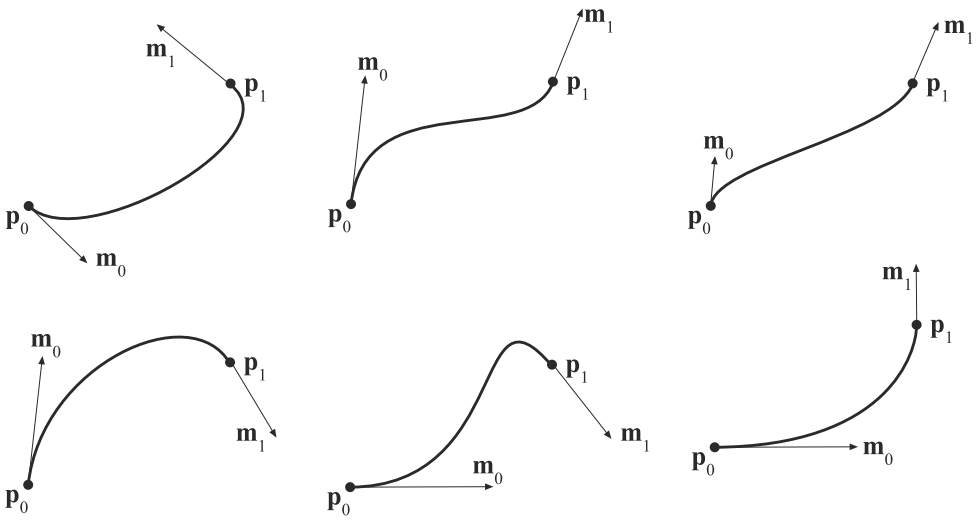

Figure 17.12. Hermite interpolation. A curve is defined by two points, $\mathbf{p}_0$ and $\mathbf{p}_1$, and a tangent, $\mathbf{m}_0$ and $\mathbf{m}_1$, at each point.

of the tangents give different results; longer tangents have a greater impact on the overall shape.

Cubic Hermite interpolation is used to render the hair in the Nalu demo [1274]. See Figure 17.2. A coarse control hair is used for animation and collision detection, tangents are computed, and cubic curves are tessellated and rendered.

17.1.5 Kochanek-Bartels Curves

When interpolating between more than two points, you can connect several Hermite curves. However, when doing this, there are degrees of freedom in selecting the shared tangents that provide different characteristics. Here, we will present one way to compute such tangents, called Kochanek-Bartels curves. Assume that we have n points, $\mathbf{p}_0, \ldots, \mathbf{p}_{n-1}$, which should be interpolated with $n-1$ Hermite curve segments. We assume that there is only one tangent at each point, and we start to look at the "inner" tangents, $\mathbf{m}_1, \ldots, \mathbf{m}_{n-2}$. A tangent at $\mathbf{p}_i$ can be computed as a combination of the two chords [917]: $\mathbf{p}_i - \mathbf{p}_{i-1}$, and $\mathbf{p}_{i+1} - \mathbf{p}_i$, as shown at the left in Figure 17.13.

First, a tension parameter, a, is introduced that modifies the length of the tangent vector. This controls how sharp the curve is going to be at the joint. The tangent is computed as

$$\mathbf{m}_i = \frac{1-a}{2}\big((\mathbf{p}_i - \mathbf{p}_{i-1}) + (\mathbf{p}_{i+1} - \mathbf{p}_i)\big). \tag{17.19}$$

The top row at the right in Figure 17.13 shows different tension parameters. The default value is $a = 0$; higher values give sharper bends (if $a > 1$, there will be a

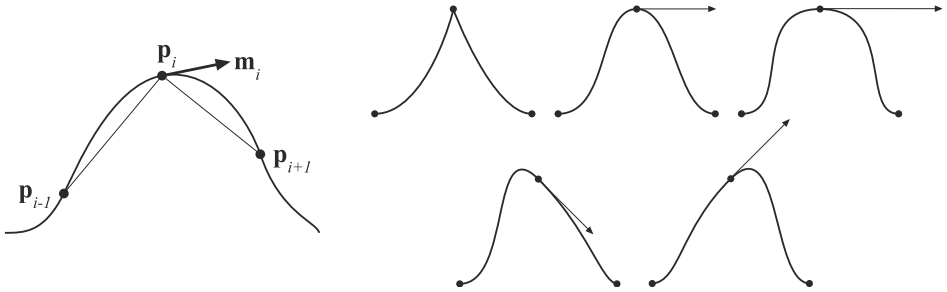

Figure 17.13. One method of computing the tangents is to use a combination of the chords (left). The upper row at the right shows three curves with different tension parameters (a). The left curve has $a \approx 1$, which means high tension; the middle curve has $a \approx 0$, which is default tension; and the right curve has $a \approx -1$, which is low tension. The bottom row of two curves at the right shows different bias parameters. The curve on the left has a negative bias, and the right curve has a positive bias.

loop at the joint), and negative values give less taut curves near the joints. Second, a bias parameter, b, is introduced that influences the direction of the tangent (and, indirectly, the length of the tangent). Using both tension and bias gives us

$$\mathbf{m}_i = \frac{(1-a)(1+b)}{2}(\mathbf{p}_i - \mathbf{p}_{i-1}) + \frac{(1-a)(1-b)}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i), \tag{17.20}$$

where the default value is $b = 0$. A positive bias gives a bend that is more directed toward the chord $\mathbf{p}_i - \mathbf{p}_{i-1}$, and a negative bias gives a bend that is more directed toward the other chord: $\mathbf{p}_{i+1} - \mathbf{p}_i$. This is shown in the bottom row on the right in Figure 17.13. The user can either set the tension and bias parameters or let them have their default values, which produces what is often called a Catmull-Rom spline [236]. The tangents at the first and the last points can also be computed with these formulae, where one of the chords is simply set to a length of zero.

Yet another parameter that controls the behavior at the joints can be incorporated into the tangent equation [917]. However, this requires the introduction of two tangents at each joint, one incoming, denoted $\mathbf{s}_i$ (for source) and one outgoing, denoted $\mathbf{d}_i$ (for destination). See Figure 17.14. Note that the curve segment between $\mathbf{p}_i$ and $\mathbf{p}_{i+1}$ uses the tangents $\mathbf{d}_i$ and $\mathbf{s}_{i+1}$. The tangents are computed as below, where c is the *continuity* parameter:

$$\begin{aligned} \mathbf{s}_i &= \frac{1-c}{2}(\mathbf{p}_i - \mathbf{p}_{i-1}) + \frac{1+c}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i), \\ \mathbf{d}_i &= \frac{1+c}{2}(\mathbf{p}_i - \mathbf{p}_{i-1}) + \frac{1-c}{2}(\mathbf{p}_{i+1} - \mathbf{p}_i). \end{aligned} \tag{17.21}$$

Again, $c = 0$ is the default value, which makes $\mathbf{s}_i = \mathbf{d}_i$. Setting $c = -1$ gives $\mathbf{s}_i = \mathbf{p}_i - \mathbf{p}_{i-1}$, and $\mathbf{d}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$, producing a sharp corner at the joint, which is only C^0. Increasing the value of c makes $\mathbf{s}_i$ and $\mathbf{d}_i$ more and more alike. For $c = 0$,

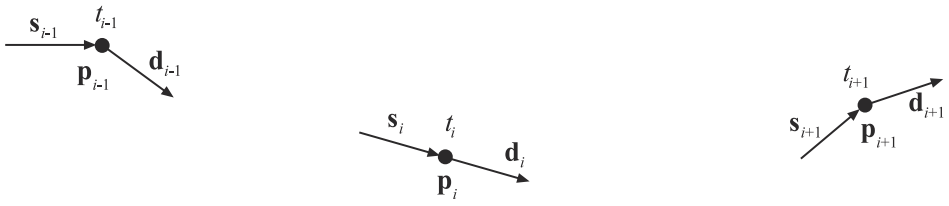

Figure 17.14. Incoming and outgoing tangents for Kochanek-Bartels curves. At each control point $\mathbf{p}_i$, its time t_i is also shown, where $t_i > t_{i-1}$, for all i.

then $\mathbf{s}_i = \mathbf{d}_i$. When $c = 1$ is reached, we get $\mathbf{s}_i = \mathbf{p}_{i+1} - \mathbf{p}_i$, and $\mathbf{d}_i = \mathbf{p}_i - \mathbf{p}_{i-1}$. Thus, the continuity parameter c is another way to give even more control to the user, and it makes it possible to get sharp corners at the joints, if desired.

The combination of tension, bias, and continuity, where the default parameter values are $a = b = c = 0$, is

$$\begin{aligned}\mathbf{s}_i &= \frac{(1-a)(1+b)(1-c)}{2}(\mathbf{p}_i-\mathbf{p}_{i-1}) + \frac{(1-a)(1-b)(1+c)}{2}(\mathbf{p}_{i+1}-\mathbf{p}_i),\\ \mathbf{d}_i &= \frac{(1-a)(1+b)(1+c)}{2}(\mathbf{p}_i-\mathbf{p}_{i-1}) + \frac{(1-a)(1-b)(1-c)}{2}(\mathbf{p}_{i+1}-\mathbf{p}_i).\end{aligned} \tag{17.22}$$

Both Equations 17.20 and 17.22 work only when all curve segments are using the same time interval length. To account for different time length of the curve segments, the tangents have to be adjusted, similar to what was done in Section 17.1.3. The adjusted tangents, denoted $\mathbf{s}'_i$ and $\mathbf{d}'_i$, are

$$\mathbf{s}'_i = \mathbf{s}_i \frac{2\Delta_i}{\Delta_{i-1} + \Delta_i} \quad \text{and} \quad \mathbf{d}'_i = \mathbf{d}_i \frac{2\Delta_{i-1}}{\Delta_{i-1} + \Delta_i}, \tag{17.23}$$

where $\Delta_i = t_{i+1} - t_i$.

17.1.6 B-Splines

Here, we will provide a brief introduction to the topic of B-splines, and we will focus in particular on cubic uniform B-splines. In general, a *B-spline* is quite similar to a Bézier curve, and can be expressed as a function of t (using shifted basis functions), β_n (weighted by control points), and c_k: e.g.,

$$s_n(t) = \sum_k c_k \beta_n(t - k). \tag{17.24}$$

In this case, this is a curve where t is the x-axis and $s_n(t)$ is the y-axis, and the control points are simply evenly spaced y-values. For much more extensive coverage, see the texts by the Killer B's [111], Farin [458], and Hoschek and Lasser [777].

