$\hat{\mathbf{q}}$, can be converted into a matrix $\mathbf{M}^q$, as expressed in Equation 4.45 [1633, 1634]:

$$\mathbf{M}^q = \begin{pmatrix} 1 - s(q_y^2 + q_z^2) & s(q_x q_y - q_w q_z) & s(q_x q_z + q_w q_y) & 0 \\ s(q_x q_y + q_w q_z) & 1 - s(q_x^2 + q_z^2) & s(q_y q_z - q_w q_x) & 0 \\ s(q_x q_z - q_w q_y) & s(q_y q_z + q_w q_x) & 1 - s(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.45}$$

Here, the scalar is $s = 2/(n(\hat{\mathbf{q}}))^2$. For unit quaternions, this simplifies to

$$\mathbf{M}^q = \begin{pmatrix} 1 - 2(q_y^2 + q_z^2) & 2(q_x q_y - q_w q_z) & 2(q_x q_z + q_w q_y) & 0 \\ 2(q_x q_y + q_w q_z) & 1 - 2(q_x^2 + q_z^2) & 2(q_y q_z - q_w q_x) & 0 \\ 2(q_x q_z - q_w q_y) & 2(q_y q_z + q_w q_x) & 1 - 2(q_x^2 + q_y^2) & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.46}$$

Once the quaternion is constructed, *no* trigonometric functions need to be computed, so the conversion process is efficient in practice.

The reverse conversion, from an orthogonal matrix, $\mathbf{M}^q$, into a unit quaternion, $\hat{\mathbf{q}}$, is a bit more involved. Key to this process are the following differences made from the matrix in Equation 4.46:

$$\begin{aligned} m_{21}^q - m_{12}^q &= 4q_w q_x, \\ m_{02}^q - m_{20}^q &= 4q_w q_y, \\ m_{10}^q - m_{01}^q &= 4q_w q_z. \end{aligned} \tag{4.47}$$

The implication of these equations is that if q_w is known, the values of the vector $\mathbf{v}_q$ can be computed, and thus $\hat{\mathbf{q}}$ derived. The trace of $\mathbf{M}^q$ is calculated by

$$\begin{aligned} \operatorname{tr}(\mathbf{M}^q) &= 4 - 2s(q_x^2 + q_y^2 + q_z^2) = 4\left(1 - \frac{q_x^2 + q_y^2 + q_z^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2}\right) \\ &= \frac{4q_w^2}{q_x^2 + q_y^2 + q_z^2 + q_w^2} = \frac{4q_w^2}{(n(\hat{\mathbf{q}}))^2}. \end{aligned} \tag{4.48}$$

This result yields the following conversion for a unit quaternion:

$$\begin{aligned} q_w &= \frac{1}{2}\sqrt{\operatorname{tr}(\mathbf{M}^q)}, & q_x &= \frac{m_{21}^q - m_{12}^q}{4q_w}, \\ q_y &= \frac{m_{02}^q - m_{20}^q}{4q_w}, & q_z &= \frac{m_{10}^q - m_{01}^q}{4q_w}. \end{aligned} \tag{4.49}$$

To have a numerically stable routine [1634], divisions by small numbers should be avoided. Therefore, first set $t = q_w^2 - q_x^2 - q_y^2 - q_z^2$, from which it follows that

$$\begin{aligned} m_{00} &= t + 2q_x^2, \\ m_{11} &= t + 2q_y^2, \\ m_{22} &= t + 2q_z^2, \\ u &= m_{00} + m_{11} + m_{22} = t + 2q_w^2, \end{aligned} \tag{4.50}$$

which in turn implies that the largest of m_{00}, m_{11}, m_{22}, and u determine which of q_x, q_y, q_z, and q_w is largest. If q_w is largest, then Equation 4.49 is used to derive the quaternion. Otherwise, we note that the following holds:

$$\begin{aligned} 4q_x^2 &= +m_{00} - m_{11} - m_{22} + m_{33}, \\ 4q_y^2 &= -m_{00} + m_{11} - m_{22} + m_{33}, \\ 4q_z^2 &= -m_{00} - m_{11} + m_{22} + m_{33}, \\ 4q_w^2 &= \mathrm{tr}(\mathbf{M}^q). \end{aligned} \tag{4.51}$$

The appropriate equation of the ones above is then used to compute the largest of q_x, q_y, and q_z, after which Equation 4.47 is used to calculate the remaining components of $\hat{\mathbf{q}}$. Schüler [1588] presents a variant that is branchless but uses four square roots instead.

Spherical Linear Interpolation

Spherical linear interpolation is an operation that, given two unit quaternions, $\hat{\mathbf{q}}$ and $\hat{\mathbf{r}}$, and a parameter $t \in [0, 1]$, computes an interpolated quaternion. This is useful for animating objects, for example. It is not as useful for interpolating camera orientations, as the camera's "up" vector can become tilted during the interpolation, usually a disturbing effect.

The algebraic form of this operation is expressed by the composite quaternion, $\hat{\mathbf{s}}$, below:

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = (\hat{\mathbf{r}}\hat{\mathbf{q}}^{-1})^t\hat{\mathbf{q}}. \tag{4.52}$$

However, for software implementations, the following form, where *slerp* stands for spherical linear interpolation, is much more appropriate:

$$\hat{\mathbf{s}}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \mathtt{slerp}(\hat{\mathbf{q}}, \hat{\mathbf{r}}, t) = \frac{\sin(\phi(1-t))}{\sin\phi}\hat{\mathbf{q}} + \frac{\sin(\phi t)}{\sin\phi}\hat{\mathbf{r}}. \tag{4.53}$$

To compute ϕ, which is needed in this equation, the following fact can be used: $\cos\phi = q_x r_x + q_y r_y + q_z r_z + q_w r_w$ [325]. For $t \in [0, 1]$, the slerp function computes (unique[2]) interpolated quaternions that together constitute the shortest arc on a four-dimensional unit sphere from $\hat{\mathbf{q}}$ ($t = 0$) to $\hat{\mathbf{r}}$ ($t = 1$). The arc is located on the circle that is formed from the intersection between the plane given by $\hat{\mathbf{q}}$, $\hat{\mathbf{r}}$, and the origin, and the four-dimensional unit sphere. This is illustrated in Figure 4.10. The computed rotation quaternion rotates around a fixed axis at constant speed. A curve such as this, that has constant speed and thus zero acceleration, is called a *geodesic* curve [229]. A *great circle* on the sphere is generated as the intersection of a plane through the origin and the sphere, and part of such a circle is called a *great arc*.

The slerp function is perfectly suited for interpolating between two orientations and it behaves well (fixed axis, constant speed). This is not the case with when

[2] If and only if $\hat{\mathbf{q}}$ and $\hat{\mathbf{r}}$ are not opposite.

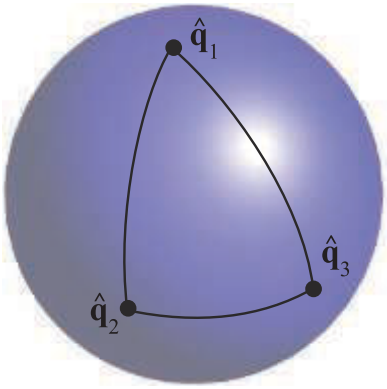

Figure 4.10. Unit quaternions are represented as points on the unit sphere. The function slerp is used to interpolate between the quaternions, and the interpolated path is a great arc on the sphere. Note that interpolating from $\hat{\mathbf{q}}_1$ to $\hat{\mathbf{q}}_2$ and interpolating from $\hat{\mathbf{q}}_1$ to $\hat{\mathbf{q}}_3$ to $\hat{\mathbf{q}}_2$ are not the same thing, even though they arrive at the same orientation.

interpolating using several Euler angles. In practice, computing a slerp directly is an expensive operation involving calling trigonometric functions. Malyshau [1114] discusses integrating quaternions into the rendering pipeline. He notes that the error in the orientation of a triangle is a maximum of 4 degrees for a 90 degree angle when, instead of using slerp, simply normalizing the quaternion in the pixel shader. This error rate can be acceptable when rasterizing a triangle. Li [1039, 1040] provides much faster incremental methods to compute slerps that do not sacrifice any accuracy. Eberly [406] presents a fast technique for computing slerps using just additions and multiplications.

When more than two orientations, say $\hat{\mathbf{q}}_0, \hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_{n-1}$, are available, and we want to interpolate from $\hat{\mathbf{q}}_0$ to $\hat{\mathbf{q}}_1$ to $\hat{\mathbf{q}}_2$, and so on until $\hat{\mathbf{q}}_{n-1}$, slerp could be used in a straightforward fashion. Now, when we approach, say, $\hat{\mathbf{q}}_i$, we would use $\hat{\mathbf{q}}_{i-1}$ and $\hat{\mathbf{q}}_i$ as arguments to slerp. After passing through $\hat{\mathbf{q}}_i$, we would then use $\hat{\mathbf{q}}_i$ and $\hat{\mathbf{q}}_{i+1}$ as arguments to slerp. This will cause sudden jerks to appear in the orientation interpolation, which can be seen in Figure 4.10. This is similar to what happens when points are linearly interpolated; see the upper right part of Figure 17.3 on page 720. Some readers may wish to revisit the following paragraph after reading about splines in Chapter 17.

A better way to interpolate is to use some sort of spline. We introduce quaternions $\hat{\mathbf{a}}_i$ and $\hat{\mathbf{a}}_{i+1}$ between $\hat{\mathbf{q}}_i$ and $\hat{\mathbf{q}}_{i+1}$. Spherical cubic interpolation can be defined within the set of quaternions $\hat{\mathbf{q}}_i$, $\hat{\mathbf{a}}_i$, $\hat{\mathbf{a}}_{i+1}$, and $\hat{\mathbf{q}}_{i+1}$. Surprisingly, these extra quaternions are computed as shown below [404][3]:

$$\hat{\mathbf{a}}_i = \hat{\mathbf{q}}_i \exp\left[-\frac{\log(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i-1}) + \log(\hat{\mathbf{q}}_i^{-1}\hat{\mathbf{q}}_{i+1})}{4}\right]. \tag{4.54}$$

[3]Shoemake [1633] gives another derivation.

The $\hat{\mathbf{q}}_i$, and $\hat{\mathbf{a}}_i$ will be used to spherically interpolate the quaternions using a smooth cubic spline, as shown in Equation 4.55:

$$\begin{aligned}&\texttt{squad}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, \hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t) = \\ &\quad \texttt{slerp}(\texttt{slerp}(\hat{\mathbf{q}}_i, \hat{\mathbf{q}}_{i+1}, t), \texttt{slerp}(\hat{\mathbf{a}}_i, \hat{\mathbf{a}}_{i+1}, t), 2t(1-t)).\end{aligned} \tag{4.55}$$

As can be seen above, the `squad` function is constructed from repeated spherical interpolation using slerp (Section 17.1.1 for information on repeated linear interpolation for points). The interpolation will pass through the initial orientations $\hat{\mathbf{q}}_i$, $i \in [0, \ldots, n-1]$, but not through $\hat{\mathbf{a}}_i$—these are used to indicate the tangent orientations at the initial orientations.

Rotation from One Vector to Another

A common operation is transforming from one direction $\mathbf{s}$ to another direction $\mathbf{t}$ via the shortest path possible. The mathematics of quaternions simplifies this procedure greatly, and shows the close relationship the quaternion has with this representation. First, normalize $\mathbf{s}$ and $\mathbf{t}$. Then compute the unit rotation axis, called $\mathbf{u}$, which is computed as $\mathbf{u} = (\mathbf{s} \times \mathbf{t})/||\mathbf{s} \times \mathbf{t}||$. Next, $e = \mathbf{s} \cdot \mathbf{t} = \cos(2\phi)$ and $||\mathbf{s} \times \mathbf{t}|| = \sin(2\phi)$, where 2ϕ is the angle between $\mathbf{s}$ and $\mathbf{t}$. The quaternion that represents the rotation from $\mathbf{s}$ to $\mathbf{t}$ is then $\hat{\mathbf{q}} = (\sin\phi\mathbf{u},\ \cos\phi)$. In fact, simplifying $\hat{\mathbf{q}} = (\frac{\sin\phi}{\sin 2\phi}(\mathbf{s} \times \mathbf{t}),\ \cos\phi)$, using the half-angle relations and the trigonometric identity, gives [1197]

$$\hat{\mathbf{q}} = (\mathbf{q}_v, q_w) = \left(\frac{1}{\sqrt{2(1+e)}}(\mathbf{s} \times \mathbf{t}), \frac{\sqrt{2(1+e)}}{2}\right). \tag{4.56}$$

Directly generating the quaternion in this fashion (versus normalizing the cross product $\mathbf{s} \times \mathbf{t}$) avoids numerical instability when $\mathbf{s}$ and $\mathbf{t}$ point in nearly the same direction [1197]. Stability problems appear for both methods when $\mathbf{s}$ and $\mathbf{t}$ point in opposite directions, as a division by zero occurs. When this special case is detected, any axis of rotation perpendicular to $\mathbf{s}$ can be used to rotate to $\mathbf{t}$.

Sometimes we need the matrix representation of a rotation from $\mathbf{s}$ to $\mathbf{t}$. After some algebraic and trigonometric simplification of Equation 4.46, the rotation matrix becomes [1233]

$$\mathbf{R}(\mathbf{s}, \mathbf{t}) = \begin{pmatrix} e + hv_x^2 & hv_xv_y - v_z & hv_xv_z + v_y & 0 \\ hv_xv_y + v_z & e + hv_y^2 & hv_yv_z - v_x & 0 \\ hv_xv_z - v_y & hv_yv_z + v_x & e + hv_z^2 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.57}$$

In this equation, we have used the following intermediate calculations:

$$\begin{aligned}\mathbf{v} &= \mathbf{s} \times \mathbf{t}, \\ e &= \cos(2\phi) = \mathbf{s} \cdot \mathbf{t}, \\ h &= \frac{1 - \cos(2\phi)}{\sin^2(2\phi)} = \frac{1-e}{\mathbf{v} \cdot \mathbf{v}} = \frac{1}{1+e}.\end{aligned} \tag{4.58}$$

As can be seen, all square roots and trigonometric functions have disappeared due to the simplifications, and so this is an efficient way to create the matrix. Note that the structure of Equation 4.57 is like that of Equation 4.30, and note how this latter form does not need trigonometric functions.

Note that care must be taken when $\mathbf{s}$ and $\mathbf{t}$ are parallel or near parallel, because then $||\mathbf{s} \times \mathbf{t}|| \approx 0$. If $\phi \approx 0$, then we can return the identity matrix. However, if $2\phi \approx \pi$, then we can rotate π radians around *any* axis. This axis can be found as the cross product between $\mathbf{s}$ and any other vector that is not parallel to $\mathbf{s}$ (Section 4.2.4). Möller and Hughes use Householder matrices to handle this special case in a different way [1233].

4.4 Vertex Blending

Imagine that an arm of a digital character is animated using two parts, a forearm and an upper arm, as shown to the left in Figure 4.11. This model could be animated using rigid-body transforms (Section 4.1.6). However, then the joint between these two parts will not resemble a real elbow. This is because two separate objects are used, and therefore, the joint consists of overlapping parts from these two separate objects. Clearly, it would be better to use just one single object. However, static model parts do not address the problem of making the joint flexible.

Vertex blending is one popular solution to this problem [1037, 1903]. This technique has several other names, such as *linear-blend skinning*, *enveloping*, or *skeleton-subspace*

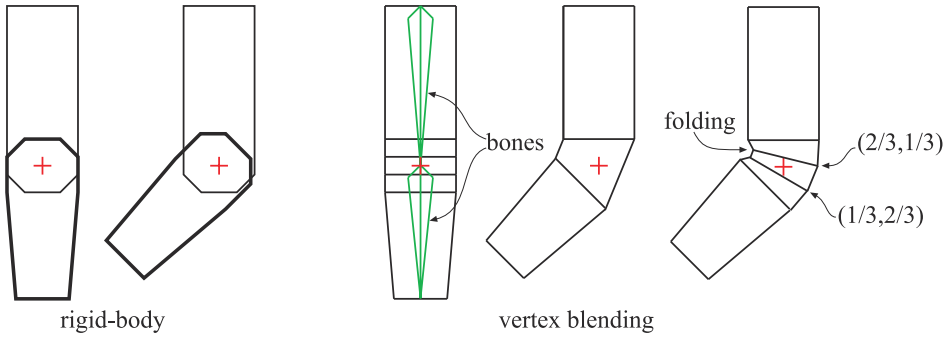

Figure 4.11. An arm consisting of a forearm and an upper arm is animated using rigid-body transforms of two separate objects to the left. The elbow does not appear realistic. To the right, vertex blending is used on one single object. The next-to-rightmost arm illustrates what happens when a simple skin directly joins the two parts to cover the elbow. The rightmost arm illustrates what happens when vertex blending is used, and some vertices are blended with different weights: $(2/3, 1/3)$ means that the vertex weighs the transform from the upper arm by $2/3$ and from the forearm by $1/3$. This figure also shows a drawback of vertex blending in the rightmost illustration. Here, folding in the inner part of the elbow is visible. Better results can be achieved with more bones, and with more carefully selected weights.

deformation. While the exact origin of the algorithm presented here is unclear, defining bones and having skin react to changes is an old concept in computer animation [1100]. In its simplest form, the forearm and the upper arm are animated separately as before, but at the joint, the two parts are connected through an elastic "skin." So, this elastic part will have one set of vertices that are transformed by the forearm matrix and another set that are transformed by the matrix of the upper arm. This results in triangles whose vertices may be transformed by different matrices, in contrast to using a single matrix per triangle. See Figure 4.11.

