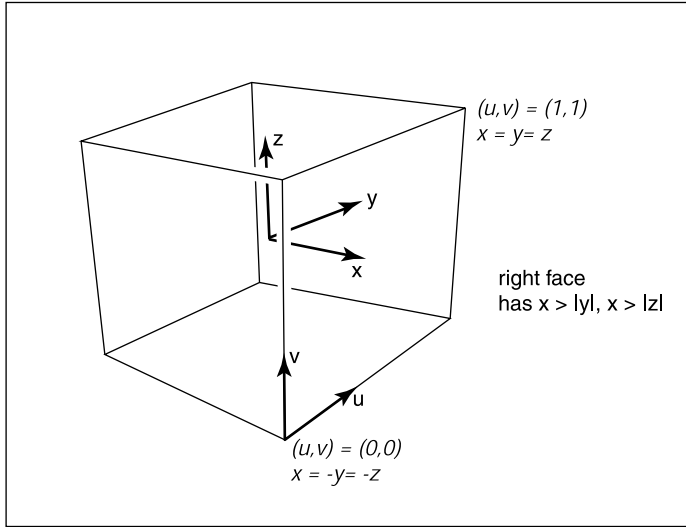


Figure 11.14. The cube map has six axis-aligned textures that store the background. The right face contains a single texture.

Figure 11.14. Here we have x_b as the maximum magnitude component. In that case, we can compute (u, v) for that texture to be

$$u = \frac{y + x}{2x},$$

$$v = \frac{z + x}{2x}.$$

There are analogous formulas for the other five faces.

So for any reflection ray $\mathbf{a} + t\mathbf{b}$ we return $\text{cubemap}(\mathbf{b})$ for the background color. In a z-buffer implementation, we need to perform this calculation on a pixel-by-pixel basis. If at a given pixel we know the viewing direction \mathbf{c} and the surface normal vector \mathbf{n} , we can compute the reflected direction \mathbf{b} (Figure 11.15). We can do this by modifying Equation (10.6) to get

$$\mathbf{b} = -\mathbf{c} + \frac{2(\mathbf{c} \cdot \mathbf{n})\mathbf{n}}{\|\mathbf{c}\|^2}. \quad (11.8)$$

Here the denominator of the fraction accounts for the fact that \mathbf{c} may not be a unit vector. Because we need to know \mathbf{b} at each pixel, we can either compute \mathbf{b} at each triangle vertex and interpolate \mathbf{b} in a perspective correct manner, or we can interpolate \mathbf{n} and compute \mathbf{b} for each pixel. This will allow us to call $\text{cubemap}(\mathbf{b})$ at each pixel.

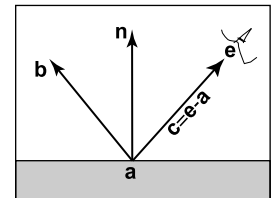


Figure 11.15. The vector \mathbf{b} is the reflection of vector \mathbf{c} with respect to the surface normal \mathbf{n} .

11.7 Shadow Maps

The basic observation to be made about a shadow map is that if we rendered the scene using the location of a light source as the eye, the visible surfaces would all be lit, and the hidden surfaces would all be in shadow. This can be used to determine whether a point being rasterized is in shadow (L. Williams, 1978). First, we rasterize the scene from the point of view of the light source using matrix \mathbf{M}_s . This matrix is just the same as the full transform matrix \mathbf{M} used for viewing in Section 7.3, but it uses the light position for the eye and the light's main direction for the view-plane normal.

Recall that the matrix \mathbf{M} takes an (x, y, z) in world coordinates and converts it to an (x', y', z') in relation to the screen. While rasterizing in a perspective correct manner, we can get the (x, y, z) that is seen through the center of each pixel. If we also rasterize that point using \mathbf{M}_s and round the resulting x - and y -coordinates, we will get

$$(i, j, \text{depth}).$$

We can compare this depth with the z -value in the shadow depth map at pixel (i, j) . If it is the same, then the point is lit, and otherwise it is in shadow. Because of computational inaccuracies, we should actually test whether the points are the same to within a small constant.

Because we typically don't want the light to only be within a square window, often a *spot light* is used. This attenuates the value of the light source based on closeness to the sides of the shadow buffer. For example, if the shadow buffer is $n \times n$ pixels, then for pixel (i, j) in the shadow buffer, we can apply the attenuation coefficient based on the fractional radius r :

$$r = \sqrt{\left(\frac{2i - n}{n}\right)^2 + \left(\frac{2j - n}{n}\right)^2}.$$

Any radially decreasing function will then give a spot-like look.

Frequently Asked Questions

- How do I implement displacement mapping in ray tracing?

There is no ideal way to do it. Generating all the triangles and caching the geometry when necessary will prevent memory overload (Pharr & Hanrahan, 1996; Pharr et al., 1997). Trying to intersect the displaced surface directly is possible



when the displacement function is restricted (Patterson et al., 1991; Heidrich & Seidel, 1998; Smits et al., 2000).

- Why don't my images with textures look realistic?

Humans are good at seeing small imperfections in surfaces. Geometric imperfections are typically absent in computer-generated images that use texture maps for details, so they look “too smooth.”

- My textured animations look bad when there are many texels visible inside a pixel. What should I do?

The problem is that the texture resolution is too high for that image. We would like a smaller down-sampled version of the texture. However, if we move closer, such a down-sampled texture would look too blurry. What we really need is to be able to dynamically choose the texture resolution based on viewing conditions so that about one texel is visible through each pixel. A common way to do that is to use *MIP-mapping* (L. Williams, 1983). That technique establishes a multi-resolution set of textures and chooses one of the textures for each polygon or pixel. Typically the resolutions vary by a factor of two, e.g., 512^2 , 256^2 , 128^2 , etc.

Notes

The discussion of perspective-correct textures is based on *Fast Shadows and Lighting Effects Using Texture Mapping* (Segal et al., 1992) and on *3D Game Engine Design* (Eberly, 2000).

Exercises

1. Find several ways to implement an infinite 2D checkerboard using surface and solid techniques. Which is best?
2. Verify that Equation (11.4) is a valid equality using brute-force algebra.
3. How could you implement solid texturing by using the z-buffer depth and a matrix transform?