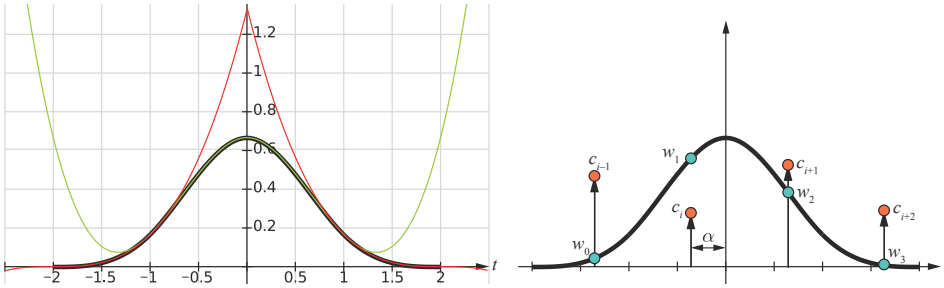

Figure 17.15. Left: the $\beta_3(t)$ basis function is shown as a fat black curve, which is constructed from two piecewise cubic functions (red and green). The green curve is used when $|t| < 1$, the red curve when $1 \leq |t| < 2$, and the curve is zero elsewhere. Right: to create a curve segment using four control points, c_k, $k \in \{i-1, i, i+1, i+2\}$, we will only obtain a curve between the t-coordinate of c_i and of $c_i + 1$. The α is fed into the w functions to evaluate the basis function, and these values are then multiplied by the corresponding control point. Finally, all values are added together, which gives us a point on the curve. See Figure 17.16. *(Illustration on the right after Ruijters et al. [1518].)*

Here, we will follow the presentation by Ruijters et al. [1518] and present the special case of a uniform cubic B-spline. The basis function, $\beta_3(t)$, is stitched together by three pieces:

$$\beta_3(t) = \begin{cases} 0, & |t| \geq 2, \\ \frac{1}{6}(2 - |t|)^3, & 1 \leq |t| < 2, \\ \frac{2}{3} - \frac{1}{2}|t|^2(2 - |t|), & |t| < 1. \end{cases} \tag{17.25}$$

The construction of this basis function is shown on the left in Figure 17.15. This function has C^2 continuity everywhere, which means that if several B-spline curve segments are stitched together, the composite curve will also be C^2. A cubic curve has C^2 continuity, and in general, a curve of degree n has C^{n-1} continuity. In general, the basis functions are created as follows. The $\beta_0(t)$ is a "square" function, i.e., it is 1 if $|t| < 0.5$, it is 0.5 if $|t| = 0.5$, and it is 0 elsewhere. The next basis function, $\beta_1(t)$ is created by integrating $\beta_0(t)$, which gives us a tent function. The basis function after that is created by integrating $\beta_1(t)$, which gives a smoother function, which is C^1. This process is repeated to get C^2, and so on.

How to evaluate a curve segment is illustrated on the right in Figure 17.15, and its formula is

$$s_3(i + \alpha) = w_0(\alpha)c_{i-1} + w_1(\alpha)c_i + w_2(\alpha)c_{i+1} + w_3(\alpha)c_{i+2}. \tag{17.26}$$

Note that only four control points will be used at any time, and this means that the curve has local support, i.e., a limited number of control points is needed. The functions $w_k(\alpha)$ are defined using $\beta_3()$ as

$$\begin{aligned} w_0(\alpha) &= \beta_3(-\alpha - 1), & w_1(\alpha) &= \beta_3(-\alpha), \\ w_2(\alpha) &= \beta_3(1 - \alpha), & w_3(\alpha) &= \beta_3(2 - \alpha). \end{aligned} \tag{17.27}$$

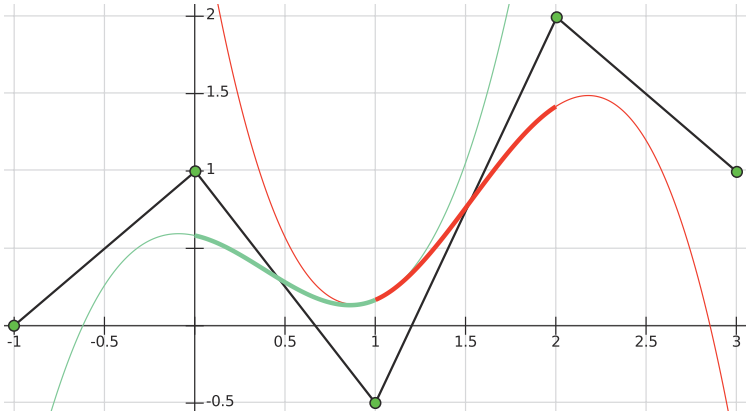

Figure 17.16. The control points c_k (green circles) define a uniform cubic spline in this example. Only the two fat curves are part of the piecewise B-spline curve. The left (green) curve is defined by the four leftmost control points, and the right (red) curve is defined by the four rightmost control points. The curves meet at $t = 1$ with C^2 continuity.

Ruijters et al. [1518] show that these can be rewritten as

$$\begin{aligned} w_0(\alpha) &= \frac{1}{6}(1-\alpha)^3, & w_1(\alpha) &= \frac{2}{3} - \frac{1}{2}\alpha^2(2-\alpha), \\ w_2(\alpha) &= \frac{2}{3} - \frac{1}{2}(1-\alpha)^2(1+\alpha), & w_3(\alpha) &= \frac{1}{6}\alpha^3. \end{aligned} \tag{17.28}$$

In Figure 17.16, we show the results of stitching two uniform cubic B-spline curves together as one. A major advantage is that the curves are continuous with the same continuity as the basis functions, $\beta(t)$, which are C^2 in the case of a cubic B-spline. As can be seen in the figure, there is no guarantee that the curve will go through any of the control points. Note that we can also create a B-spline for the x-coordinates, which would give a general curve in the plane (and not just a function). The resulting two-dimensional points would then be $(s_3^x(i+\alpha), s_3^y(i+\alpha))$, i.e., simply two different evaluations of Equation 17.26, one for x and one for y.

We have shown how to use only B-splines that are uniform. If the spacing between the control points is nonuniform, the equations become a bit more elaborate but more flexible [111, 458, 777].

17.2 Parametric Curved Surfaces

A natural extension of parametric curves is parametric surfaces. An analogy is that a triangle or polygon is an extension of a line segment, in which we go from one to two

dimensions. Parametric surfaces can be used to model objects with curved surfaces. A parametric surface is defined by a small number of control points. Tessellation of a parametric surface is the process of evaluating the surface representation at several positions, and connecting these to form triangles that approximate the true surface. This is done because graphics hardware can efficiently render triangles. At runtime, the surface can then be tessellated into as many triangles as desired. Thus, parametric surfaces are perfect for making a trade-off between quality and speed, since more triangles take more time to render, but give better shading and silhouettes. Another advantage of parametric surfaces is that the control points can be animated and then the surface can be tessellated. This is in contrast to animating a large triangle mesh directly, which can be more expensive.

This section starts by introducing *Bézier patches*, which are curved surfaces with rectangular domains. These are also called *tensor-product Bézier surfaces*. Then *Bézier triangles* are presented, which have triangular domains, followed by a discussion about continuity in Section 17.2.3. In Sections 17.2.4 and 17.2.5, two methods are presented that replace each input triangle with a Bézier triangle. These techniques are called PN triangles and Phong tessellation, respectively. Finally, B-spline patches are presented in Section 17.2.6.

17.2.1 Bézier Patches

The concept of Bézier curves, introduced in Section 17.1.1, can be extended from using one parameter to using two parameters, thus forming surfaces instead of curves. Let us start with extending linear interpolation to *bilinear interpolation*. Now, instead of just using two points, we use four points, called $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$, and $\mathbf{d}$, as shown in Figure 17.17. Instead of using one parameter called t, we now use two parameters (u, v). Using u to linearly interpolate $\mathbf{a}$ & $\mathbf{b}$ and $\mathbf{c}$ & $\mathbf{d}$ gives $\mathbf{e}$ and $\mathbf{f}$:

$$\mathbf{e} = (1-u)\mathbf{a} + u\mathbf{b}, \qquad \mathbf{f} = (1-u)\mathbf{c} + u\mathbf{d}. \tag{17.29}$$

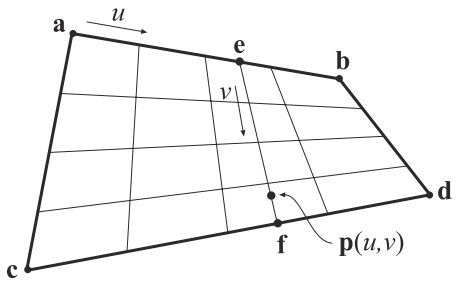

Figure 17.17. Bilinear interpolation using four points.

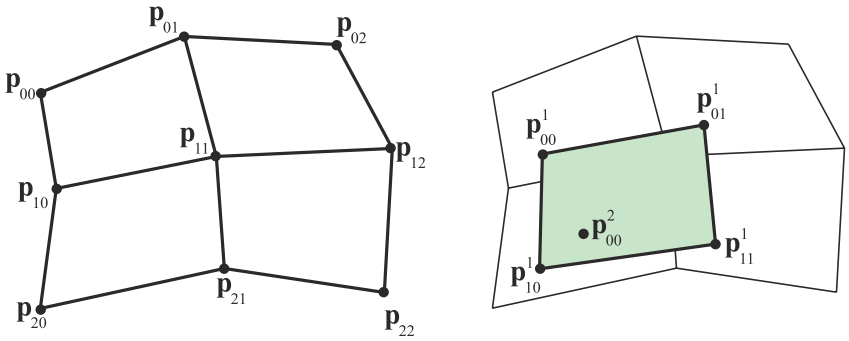

Figure 17.18. Left: a biquadratic Bézier surface, defined by nine control points, $\mathbf{p}_{ij}$. Right: to generate a point on the Bézier surface, four points $\mathbf{p}^1_{ij}$ are first created using bilinear interpolation from the nearest control points. Finally, the point surface $\mathbf{p}(u,v) = \mathbf{p}^2_{00}$ is bilinearly interpolated from these created points.

Next, the linearly interpolated points, $\mathbf{e}$ and $\mathbf{f}$, are linearly interpolated in the other direction, using v. This yields bilinear interpolation:

$$\begin{aligned}\mathbf{p}(u,v) &= (1-v)\mathbf{e} + v\mathbf{f} \\ &= (1-u)(1-v)\mathbf{a} + u(1-v)\mathbf{b} + (1-u)v\mathbf{c} + uv\mathbf{d}.\end{aligned} \tag{17.30}$$

Note that this is the same type of equation used for bilinear interpolation for texture mapping (Equation 6.1 on page 179). Equation 17.30 describes the simplest nonplanar parametric surface, where different points on the surface are generated using different values of (u,v). The domain, i.e., the set of valid values, is $(u,v) \in [0,1] \times [0,1]$, which means that both u and v should belong to $[0,1]$. When the domain is rectangular, the resulting surface is often called a *patch*.

To extend a Bézier curve from linear interpolation, more points were added and the interpolation repeated. The same strategy can be used for patches. Assume nine points, arranged in a 3×3 grid, are used. This is shown in Figure 17.18, where the notation is shown as well. To form a biquadratic Bézier patch from these points, we first need to bilinearly interpolate four times to create four intermediate points, also shown in Figure 17.18. Next, the final point on the surface is bilinearly interpolated from the previously created points.