By taking this one step further, one can allow a single vertex to be transformed by several different matrices, with the resulting locations weighted and blended together. This is done by having a skeleton of bones for the animated object, where each bone's transform may influence each vertex by a user-defined weight. Since the entire arm may be "elastic," i.e., all vertices may be affected by more than one matrix, the entire mesh is often called a *skin* (over the bones). See Figure 4.12. Many commercial modeling systems have this same sort of skeleton-bone modeling feature. Despite their name, bones do not need to necessarily be rigid. For example, Mohr and Gleicher [1230] present the idea of adding additional joints to enable effects such as muscle bulge. James and Twigg [813] discuss animation skinning using bones that can squash and stretch.

Mathematically, this is expressed in Equation 4.59, where $\mathbf{p}$ is the original vertex, and $\mathbf{u}(t)$ is the transformed vertex whose position depends on time t:

$$\mathbf{u}(t) = \sum_{i=0}^{n-1} w_i \mathbf{B}_i(t) \mathbf{M}_i^{-1} \mathbf{p}, \quad \text{where} \quad \sum_{i=0}^{n-1} w_i = 1, \quad w_i \geq 0. \tag{4.59}$$

There are n bones influencing the position of $\mathbf{p}$, which is expressed in world coordinates. The value w_i is the weight of bone i for vertex $\mathbf{p}$. The matrix $\mathbf{M}_i$ transforms from the initial bone's coordinate system to world coordinates. Typically a bone has its controlling joint at the origin of its coordinate system. For example, a forearm bone would move its elbow joint to the origin, with an animated rotation matrix moving this part of the arm around the joint. The $\mathbf{B}_i(t)$ matrix is the ith bone's world transform that changes with time to animate the object, and is typically a concatenation of several matrices, such as the hierarchy of previous bone transforms and the local animation matrix.

One method of maintaining and updating the $\mathbf{B}_i(t)$ matrix animation functions is discussed in depth by Woodland [1903]. Each bone transforms a vertex to a location with respect to its own frame of reference, and the final location is interpolated from the set of computed points. The matrix $\mathbf{M}_i$ is not explicitly shown in some discussions of skinning, but rather is considered as being a part of $\mathbf{B}_i(t)$. We present it here as it is a useful matrix that is almost always a part of the matrix concatenation process.

In practice, the matrices $\mathbf{B}_i(t)$ and $\mathbf{M}_i^{-1}$ are concatenated for each bone for each frame of animation, and each resulting matrix is used to transform the vertices. The vertex $\mathbf{p}$ is transformed by the different bones' concatenated matrices, and then

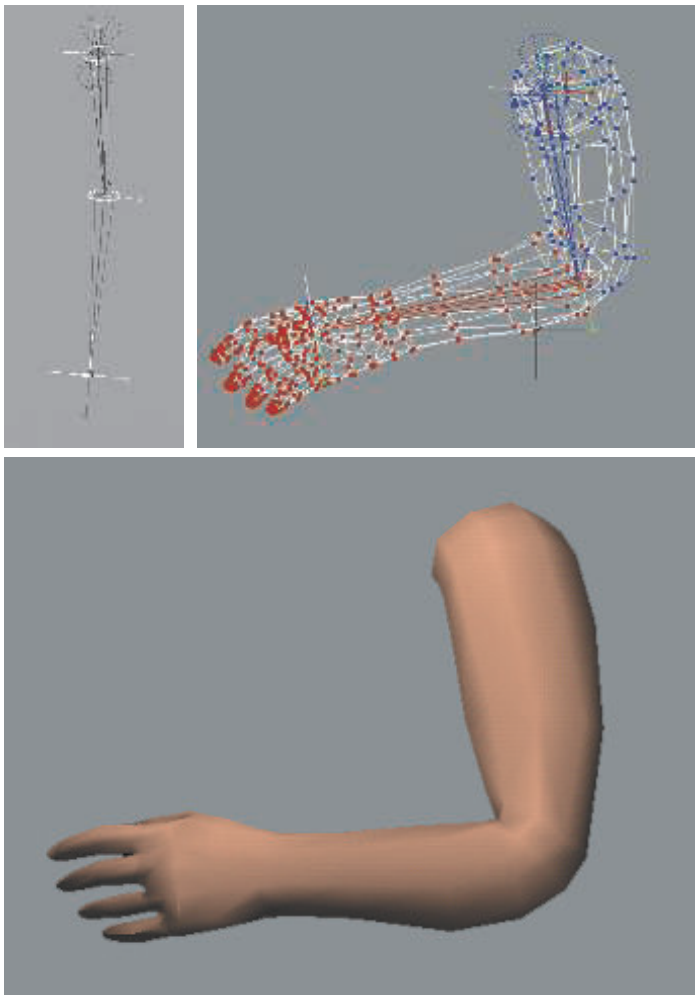

Figure 4.12. A real example of vertex blending. The top left image shows the two bones of an arm, in an extended position. On the top right, the mesh is shown, with color denoting which bone owns each vertex. Bottom: the shaded mesh of the arm in a slightly different position. *(Images courtesy of Jeff Lander [968].)*

blended using the weights w_i—thus the name *vertex blending*. The weights are non-negative and sum to one, so what is occurring is that the vertex is transformed to a few positions and then interpolated among them. As such, the transformed point $\mathbf{u}$ will lie in the convex hull of the set of points $\mathbf{B}_i(t)\mathbf{M}_i^{-1}\mathbf{p}$, for all $i = 0 \ldots n-1$ (fixed t). The normals usually can also be transformed using Equation 4.59. Depending on the transforms used (e.g., if a bone is stretched or squished a considerable amount),

the transpose of the inverse of the $\mathbf{B}_i(t)\mathbf{M}_i^{-1}$ may be needed instead, as discussed in Section 4.1.7.

Vertex blending is well suited for use on the GPU. The set of vertices in the mesh can be placed in a static buffer that is sent to the GPU one time and reused. In each frame, only the bone matrices change, with a vertex shader computing their effect on the stored mesh. In this way, the amount of data processed on and transferred from the CPU is minimized, allowing the GPU to efficiently render the mesh. It is easiest if the model's whole set of bone matrices can be used together; otherwise the model must be split up and some bones replicated. Alternately the bone transforms can be stored in textures that the vertices access, which avoids hitting register storage limits. Each transform can be stored in just two textures by using quaternions to represent rotation [1639]. If available, unordered access view storage allows the reuse of skinning results [146].

It is possible to specify sets of weights that are outside the range $[0, 1]$ or do not sum to one. However, this makes sense only if some other blending algorithm, such as *morph targets* (Section 4.5), is being used.

One drawback of basic vertex blending is that unwanted folding, twisting, and self-intersection can occur [1037]. See Figure 4.13. A better solution is to use *dual quaternions* [872, 873]. This technique to perform skinning helps to preserve the rigidity of the original transforms, so avoiding "candy wrapper" twists in limbs. Computation is less than 1.5× the cost for linear skin blending and the results are good, which has led to rapid adoption of this technique. However, dual quaternion skinning can lead to bulging effects, and Le and Hodgins [1001] present center-of-rotation skinning as a better alternative. They rely on the assumptions that local transforms should be rigid-body and that vertices with similar weights, w_i, should have similar transforms. Centers of rotation are precomputed for each vertex while orthogonal (rigid body) constraints are imposed to prevent elbow collapse and candy wrapper twist artifacts. At runtime, the algorithm is similar to linear blend skinning in that a GPU implementation performs linear blend skinning on the centers of rotation followed by a quaternion blending step.

4.5 Morphing

Morphing from one three-dimensional model to another can be useful when performing animations [28, 883, 1000, 1005]. Imagine that one model is displayed at time t_0 and we wish it to change into another model by time t_1. For all times between t_0 and t_1, a continuous "mixed" model is obtained, using some kind of interpolation. An example of morphing is shown in Figure 4.14.

Morphing involves solving two major problems, namely, the *vertex correspondence* problem and the *interpolation* problem. Given two arbitrary models, which may have different topologies, different number of vertices, and different mesh connectivity, one usually has to begin by setting up these vertex correspondences. This is a difficult

Figure 4.13. The left side shows problems at the joints when using linear blend skinning. On the right, blending using dual quaternions improves the appearance. *(Images courtesy of Ladislav Kavan et al., model by Paul Steed [1693].)*

problem, and there has been much research in this field. We refer the interested reader to Alexa's survey [28].

However, if there already is a one-to-one vertex correspondence between the two models, then interpolation can be done on a per-vertex basis. That is, for each vertex in the first model, there must exist only one vertex in the second model, and vice versa. This makes interpolation an easy task. For example, linear interpolation can be used directly on the vertices (Section 17.1 for other ways of doing interpolation). To

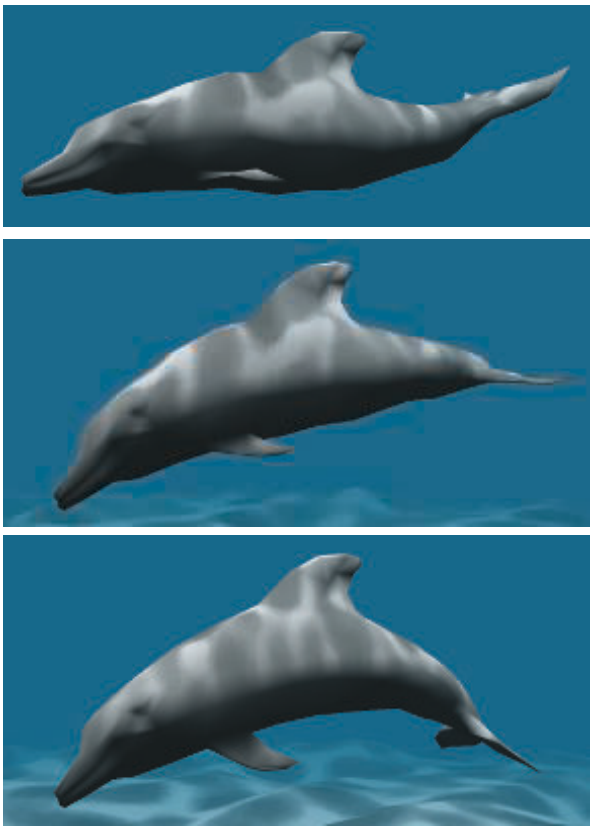

Figure 4.14. Vertex morphing. Two locations and normals are defined for every vertex. In each frame, the intermediate location and normal are linearly interpolated by the vertex shader. *(Images courtesy of NVIDIA Corporation.)*

compute a morphed vertex for time $t \in [t_0, t_1]$, we first compute $s = (t - t_0)/(t_1 - t_0)$, and then the linear vertex blend,

$$\mathbf{m} = (1 - s)\mathbf{p}_0 + s\mathbf{p}_1, \tag{4.60}$$

where $\mathbf{p}_0$ and $\mathbf{p}_1$ correspond to the same vertex but at different times, t_0 and t_1.

A variant of morphing where the user has more intuitive control is referred to as *morph targets* or *blend shapes* [907]. The basic idea can be explained using Figure 4.15.

We start out with a neutral model, which in this case is a face. Let us denote this model by $\mathcal{N}$. In addition, we also have a set of different face poses. In the example illustration, there is only one pose, which is a smiling face. In general, we can allow $k \geq 1$ different poses, which are denoted $\mathcal{P}_i$, $i \in [1, \ldots, k]$. As a preprocess, the

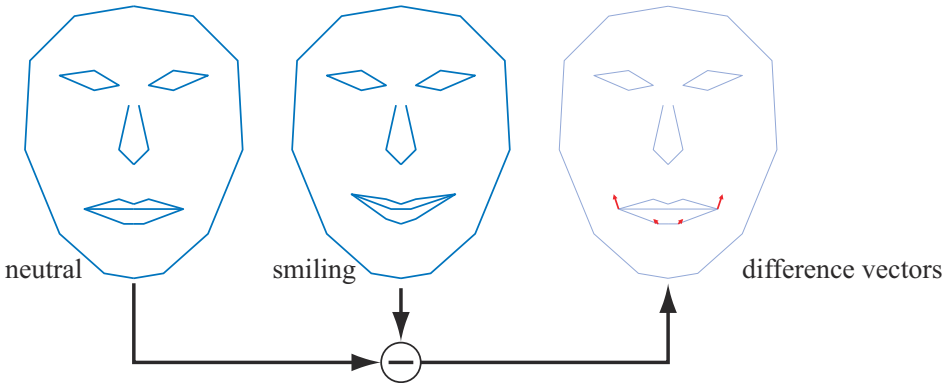

Figure 4.15. Given two mouth poses, a set of difference vectors is computed to control interpolation, or even extrapolation. In morph targets, the difference vectors are used to "add" movements onto the neutral face. With positive weights for the difference vectors, we get a smiling mouth, while negative weights can give the opposite effect.

"difference faces" are computed as: $\mathcal{D}_i = \mathcal{P}_i - \mathcal{N}$, i.e., the neutral model is subtracted from each pose.

At this point, we have a neutral model, $\mathcal{N}$, and a set of difference poses, $\mathcal{D}_i$. A morphed model $\mathcal{M}$ can then be obtained using the following formula:

$$\mathcal{M} = \mathcal{N} + \sum_{i=1}^{k} w_i \mathcal{D}_i. \tag{4.61}$$

This is the neutral model, and on top of that we add the features of the different poses as desired, using the weights, w_i. For Figure 4.15, setting $w_1 = 1$ gives us exactly the smiling face in the middle of the illustration. Using $w_1 = 0.5$ gives us a half-smiling face, and so on. One can also use negative weights and weights greater than one.

For this simple face model, we could add another face having "sad" eyebrows. Using a negative weight for the eyebrows could then create "happy" eyebrows. Since displacements are additive, this eyebrow pose could be used in conjunction with the pose for a smiling mouth.

Morph targets are a powerful technique that provides the animator with much control, since different features of a model can be manipulated independently of the others. Lewis et al. [1037] introduce *pose-space deformation*, which combines vertex blending and morph targets. Senior [1608] uses precomputed vertex textures to store and retrieve displacements between target poses. Hardware supporting stream-out and the ID of each vertex allow many more targets to be used in a single model and the effects to be computed exclusively on the GPU [841, 1074]. Using a low-resolution mesh and then generating a high-resolution mesh via the tessellation stage and displacement mapping avoids the cost of skinning every vertex in a highly detailed model [1971].

Figure 4.16. The Delsin character's face, in *inFAMOUS Second Son*, is animated using blend shapes. The same resting pose face is used for all of these shots, and then different weights are modified to make the face appear differently. *(Images provided courtesy of Naughty Dog LLC. inFAMOUS Second Son © 2014 Sony Interactive Entertainment LLC. inFAMOUS Second Son is a trademark of Sony Interactive Entertainment LLC. Developed by Sucker Punch Productions LLC.)*

A real example of using both skinning and morphing is shown in Figure 4.16. Weronko and Andreason [1872] used skinning and morphing in *The Order: 1886*.

4.6 Geometry Cache Playback

In cut scenes, it may be desirable to use extremely high-quality animations, e.g., for movements that cannot be represented using any of the methods above. A naive approach is to store all the vertices for all frames, reading them from disk and updating the mesh. However, this can amount to 50 MB/s for a simple model of 30,000 vertices used in a short animation. Gneiting [545] presents several ways to reduce memory costs down to about 10%.