The repeated bilinear interpolation described above is the extension of de Casteljau's algorithm to patches. At this point we need to define some notation. The degree of the surface is n. The control points are $\mathbf{p}_{i,j}$, where i and j belong to $[0 \ldots n]$. Thus, $(n+1)^2$ control points are used for a patch of degree n. Note that the control points should be superscripted with a zero, i.e., $\mathbf{p}^0_{i,j}$, but this is often omitted, and sometimes we use the subscript $_{ij}$ instead of $_{i,j}$ when there can be no confusion. The Bézier patch using de Casteljau's algorithm is described in the equation that follows:

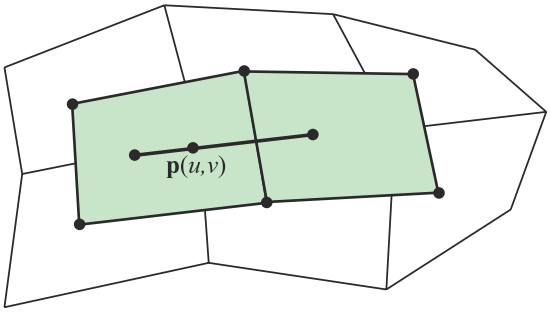

Figure 17.19. Different degrees in different directions.

de Casteljau [patches]:

$$\begin{aligned}\mathbf{p}_{i,j}^{k}(u,v) &= (1-u)(1-v)\mathbf{p}_{i,j}^{k-1} + u(1-v)\mathbf{p}_{i,j+1}^{k-1} + (1-u)v\mathbf{p}_{i+1,j}^{k-1} + uv\mathbf{p}_{i+1,j+1}^{k-1} \\ & k = 1\ldots n, \quad i = 0\ldots n-k, \quad j = 0\ldots n-k.\end{aligned} \tag{17.31}$$

Similar to the Bézier curve, the point at (u, v) on the Bézier patch is $\mathbf{p}_{0,0}^{n}(u, v)$. The Bézier patch can also be described in Bernstein form using Bernstein polynomials, as shown in Equation 17.32:

Bernstein [patches]:

$$\begin{aligned}\mathbf{p}(u,v) &= \sum_{i=0}^{m} B_i^m(u) \sum_{j=0}^{n} B_j^n(v)\mathbf{p}_{i,j} = \sum_{i=0}^{m}\sum_{j=0}^{n} B_i^m(u)B_j^n(v)\mathbf{p}_{i,j}, \\ &= \sum_{i=0}^{m}\sum_{j=0}^{n} \binom{m}{i}\binom{n}{j} u^i(1-u)^{m-i}v^j(1-v)^{n-j}\mathbf{p}_{i,j}.\end{aligned} \tag{17.32}$$

Note that in Equation 17.32, there are two parameters, m and n, for the degree of the surface. The "compound" degree is sometimes denoted $m \times n$. Most often $m = n$, which simplifies the implementation a bit. The consequence of, say, $m > n$ is to first bilinearly interpolate n times, and then linearly interpolate $m - n$ times. This is shown in Figure 17.19. A different interpretation of Equation 17.32 is found by rewriting it as

$$\mathbf{p}(u,v) = \sum_{i=0}^{m} B_i^m(u) \sum_{j=0}^{n} B_j^n(v)\mathbf{p}_{i,j} = \sum_{i=0}^{m} B_i^m(u)\mathbf{q}_i(v). \tag{17.33}$$

Here, $\mathbf{q}_i(v) = \sum_{j=0}^{n} B_j^n(v)\mathbf{p}_{i,j}$ for $i = 0\ldots m$. As can be seen in the bottom row in Equation 17.33, this is just a Bézier curve when we fix a v-value. Assuming $v = 0.35$, the points $\mathbf{q}_i(0.35)$ can be computed from a Bézier curve, and then Equation 17.33 describes a Bézier curve on the Bézier surface, for $v = 0.35$.

Next, some useful properties of Bézier patches will be presented. By setting $(u, v) = (0, 0)$, $(u, v) = (0, 1)$, $(u, v) = (1, 0)$, and $(u, v) = (1, 1)$ in Equation 17.32, it is simple to prove that a Bézier patch interpolates, that is, goes through, the corner control points, $\mathbf{p}_{0,0}$, $\mathbf{p}_{0,n}$, $\mathbf{p}_{n,0}$, and $\mathbf{p}_{n,n}$. Also, each boundary of the patch is described by a Bézier curve of degree n formed by the control points on the boundary. Therefore, the tangents at the corner control points are defined by these boundary Bézier curves. Each corner control point has two tangents, one in each of the u- and v-directions. As was the case for Bézier curves, the patch also lies within the convex hull of its control points, and

$$\sum_{i=0}^{m}\sum_{j=0}^{n} B_i^m(u)B_j^n(v) = 1 \tag{17.34}$$

for $(u, v) \in [0, 1]\times[0, 1]$. Finally, rotating the control points and then generating points on the patch is the same mathematically as (though usually faster than) generating points on the patch and then rotating these.

Partially differentiating Equation 17.32 gives [458] the equations below:

Derivatives [patches]:

$$\begin{aligned}\frac{\partial \mathbf{p}(u, v)}{\partial u} &= m\sum_{j=0}^{n}\sum_{i=0}^{m-1} B_i^{m-1}(u)B_j^n(v)[\mathbf{p}_{i+1,j} - \mathbf{p}_{i,j}],\\ \frac{\partial \mathbf{p}(u, v)}{\partial v} &= n\sum_{i=0}^{m}\sum_{j=0}^{n-1} B_i^m(u)B_j^{n-1}(v)[\mathbf{p}_{i,j+1} - \mathbf{p}_{i,j}].\end{aligned} \tag{17.35}$$

As can be seen, the degree of the patch is reduced by one in the direction that is differentiated. The unnormalized normal vector is then formed as

$$\mathbf{n}(u, v) = \frac{\partial \mathbf{p}(u, v)}{\partial u} \times \frac{\partial \mathbf{p}(u, v)}{\partial v}. \tag{17.36}$$

In Figure 17.20, the control mesh together with the actual Bézier patch is shown. The effect of moving a control point is shown in Figure 17.21.

Rational Bézier Patches

Just as the Bézier curve could be extended into a rational Bézier curve (Section 17.1.1), and thus introduce more degrees of freedom, so can the Bézier patch be extended into a rational Bézier patch:

$$\mathbf{p}(u, v) = \frac{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{i,j}B_i^m(u)B_j^n(v)\mathbf{p}_{i,j}}{\sum_{i=0}^{m}\sum_{j=0}^{n} w_{i,j}B_i^m(u)B_j^n(v)}. \tag{17.37}$$

Consult Farin's book [458] and Hochek and Lasser's book [777] for information about this type of patch. Similarly, the rational Bézier triangle is an extension of the Bézier triangle, treated next.

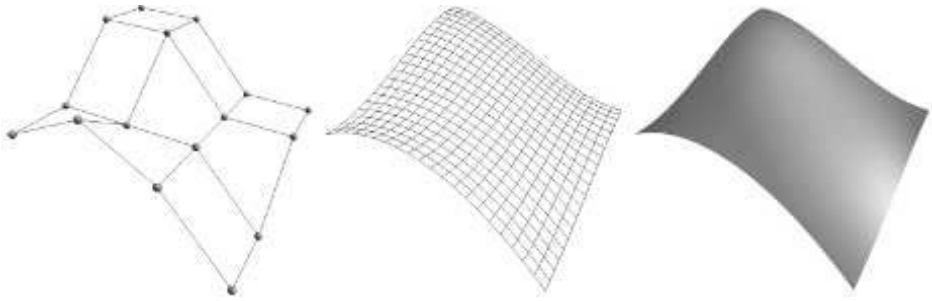

Figure 17.20. Left: control mesh of a 4×4 Bézier patch of degree 3×3. Middle: the actual quadrilaterals that were generated on the surface. Right: shaded Bézier patch.

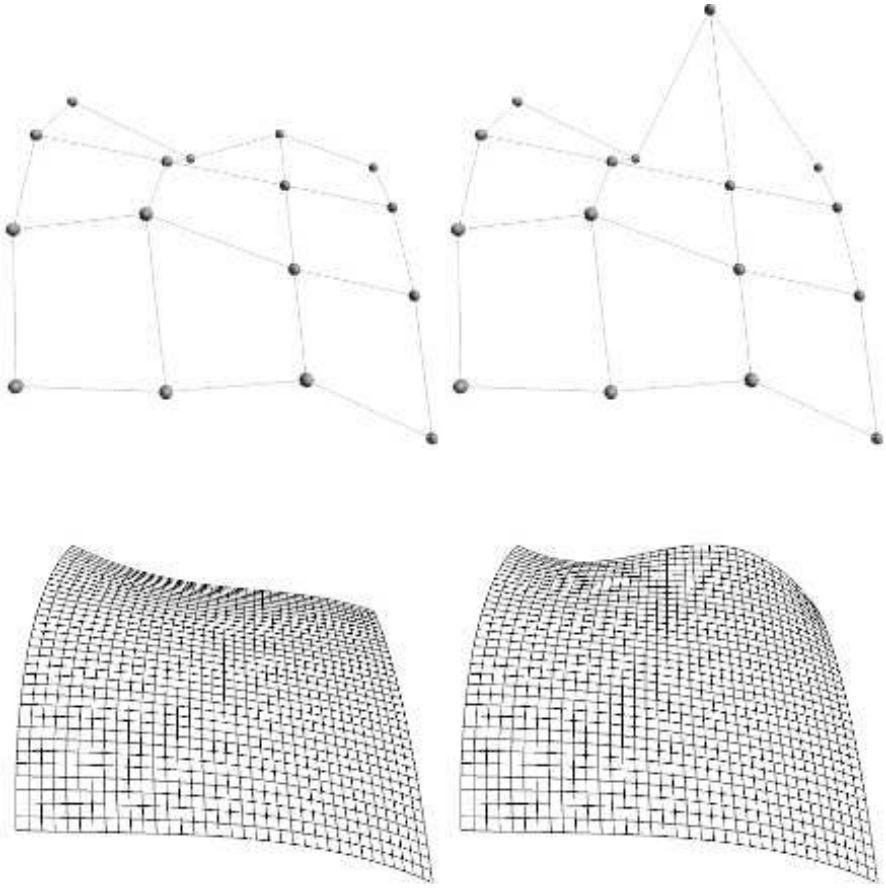

Figure 17.21. This set of images shows what happens to a Bézier patch when one control point is moved. Most of the change is near the moved control point.

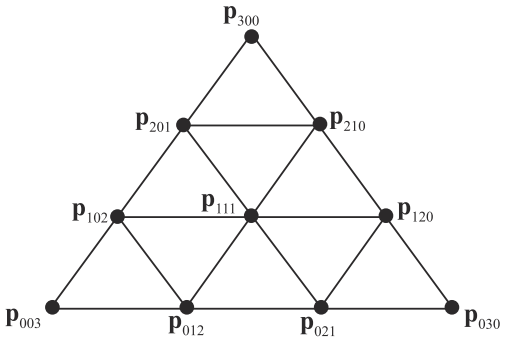

Figure 17.22. The control points of a Bézier triangle with degree three (cubic).

17.2.2 Bézier Triangles

Even though the triangle often is considered a simpler geometric primitive than the rectangle, this is not the case when it comes to Bézier surfaces: Bézier triangles are not as straightforward as Bézier patches. This type of patch is worth presenting as it is used in forming PN triangles and for Phong tessellation, which are fast and simple. Note that some game engines, such as the Unreal Engine, Unity, and Lumberyard, support Phong tessellation and PN triangles.

The control points are located in a triangular grid, as shown in Figure 17.22. The degree of the Bézier triangle is n, and this implies that there are $n+1$ control points per side. These control points are denoted $\mathbf{p}^0_{i,j,k}$ and sometimes abbreviated to $\mathbf{p}_{ijk}$. Note that $i+j+k=n$, and $i,j,k \geq 0$ for all control points. Thus, the total number of control points is

$$\sum_{x=1}^{n+1} x = \frac{(n+1)(n+2)}{2}. \tag{17.38}$$

It should come as no surprise that Bézier triangles also are based on repeated interpolation. However, due to the triangular shape of the domain, barycentric coordinates (Section 22.8) must be used for the interpolation. Recall that a point within a triangle $\Delta\mathbf{p}_0\mathbf{p}_1\mathbf{p}_2$, can be described as $\mathbf{p}(u,v) = \mathbf{p}_0 + u(\mathbf{p}_1 - \mathbf{p}_0) + v(\mathbf{p}_2 - \mathbf{p}_0) = (1-u-v)\mathbf{p}_0 + u\mathbf{p}_1 + v\mathbf{p}_2$, where (u,v) are the barycentric coordinates. For points inside the triangle the following must hold: $u \geq 0$, $v \geq 0$, and $1-(u+v) \geq 0 \Leftrightarrow u+v \leq 1$. Based on this, the de Casteljau algorithm for Bézier triangles is

de Casteljau [triangles]:

$$\begin{aligned} \mathbf{p}^l_{i,j,k}(u,v) &= u\mathbf{p}^{l-1}_{i+1,j,k} + v\mathbf{p}^{l-1}_{i,j+1,k} + (1-u-v)\mathbf{p}^{l-1}_{i,j,k+1}, \\ & l = 1 \ldots n, \quad i+j+k = n-l. \end{aligned} \tag{17.39}$$

The final point on the Bézier triangle at (u,v) is $\mathbf{p}^n_{000}(u,v)$. The Bézier triangle in Bernstein form is

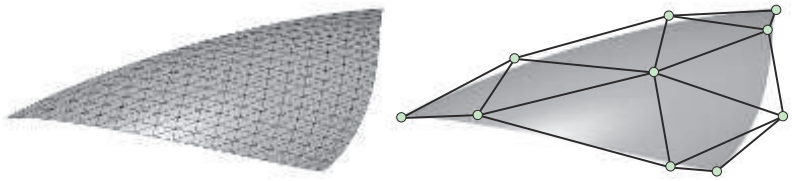

Figure 17.23. Left: wireframe of a tessellated Bézier triangle. Right: shaded surface together with control points.

Bernstein [triangles]:
$$\mathbf{p}(u,v) = \sum_{i+j+k=n} B_{ijk}^n(u,v)\mathbf{p}_{ijk}. \tag{17.40}$$

The Bernstein polynomials now depend on both u and v, and are therefore computed differently, as shown below:

$$B_{ijk}^n(u,v) = \frac{n!}{i!j!k!}u^i v^j (1-u-v)^k, \quad i+j+k=n. \tag{17.41}$$

The partial derivatives are [475]

Derivatives [triangles]:
$$\begin{aligned}\frac{\partial \mathbf{p}(u,v)}{\partial u} &= \sum_{i+j+k=n-1} nB_{ijk}^{n-1}(u,v)\big(\mathbf{p}_{i+1,j,k} - \mathbf{p}_{i,j,k+1}\big),\\ \frac{\partial \mathbf{p}(u,v)}{\partial v} &= \sum_{i+j+k=n-1} nB_{ijk}^{n-1}(u,v)\big(\mathbf{p}_{i,j+1,k} - \mathbf{p}_{i,j,k+1}\big).\end{aligned} \tag{17.42}$$

Some unsurprising properties of Bézier triangles are that they interpolate (pass through) the three corner control points, and that each boundary is a Bézier curve described by the control points on that boundary. Also, the surfaces lies in the convex hull of the control points. A Bézier triangle is shown in Figure 17.23.

17.2.3 Continuity

When constructing a complex object from Bézier surfaces, one often wants to stitch together several different Bézier surfaces to form one composite surface. To get a good-looking result, care must be taken to ensure that reasonable continuity is obtained across the surfaces. This is in the same spirit as for curves, in Section 17.1.3.

Assume two bicubic Bézier patches should be pieced together. These have 4×4 control points each. This is illustrated in Figure 17.24, where the left patch has control points, $\mathbf{a}_{ij}$, and the right has control points, $\mathbf{b}_{ij}$, for $0 \leq i,j \leq 3$. To ensure C^0 continuity, the patches must share the same control points at the border, that is, $\mathbf{a}_{3j} = \mathbf{b}_{0j}$.

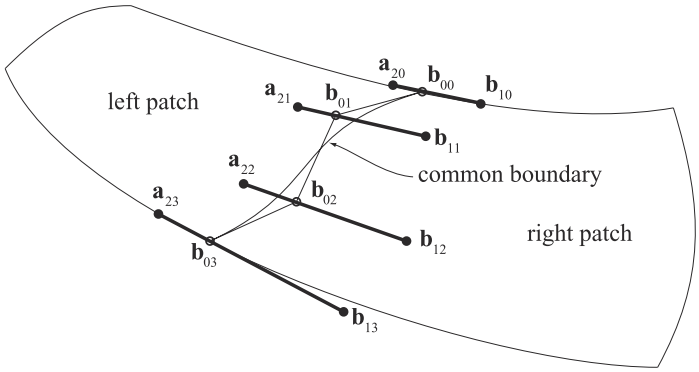

Figure 17.24. How to stitch together two Bézier patches with C^1 continuity. All control points on bold lines must be collinear, and they must have the same ratio between the two segment lengths. Note that $\mathbf{a}_{3j} = \mathbf{b}_{0j}$ to get a shared boundary between patches. This can also be seen to the right in Figure 17.25.

However, this is not sufficient to get a nice looking composite surface. Instead, a simple technique will be presented that gives C^1 continuity [458]. To achieve this we must constrain the position of the two rows of control points closest to the shared control points. These rows are $\mathbf{a}_{2j}$ and $\mathbf{b}_{1j}$. For each j, the points $\mathbf{a}_{2j}$, $\mathbf{b}_{0j}$, and $\mathbf{b}_{1j}$ must be collinear, that is, they must lie on a line. Moreover, they must have the same ratio, which means that $||\mathbf{a}_{2j} - \mathbf{b}_{0j}|| = k||\mathbf{b}_{0j} - \mathbf{b}_{1j}||$. Here, k is a constant, and it must be the same for all j. Examples are shown in Figure 17.24 and 17.25.

This sort of construction uses up many degrees of freedom of setting the control points. This can be seen even more clearly when stitching together four patches, sharing one common corner. The construction is visualized in Figure 17.26. The result is shown to the right in this figure, where the locations of the eight control points around the shared control point are shown. These nine points must all lie in the same plane, and they must form a bilinear patch, as shown in Figure 17.17. If one is satisfied with G^1 continuity at the corners (and only there), it suffices to make the nine points coplanar. This uses fewer degrees of freedom.

Continuity for Bézier triangles is generally more complex, as well as the G^1 conditions for both Bézier patches and triangles [458, 777]. When constructing a complex object of many Bézier surfaces, it is often hard to see to it that reasonable continuity is obtained across all borders. One solution to this is to turn to subdivision surfaces, treated in Section 17.5.

Note that C^1 continuity is required for good-looking texturing across borders. For reflections and shading, a reasonable result is obtained with G^1 continuity. C^1 or higher gives even better results. An example is shown in Figure 17.25.

In the following two subsections, we will present two methods that exploit the normals at triangle vertices to derive a Bézier triangle per input (flat) triangle.

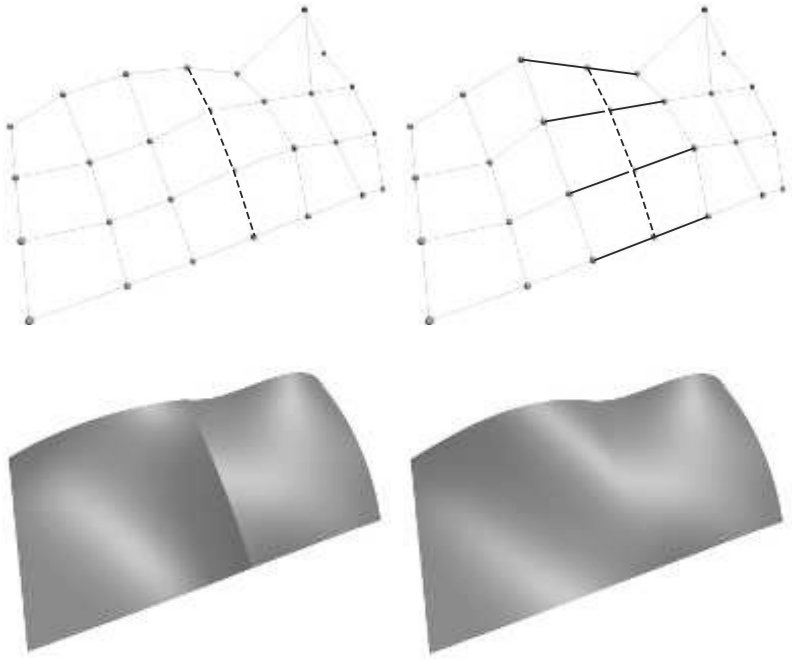

Figure 17.25. The left column shows two Bézier patches joined with only C^0 continuity. Clearly, there is a shading discontinuity between the patches. The right column shows similar patches joined with C^1 continuity, which looks better. In the top row, the dashed lines indicate the border between the two joined patches. To the upper right, the black lines show the collinearity of the control points of the joining patches.

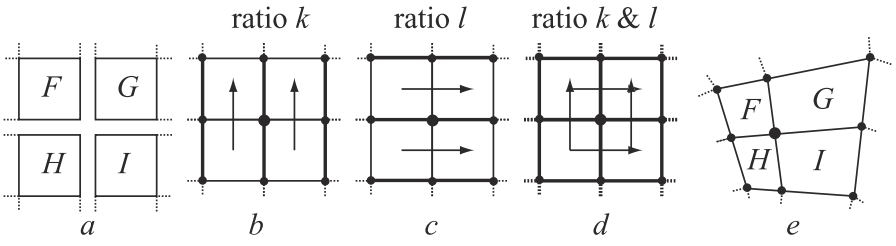

Figure 17.26. (a) Four patches, F, G, H, and I, are to be stitched together, where all patches share one corner. (b) In the vertical direction, the three sets of three points (on each bold line) must use the same ratio, k. This relationship is not shown here; see the rightmost figure. A similar process is done for (c), where, in the horizontal direction, both patches must use the same ratio, l. (d) When stitched together, all four patches must use ratio k vertically, and l horizontally. (e) The result is shown, in which the ratios are correctly computed for the nine control points closest to (and including) the shared control point.

17.2.4 PN Triangles

Given an input triangle mesh with normals at each vertex, the goal of the *PN triangle* scheme by Vlachos et al. [1819] is to construct a better-looking surface compared to using just triangles. The letters "PN" are short for "point and normal," since that is all the data you need to generate the surfaces. They are also called *N-patches*. This scheme attempts to improve the triangle mesh's shading and silhouettes by creating a curved surface to replace each triangle. Tessellation hardware is able to make each surface on the fly because the tessellation is generated from each triangle's points and normals, with no neighbor information needed. See Figure 17.27 for an example. The algorithm presented here builds upon work by van Overveld and Wyvill [1341].