First, quantization is used. For example, positions and texture coordinates are stored using 16-bit integers for each coordinate. This step is lossy in the sense that one cannot recover the original data after compression is performed. To reduce data further, spatial and temporal predictions are made and the differences encoded. For spatial compression, parallelogram prediction can be used [800]. For a triangle strip, the next vertex's predicted position is simply the current triangle reflected in the triangle's plane around the current triangle edge, which forms a parallelogram. The differences from this new position is then encoded. With good predictions, most values will be close to zero, which is ideal for many commonly used compression schemes. Similar to MPEG compression, prediction is also done in the temporal dimension. That is, every n frames, spatial compression is performed. In between, predictions are done in the temporal dimension, e.g., if a certain vertex moved by delta vector from frame $n-1$ to frame n, then it is likely to move by a similar amount to frame $n+1$. These techniques reduced storage sufficiently so this system could be used for streaming data in real time.

4.7 Projections

Before one can actually render a scene, all relevant objects in the scene must be projected onto some kind of plane or into some type of simple volume. After that, clipping and rendering are performed (Section 2.3).

The transforms seen so far in this chapter have left the fourth coordinate, the w-component, unaffected. That is, points and vectors have retained their types after the transform. Also, the bottom row in the 4×4 matrices has always been $(0 \;\; 0 \;\; 0 \;\; 1)$. *Perspective projection matrices* are exceptions to both of these properties: The bottom row contains vector and point manipulating numbers, and the homogenization process is often needed. That is, w is often not 1, so a division by w is needed to obtain the nonhomogeneous point. *Orthographic projection*, which is dealt with first in this section, is a simpler kind of projection that is also commonly used. It does not affect the w-component.

In this section, it is assumed that the viewer is looking along the camera's negative z-axis, with the y-axis pointing up and the x-axis to the right. This is a right-handed coordinate system. Some texts and software, e.g., DirectX, use a left-handed system in which the viewer looks along the camera's positive z-axis. Both systems are equally valid, and in the end, the same effect is achieved.

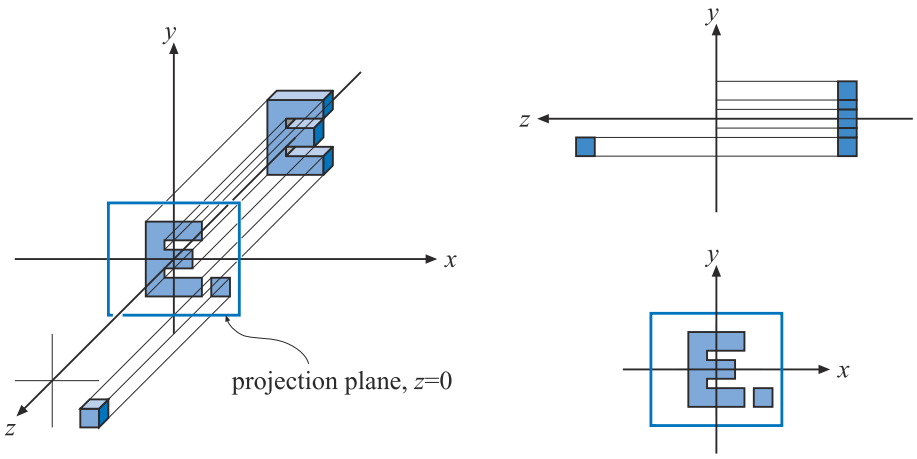

Figure 4.17. Three different views of the simple orthographic projection generated by Equation 4.62. This projection can be seen as the viewer is looking along the negative z-axis, which means that the projection simply skips (or sets to zero) the z-coordinate while keeping the x- and y-coordinates. Note that objects on both sides of $z = 0$ are projected onto the projection plane.

4.7.1 Orthographic Projection

A characteristic of an orthographic projection is that parallel lines remain parallel after the projection. When orthographic projection is used for viewing a scene, objects maintain the same size regardless of distance to the camera. Matrix $\mathbf{P}_o$, shown below, is a simple orthographic projection matrix that leaves the x- and y-components of a point unchanged, while setting the z-component to zero, i.e., it orthographically projects onto the plane $z = 0$:

$$\mathbf{P}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.62}$$

The effect of this projection is illustrated in Figure 4.17. Clearly, $\mathbf{P}_o$ is non-invertible, since its determinant $|\mathbf{P}_o| = 0$. In other words, the transform drops from three to two dimensions, and there is no way to retrieve the dropped dimension. A problem with using this kind of orthographic projection for viewing is that it projects both points with positive and points with negative z-values onto the projection plane. It is usually useful to restrict the z-values (and the x- and y-values) to a certain interval, from, say n (near plane) to f (far plane).[4] This is the purpose of the next transformation.

A more common matrix for performing orthographic projection is expressed by the six-tuple, (l, r, b, t, n, f), denoting the left, right, bottom, top, near, and far planes. This matrix scales and translates the *axis-aligned bounding box* (AABB; see the definition in Section 22.2) formed by these planes into an axis-aligned cube centered around

[4] The near plane is also called the *front plane* or *hither*; the far plane is also the *back plane* or *yon*.

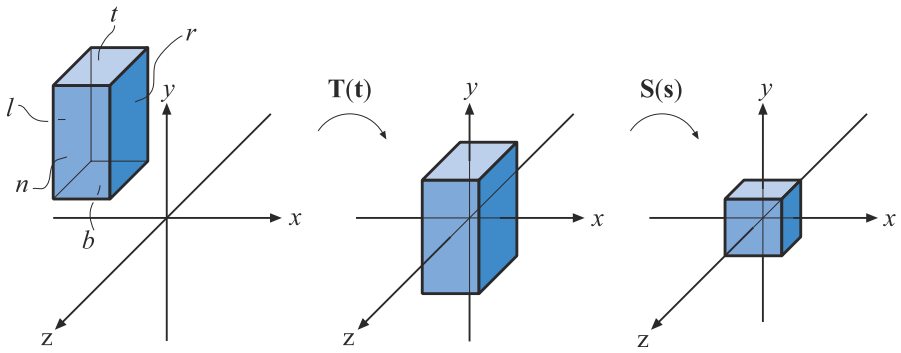

Figure 4.18. Transforming an axis-aligned box on the canonical view volume. The box on the left is first translated, making its center coincide with the origin. Then it is scaled to get the size of the canonical view volume, shown at the right.

the origin. The minimum corner of the AABB is (l, b, n) and the maximum corner is (r, t, f). It is important to realize that $n > f$, because we are looking down the negative z-axis at this volume of space. Our common sense says that the near value should be a lower number than the far, so one may let the user supply them as such, and then internally negate them.

In OpenGL the axis-aligned cube has a minimum corner of $(-1, -1, -1)$ and a maximum corner of $(1, 1, 1)$; in DirectX the bounds are $(-1, -1, 0)$ to $(1, 1, 1)$. This cube is called the *canonical view volume* and the coordinates in this volume are called *normalized device coordinates*. The transformation procedure is shown in Figure 4.18. The reason for transforming into the canonical view volume is that clipping is more efficiently performed there.

After the transformation into the canonical view volume, vertices of the geometry to be rendered are clipped against this cube. The geometry not outside the cube is finally rendered by mapping the remaining unit square to the screen. This orthographic transform is shown here:

$$\begin{aligned}\mathbf{P}_o = \mathbf{S}(\mathbf{s})\mathbf{T}(\mathbf{t}) &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & 0 \\ 0 & \frac{2}{t-b} & 0 & 0 \\ 0 & 0 & \frac{2}{f-n} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 & -\frac{l+r}{2} \\ 0 & 1 & 0 & -\frac{t+b}{2} \\ 0 & 0 & 1 & -\frac{f+n}{2} \\ 0 & 0 & 0 & 1 \end{pmatrix} \\ &= \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{2}{f-n} & -\frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}.\end{aligned} \tag{4.63}$$

As suggested by this equation, $\mathbf{P}_o$ can be written as the concatenation of a translation, $\mathbf{T}(\mathbf{t})$, followed by a scaling matrix, $\mathbf{S}(\mathbf{s})$, where $\mathbf{s} = (2/(r-l), 2/(t-b), 2/(f-n))$, and $\mathbf{t} = (-(r+l)/2, -(t+b)/2, -(f+n)/2)$. This matrix is invertible,[5] i.e., $\mathbf{P}_o^{-1} = \mathbf{T}(-\mathbf{t})\mathbf{S}((r-l)/2, (t-b)/2, (f-n)/2)$.

In computer graphics, a left-hand coordinate system is most often used after projection—i.e., for the viewport, the x-axis goes to the right, y-axis goes up, and the z-axis goes into the viewport. Because the far value is less than the near value for the way we defined our AABB, the orthographic transform will always include a mirroring transform. To see this, say the original AABBs is the same size as the goal, the canonical view volume. Then the AABB's coordinates are $(-1, -1, 1)$ for (l, b, n) and $(1, 1, -1)$ for (r, t, f). Applying that to Equation 4.63 gives us

$$\mathbf{P}_o = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \tag{4.64}$$

which is a mirroring matrix. It is this mirroring that converts from the right-handed viewing coordinate system (looking down the negative z-axis) to left-handed normalized device coordinates.

DirectX maps the z-depths to the range $[0, 1]$ instead of OpenGL's $[-1, 1]$. This can be accomplished by applying a simple scaling and translation matrix applied after the orthographic matrix, that is,

$$\mathbf{M}_{st} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0.5 & 0.5 \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.65}$$

So, the orthographic matrix used in DirectX is

$$\mathbf{P}_{o[0,1]} = \begin{pmatrix} \frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\ 0 & 0 & \frac{1}{f-n} & -\frac{n}{f-n} \\ 0 & 0 & 0 & 1 \end{pmatrix}. \tag{4.66}$$

which is normally presented in transposed form, as DirectX uses a row-major form for writing matrices.

[5] If and only if $n \neq f$, $l \neq r$, and $t \neq b$; otherwise, no inverse exists.

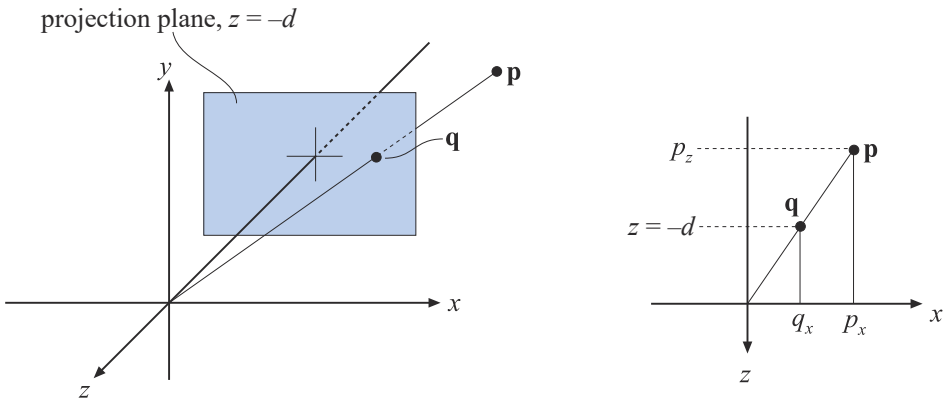

Figure 4.19. The notation used for deriving a perspective projection matrix. The point $\mathbf{p}$ is projected onto the plane $z = -d$, $d > 0$, which yields the projected point $\mathbf{q}$. The projection is performed from the perspective of the camera's location, which in this case is the origin. The similar triangle used in the derivation is shown for the x-component at the right.

4.7.2 Perspective Projection

A more complex transform than orthographic projection is perspective projection, which is commonly used in most computer graphics applications. Here, parallel lines are generally not parallel after projection; rather, they may converge to a single point at their extreme. Perspective more closely matches how we perceive the world, i.e., objects farther away are smaller.

First, we shall present an instructive derivation for a perspective projection matrix that projects onto a plane $z = -d$, $d > 0$. We derive from world space to simplify understanding of how the world-to-view conversion proceeds. This derivation is followed by the more conventional matrices used in, for example, OpenGL [885].

Assume that the camera (viewpoint) is located at the origin, and that we want to project a point, $\mathbf{p}$, onto the plane $z = -d$, $d > 0$, yielding a new point $\mathbf{q} = (q_x, q_y, -d)$. This scenario is depicted in Figure 4.19. From the similar triangles shown in this figure, the following derivation, for the x-component of $\mathbf{q}$, is obtained:

$$\frac{q_x}{p_x} = \frac{-d}{p_z} \qquad \Longleftrightarrow \qquad q_x = -d\frac{p_x}{p_z}. \tag{4.67}$$

The expressions for the other components of $\mathbf{q}$ are $q_y = -dp_y/p_z$ (obtained similarly to q_x), and $q_z = -d$. Together with the above formula, these give us the perspective projection matrix, $\mathbf{P}_p$, as shown here:

$$\mathbf{P}_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix}. \tag{4.68}$$

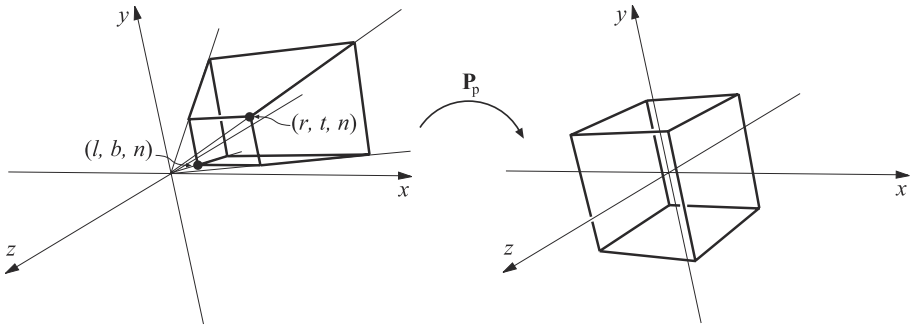

Figure 4.20. The matrix $\mathbf{P}_p$ transforms the view frustum into the unit cube, which is called the canonical view volume.

That this matrix yields the correct perspective projection is confirmed by

$$\mathbf{q} = \mathbf{P}_p\mathbf{p} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1/d & 0 \end{pmatrix} \begin{pmatrix} p_x \\ p_y \\ p_z \\ 1 \end{pmatrix} = \begin{pmatrix} p_x \\ p_y \\ p_z \\ -p_z/d \end{pmatrix} \Rightarrow \begin{pmatrix} -dp_x/p_z \\ -dp_y/p_z \\ -d \\ 1 \end{pmatrix}. \tag{4.69}$$

The last step comes from the fact that the whole vector is divided by the w-component (in this case $-p_z/d$), to get a 1 in the last position. The resulting z value is always $-d$ since we are projecting onto this plane.

Intuitively, it is easy to understand why homogeneous coordinates allow for projection. One geometrical interpretation of the homogenization process is that it projects the point (p_x, p_y, p_z) onto the plane $w = 1$.

As with the orthographic transformation, there is also a perspective transform that, rather than actually projecting onto a plane (which is noninvertible), transforms the view frustum into the canonical view volume described previously. Here the view frustum is assumed to start at $z = n$ and end at $z = f$, with $0 > n > f$. The rectangle at $z = n$ has the minimum corner at (l, b, n) and the maximum corner at (r, t, n). This is shown in Figure 4.20.

The parameters (l, r, b, t, n, f) determine the view frustum of the camera. The horizontal field of view is determined by the angle between the left and the right planes (determined by l and r) of the frustum. In the same manner, the vertical field of view is determined by the angle between the top and the bottom planes (determined by t and b). The greater the field of view, the more the camera "sees." Asymmetric frusta can be created by $r \neq -l$ or $t \neq -b$. Asymmetric frusta are, for example, used for stereo viewing and for virtual reality (Section 21.2.3).

The field of view is an important factor in providing a sense of the scene. The eye itself has a physical field of view compared to the computer screen. This relationship is

$$\phi = 2\arctan(w/(2d)), \tag{4.70}$$

where ϕ is the field of view, w is the width of the object perpendicular to the line of sight, and d is the distance to the object. For example, a 25-inch monitor is about 22 inches wide. At 12 inches away, the horizontal field of view is 85 degrees; at 20 inches, it is 58 degrees; at 30 inches, 40 degrees. This same formula can be used to convert from camera lens size to field of view, e.g., a standard 50mm lens for a 35mm camera (which has a 36mm wide frame size) gives $\phi = 2\arctan(36/(2 \cdot 50)) = 39.6$ degrees.