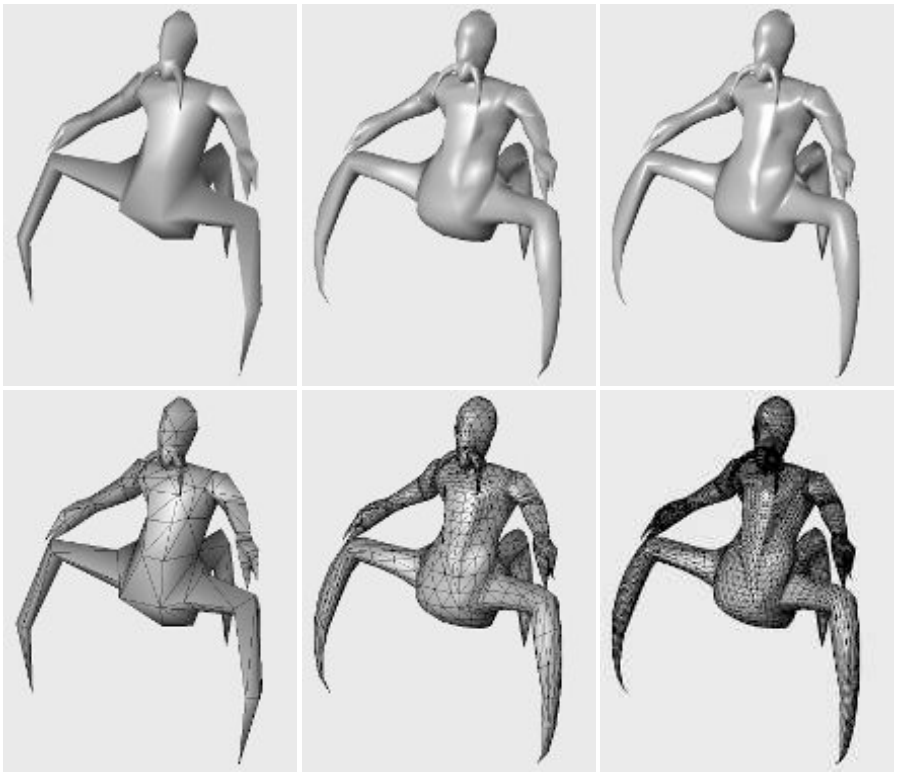

Figure 17.27. The columns show different levels of detail of the same model. The original triangle data, consisting of 414 triangles, is shown on the left. The middle model has 3,726 triangles, while the right has 20,286 triangles, all generated with the presented algorithm. Note how the silhouette and the shading improve. The bottom row shows the models in wireframe, which reveals that each original triangle generates the same amount of subtriangles. *(Model courtesy of id Software. Images from ATI Technologies Inc. demo.)*

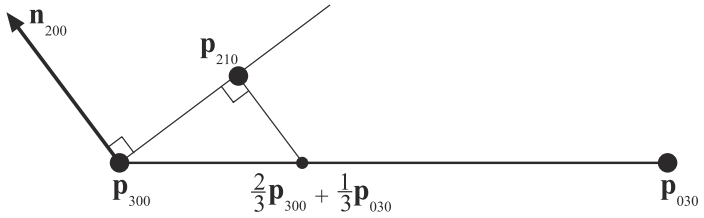

Figure 17.28. How the Bézier point $\mathbf{p}_{210}$ is computed using the normal $\mathbf{n}_{200}$ at $\mathbf{p}_{300}$, and the two corner points $\mathbf{p}_{300}$ and $\mathbf{p}_{030}$.

Assume that we have a triangle with vertices $\mathbf{p}_{300}$, $\mathbf{p}_{030}$, and $\mathbf{p}_{003}$ with normals $\mathbf{n}_{200}$, $\mathbf{n}_{020}$, and $\mathbf{n}_{002}$. The basic idea is to use this information to create a cubic Bézier triangle for each original triangle, and to generate as many triangles as we wish from the Bézier triangle.

To shorten notation, $w = 1-u-v$ will be used. A cubic Bézier triangle is given by

$$\begin{aligned}\mathbf{p}(u,v) &= \sum_{i+j+k=3} B_{ijk}^3(u,v)\mathbf{p}_{ijk} \\ &= u^3\mathbf{p}_{300} + v^3\mathbf{p}_{030} + w^3\mathbf{p}_{003} + 3u^2v\mathbf{p}_{210} + 3u^2w\mathbf{p}_{201} \\ &\quad + 3uv^2\mathbf{p}_{120} + 3v^2w\mathbf{p}_{021} + 3vw^2\mathbf{p}_{012} + 3uw^2\mathbf{p}_{102} + 6uvw\mathbf{p}_{111}.\end{aligned} \tag{17.43}$$

See Figure 17.22. To ensure C^0 continuity at the borders between two PN triangles, the control points on the edge can be determined from the corner control points and the normals at those corners. (assuming that normals are shared between adjacent triangles).

Say that we want to compute $\mathbf{p}_{210}$ using the control points $\mathbf{p}_{300}$, $\mathbf{p}_{030}$ and the normal $\mathbf{n}_{200}$ at $\mathbf{p}_{300}$, as illustrated in Figure 17.28. Simply take the point $\frac{2}{3}\mathbf{p}_{300}+\frac{1}{3}\mathbf{p}_{030}$ and project it in the direction of the normal, $\mathbf{n}_{200}$, onto the tangent plane defined by $\mathbf{p}_{300}$ and $\mathbf{n}_{200}$ [457, 458, 1819]. Assuming normalized normals, the point $\mathbf{p}_{210}$ is computed as

$$\mathbf{p}_{210} = \frac{1}{3}\big(2\mathbf{p}_{300} + \mathbf{p}_{030} - (\mathbf{n}_{200}\cdot(\mathbf{p}_{030}-\mathbf{p}_{300}))\mathbf{n}_{200}\big). \tag{17.44}$$

The other border control points can be computed similarly, so it only remains to compute the interior control point, $\mathbf{p}_{111}$. This is done as shown in the next equation, and this choice follows a quadratic polynomial [457, 458]:

$$\mathbf{p}_{111} = \frac{1}{4}(\mathbf{p}_{210}+\mathbf{p}_{120}+\mathbf{p}_{102}+\mathbf{p}_{201}+\mathbf{p}_{021}+\mathbf{p}_{012}) - \frac{1}{6}(\mathbf{p}_{300}+\mathbf{p}_{030}+\mathbf{p}_{003}). \tag{17.45}$$

Instead of using Equation 17.42 to compute the two tangents on the surface, and subsequently the normal, Vlachos et al. [1819] choose to interpolate the normal using

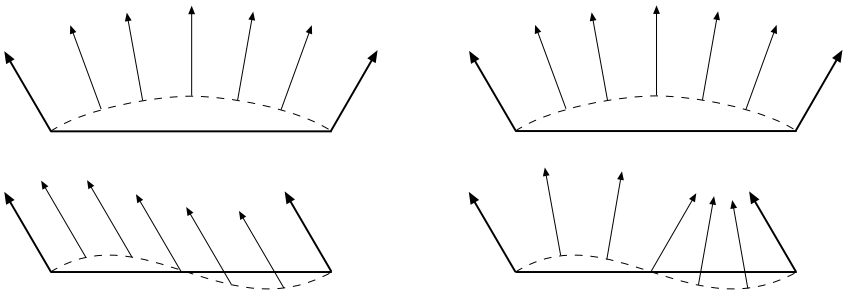

Figure 17.29. This figure illustrates why quadratic interpolation of normals is needed, and why linear interpolation is not sufficient. The left column shows what happens when linear interpolation of normals is used. This works fine when the normals describe a convex surface (top), but breaks down when the surface has an inflection (bottom). The right column illustrates quadratic interpolation. *(Illustration after van Overveld and Wyvill [1342].)*

a quadratic scheme, as shown here:

$$\begin{aligned}\mathbf{n}(u,v) &= \sum_{i+j+k=2} B^2_{ijk}(u,v)\mathbf{n}_{ijk} \\ &= u^2\mathbf{n}_{200} + v^2\mathbf{n}_{020} + w^2\mathbf{n}_{002} + 2(uv\mathbf{n}_{110} + uw\mathbf{n}_{101} + vw\mathbf{n}_{011}).\end{aligned} \tag{17.46}$$

This can be thought of as a Bézier triangle of degree two, where the control points are six different normals. In Equation 17.46, the choice of the degree, i.e., quadratic, is quite natural, since the derivatives are of one degree lower than the actual Bézier triangle, and because linear interpolation of the normals cannot describe an inflection. See Figure 17.29.

To be able to use Equation 17.46, the normal control points $\mathbf{n}_{110}$, $\mathbf{n}_{101}$, and $\mathbf{n}_{011}$ need to be computed. One intuitive, but flawed, solution is to use the average of $\mathbf{n}_{200}$ and $\mathbf{n}_{020}$ (normals at the vertices of the original triangle) to compute $\mathbf{n}_{110}$. However, when $\mathbf{n}_{200} = \mathbf{n}_{020}$, then the problem shown at the lower left in Figure 17.29 will once again be encountered. Instead, $\mathbf{n}_{110}$ is constructed by first taking the average of $\mathbf{n}_{200}$ and $\mathbf{n}_{020}$, and then reflecting this normal in the plane π, which is shown in Figure 17.30. This plane has a normal parallel to the difference between the endpoints $\mathbf{p}_{300}$ and $\mathbf{p}_{030}$. Since only normal vectors will be reflected in π, we can assume that π passes through the origin because normals are independent of the position on the plane. Also, note that each normal should be normalized. Mathematically, the unnormalized version of $\mathbf{n}_{110}$ is expressed as [1819]

$$\mathbf{n}'_{110} = \mathbf{n}_{200} + \mathbf{n}_{020} - 2\frac{(\mathbf{p}_{030} - \mathbf{p}_{300}) \cdot (\mathbf{n}_{200} + \mathbf{n}_{020})}{(\mathbf{p}_{030} - \mathbf{p}_{300}) \cdot (\mathbf{p}_{030} - \mathbf{p}_{300})}(\mathbf{p}_{030} - \mathbf{p}_{300}). \tag{17.47}$$

Originally, van Overveld and Wyvill used a factor of 3/2 instead of the 2 in this equation. Which value is best is hard to judge from looking at images, but using 2 gives the nice interpretation of a true reflection in the plane.

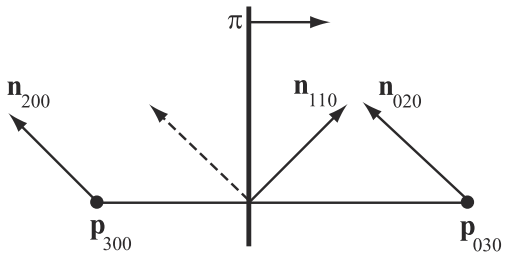

Figure 17.30. Construction of $\mathbf{n}_{110}$ for PN triangles. The dashed normal is the average of $\mathbf{n}_{200}$ and $\mathbf{n}_{020}$, and $\mathbf{n}_{110}$ is this normal reflected in the plane π. The plane π has a normal that is parallel to $\mathbf{p}_{030} - \mathbf{p}_{300}$.

At this point, all Bézier points of the cubic Bézier triangle and all the normal vectors for quadratic interpolation have been computed. It only remains to create triangles on the Bézier triangle so that they can be rendered. Advantages of this approach are that the surface gets a better silhouette and shape for relatively lower cost.

One way to specify levels of detail is the following. The original triangle data are considered LOD 0. The LOD number then increases with the number of newly introduced vertices on a triangle edge. So, LOD 1 introduces one new vertex per edge, and so creates four subtriangles on the Bézier triangle, and LOD 2 introduces two new vertices per edge, generating nine subtriangles. In general, LOD n generates $(n+1)^2$ subtriangles. To prevent cracking between Bézier triangles, each triangle in the mesh must be tessellated with the same LOD. This is a serious disadvantage, since a tiny triangle will be tessellated as much as a large triangle. Techniques such as adaptive tessellation (Section 17.6.2) and fractional tessellation (Section 17.6.1) can be used to avoid these problems.

One problem with PN triangles is that creases are hard to control, and often one needs to insert extra triangles near a desired crease. The continuity between Bézier triangles is only C^0, but they still look acceptable in many cases. This is mainly because the normals are continuous across triangles, so that a set of PN triangles mimics a G^1 surface. A better solution is suggested by Boubekeur et al. [181], where a vertex can have two normals, and two such connected vertices generate a crease. Note that to get good-looking texturing, C^1 continuity is required across borders between triangles (or patches). Also worth knowing is that cracks will appear if two adjacent triangles do not share the same normals. A technique to further improve the quality of the continuity across PN triangles is described by Grün [614]. Dyken et al. [401] present a technique inspired by PN triangles, where only the silhouettes as seen from the viewer are adaptively tessellated and, hence, become more curved. These silhouette curves are derived in similar ways as the PN triangle curves. To get smooth transitions, they blend between coarse silhouettes and tessellated silhouettes. For improved continuity, Fünfzig et al. [505] present PNG1 triangles, which is a modification of PN triangles that have G^1 continuity everywhere. McDonald and Kilgard [1164] present another extension of PN triangles, which can handle different normals on adjacent triangles.

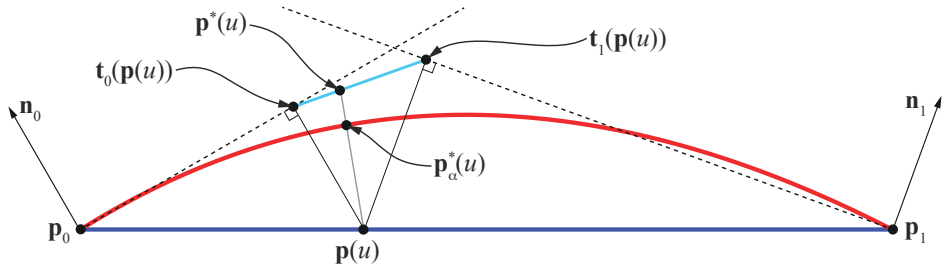

Figure 17.31. Phong tessellation construction illustrated using curves instead of surfaces, which means that $\mathbf{p}(u)$ is only a function of u instead of (u, v), and similarly for $\mathbf{t}_i$. Note that $\mathbf{p}(u)$ is first projected onto the tangent planes, which generates $\mathbf{t}_0$ and $\mathbf{t}_1$. After that,$\mathbf{p}^*(u)$ is created by a linear interpolation from $\mathbf{t}_0$ and $\mathbf{t}_1$. As a final step, a shape factor α is used to blend between the base triangle and $\mathbf{p}^*(u)$. In this example, we used $\alpha = 0.75$.

17.2.5 Phong Tessellation

Boubekeur and Alexa [182] presented a surface construction called *Phong tessellation*, which has many similarities with PN triangles, but is faster to evaluate and simpler to implement. Call the vertices of a base triangle $\mathbf{p}_0$, $\mathbf{p}_1$, and $\mathbf{p}_2$, and let the corresponding normalized normals be $\mathbf{n}_0$, $\mathbf{n}_1$, and $\mathbf{n}_2$. First, recall that a point on the base triangle at the barycentric coordinates (u, v) is computed as

$$\mathbf{p}(u, v) = (u, v, 1 - u - v) \cdot (\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2). \tag{17.48}$$

In Phong shading, the normals are interpolated over the flat triangle, also using the equation above, but with the points replaced by normals. Phong tessellation attempts to create a geometric version of Phong shading normal interpolation using repeated interpolation, which results in a Bézier triangle. For this discussion, we will refer to Figure 17.31. The first step is to create a function that projects a point $\mathbf{q}$ on the base triangle up to the tangent plane defined by a point and a normal. This is done as

$$\mathbf{t}_i(\mathbf{q}) = \mathbf{q} - \big((\mathbf{q} - \mathbf{p}_i) \cdot \mathbf{n}_i\big)\mathbf{n}_i. \tag{17.49}$$

Instead of using the triangle vertices to perform linear interpolation (Equation 17.48), linear interpolation is done using the function $\mathbf{t}_i$, which results in

$$\mathbf{p}^*(u, v) = (u, v, 1 - u - v) \cdot \big(\mathbf{t}_0(u, v), \mathbf{t}_1(u, v), \mathbf{t}_2(u, v)\big). \tag{17.50}$$

To add some flexibility, a shape factor α is added that interpolates between the base triangle and Equation 17.50, which results in the final formula for Phong tessellation:

$$\mathbf{p}^*_\alpha(u, v) = (1 - \alpha)\mathbf{p}(u, v) + \alpha\mathbf{p}^*(u, v), \tag{17.51}$$

where $\alpha = 0.75$ is a recommended setting [182]. The only information needed to generate this surface is the vertices and normals of the base triangle and a user-

Figure 17.32. Phong tessellation applied to the monster frog. From left to right: base mesh with flat shading, base mesh with Phong shading, and finally, Phong tessellation applied to the base mesh. Note the improved silhouettes. In this example, we used $\alpha = 0.6$. *(Images generated using Tamy Boubekeur's demo program.)*

supplied α, which makes evaluation of this surface fast. The resulting triangular path is quadratic, i.e., of lower degree than PN triangles. The normals are simply linearly interpolated, just as in standard Phong shading. See Figure 17.32 for an example illustrating the effect of Phong tessellation applied to a mesh.

17.2.6 B-Spline Surfaces

Section 17.1.6 briefly introduced B-spline curves, and here we will do the same for introducing B-spline surfaces. Equation 17.24 on page 732 can be generalized to B-spline patches as

$$\mathbf{s}_n(u,v) = \sum_k \sum_l \mathbf{c}_{k,l}\beta_n(u-k)\beta_n(v-l), \tag{17.52}$$

which is fairly similar to the Bézier patch formula (Equation 17.32). Note that $\mathbf{s}_n(u,v)$ is a three-dimensional point on the surface. If this function were to be used for texture filtering, Equation 17.52 would be a height field and $c_{k,l}$ would be one-dimensional, i.e., heights.

For a bicubic B-spline patch, the $\beta_3(t)$ function from Equation 17.25 would be used in Equation 17.52. A total of 4×4 control points, $\mathbf{c}_{k,l}$, would be needed, and the actual surface patch described by Equation 17.52 would be inside the innermost 2×2 control points. This is illustrated in Figure 17.33. Note that bi-cubic B-spline patches are essential to Catmull-Clark subdivision surfaces (Section 17.5.2), as well. There are many good books with more information about B-spline surfaces [111, 458, 777].

17.3 Implicit Surfaces

To this point, only parametric curves and surfaces have been discussed. *Implicit surfaces* form another useful class for representing models. Instead of using some

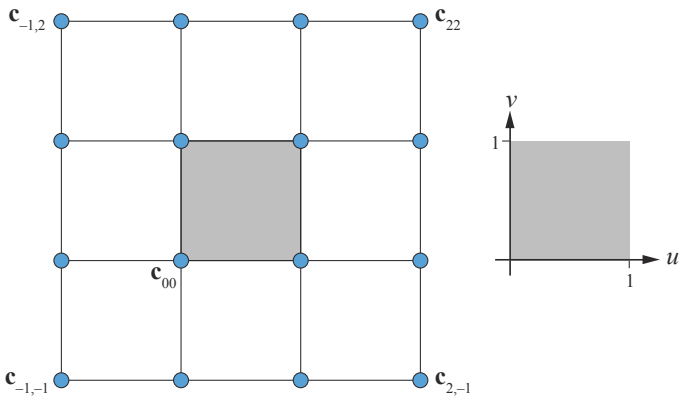

Figure 17.33. The setting for a bicubic B-spline patch, which has 4×4 control points, $\mathbf{c}_{k,l}$. The domain for (u, v) is the unit square as shown to the right.

parameters, say u and v, to explicitly describe a point on the surface, the following form, called the implicit function, is used:

$$f(x, y, z) = f(\mathbf{p}) = 0. \tag{17.53}$$

This is interpreted as follows: A point $\mathbf{p}$ is on the implicit surface if the result is zero when the point is inserted into the implicit function f. Implicit surfaces are often used in intersection testing with rays (Sections 22.6–22.9), as they can be simpler to intersect than the corresponding (if any) parametric surface. Another advantage of implicit surfaces is that *constructive solid geometry* algorithms can be applied easily to them, that is, objects can be subtracted from each other, logically `AND`:ed or `OR`:ed with each other. Also, objects can be easily blended and deformed.

Some examples of implicit surfaces, located at the origin, are

$$\begin{array}{ll} f_s(\mathbf{p}, r) = ||\mathbf{p}|| - r, & \text{sphere;} \\ f_{xz}(\mathbf{p}) = p_y, & \text{plane in } xz; \\ f_{rb}(\mathbf{p}, \mathbf{d}, r) = ||\max(|\mathbf{p}| - \mathbf{d}, 0)|| - r, & \text{rounded box.} \end{array} \tag{17.54}$$

These deserve some explanation. The sphere is simply the distance from $\mathbf{p}$ to the origin subtracted by the radius, so $f_s(\mathbf{p}, r)$ is equal to 0 if $\mathbf{p}$ is on the sphere with radius r. Otherwise, a signed distance will be returned where negative means that $\mathbf{p}$ is inside the sphere, and positive outside. Therefore, these functions are sometimes also called *signed distance functions* (SDFs). The plane $f_{xz}(\mathbf{p})$ is just the y-coordinate of $\mathbf{p}$, i.e., the side where the y-axis is positive. For the expression for the rounded box, we assume that the absolute value ($|\mathbf{p}|$) and the maximum of a vector are calculated per component. Also, $\mathbf{d}$ is a vector of the half sides of the box. See the rounded box illustrated in Figure 17.34; the formula is explained in the caption. To get a non-rounded box, simply set $r = 0$.

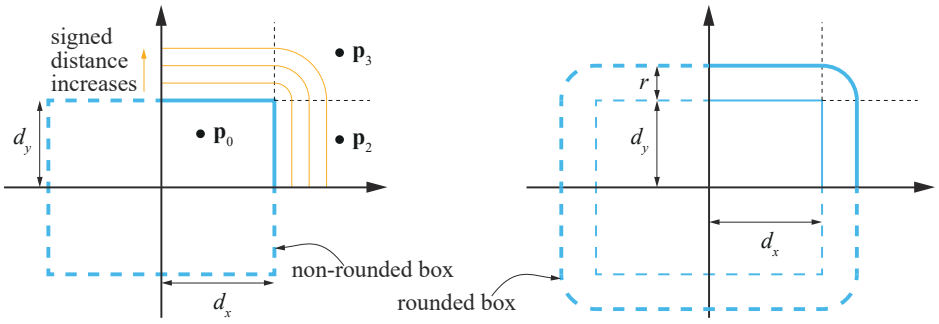

Figure 17.34. Left: a non-rounded box, whose signed distance function is $||\max(|\mathbf{p}| - \mathbf{d}, 0)||$, where $\mathbf{p}$ is the point to be tested and $\mathbf{d}$'s components are the half sides as shown. Note that $|\mathbf{p}|$ makes the rest of the computations occur in the top right quadrant (in 2D). Subtracting $\mathbf{d}$ means that $|p_x| - d_x$ will be negative if $\mathbf{p}$ is inside the box along x, and likewise for the other axes. Only positive values are retained, while negative ones are clamped to zero by max(). Hence, $||\max(|\mathbf{p}| - \mathbf{d}, 0)||$ computes the closest distance to the box sides, and this means that the signed distance field outside the box gets rounded if more than one value is positive after evaluating max(). Right: the rounded box is obtained by subtracting r from the non-rounded box, which expands the box by r is all directions.