Using a narrower field of view compared to the physical setup will lessen the perspective effect, as the viewer will be zoomed in on the scene. Setting a wider field of view will make objects appear distorted (like using a wide angle camera lens), especially near the screen's edges, and will exaggerate the scale of nearby objects. However, a wider field of view gives the viewer a sense that objects are larger and more impressive, and has the advantage of giving the user more information about the surroundings.

The perspective transform matrix that transforms the frustum into a unit cube is given by Equation 4.71:

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{4.71}$$

After applying this transform to a point, we will get another point $\mathbf{q} = (q_x, q_y, q_z, q_w)^T$. The w-component, q_w, of this point will (most often) be nonzero and not equal to one. To get the projected point, $\mathbf{p}$, we need to divide by q_w, i.e.,

$$\mathbf{p} = (q_x/q_w, q_y/q_w, q_z/q_w, 1). \tag{4.72}$$

The matrix $\mathbf{P}_p$ always sees to it that $z = f$ maps to $+1$ and $z = n$ maps to -1.

Objects beyond the far plane will be clipped and so will not appear in the scene. The perspective projection can handle a far plane taken to infinity, which makes Equation 4.71 become

$$\mathbf{P}_p = \begin{pmatrix} \frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & 1 & -2n \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{4.73}$$

To sum up, the perspective transform (in any form), $\mathbf{P}_p$, is applied, followed by clipping and homogenization (division by w), which results in normalized device coordinates.

To get the perspective transform used in OpenGL, first multiply with $\mathbf{S}(1, 1, -1, 1)$, for the same reasons as for the orthographic transform. This simply negates the values in the third column of Equation 4.71. After this mirroring transform has been applied, the near and far values are entered as positive values, with $0 < n' < f'$, as they would traditionally be presented to the user. However, they still represent distances along the world's negative z-axis, which is the direction of view. For reference purposes, here is the OpenGL equation:

$$\mathbf{P}_{\text{OpenGL}} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2f'n'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{pmatrix}. \tag{4.74}$$

A simpler setup is to provide just the vertical field of view, ϕ, the aspect ratio $a = w/h$ (where $w \times h$ is the screen resolution), n', and f'. This results in

$$\mathbf{P}_{\text{OpenGL}} = \begin{pmatrix} c/a & 0 & 0 & 0 \\ 0 & c & 0 & 0 \\ 0 & 0 & -\frac{f'+n'}{f'-n'} & -\frac{2f'n'}{f'-n'} \\ 0 & 0 & -1 & 0 \end{pmatrix}, \tag{4.75}$$

where $c = 1.0/\tan(\phi/2)$. This matrix does exactly what the old `gluPerspective()` did, which is part of the OpenGL Utility Library (GLU).

Some APIs (e.g., DirectX) map the near plane to $z = 0$ (instead of $z = -1$) and the far plane to $z = 1$. In addition, DirectX uses a left-handed coordinate system to define its projection matrix. This means DirectX looks along the positive z-axis and presents the near and far values as positive numbers. Here is the DirectX equation:

$$\mathbf{P}_{p[0,1]} = \begin{pmatrix} \frac{2n'}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\ 0 & \frac{2n'}{t-b} & -\frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{f'}{f'-n'} & -\frac{f'n'}{f'-n'} \\ 0 & 0 & 1 & 0 \end{pmatrix}. \tag{4.76}$$

DirectX uses row-major form in its documentation, so this matrix is normally presented in transposed form.

One effect of using a perspective transformation is that the computed depth value does not vary linearly with the input p_z value. Using any of Equations 4.74–4.76 to

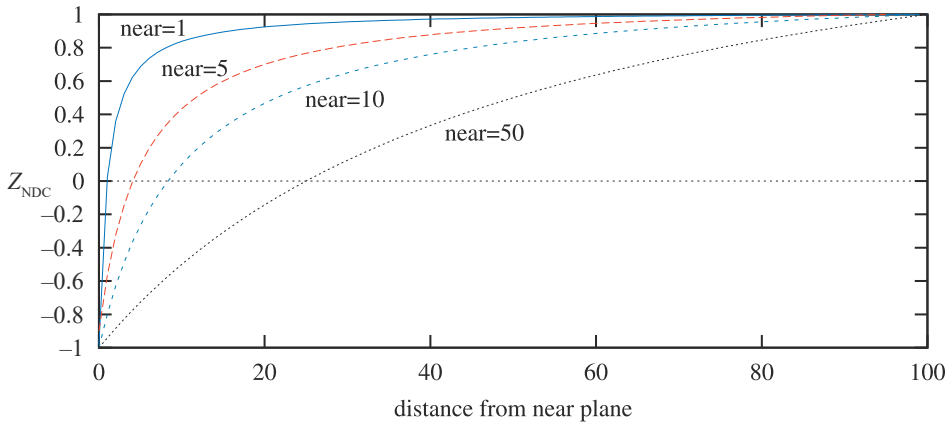

Figure 4.21. The effect of varying the distance of the near plane from the origin. The distance $f' - n'$ is kept constant at 100. As the near plane becomes closer to the origin, points nearer the far plane use a smaller range of the normalized device coordinate (NDC) depth space. This has the effect of making the z-buffer less accurate at greater distances.

multiply with a point $\mathbf{p}$, we can see that

$$\mathbf{v} = \mathbf{P}\mathbf{p} = \begin{pmatrix} \cdots \\ \cdots \\ dp_z + e \\ \pm p_z \end{pmatrix}, \tag{4.77}$$

where the details of v_x and v_y have been omitted, and the constants d and f depend on the chosen matrix. If we use Equation 4.74, for example, then $d = -(f'+n')/(f'-n')$, $e = -2f'n'/(f' - n')$, and $v_x = -p_z$. To obtain the depth in normalized device coordinates (NDC), we need to divide by the w-component, which results in

$$z_{\text{NDC}} = \frac{dp_z + e}{-p_z} = d - \frac{e}{p_z}, \tag{4.78}$$

where $z_{\text{NDC}} \in [-1, +1]$ for the OpenGL projection. As can be seen, the output depth z_{NDC} is inversely proportional to the input depth, p_z.

For example, if $n' = 10$ and $f' = 110$ (using the OpenGL terminology), when p_z is 60 units down the negative z-axis (i.e., the halfway point) the normalized device coordinate depth value is 0.833, not 0. Figure 4.21 shows the effect of varying the distance of the near plane from the origin. Placement of the near and far planes affects the precision of the z-buffer. This effect is discussed further in Section 23.7.

There are several ways to increase the depth precision. A common method, which we call *reversed z*, is to store $1.0 - z_{\text{NDC}}$ [978] either with floating point depth or with integers. A comparison is shown in Figure 4.22. Reed [1472] shows with simulations

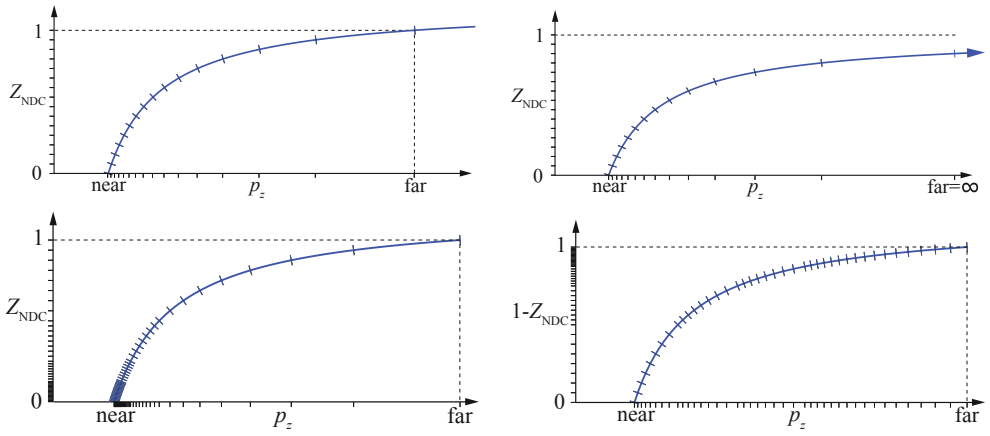

Figure 4.22. Different ways to set up the depth buffer with the DirectX transform, i.e., $z_{\text{NDC}} \in [0, +1]$. Top left: standard integer depth buffer, shown here with 4 bits of precision (hence the 16 marks on the y-axis). Top right: the far plane set to ∞, the small shifts on both axes showing that one does not lose much precision by doing so. Bottom left: with 3 exponent bits and 3 mantissa bits for floating point depth. Notice how the distribution is nonlinear on the y-axis, which makes it even worse on the x-axis. Bottom right: reversed floating point depth, i.e., $1 - z_{\text{NDC}}$, with a much better distribution as a result. *(Illustrations courtesy of Nathan Reed.)*

that using a floating point buffer with reversed z provides the best accuracy, and this is also the preferred method for integer depth buffers (which usually have 24 bits per depth). For the standard mapping (i.e., non-reversed z), separating the projection matrix in the transform decreases the error rate, as suggested by Upchurch and Desbrun [1803]. For example, it can be better to use $\mathbf{P}(\mathbf{Mp})$ than $\mathbf{Tp}$, where $\mathbf{T} = \mathbf{PM}$. Also, in the range of $[0.5, 1.0]$, fp32 and int24 are quite similar in accuracy, since fp32 has a 23-bit mantissa. The reason for having z_{NDC} proportional to $1/p_z$ is that it makes hardware simpler and compression of depth more successful, which is discussed more in Section 23.7.

Lloyd [1063] proposed to use a logarithm of the depth values to improve precision for shadow maps. Lauritzen et al. [991] use the previous frame's z-buffer to determine a maximum near plane and minimum far plane. For screen-space depth, Kemen [881] proposes to use the following remapping per vertex:

$$\begin{aligned} z &= w\left(\log_2\left(\max(10^{-6}, 1+w)\right)f_c - 1\right), && \text{[OpenGL]} \\ z &= w\log_2\left(\max(10^{-6}, 1+w)\right)f_c/2, && \text{[DirectX]} \end{aligned} \tag{4.79}$$

where w is the w-value of the vertex after the projection matrix, and z is the output z from the vertex shader. The constant f_c is $f_c = 2/\log_2(f+1)$, where f is the far plane. When this transform is applied in the vertex shader only, depth will still be interpolated linearly over the triangle by the GPU in between the nonlinearly transformed depths at vertices (Equation 4.79). Since the logarithm is a monotonic

function, occlusion culling hardware and depth compression techniques will still work as long as the difference between the piecewise linear interpolation and the accurate nonlinearly transformed depth value is small. That's true for most cases with sufficient geometry tessellation. However, it is also possible to apply the transform per fragment. This is done by outputting a per-vertex value of $e = 1 + w$, which is then interpolated by the GPU over the triangle. The pixel shader then modifies the fragment depth as $\log_2(e_i)f_c/2$, where e_i is the interpolated value of e. This method is a good alternative when there is no floating point depth in the GPU and when rendering using large distances in depth.

Cozzi [1605] proposes to use multiple frusta, which can improve accuracy to effectively any desired rate. The view frustum is divided in the depth direction into several non-overlapping smaller sub-frusta whose union is exactly the frustum. The sub-frusta are rendered to in back-to-front order. First, both the color and depth buffers are cleared, and all objects to be rendered are sorted into each sub-frusta that they overlap. For each sub-frusta, its projection matrix is set up, the depth buffer is cleared, and then the objects that overlap the sub-frusta are rendered.

Further Reading and Resources

The *immersive linear algebra* site [1718] provides an interactive book about the basics of this subject, helping build intuition by encouraging you to manipulate the figures. Other interactive learning tools and transform code libraries are linked from realtimerendering.com.

One of the best books for building up one's intuition about matrices in a painless fashion is Farin and Hansford's *The Geometry Toolbox* [461]. Another useful work is Lengyel's *Mathematics for 3D Game Programming and Computer Graphics* [1025]. For a different perspective, many computer graphics texts, such as Hearn and Baker [689], Marschner and Shirley [1129], and Hughes et al. [785] also cover matrix basics. The course by Ochiai et al. [1310] introduces the matrix foundations as well as the exponential and logarithm of matrices, with uses for computer graphics. The *Graphics Gems* series [72, 540, 695, 902, 1344] presents various transform-related algorithms and has code available online for many of these. Golub and Van Loan's *Matrix Computations* [556] is the place to start for a serious study of matrix techniques in general. More on skeleton-subspace deformation/vertex blending and shape interpolation can be read in Lewis et al.'s SIGGRAPH paper [1037].

Hart et al. [674] and Hanson [663] provide visualizations of quaternions. Pletinckx [1421] and Schlag [1566] present different ways of interpolating smoothly between a set of quaternions. Vlachos and Isidoro [1820] derive formulae for C^2 interpolation of quaternions. Related to quaternion interpolation is the problem of computing a consistent coordinate system along a curve. This is treated by Dougan [374].

Alexa [28] and Lazarus and Verroust [1000] present surveys on many different morphing techniques. Parent's book [1354] is an excellent source for techniques about computer animation.

Chapter 5
Shading Basics

"A good picture is equivalent to a good deed."
—Vincent Van Gogh

When you render images of three-dimensional objects, the models should not only have the proper geometrical shape, they should also have the desired visual appearance. Depending on the application, this can range from photorealism—an appearance nearly identical to photographs of real objects—to various types of stylized appearance chosen for creative reasons. See Figure 5.1 for examples of both.

This chapter will discuss those aspects of shading that are equally applicable to photorealistic and stylized rendering. Chapter 15 is dedicated specifically to stylized rendering, and a significant part of the book, Chapters 9 through 14, focuses on physically based approaches commonly used for photorealistic rendering.

5.1 Shading Models

The first step in determining the appearance of a rendered object is to choose a *shading model* to describe how the object's color should vary based on factors such as surface orientation, view direction, and lighting.

As an example, we will use a variation on the *Gooch shading model* [561]. This is a form of non-photorealistic rendering, the subject of Chapter 15. The Gooch shading model was designed to increase legibility of details in technical illustrations.

The basic idea behind Gooch shading is to compare the surface normal to the light's location. If the normal points toward the light, a warmer tone is used to color the surface; if it points away, a cooler tone is used. Angles in between interpolate between these tones, which are based on a user-supplied surface color. In this example, we add a stylized "highlight" effect to the model to give the surface a shiny appearance. Figure 5.2 shows the shading model in action.

Shading models often have properties used to control appearance variation. Setting the values of these properties is the next step in determining object appearance. Our example model has just one property, surface color, as shown in the bottom image of Figure 5.2.

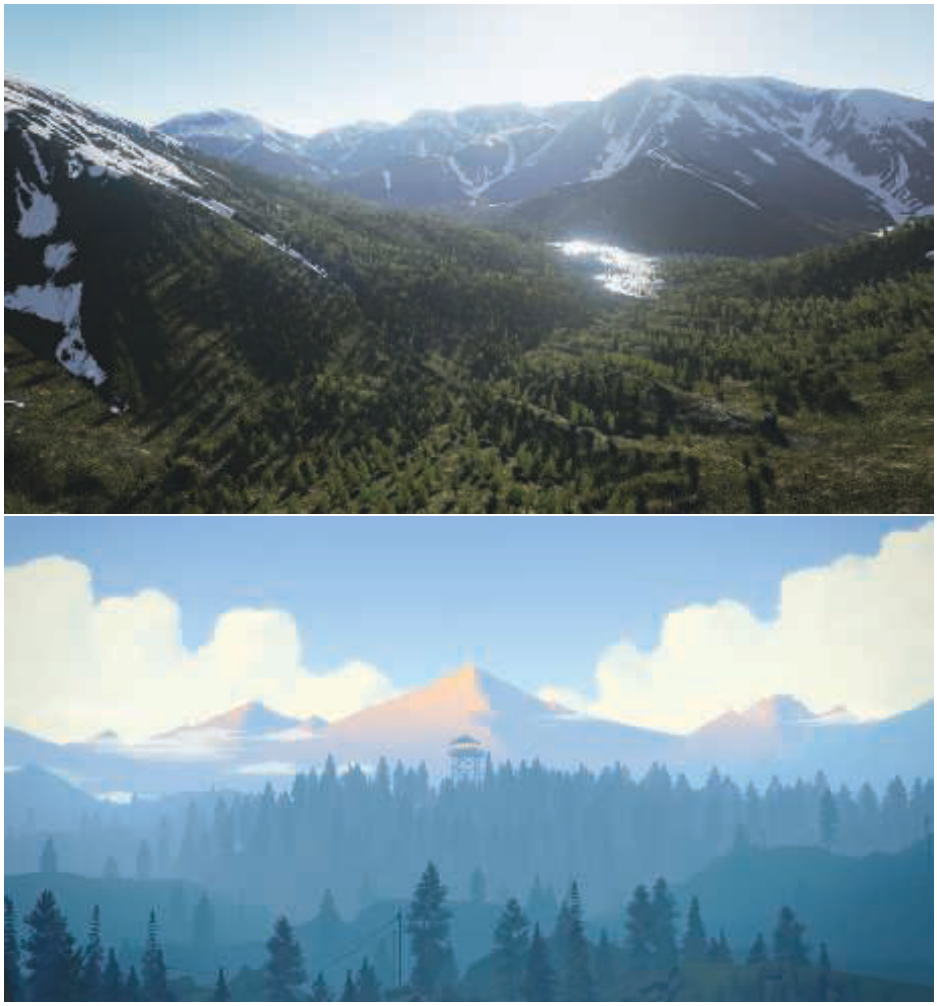

Figure 5.1. The top image is from a realistic landscape scene rendered using the Unreal Engine. The bottom image is from the game *Firewatch* by Campo Santo, which was designed with a illustrative art style. *(Upper image courtesy of Gökhan Karadayi, lower image courtesy of Campo Santo.)*

Like most shading models, this example is affected by the surface orientation relative to the view and lighting directions. For shading purposes, these directions are commonly expressed as normalized (unit-length) vectors, as illustrated in Figure 5.3.

Now that we have defined all the inputs to our shading model, we can look at the mathematical definition of the model itself:

$$\mathbf{c}_{\text{shaded}} = s\,\mathbf{c}_{\text{highlight}} + (1-s)\left(t\,\mathbf{c}_{\text{warm}} + (1-t)\,\mathbf{c}_{\text{cool}}\right). \tag{5.1}$$

Figure 5.2. A stylized shading model combining Gooch shading with a highlight effect. The top image shows a complex object with a neutral surface color. The bottom image shows spheres with various different surface colors. *(Chinese Dragon mesh from Computer Graphics Archive [1172], original model from Stanford 3D Scanning Repository.)*

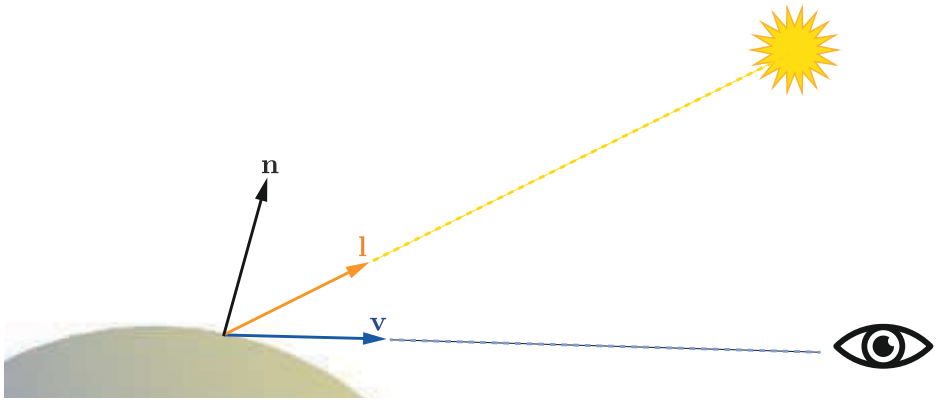

Figure 5.3. Unit-length vector inputs to the example shading model (and most others): surface normal $\mathbf{n}$, view vector $\mathbf{v}$, and light direction $\mathbf{l}$.

In this equation, we have used the following intermediate calculations:

$$\begin{aligned}
\mathbf{c}_{\text{cool}} &= (0, 0, 0.55) + 0.25\,\mathbf{c}_{\text{surface}}, \\
\mathbf{c}_{\text{warm}} &= (0.3, 0.3, 0) + 0.25\,\mathbf{c}_{\text{surface}}, \\
\mathbf{c}_{\text{highlight}} &= (1, 1, 1), \\
t &= \frac{(\mathbf{n} \cdot \mathbf{l}) + 1}{2}, \\
\mathbf{r} &= 2\,(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l}, \\
s &= \big(100\,(\mathbf{r} \cdot \mathbf{v}) - 97\big)^{\overline{+}}.
\end{aligned} \tag{5.2}$$

Several of the mathematical expressions in this definition are often found in other shading models as well. Clamping operations, typically clamping to 0 or clamping between 0 and 1, are common in shading. Here we use the $x^{\overline{+}}$ notation, introduced in Section 1.2, for the clamp between 0 and 1 used in the computation of the highlight blend factor s. The dot product operator appears three times, in each case between two unit-length vectors; this is an extremely common pattern. The dot product of two vectors is the product of their lengths and the cosine of the angle between them. So, the dot product of two unit-length vectors is simply the cosine, which is a useful measure of the degree to which two vectors are aligned with each other. Simple functions composed of cosines are often the most pleasing and accurate mathematical expressions to account for the relationship between two directions, e.g., light direction and surface normal, in a shading model.

Another common shading operation is to interpolate linearly between two colors based on a scalar value between 0 and 1. This operation takes the form $t\mathbf{c}_{\text{a}} + (1 - t)\mathbf{c}_{\text{b}}$ that interpolates between $\mathbf{c}_{\text{a}}$ and $\mathbf{c}_{\text{b}}$ as the value of t moves between 1 and 0, respectively. This pattern appears twice in this shading model, first to interpolate between $\mathbf{c}_{\text{warm}}$ and $\mathbf{c}_{\text{cool}}$ and second to interpolate between the result of the previous interpolation and $\mathbf{c}_{\text{highlight}}$. Linear interpolation appears so often in shaders that it is a built-in function, called `lerp` or `mix`, in every shading language we have seen.

The line "$\mathbf{r} = 2\,(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l}$" computes the reflected light vector, reflecting $\mathbf{l}$ about $\mathbf{n}$. While not quite as common as the previous two operations, this is common enough for most shading languages to have a built-in `reflect` function as well.

By combining such operations in different ways with various mathematical expressions and shading parameters, shading models can be defined for a huge variety of stylized and realistic appearances.

5.2 Light Sources

The impact of lighting on our example shading model was quite simple; it provided a dominant direction for shading. Of course, lighting in the real world can be quite complex. There can be multiple light sources each with its own size, shape, color,

and intensity; indirect lighting adds even more variation. As we will see in Chapter 9, physically based, photorealistic shading models need to take all these parameters into account.

In contrast, stylized shading models may use lighting in many different ways, depending on the needs of the application and visual style. Some highly stylized models may have no concept of lighting at all, or (like our Gooch shading example) may only use it to provide some simple directionality.

The next step in lighting complexity is for the shading model to react to the presence or absence of light in a binary way. A surface shaded with such a model would have one appearance when lit and a different appearance when unaffected by light. This implies some criteria for distinguishing the two cases: distance from light sources, shadowing (which will be discussed in Chapter 7), whether the surface is facing away from the light source (i.e., the angle between the surface normal $\mathbf{n}$ and the light vector $\mathbf{l}$ is greater than 90°), or some combination of these factors.

It is a small step from the binary presence or absence of light to a continuous scale of light intensities. This could be expressed as a simple interpolation between absence and full presence, which implies a bounded range for the intensity, perhaps 0 to 1, or as an unbounded quantity that affects the shading in some other way. A common option for the latter is to factor the shading model into lit and unlit parts, with the light intensity k_{light} linearly scaling the lit part:

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + k_{\text{light}} f_{\text{lit}}(\mathbf{l}, \mathbf{n}, \mathbf{v}). \tag{5.3}$$

This easily extends to an RGB light color $\mathbf{c}_{\text{light}}$,

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + \mathbf{c}_{\text{light}} f_{\text{lit}}(\mathbf{l}, \mathbf{n}, \mathbf{v}), \tag{5.4}$$

and to multiple light sources,

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + \sum_{i=1}^{n} \mathbf{c}_{\text{light}_i} f_{\text{lit}}(\mathbf{l}_i, \mathbf{n}, \mathbf{v}). \tag{5.5}$$

The unlit part $f_{\text{unlit}}(\mathbf{n}, \mathbf{v})$ corresponds to the "appearance when unaffected by light" of shading models that treat light as a binary. It can have various forms, depending on the desired visual style and the needs of the application. For example, $f_{\text{unlit}}() = (0, 0, 0)$ will cause any surface unaffected by a light source to be colored pure black. Alternately, the unlit part could express some form of stylized appearance for unlit objects, similar to the Gooch model's cool color for surfaces facing away from light. Often, this part of the shading model expresses some form of lighting that does not come directly from explicitly placed light sources, such as light from the sky or light bounced from surrounding objects. These other forms of lighting will be discussed in Chapters 10 and 11.

We mentioned earlier that a light source does not affect a surface point if the light direction $\mathbf{l}$ is more than 90° from the surface normal $\mathbf{n}$, in effect coming from

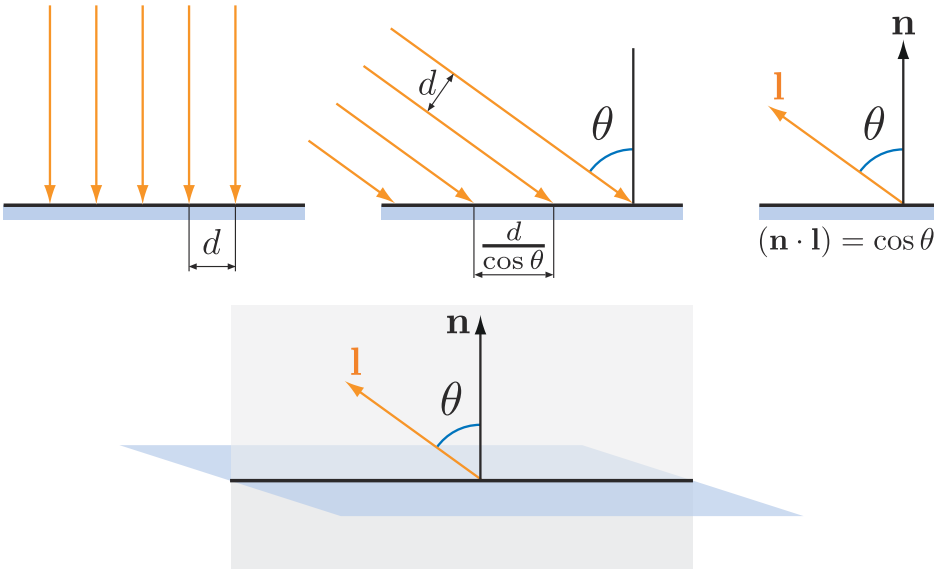

Figure 5.4. The upper row of drawings shows a cross-section view of light on a surface. On the left the light rays hit the surface straight on, in the center they hit the surface at an angle, and on the right we see the use of vector dot products to compute the angle cosine. The bottom drawing shows the cross-section plane (which includes the light and view vectors) in relation to the full surface.

underneath the surface. This can be thought of as a special case of a more general relationship between the light's direction, relative to the surface, and its effect on shading. Although physically based, this relationship can be derived from simple geometrical principles and is useful for many types of non-physically based, stylized shading models as well.

The effect of light on a surface can be visualized as a set of rays, with the density of rays hitting the surface corresponding to the light intensity for surface shading purposes. See Figure 5.4, which shows a cross section of a lit surface. The spacing between light rays hitting the surface along that cross section is inversely proportional to the cosine of the angle between $\mathbf{l}$ and $\mathbf{n}$. So, the overall density of light rays hitting the surface is proportional to the cosine of the angle between $\mathbf{l}$ and $\mathbf{n}$, which, as we have seen earlier, is equal to the dot product between those two unit-length vectors. Here we see why it is convenient to define the light vector $\mathbf{l}$ opposite to the light's direction of travel; otherwise we would have to negate it before performing the dot product.

More precisely, the ray density (and thus the light's contribution to shading) is proportional to the dot product when it is positive. Negative values correspond to light rays coming from behind the surface, which have no effect. So, before multiplying the light's shading by the lighting dot product, we need to first clamp the dot product

to 0. Using the x^+ notation introduced in Section 1.2, which means clamping negative values to zero, we have

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + \sum_{i=1}^{n} (\mathbf{l}_i \cdot \mathbf{n})^+ \mathbf{c}_{\text{light}_i} f_{\text{lit}}(\mathbf{l}_i, \mathbf{n}, \mathbf{v}). \tag{5.6}$$

Shading models that support multiple light sources will typically use one of the structures from Equation 5.5, which is more general, or Equation 5.6, which is required for physically based models. It can be advantageous for stylized models as well, since it helps ensure an overall consistency to the lighting, especially for surfaces that are facing away from the light or are shadowed. However, some models are not a good fit for that structure; such models would use the structure in Equation 5.5.

The simplest possible choice for the function $f_{\text{lit}}()$ is to make it a constant color,

$$f_{\text{lit}}() = \mathbf{c}_{\text{surface}}, \tag{5.7}$$

which results in the following shading model:

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + \sum_{i=1}^{n} (\mathbf{l}_i \cdot \mathbf{n})^+ \mathbf{c}_{\text{light}_i} \mathbf{c}_{\text{surface}}. \tag{5.8}$$

The lit part of this model corresponds to the *Lambertian* shading model, after Johann Heinrich Lambert [967], who published it in 1760! This model works in the context of ideal diffusely reflecting surfaces, i.e., surfaces that are perfectly matte. We present here a somewhat simplified explanation of Lambert's model, which will be covered with more rigor in Chapter 9. The Lambertian model can be used by itself for simple shading, and it is a key building block in many shading models.

We can see from Equations 5.3–5.6 that a light source interacts with the shading model via two parameters: the vector $\mathbf{l}$ pointing toward the light and the light color $\mathbf{c}_{\text{light}}$. There are various different types of light sources, which differ primarily in how these two parameters vary over the scene.

We will next discuss several popular types of light sources, which have one thing in common: At a given surface location, each light source illuminates the surface from only one direction $\mathbf{l}$. In other words, the light source, as seen from the shaded surface location, is an infinitesimally small point. This is not strictly true for real-world lights, but most light sources are small relative to their distance from illuminated surfaces, making this a reasonable approximation. In Sections 7.1.2 and 10.1, we will discuss light sources that illuminate a surface location from a range of directions, i.e., "area lights."

5.2.1 Directional Lights

Directional light is the simplest model of a light source. Both $\mathbf{l}$ and $\mathbf{c}_{\text{light}}$ are constant over the scene, except that $\mathbf{c}_{\text{light}}$ may be attenuated by shadowing. Directional lights

have no location. Of course, actual light sources do have specific locations in space. Directional lights are abstractions, which work well when the distance to the light is large relative to the scene size. For example, a floodlight 20 feet away illuminating a small tabletop diorama could be represented as a directional light. Another example is pretty much any scene lit by the sun, unless the scene in question is something such as the inner planets of the solar system.

The concept of a directional light can be somewhat extended to allow varying the value of $\mathbf{c}_{\text{light}}$ while the light direction $\mathbf{l}$ remains constant. This is most often done to bound the effect of the light to a particular part of the scene for performance or creative reasons. For example, a region could be defined with two nested (one inside the other) box-shaped volumes, where $\mathbf{c}_{\text{light}}$ is equal to $(0, 0, 0)$ (pure black) outside the outer box, is equal to some constant value inside the inner box, and smoothly interpolates between those extremes in the region between the two boxes.

5.2.2 Punctual Lights

A *punctual light* is not one that is on time for its appointments, but rather a light that has a location, unlike directional lights. Such lights also have no dimensions to them, no shape or size, unlike real-world light sources. We use the term "punctual," from the Latin *punctus* meaning "point," for the class consisting of all sources of illumination that originate from a single, local position. We use the term "point light" to mean a specific kind of emitter, one that shines light equally in all directions. So, point and spotlight are two different forms of punctual lights. The light direction vector $\mathbf{l}$ varies depending on the location of the currently shaded surface point $\mathbf{p}_0$ relative to the punctual light's position $\mathbf{p}_{\text{light}}$:

$$\mathbf{l} = \frac{\mathbf{p}_{\text{light}} - \mathbf{p}_0}{\|\mathbf{p}_{\text{light}} - \mathbf{p}_0\|}. \tag{5.9}$$

This equation is an example of vector normalization: dividing a vector by its length to produce a unit-length vector pointing in the same direction. This is another common shading operation, and, like the shading operations we have seen in the previous section, it is a built-in function in most shading languages. However, sometimes an intermediate result from this operation is needed, which requires performing the normalization explicitly, in multiple steps, using more basic operations. Applying this to the punctual light direction computation gives us the following:

$$\begin{aligned} \mathbf{d} &= \mathbf{p}_{\text{light}} - \mathbf{p}_0, \\ r &= \sqrt{\mathbf{d} \cdot \mathbf{d}}, \\ \mathbf{l} &= \frac{\mathbf{d}}{r}. \end{aligned} \tag{5.10}$$

Since the dot product of two vectors is equal to the product of the two vector's lengths with the cosine of the angle between them, and the cosine of 0° is 1.0, the dot product

of a vector with itself is the square of its length. So, to find the length of any vector, we just dot it with itself and take the square root of the result.