The normal of an implicit surface is described by the partial derivatives, called the gradient and denoted ∇f:

$$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right). \tag{17.55}$$

To be able to evaluate it exactly, f in Equation 17.55 must be differentiable, and thus also continuous. In practice, one often use a numerical technique called central differences, which samples using the scene function f [495]:

$$\nabla f_x \approx f(\mathbf{p} + \epsilon \mathbf{e}_x) - f(\mathbf{p} - \epsilon \mathbf{e}_x), \tag{17.56}$$

and similarly for ∇f_y and ∇f_z. Recall that $\mathbf{e}_x = (1, 0, 0)$, $\mathbf{e}_y = (0, 1, 0)$, and $\mathbf{e}_z = (0, 0, 1)$ and that ϵ is a small number.

To build a scene with the primitives in Equation 17.54, the union operator, $\cup$, is used. For example, $f(\mathbf{p}) = f_s(\mathbf{p}, 1) \cup f_{xz}(\mathbf{p})$ is a scene consisting of a sphere and a plane. The union operator is implemented by taking the smallest of its two operands, since we want to find the surface closest to $\mathbf{p}$. Translation is done by translating $\mathbf{p}$ before calling the signed distance function, i.e., $f_s(\mathbf{p} - \mathbf{t}, 1)$ is a sphere translated by $\mathbf{t}$. Rotations and other transforms can be done in the same spirit, i.e., with the inverse transform applied to $\mathbf{p}$. It is also straightforward to repeat an object over the entire space by using $\mathbf{r} = \text{mod}(\mathbf{p}, \mathbf{c}) - 0.5\mathbf{c}$ instead of $\mathbf{p}$ as the argument to the signed distance function.

Blending of implicit surfaces is a nice feature that can be used in what is often referred to as blobby modeling [161], soft objects, or metaballs [67, 558]. See Figure 17.35 for some examples. The basic idea is to use several simple primitives, such

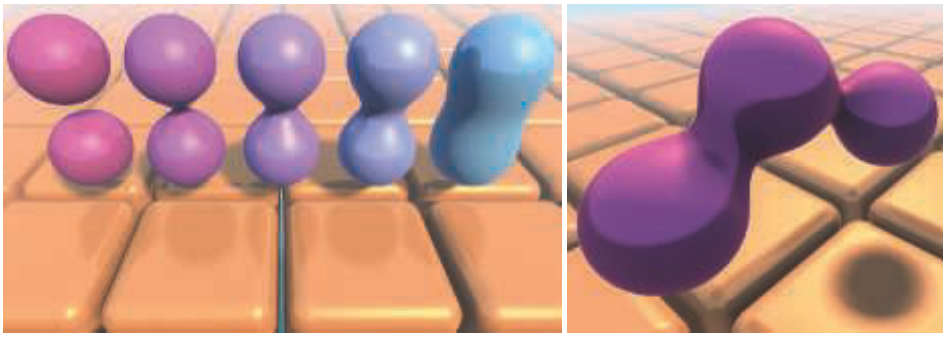

Figure 17.35. Left: pairs of spheres blended with different increasing (from left to right) blend radii and with a ground floor composed of repeated rounded boxes. Right: three spheres blended together.

as spheres, ellipsoids, or whatever is available, and blend these smoothly. Each object can be seen as an atom, and after blending the molecule of the atoms is obtained. Blending can be done in many different ways. An often-used method [1189, 1450] to blend two distances, d_1 and d_2, with a blend radius, r_b, is

$$\begin{aligned} h &= \min\big(\max(0.5 + 0.5(d_2 - d_1)/r_b, 0.0), 1.0\big), \\ d &= (1-h)d_2 + hd_1 + r_b h(1-h), \end{aligned} \tag{17.57}$$

where d is the blended distance. While this function only blends the shortest distances to two objects, the function can be use repeatedly to blend more objects (see the right part of Figure 17.35).

To visualize a set of implicit functions, the usual method used is ray marching [673]. Once you can ray-march through a scene, it is also possible to generate shadows, reflections, ambient occlusion, and other effects. Ray marching within a signed distance field is illustrated in Figure 17.36. At the first point, $\mathbf{p}$, on the ray, we evaluate the

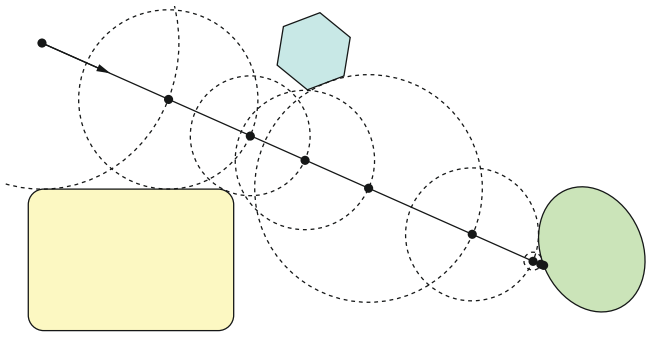

Figure 17.36. Ray marching with signed distance fields. The dashed circles indicate the distance to the closest surface from their centers. A position can advance along the ray to the border of the previous position's circle.

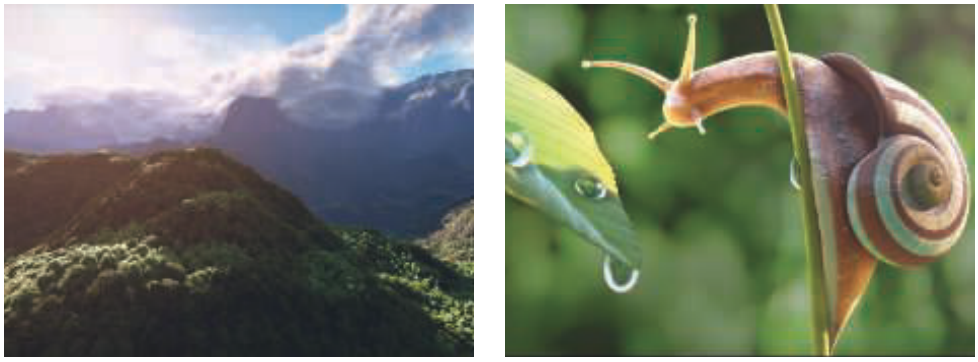

Figure 17.37. Rain forest (left) and a snail (right) created procedurally using signed distance functions and ray marching. The trees were generated using ellipsoids displaced with procedural noise. *(Images generated with Shadertoy using programs from Iñigo Quilez.)*

shortest distance, d, to the scene. Since this indicates that there is a sphere around $\mathbf{p}$ of radius d with no other object being any closer, we can move the ray d units along the ray direction and so on until we reach the surface within some epsilon, or when a predefined ray-march steps have been met, in which case we can assume that the background is hit. Two excellent examples are shown in Figure 17.37.

Every implicit surface can also be turned into a surface consisting of triangles. There are several algorithms available for performing this operation [67, 558]. One well-known example is the marching cubes algorithm, described in Section 13.10. Code for performing polygonalization using algorithms by Wyvill and Bloomenthal is available on the web [171], and de Araújo et al. [67] present a survey on recent techniques for polygonalization of implicit surfaces. Tatarchuk and Shopf [1744] describe a technique they call *marching tetrahedra*, in which the GPU can be used to find isosurfaces in a three-dimensional data set. Figure 3.13 on page 48 shows an example of isosurface extraction using the geometry shader. Xiao et al. [1936] present a fluid simulation system in which the GPU computes the locations of 100k particles and uses them to display the isosurface, all at interactive rates.

17.4 Subdivision Curves

Subdivision techniques are used to create smooth curves and surfaces. One reason why they are used in modeling is that they bridge the gap between discrete surfaces (triangle meshes) and continuous surfaces (e.g., a collection of Bézier patches), and can therefore be used for level of detail techniques (Section 19.9). Here, we will first describe how subdivision curves work, and then discuss the more popular subdivision surface schemes.

Subdivision curves are best explained by an example that uses *corner cutting*. See Figure 17.38. The corners of the leftmost polygon are cut off, creating a new polygon

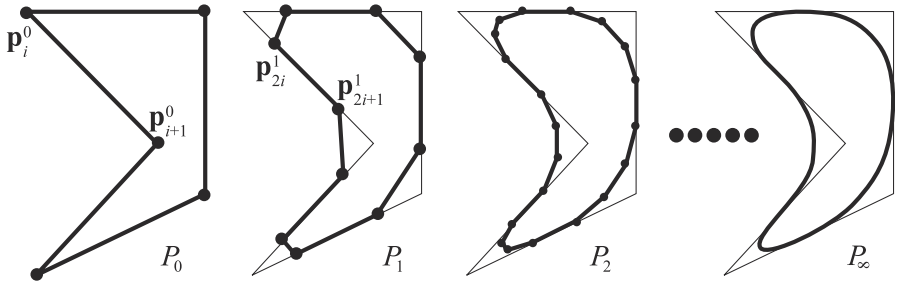

Figure 17.38. Chaikin's subdivision scheme in action. The initial control polygon P_0 is subdivided once into P_1, and then again into P_2. As can be seen, the corners of each polygon, P_i, are cut off during subdivision. After infinitely many subdivisions, the limit curve P_∞ is obtained. This is an approximating scheme as the curve does not go through the initial points.

with twice as many vertices. Then the corners of this new polygon are cut off, and so on to infinity (or, more practically, until we cannot see any difference). The resulting curve, called the *limit curve*, is smooth since all corners are cut off. This process can also be thought of as a low-pass filter since all sharp corners (high frequency) are removed. This process is often written as $P_0 \to P_1 \to P_2 \cdots \to P_\infty$, where P_0 is the starting polygon, also called the *control polygon*, and P_∞ is the limit curve.

This subdivision process can be done in many different ways, and each is characterized by a subdivision scheme. The one shown in Figure 17.38 is called Chaikin's scheme [246] and works as follows. Assume the n vertices of a polygon are $P_0 = \{\mathbf{p}_0^0, \ldots, \mathbf{p}_{n-1}^0\}$, where the superscript denotes the level of subdivision. Chaikin's scheme creates two new vertices between each subsequent pair of vertices, say $\mathbf{p}_i^k$ and $\mathbf{p}_{i+1}^k$, of the original polygon as

$$\mathbf{p}_{2i}^{k+1} = \frac{3}{4}\mathbf{p}_i^k + \frac{1}{4}\mathbf{p}_{i+1}^k \quad \text{and} \quad \mathbf{p}_{2i+1}^{k+1} = \frac{1}{4}\mathbf{p}_i^k + \frac{3}{4}\mathbf{p}_{i+1}^k. \tag{17.58}$$

As can be seen, the superscript changes from k to $k+1$, which means that we go from one subdivision level to the next, i.e., $P_k \to P_{k+1}$. After such a subdivision step is performed, the original vertices are discarded and the new points are reconnected. This kind of behavior can be seen in Figure 17.38, where new points are created 1/4 away from the original vertices toward neighboring vertices. The beauty of subdivision schemes comes from the simplicity of rapidly generating smooth curves. However, you do not immediately have a parametric form of the curve as in Section 17.1, though it can be shown that Chaikin's algorithm generates a quadratic B-spline [111, 458, 777, 1847]. So far, the presented scheme works for (closed) polygons, but most schemes can be extended to work for open polylines as well. In the case of Chaikin, the only difference is that the two endpoints of the polyline are kept in each subdivision step (instead of being discarded). This makes the curve go through the endpoints.