The intermediate value that we need is r, the distance between the punctual light source and the currently shaded point. Besides its use in normalizing the light vector, the value of r is also needed to compute the attenuation (darkening) of the light color $\mathbf{c}_{\text{light}}$ as a function of distance. This will be discussed further in the following section.

Point/Omni Lights

Punctual lights that emit light uniformly in all directions are known as *point lights* or *omni lights*. For point lights, $\mathbf{c}_{\text{light}}$ varies as a function of the distance r, with the only source of variation being the distance attenuation mentioned above. Figure 5.5 shows why this darkening occurs, using similar geometric reasoning as the demonstration of the cosine factor in Figure 5.4. At a given surface, the spacing between rays from a point light is proportional to the distance from the surface to the light. Unlike the cosine factor in Figure 5.4, this spacing increase happens along both dimensions of the surface, so the ray density (and thus the light color $\mathbf{c}_{\text{light}}$) is proportional to the inverse square distance $1/r^2$. This enables us to specify the spatial variation in $\mathbf{c}_{\text{light}}$ with a single light property, $\mathbf{c}_{\text{light}_0}$, which is defined as the value of $\mathbf{c}_{\text{light}}$ at a fixed reference distance r_0:

$$\mathbf{c}_{\text{light}}(r) = \mathbf{c}_{\text{light}_0}\left(\frac{r_0}{r}\right)^2. \tag{5.11}$$

Equation 5.11 is often referred to as *inverse-square light attenuation*. Although technically the correct distance attenuation for a point light, there are some issues that make this equation less than ideal for practical shading use.

The first issue occurs at relatively small distances. As the value of r tends to 0, the value of $\mathbf{c}_{\text{light}}$ will increase in an unbounded manner. When r reaches 0, we will have a divide-by-zero singularity. To address this, one common modification is to add a small value ϵ to the denominator [861]:

$$\mathbf{c}_{\text{light}}(r) = \mathbf{c}_{\text{light}_0}\frac{r_0^2}{r^2 + \epsilon}. \tag{5.12}$$

The exact value used for ϵ depends on the application; for example, the Unreal game engine uses $\epsilon = 1$ cm [861].

An alternative modification, used in the CryEngine [1591] and Frostbite [960] game engines, is to clamp r to a minimum value r_{min}:

$$\mathbf{c}_{\text{light}}(r) = \mathbf{c}_{\text{light}_0}\left(\frac{r_0}{\max(r, r_{\text{min}})}\right)^2. \tag{5.13}$$

Unlike the somewhat arbitrary ϵ value used in the previous method, the value of r_{min} has a physical interpretation: the radius of the physical object emitting the light. Values of r smaller than r_{min} correspond to the shaded surface penetrating inside the physical light source, which is impossible.

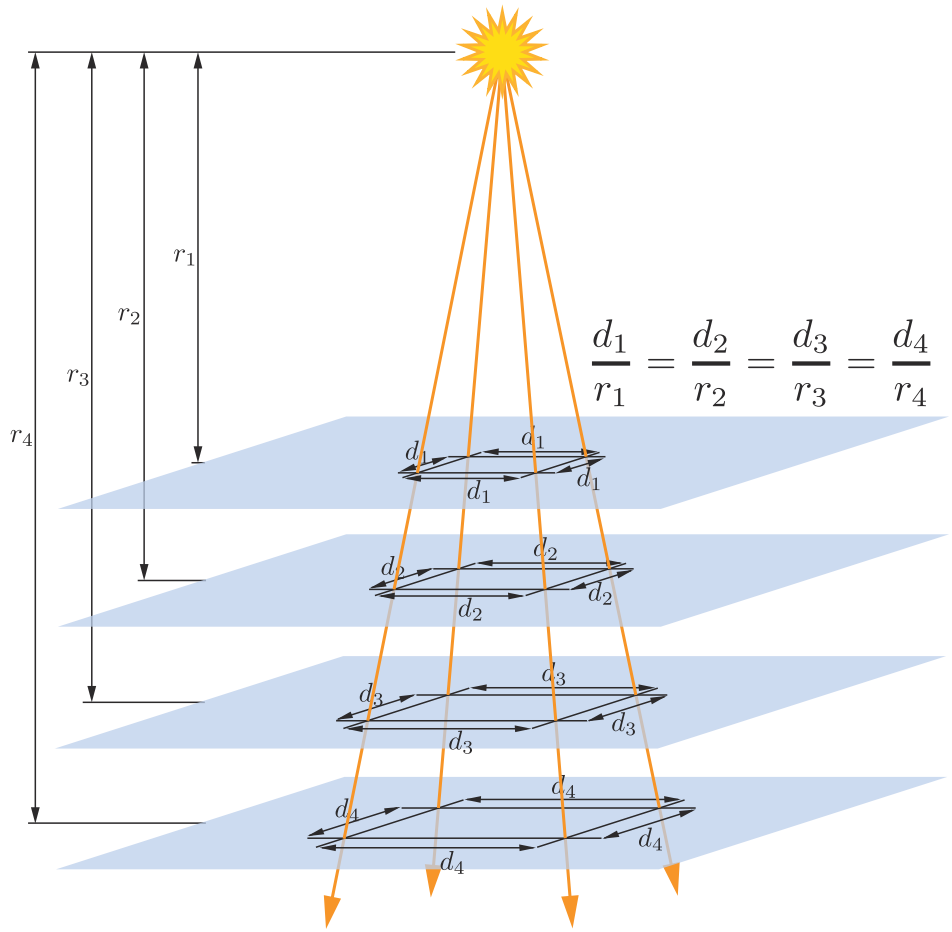

Figure 5.5. The spacing between light rays from a point light increases proportionally to the distance r. Since the spacing increase occurs in two dimensions, the density of rays (and thus the light intensity) decreases proportionally to $1/r^2$.

In contrast, the second issue with inverse-square attenuation occurs at relatively large distances. The problem is not with visuals but with performance. Although light intensity keeps decreasing with distance it never goes to 0. For efficient rendering, it is desirable for lights to reach 0 intensity at some finite distance (Chapter 20). There are many different ways in which the inverse-square equation could be modified to achieve this. Ideally the modification should introduce as little change as possible. To avoid a sharp cutoff at the boundary of the light's influence, it is also preferable for the derivative and value of the modified function to reach 0 at the same distance. One solution is to multiply the inverse-square equation by a *windowing function* with the

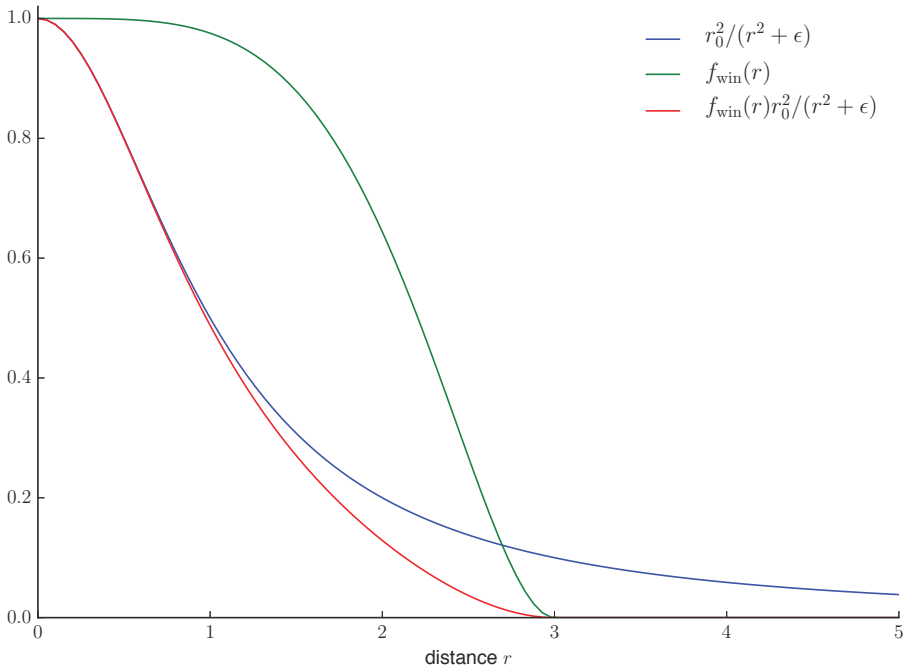

Figure 5.6. This graph shows an inverse-square curve (using the ϵ method to avoid singularities, with an ϵ value of 1), the windowing function described in Equation 5.14 (with r_{max} set to 3), and the windowed curve.

desired properties. One such function [860] is used by both the Unreal Engine [861] and Frostbite [960] game engines:

$$f_{\text{win}}(r) = \left(1 - \left(\frac{r}{r_{\text{max}}}\right)^4\right)^{+2}. \tag{5.14}$$

The +2 means to clamp the value, if negative, to 0 before squaring it. Figure 5.6 shows an example inverse-square curve, the windowing function from Equation 5.14, and the result of multiplying the two.

Application requirements will affect the choice of method used. For example, having the derivative equal to 0 at r_{max} is particularly important when the distance attenuation function is sampled at a relatively low spatial frequency (e.g., in light maps or per-vertex). CryEngine does not use light maps or vertex lighting, so it employs a simpler adjustment, switching to linear falloff in the range between $0.8r_{\text{max}}$ and r_{max} [1591].

For some applications, matching the inverse-square curve is not a priority, so some other function entirely is used. This effectively generalizes Equations 5.11–5.14 to the following:

$$\mathbf{c}_{\text{light}}(r) = \mathbf{c}_{\text{light}_0} f_{\text{dist}}(r), \tag{5.15}$$

where $f_{\text{dist}}(r)$ is some function of distance. Such functions are called *distance falloff functions*. In some cases, the use of non-inverse-square falloff functions is driven by performance constraints. For example, the game *Just Cause 2* needed lights that were extremely inexpensive to compute. This dictated a falloff function that was simple to compute, while also being smooth enough to avoid per-vertex lighting artifacts [1379]:

$$f_{\text{dist}}(r) = \left(1 - \left(\frac{r}{r_{\text{max}}}\right)^2\right)^{+2}. \tag{5.16}$$

In other cases, the choice of falloff function may be driven by creative considerations. For example, the Unreal Engine, used for both realistic and stylized games, has two modes for light falloff: an inverse-square mode, as described in Equation 5.12, and an exponential falloff mode that can be tweaked to create a variety of attenuation curves [1802]. The developers of the game *Tomb Raider* (2013) used spline-editing tools to author falloff curves [953], allowing for even greater control over the curve shape.

Spotlights

Unlike point lights, illumination from nearly all real-world light sources varies by direction as well as distance. This variation can be expressed as a directional falloff function $f_{\text{dir}}(\mathbf{l})$, which combines with the distance falloff function to define the overall spatial variation in light intensity:

$$\mathbf{c}_{\text{light}} = \mathbf{c}_{\text{light}_0} f_{\text{dist}}(r) f_{\text{dir}}(\mathbf{l}). \tag{5.17}$$

Different choices of $f_{\text{dir}}(\mathbf{l})$ can produce various lighting effects. One important type of effect is the *spotlight*, which projects light in a circular cone. A spotlight's directional falloff function has rotational symmetry around a spotlight direction vector $\mathbf{s}$, and thus can be expressed as a function of the angle θ_s between $\mathbf{s}$ and the reversed light vector $-\mathbf{l}$ to the surface. The light vector needs to be reversed because we define $\mathbf{l}$ at the surface as pointing toward the light, and here we need the vector pointing away from the light.

Most spotlight functions use expressions composed of the cosine of θ_s, which (as we have seen earlier) is the most common form for angles in shading. Spotlights typically have an *umbra angle* θ_u, which bounds the light such that $f_{\text{dir}}(\mathbf{l}) = 0$ for all $\theta_s \geq \theta_u$. This angle can be used for culling in a similar manner to the maximum falloff distance r_{max} seen earlier. It is also common for spotlights to have a *penumbra angle* θ_p, which defines an inner cone where the light is at its full intensity. See Figure 5.7.

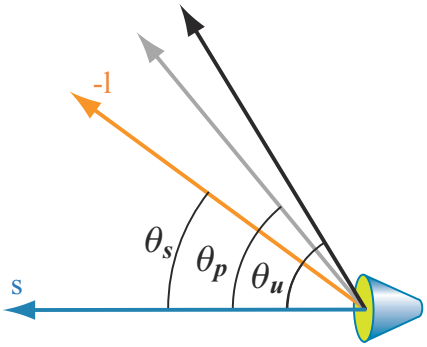

Figure 5.7. A spotlight: θ_s is the angle from the light's defined direction $\mathbf{s}$ to the vector $-\mathbf{l}$, the direction to the surface; θ_p shows the penumbra; and θ_u shows the umbra angles defined for the light.

Various directional falloff functions are used for spotlights, but they tend to be roughly similar. For example, the function $f_{\mathrm{dir_F}}(\mathbf{l})$ is used in the Frostbite game engine [960], and the function $f_{\mathrm{dir_T}}(\mathbf{l})$ is used in the three.js browser graphics library [218]:

$$\begin{aligned} t &= \left(\frac{\cos\theta_s - \cos\theta_u}{\cos\theta_p - \cos\theta_u}\right)^{\overline{+}}, \\ f_{\mathrm{dir_F}}(\mathbf{l}) &= t^2, \\ f_{\mathrm{dir_T}}(\mathbf{l}) &= \mathrm{smoothstep}(t) = t^2(3-2t). \end{aligned} \tag{5.18}$$

Recall that $x^{\overline{+}}$ is our notation for clamping x between 0 and 1, as introduced in Section 1.2. The smoothstep function is a cubic polynomial that is often used for smooth interpolation in shading. It is a built-in function in most shading languages.

Figure 5.8 shows some of the light types we have discussed so far.

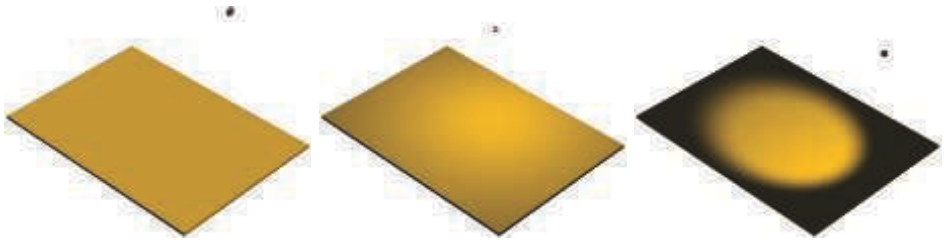

Figure 5.8. Some types of lights. From left to right: directional, point light with no falloff, and spotlight with a smooth transition. Note that the point light dims toward the edges due to the changing angle between the light and the surface.

Other Punctual Lights

There are many other ways in which the $\mathbf{c}_{\text{light}}$ value of a punctual light can vary.

The $f_{\text{dir}}(\mathbf{l})$ function is not limited to the simple spotlight falloff functions discussed above; it can represent any type of directional variation, including complex tabulated patterns measured from real-world light sources. The Illuminating Engineering Society (IES) have defined a standard file format for such measurements. IES profiles are available from many lighting manufacturers and have been used in the game *Killzone: Shadow Fall* [379, 380], as well as the Unreal [861] and Frostbite [960] game engines, among others. Lagarde gives a good summary [961] of issues relating to parsing and using this file format.

The game *Tomb Raider* (2013) [953] has a type of punctual light that applies independent falloff functions for distance along the x, y, and z world axes. In *Tomb Raider* curves can also be applied to vary light intensity over time, e.g., to produce a flickering torch.

In Section 6.9 we will discuss how light intensity and color can be varied via the use of textures.

5.2.3 Other Light Types

Directional and punctual lights are primarily characterized by how the light direction $\mathbf{l}$ is computed. Different types of lights can be defined by using other methods to compute the light direction. For example, in addition to the light types mentioned earlier, *Tomb Raider* also has capsule lights that use a line segment as the source instead of a point [953]. For each shaded pixel, the direction to the closest point on the line segment is used as the light direction $\mathbf{l}$.

As long as the shader has $\mathbf{l}$ and $\mathbf{c}_{\text{light}}$ values for use in evaluating the shading equation, any method can be used to compute those values.

The types of light discussed so far are abstractions. In reality, light sources have size and shape, and they illuminate surface points from multiple directions. In rendering, such lights are called *area lights*, and their use in real-time applications is steadily increasing. Area-light rendering techniques fall into two categories: those that simulate the softening of shadow edges that results from the area light being partially occluded (Section 7.1.2) and those that simulate the effect of the area light on surface shading (Section 10.1). This second category of lighting is most noticeable for smooth, mirror-like surfaces, where the light's shape and size can be clearly discerned in its reflection. Directional and punctual lights are unlikely to fall into disuse, though they are no longer as ubiquitous as in the past. Approximations accounting for a light's area have been developed that are relatively inexpensive to implement, and so are seeing wider use. Increased GPU performance also allows for more elaborate techniques than in the past.

5.3 Implementing Shading Models

To be useful, these shading and lighting equations must of course be implemented in code. In this section we will go over some key considerations for designing and writing such implementations. We will also walk through a simple implementation example.

5.3.1 Frequency of Evaluation

When designing a shading implementation, the computations need to be divided according to their *frequency of evaluation*. First, determine whether the result of a given computation is always constant over an entire draw call. In this case, the computation can be performed by the application, typically on the CPU, though a GPU compute shader could be used for especially costly computations. The results are passed to the graphics API via uniform shader inputs.

Even within this category, there is a broad range of possible frequencies of evaluation, starting from "once ever." The simplest such case would be a constant subexpression in the shading equation, but this could apply to any computation based on rarely changing factors such as the hardware configuration and installation options. Such shading computations might be resolved when the shader is compiled, in which case there is no need to even set a uniform shader input. Alternatively, the computation might be performed in an offline precomputation pass, at installation time, or when the application is loaded.

Another case is when the result of a shading computation changes over an application run, but so slowly that updating it every frame is not necessary. For example, lighting factors that depend on the time of day in a virtual game world. If the computation is costly, it may be worthwhile to amortize it over multiple frames.

Other cases include computations that are performed once per frame, such as concatenating the view and perspective matrices; or once per model, such as updating model lighting parameters that depend on location; or once per draw call, e.g., updating parameters for each material within a model. Grouping uniform shader inputs by frequency of evaluation is useful for application efficiency, and can also help GPU performance by minimizing constant updates [1165].

If the result of a shading computation changes within a draw call, it cannot be passed to the shader through a uniform shader input. Instead, it must be computed by one of the programmable shader stages described in Chapter 3 and, if needed, passed to other stages via varying shader inputs. In theory, shading computations can be performed on any of the programmable stages, each one corresponding to a different evaluation frequency:

- Vertex shader—Evaluation per pre-tessellation vertex.
- Hull shader—Evaluation per surface patch.
- Domain shader—Evaluation per post-tessellation vertex.

Figure 5.9. A comparison of per-pixel and per-vertex evaluations for the example shading model from Equation 5.19, shown on three models of varying vertex density. The left column shows the results of per-pixel evaluation, the middle column shows per-vertex evaluation, and the right column presents wireframe renderings of each model to show vertex density. *(Chinese Dragon mesh from Computer Graphics Archive [1172], original model from Stanford 3D Scanning Repository.)*

- Geometry shader—Evaluation per primitive.
- Pixel shader—Evaluation per pixel.

In practice most shading computations are performed per pixel. While these are typically implemented in the pixel shader, compute shader implementations are increasingly common; several examples will be discussed in Chapter 20. The other stages are primarily used for geometric operations such as transformation and deformation. To understand why this is the case, we will compare the results of per-vertex and per-pixel shading evaluations. In older texts, these are sometimes referred to as *Gouraud shading* [578] and *Phong shading* [1414], respectively, though those terms are not often used today. This comparison uses a shading model somewhat similar to the one in Equation 5.1, but modified to work with multiple light sources. The full model will be given a bit later, when we cover an example implementation in detail.

Figure 5.9 shows the results of per-pixel and per-vertex shading on models with a wide range of vertex densities. For the dragon, an extremely dense mesh, the difference between the two is small. But on the teapot, vertex shading evaluation causes visible errors such as angularly shaped highlights, and on the two-triangle plane the vertex-shaded version is clearly incorrect. The cause of these errors is that parts of the shading equation, the highlight in particular, have values that vary nonlinearly over

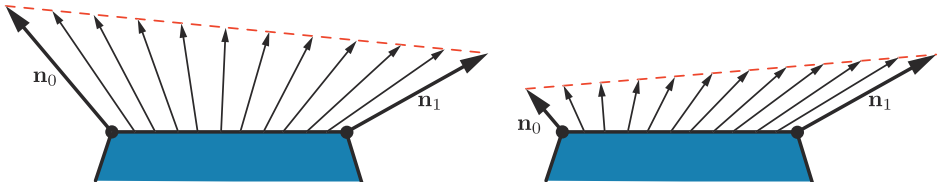

Figure 5.10. On the left, we see that linear interpolation of unit normals across a surface results in interpolated vectors with lengths less than one. On the right, we see that linear interpolation of normals with significantly different lengths results in interpolated directions that are skewed toward the longer of the two normals.

the mesh surface. This makes them a poor fit for the vertex shader, the results of which are interpolated linearly over the triangle before being fed to the pixel shader.

In principle, it would be possible to compute only the *specular highlight* part of the shading model in the pixel shader, and calculate the rest in the vertex shader. This would likely not result in visual artifacts and in theory would save some computation. In practice, this kind of hybrid implementation is often not optimal. The linearly varying parts of the shading model tend to be the least computationally costly, and splitting up the shading computation in this way tends to add enough overhead, such as duplicated computations and additional varying inputs, to outweigh any benefit.

As we mentioned earlier, in most implementations the vertex shader is responsible for non-shading operations such as geometry transformation and deformation. The resulting geometric surface properties, transformed into the appropriate coordinate system, are written out by the vertex shader, linearly interpolated over the triangle, and passed into the pixel shader as varying shader inputs. These properties typically include the position of the surface, the surface normal, and optionally surface tangent vectors, if needed for normal mapping.

Note that even if the vertex shader always generates unit-length surface normals, interpolation can change their length. See the left side of Figure 5.10. For this reason the normals need to be renormalized (scaled to length 1) in the pixel shader. However, the length of the normals generated by the vertex shader still matters. If the normal length varies significantly between vertices, e.g., as a side effect of vertex blending, this will skew the interpolation. This can be seen in the right side of Figure 5.10. Due to these two effects, implementations often normalize interpolated vectors before and after interpolation, i.e., in both the vertex and pixel shaders.

Unlike the surface normals, vectors that point toward specific locations, such as the view vector and the light vector for punctual lights, are typically not interpolated. Instead, the interpolated surface position is used to compute these vectors in the pixel shader. Other than the normalization, which as we have seen needs to be performed in the pixel shader in any case, each of these vectors is computed with a vector subtraction, which is quick. If for some reason it is necessary to interpolate these

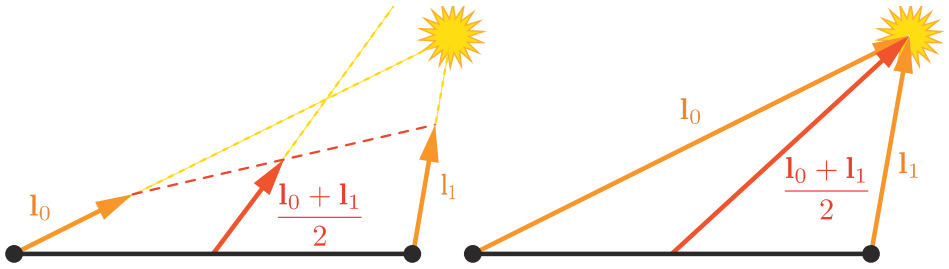

Figure 5.11. Interpolation between two light vectors. On the left, normalizing them before interpolation causes the direction to be incorrect after interpolation. On the right, interpolating the non-normalized vectors yields correct results.

vectors, do not normalize them beforehand. This will yield incorrect results, as shown in Figure 5.11.

Earlier we mentioned that the vertex shader transforms the surface geometry into "the appropriate coordinate system." The camera and light positions, passed to the pixel shader through uniform variables, are typically transformed by the application into the same coordinate system. This minimizes work done by the pixel shader to bring all the shading model vectors into the same coordinate space. But which coordinate system is the "appropriate" one? Possibilities include the global world space as well as the local coordinate system of the camera or, more rarely, that of the currently rendered model. The choice is typically made for the rendering system as a whole, based on systemic considerations such as performance, flexibility, and simplicity. For example, if rendered scenes are expected to include huge numbers of lights, world space might be chosen to avoid transforming the light positions. Alternately, camera space might be preferred, to better optimize pixel shader operations relating to the view vector and to possibly improve precision (Section 16.6).

Although most shader implementations, including the example implementation we are about to discuss, follow the general outline described above, there are certainly exceptions. For example, some applications choose the faceted appearance of per-primitive shading evaluation for stylistic reasons. This style is often referred to as *flat shading*. Two examples are shown in Figure 5.12.

In principle, flat shading could be performed in the geometry shader, but recent implementations typically use the vertex shader. This is done by associating each primitive's properties with its first vertex and disabling vertex value interpolation. Disabling interpolation (which can be done for each vertex value separately) causes the value from the first vertex to be passed to all pixels in the primitive.

5.3.2 Implementation Example

We will now present an example shading model implementation. As mentioned earlier, the shading model we are implementing is similar to the extended Gooch model from

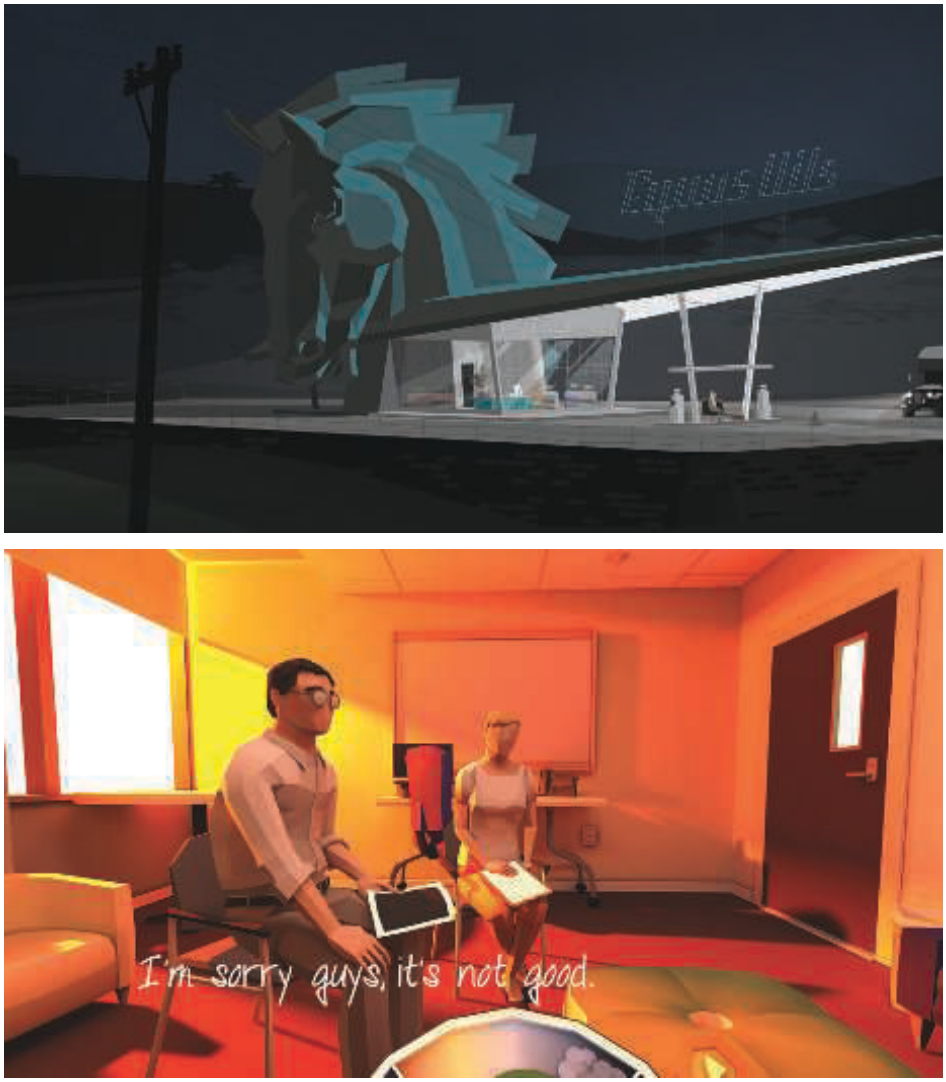

Figure 5.12. Two games that use flat shading as a stylistic choice: *Kentucky Route Zero*, top, and *That Dragon, Cancer*, bottom. *(Upper image courtesy of Cardboard Computer, lower courtesy of Numinous Games.)*

Equation 5.1, but modified to work with multiple light sources. It is described by

$$\mathbf{c}_{\text{shaded}} = \frac{1}{2}\mathbf{c}_{\text{cool}} + \sum_{i=1}^{n}(\mathbf{l}_i \cdot \mathbf{n})^{+}\mathbf{c}_{\text{light}_i}\left(s_i\,\mathbf{c}_{\text{highlight}} + (1 - s_i)\,\mathbf{c}_{\text{warm}}\right), \tag{5.19}$$

with the following intermediate calculations:

$$\begin{aligned}
\mathbf{c}_{\text{cool}} &= (0, 0, 0.55) + 0.25\,\mathbf{c}_{\text{surface}}, \\
\mathbf{c}_{\text{warm}} &= (0.3, 0.3, 0) + 0.25\,\mathbf{c}_{\text{surface}}, \\
\mathbf{c}_{\text{highlight}} &= (2, 2, 2), \\
\mathbf{r}_i &= 2\,(\mathbf{n} \cdot \mathbf{l}_i)\mathbf{n} - \mathbf{l}_i, \\
s_i &= \big(100\,(\mathbf{r}_i \cdot \mathbf{v}) - 97\big)^{\overline{+}}.
\end{aligned} \tag{5.20}$$

This formulation fits the multi-light structure in Equation 5.6, repeated here for convenience:

$$\mathbf{c}_{\text{shaded}} = f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) + \sum_{i=1}^{n} (\mathbf{l}_i \cdot \mathbf{n})^{+} \mathbf{c}_{\text{light}_i} f_{\text{lit}}(\mathbf{l}_i, \mathbf{n}, \mathbf{v}).$$

The lit and unlit terms in this case are

$$\begin{aligned}
f_{\text{unlit}}(\mathbf{n}, \mathbf{v}) &= \frac{1}{2}\mathbf{c}_{\text{cool}}, \\
f_{\text{lit}}(\mathbf{l}_i, \mathbf{n}, \mathbf{v}) &= s_i\,\mathbf{c}_{\text{highlight}} + (1 - s_i)\,\mathbf{c}_{\text{warm}},
\end{aligned} \tag{5.21}$$

with the cool color's unlit contribution adjusted to make results look more like the original equation.

In most typical rendering applications, varying values for material properties such as $\mathbf{c}_{\text{surface}}$ would be stored in vertex data or, more commonly, in textures (Chapter 6). However, to keep this example implementation simple, we will assume that $\mathbf{c}_{\text{surface}}$ is constant across the model.

This implementation will use the shader's dynamic branching capabilities to loop over all light sources. While this straightforward approach can work well for reasonably simple scenes, it does not scale well to large and geometrically complex scenes with many light sources. Rendering techniques to efficiently handle large light counts will be covered in Chapter 20. Also, in the interest of simplicity, we will only support one type of light source: point lights. Although the implementation is quite simple, it follows the best practices covered earlier.

Shading models are not implemented in isolation, but in the context of a larger rendering framework. This example is implemented inside a simple WebGL 2 application, modified from the "Phong-shaded Cube" WebGL 2 sample by Tarek Sherif [1623], but the same principles apply to more complex frameworks as well.

We will be discussing some samples of GLSL shader code and JavaScript WebGL calls from the application. The intent is not to teach the specifics of the WebGL API but to show general implementation principles. We will go through the implementation in "inside out" order, starting with the pixel shader, then the vertex shader, and finally the application-side graphics API calls.

Before the shader code proper, the shader source includes definitions of the shader inputs and outputs. As discussed earlier in Section 3.3, using GLSL terminology,

shader inputs fall into two categories. One is the set of *uniform* inputs, which have values set by the application and which remain constant over a draw call. The second type consists of *varying* inputs, which have values that can change between shader invocations (pixels or vertices). Here we see the definitions of the pixel shader's varying inputs, which in GLSL are marked `in`, as well as its outputs:

```
in vec3 vPos;
in vec3 vNormal;
out vec4 outColor;
```

This pixel shader has a single output, which is the final shaded color. The pixel shader inputs match the vertex shader outputs, which are interpolated over the triangle before being fed into the pixel shader. This pixel shader has two varying inputs: surface position and surface normal, both in the application's world-space coordinate system. The number of uniform inputs is much larger, so for brevity we will only show the definitions of two, both related to light sources:

```
struct Light {
    vec4 position;
    vec4 color;
};
uniform LightUBlock {
    Light uLights[MAXLIGHTS];
};

uniform uint uLightCount;
```

Since these are point lights, the definition for each one includes a position and a color. These are defined as `vec4` instead of `vec3` to conform to the restrictions of the GLSL `std140` data layout standard. Although, as in this case, the `std140` layout can lead to some wasted space, it simplifies the task of ensuring consistent data layout between CPU and GPU, which is why we use it in this sample. The array of `Light` structs is defined inside a named uniform block, which is a GLSL feature for binding a group of uniform variables to a buffer object for faster data transfer. The array length is defined to be equal to the maximum number of lights that the application allows in a single draw call. As we will see later, the application replaces the `MAXLIGHTS` string in the shader source with the correct value (10 in this case) before shader compilation. The uniform integer `uLightCount` is the actual number of active lights in the draw call.

Next, we will take a look at the pixel shader code:

```
vec3 lit(vec3 l, vec3 n, vec3 v) {
    vec3 r_l = reflect(-l, n);
    float s = clamp(100.0 * dot(r_l, v) - 97.0, 0.0, 1.0);
    vec3 highlightColor = vec3(2,2,2);
    return mix(uWarmColor, highlightColor, s);
}

void main() {
    vec3 n = normalize(vNormal);
    vec3 v = normalize(uEyePosition.xyz - vPos);
```

```
    outColor = vec4(uFUnlit, 1.0);

    for (uint i = 0u; i < uLightCount; i++) {
        vec3 l = normalize(uLights[i].position.xyz - vPos);
        float NdL = clamp(dot(n, l), 0.0, 1.0);
        outColor.rgb += NdL * uLights[i].color.rgb * lit(l,n,v);
    }
}
```

We have a function definition for the `lit` term, which is called by the `main()` function. Overall, this is a straightforward GLSL implementation of Equations 5.20 and 5.21. Note that the values of $f_{\text{unlit}}()$ and $\mathbf{c}_{\text{warm}}$ are passed in as uniform variables. Since these are constant over the entire draw call, the application can compute these values, saving some GPU cycles.

This pixel shader uses several built-in GLSL functions. The `reflect()` function reflects one vector, in this case the light vector, in the plane defined by a second vector, in this case the surface normal. Since we want both the light vector and reflected vector to point away from the surface, we need to negate the former before passing it into `reflect()`. The `clamp()` function has three inputs. Two of them define a range to which the third input is clamped. The special case of clamping to the range between 0 and 1 (which corresponds to the HLSL `saturate()` function) is quick, often effectively free, on most GPUs. This is why we use it here, although we only need to clamp the value to 0, as we know it will not exceed 1. The function `mix()` also has three inputs and linearly interpolates between two of them, the warm color and the highlight color in this case, based on the value of the third, a mixing parameter between 0 and 1. In HLSL this function is called `lerp()`, for "linear interpolation." Finally, `normalize()` divides a vector by its length, scaling it to a length of 1.

Now let us look at the vertex shader. We will not show any of its uniform definitions since we already saw some example uniform definitions for the pixel shader, but the varying input and output definitions are worth examining:

```
layout(location=0) in vec4 position;
layout(location=1) in vec4 normal;
out vec3 vPos;
out vec3 vNormal;
```

Note that, as mentioned earlier, the vertex shader outputs match the pixel shader varying inputs. The inputs include directives that specify how the data are laid out in the vertex array. The vertex shader code comes next:

```
void main() {
    vec4 worldPosition = uModel * position;
    vPos = worldPosition.xyz;
    vNormal = (uModel * normal).xyz;
    gl_Position = viewProj * worldPosition;
}
```

These are common operations for a vertex shader. The shader transforms the surface position and normal into world space and passes them to the pixel shader

for use in shading. Finally, the surface position is transformed into clip space and passed into `gl_Position`, a special system-defined variable used by the rasterizer. The `gl_Position` variable is the one required output from any vertex shader.

Note that the normal vectors are not normalized in the vertex shader. They do not need to be normalized since they have a length of 1 in the original mesh data and this application does not perform any operations, such as vertex blending or nonuniform scaling, that could change their length unevenly. The model matrix could have a uniform scale factor, but that would change the length of all normals proportionally and thus not result in the problem shown on the right side of Figure 5.10.

The application uses the WebGL API for various rendering and shader setup. Each of the programmable shader stages are set up individually, and then they are all bound to a program object. Here is the pixel shader setup code:

```
var fSource = document.getElementById("fragment").text.trim();

var maxLights = 10;
fSource = fSource.replace(/MAXLIGHTS/g, maxLights.toString());

var fragmentShader = gl.createShader(gl.FRAGMENT_SHADER);
gl.shaderSource(fragmentShader, fSource);
gl.compileShader(fragmentShader);
```

Note the "fragment shader" references. This term is used by WebGL (and OpenGL, on which it is based). As noted earlier in this book, although "pixel shader" is less precise in some ways, it is the more common usage, which we follow in this book. This code is also where the `MAXLIGHTS` string is replaced with the appropriate numerical value. Most rendering frameworks perform similar pre-compilation shader manipulations.

There is more application-side code for setting uniforms, initializing vertex arrays, clearing, drawing, and so on, which you can view in the program [1623] and which are explained by numerous API guides. Our goal here is to give a sense of how shaders are treated as separate processors, with their own programming environment. We thus end our walkthrough at this point.

5.3.3 Material Systems

Rendering frameworks rarely implement just a single shader, as in our simple example. Typically, a dedicated system is needed to handle the variety of materials, shading models, and shaders used by the application.

As explained in earlier chapters, a shader is a program for one of the GPU's programmable shader stages. As such, it is a low-level graphics API resource and not something with which artists would interact directly. In contrast, a *material* is an artist-facing encapsulation of the visual appearance of a surface. Materials sometimes also describe non-visual aspects, such as collision properties, which we will not discuss further because they are outside the scope of this book.

While materials are implemented via shaders, this is not a simple one-to-one correspondence. In different rendering situations, the same material may use different shaders. A shader can also be shared by multiple materials. The most common case is parameterized materials. In its simplest form, material parameterization requires two types of material entities: *material templates* and *material instances*. Each material template describes a class of materials and has a set of parameters that can be assigned numerical, color, or texture values depending on the parameter type. Each material instance corresponds to a material template plus a specific set of values for all of its parameters. Some rendering frameworks such as the Unreal Engine [1802] allow for a more complex, hierarchical structure, with material templates deriving from other templates at multiple levels.

Parameters may be resolved at runtime, by passing uniform inputs to the shader program, or at compile time, by substituting values before the shader is compiled. A common type of compile-time parameter is a boolean switch that controls the activation of a given material feature. This can be set by artists via a checkbox in the material user interface or procedurally by the material system, e.g., to reduce shader cost for distant objects where the visual effect of the feature is negligible.

While the material parameters may correspond one-to-one with the parameters of the shading model, this is not always the case. A material may fix the value of a given shading model parameter, such as the surface color, to a constant value. Alternately, a shading model parameter may be computed as the result of a complex series of operations taking multiple material parameters, as well as interpolated vertex or texture values, as inputs. In some cases, parameters such as surface position, surface orientation, and even time may also factor into the calculation. Shading based on surface position and orientation is especially common in terrain materials. For example, the height and surface normal can be used to control a snow effect, blending in a white surface color on high-altitude horizontal and almost-horizontal surfaces. Time-based shading is common in animated materials, such as a flickering neon sign.

One of the most important tasks of a material system is dividing various shader functions into separate elements and controlling how these are combined. There are many cases where this type of composition is useful, including the following:

- Composing surface shading with geometric processing, such as rigid transforms, vertex blending, morphing, tessellation, instancing, and clipping. These bits of functionality vary independently: Surface shading depends on the material, and geometry processing depends on the mesh. So, it is convenient to author them separately and have the material system compose them as needed.

- Composing surface shading with compositing operations such as pixel discard and blending. This is particularly relevant to mobile GPUs, where blending is typically performed in the pixel shader. It is often desirable to select these operations independently of the material used for surface shading.

- Composing the operations used to compute the shading model parameters with the computation of the shading model itself. This allows authoring the shading model implementation once and reusing it in combination with various different methods for computing the shading model parameters.
- Composing individually selectable material features with each other, the selection logic, and the rest of the shader. This enables writing the implementation of each feature separately.
- Composing the shading model and computation of its parameters with light source evaluation: computing the values of $\mathbf{c}_{\text{light}}$ and $\mathbf{l}$ at the shaded point for each light source. Techniques such as deferred rendering (discussed in Chapter 20) change the structure of this composition. In rendering frameworks that support multiple such techniques, this adds an additional layer of complexity.

It would be convenient if the graphics API provided this type of shader code modularity as a core feature. Sadly, unlike CPU code, GPU shaders do not allow for post-compilation linking of code fragments. The program for each shader stage is compiled as a unit. The separation between shader stages does offer some limited modularity, which somewhat fits the first item on our list: composing surface shading (typically performed in the pixel shader) with geometric processing (typically performed in other shader stages). But the fit is not perfect, since each shader performs other operations as well, and the other types of composition still need to be handled. Given these limitations, the only way that the material system can implement all these types of composition is at the source-code level. This primarily involves string operations such as concatenation and replacement, often performed via C-style preprocessing directives such as `#include`, `#if`, and `#define`.

Early rendering systems had a relatively small number of shader variants, and often each one was written manually. This has some benefits. For example, each variant can be optimized with full knowledge of the final shader program. However, this approach quickly becomes impractical as the number of variants grows. When taking all the different parts and options into account, the number of possible different shader variants is huge. This is why modularity and composability are so crucial.

The first question to be resolved when designing a system for handling shader variants is whether selection between different options is performed at runtime via dynamic branching, or at compile time via conditional preprocessing. On older hardware, dynamic branching was often impossible or extremely slow, so runtime selection was not an option. Variants were then all handled at compile time, including all possible combinations of counts of the different light types [1193].

In contrast, current GPUs handle dynamic branching quite well, especially when the branch behaves the same for all pixels in a draw call. Today much of the functionality variation, such as the number of lights, is handled at runtime. However, adding a large amount of functional variation to a shader incurs a different cost: an increase in register count and a corresponding reduction in occupancy, and thus performance.

See Section 18.4.5 for more details. So, compile-time variation is still valuable. It avoids including complex logic that will never be executed.

As an example, let us imagine an application that supports three different types of lights. Two light types are simple: point and directional. The third type is a generalized spotlight that supports tabulated illumination patterns and other complex features, requiring a significant amount of shader code to implement. However, say the generalized spotlight is used relatively rarely, with less than 5% of the lights in the application being this type. In the past, a separate shader variant would be compiled for each possible combination of counts of the three light types, to avoid dynamic branching. While this would not be needed today, it may still be beneficial to compile two separate variants, one for the case when the count of generalized spotlights is equal to or greater than 1, and one for the case where the count of such lights is exactly 0. Due to its simpler code, the second variant (which is most commonly used) is likely to have lower register occupancy and thus higher performance.

Modern material systems employ both runtime and compile-time shader variation. Even though the full burden is no longer handled only at compile time, the overall complexity and number of variations keep increasing, so a large number of shader variants still need to be compiled. For example, in some areas of the game *Destiny: The Taken King*, over 9000 compiled shader variations were used in a single frame [1750]. The number of possible variations can be much larger, e.g., the Unity rendering system has shaders with close to 100 billion possible variants. Only the variants that are actually used are compiled, but the shader compilation system had to be redesigned to handle the huge number of possible variants [1439].

Material-system designers employ different strategies to address these design goals. Although these are sometimes presented as mutually exclusive system architectures [342], these strategies can be—and usually are—combined in the same system. These strategies include the following:

- Code reuse—Implementing functions in shared files, using `#include` preprocessor directives to access those functions from any shader that needs them.
- Subtractive—A shader, often referred to as an *übershader* or *supershader* [1170, 1784], that aggregates a large set of functionality, using a combination of compile-time preprocessor conditionals and dynamic branching to remove unused parts and to switch between mutually exclusive alternatives.
- Additive—Various bits of functionality are defined as nodes with input and output connectors, and these are composed together. This is similar to the code reuse strategy but is more structured. The composition of nodes can be done via text [342] or a visual graph editor. The latter is intended to make it easier for non-engineers, such as technical artists, to author new material templates [1750, 1802]. Typically only part of the shader is accessible to visual graph authoring. For example, in the Unreal Engine the graph editor can only affect the computation of shading model inputs [1802]. See Figure 5.13.

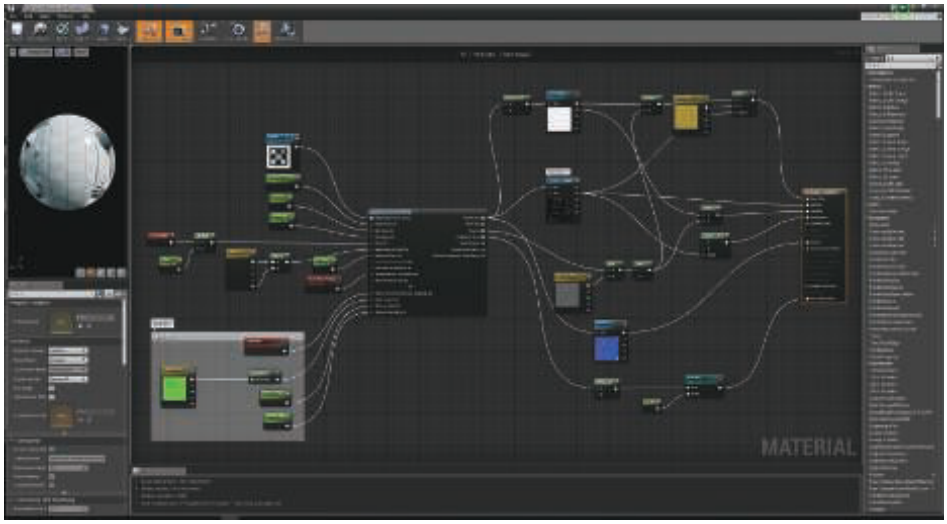

Figure 5.13. The Unreal Engine material editor. Note the tall node on the right side of the node graph. The input connectors of this node correspond to various shading inputs used by the rendering engine, including all the shading model parameters. *(Material sample courtesy of Epic Games.)*

- Template-based—An interface is defined, into which different implementations can be plugged as long as they conform to that interface. This is a bit more formal than the additive strategy and is typically used for larger chunks of functionality. A common example for such an interface is the separation between the calculation of shading model parameters and the computation of the shading model itself. The Unreal Engine [1802] has different "material domains," including the Surface domain for computing shading model parameters and the Light Function domain for computing a scalar value that modulates $\mathbf{c}_{\text{light}}$ for a given light source. A similar "surface shader" structure also exists in Unity [1437]. Note that deferred shading techniques (discussed in Chapter 20) enforce a similar structure, with the G-buffer serving as the interface.

For more specific examples, several chapters in the (now free) book *WebGL Insights* [301] discuss how a variety of engines control their shader pipelines. Besides composition, there are several other important design considerations for modern material systems, such as the need to support multiple platforms with minimal duplication of shader code. This includes variations in functionality to account for performance and capability differences among platforms, shading languages, and APIs. The *Destiny* shader system [1750] is a representative solution to this type of problem. It uses a proprietary preprocessor layer that takes shaders written in a custom shading language dialect. This allows writing platform-independent materials with automatic translation to different shading languages and implementations. The Unreal Engine [1802] and Unity [1436] have similar systems.