There are two different classes of subdivision schemes, namely *approximating* and *interpolating*. Chaikin's scheme is approximating, as the limit curve, in general, does

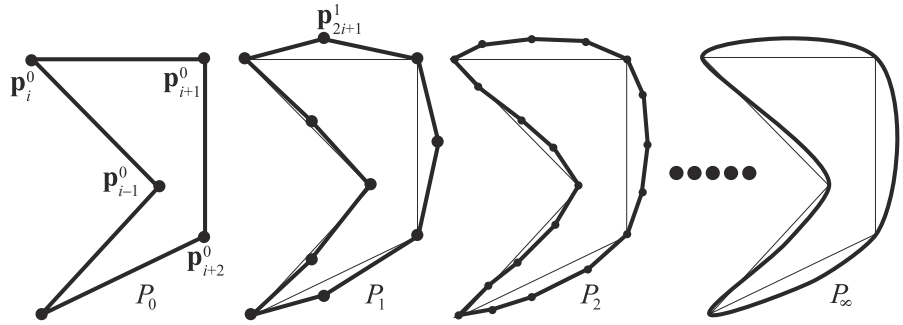

Figure 17.39. The 4-point subdivision scheme in action. This is an interpolating scheme as the curve goes through the initial points, and in general curve P_{i+1} goes through the points of P_i. Note that the same control polygon is used in Figure 17.38.

not lie on the vertices of the initial polygon. This is because the vertices are discarded (or updated, for some schemes). In contrast, an interpolating scheme keeps all the points from the previous subdivision step, and so the limit curve P_∞ goes through all the points of P_0, P_1, P_2, and so on. This means that the scheme interpolates the initial polygon. An example, using the same polygon as in Figure 17.38, is shown in Figure 17.39. This scheme uses the four nearest points to create a new point [402]:

$$\begin{aligned} \mathbf{p}^{k+1}_{2i} &= \mathbf{p}^k_i, \\ \mathbf{p}^{k+1}_{2i+1} &= \left(\frac{1}{2}+w\right)(\mathbf{p}^k_i + \mathbf{p}^k_{i+1}) - w(\mathbf{p}^k_{i-1} + \mathbf{p}^k_{i+2}). \end{aligned} \tag{17.59}$$

The first line in Equation 17.59 simply means that we keep the points from the previous step without changing them (i.e., interpolating), and the second line is for creating a new point in between $\mathbf{p}^k_i$ and $\mathbf{p}^k_{i+1}$. The weight w is called a *tension parameter*. When $w = 0$, linear interpolation is the result, but when $w = 1/16$, we get the kind of behavior shown in Figure 17.39. It can be shown [402] that the resulting curve is C^1 when $0 < w < 1/8$. For open polylines we run into problems at the endpoints because we need two points on both sides of the new point, and we only have one. This can be solved if the point next to the endpoint is reflected across the endpoint. So, for the start of the polyline, $\mathbf{p}_1$ is reflected across $\mathbf{p}_0$ to obtain $\mathbf{p}_{-1}$. This point is then used in the subdivision process. The creation of $\mathbf{p}_{-1}$ is shown in Figure 17.40.

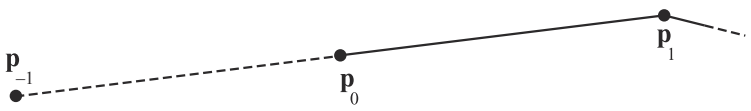

Figure 17.40. The creation of a reflection point, $\mathbf{p}_{-1}$, for open polylines. The reflection point is computed as: $\mathbf{p}_{-1} = \mathbf{p}_0 - (\mathbf{p}_1 - \mathbf{p}_0) = 2\mathbf{p}_0 - \mathbf{p}_1$.

Another approximating scheme uses the following subdivision rules:

$$\begin{aligned}\mathbf{p}_{2i}^{k+1} &= \frac{3}{4}\mathbf{p}_i^k + \frac{1}{8}(\mathbf{p}_{i-1}^k + \mathbf{p}_{i+1}^k),\\ \mathbf{p}_{2i+1}^{k+1} &= \frac{1}{2}(\mathbf{p}_i^k + \mathbf{p}_{i+1}^k).\end{aligned} \tag{17.60}$$

The first line updates the existing points, and the second computes the midpoint on the line segment between two neighboring points. This scheme generates a cubic B-spline curve (Section 17.1.6). Consult the SIGGRAPH course on subdivision [1977], the Killer B's book [111], Warren and Weimer's subdivision book [1847], or Farin's CAGD book [458] for more about these curves.

Given a point $\mathbf{p}$ and its neighboring points, it is possible to directly "push" that point to the limit curve, i.e., determine what the coordinates of $\mathbf{p}$ would be on P_∞. This is also possible for tangents. See, for example, Joy's online introduction to this topic [843].

Many of the concepts for subdivision curves also apply to subdivision surfaces, which are presented next.

17.5 Subdivision Surfaces

Subdivision surfaces are a powerful paradigm for defining smooth, continuous, crackless surfaces from meshes with arbitrary topology. As with all other surfaces in this chapter, subdivision surfaces also provide infinite level of detail. That is, you can generate as many triangles or polygons as you wish, and the original surface representation is compact. An example of a surface being subdivided is shown in Figure 17.41. Another advantage is that subdivision rules are simple and easily implemented. A disadvantage is that the analysis of surface continuity often is mathematically involved. However, this sort of analysis is often of interest only to those who wish to create new subdivision schemes, and is out of the scope of this book. For such details, consult Warren and Weimer's book [1847] and the SIGGRAPH course on subdivision [1977].

In general, the subdivision of surfaces (and curves) can be thought of as a two-phase process [915]. Starting with a polygonal mesh, called the *control mesh* or the *control cage*, the first phase, called the *refinement phase*, creates new vertices and reconnects to create new, smaller triangles. The second, called the *smoothing phase*, typically computes new positions for some or all vertices in the mesh. This is illustrated in Figure 17.42. It is the details of these two phases that characterize a subdivision scheme. In the first phase, a polygon can be split in different ways, and in the second phase, the choice of subdivision rules give different characteristics such as the level of continuity, and whether the surface is approximating or interpolating, which are properties described in Section 17.4.

A subdivision scheme can be characterized by being *stationary* or *non-stationary*, by being *uniform* or *nonuniform*, and whether it is *triangle-based* or *polygon-based*. A stationary scheme uses the same subdivision rules at every subdivision step, while

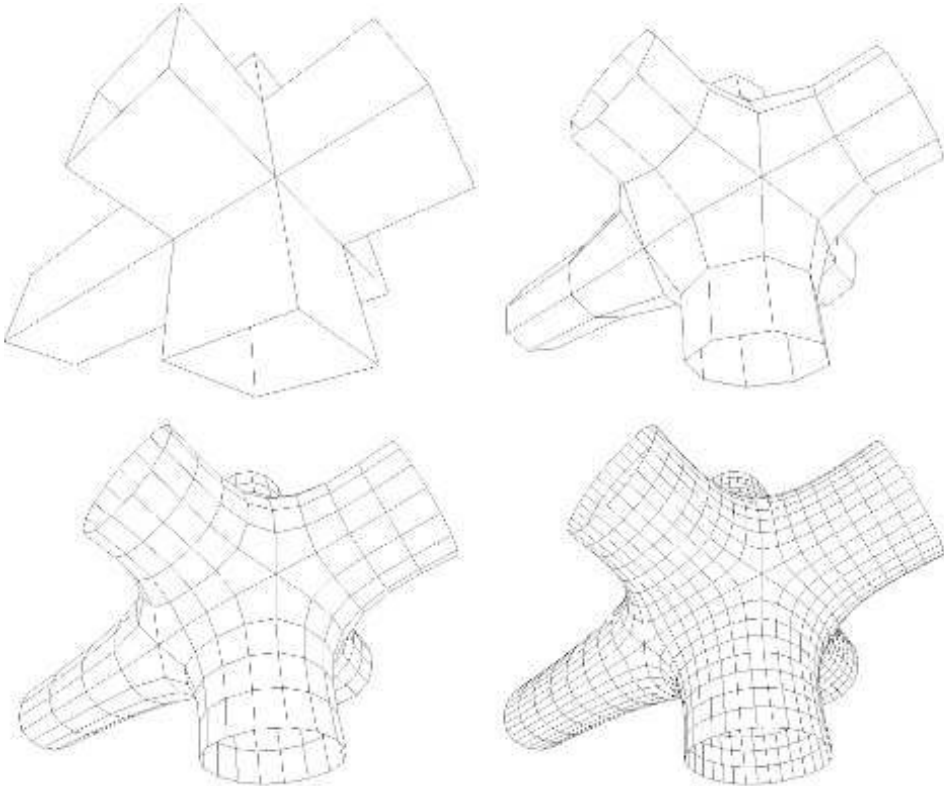

Figure 17.41. The top left image shows the control mesh, i.e., that original mesh, which is the only geometrical data that describes the resulting subdivision surface. The following images are subdivided one, two, and three times. As can be seen, more and more polygons are generated and the surface gets smoother and smoother. The scheme used here is the Catmull-Clark scheme, described in Section 17.5.2.

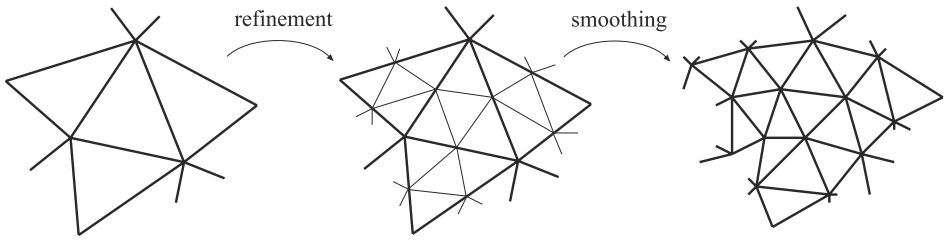

Figure 17.42. Subdivision as refinement and smoothing. The refinement phase creates new vertices and reconnects to create new triangles, and the smoothing phase computes new positions for the vertices.

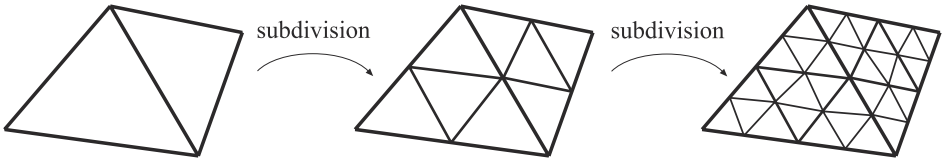

Figure 17.43. The connectivity of two subdivision steps for schemes such as Loop's method. Each triangle generates four new triangles.

a nonstationary may change the rules depending on which step currently is being processed. The schemes treated below are all stationary. A uniform scheme uses the same rules for every vertex or edge, while a nonuniform scheme may use different rules for different vertices or edges. As an example, a different set of rules is often used for edges that are on the boundaries of a surface. A triangle-based scheme only operates on triangles, and thus only generates triangles, while a polygon-based scheme operates on arbitrary polygons.

Several different subdivision schemes are presented next. Following these, two techniques are presented that extend the use of subdivision surfaces, along with methods for subdividing normals, texture coordinates, and colors. Finally, some practical algorithms for subdivision and rendering are presented.

17.5.1 Loop Subdivision

Loop's method [767, 1067] was the first subdivision scheme for triangles. It is like the last scheme in Section 17.4 in that it is approximating, and that it updates each existing vertex and creates a new vertex for each edge. The connectivity for this scheme is shown in Figure 17.43. As can be seen, each triangle is subdivided into four new triangles, so after n subdivision steps, a triangle has been subdivided into 4^n triangles.

First, let us focus on an existing vertex $\mathbf{p}^k$, where k is the number of subdivision steps. This means that $\mathbf{p}^0$ is the vertex of the control mesh.

After one subdivision step, $\mathbf{p}^0$ turns into $\mathbf{p}^1$. In general, $\mathbf{p}^0 \rightarrow \mathbf{p}^1 \rightarrow \mathbf{p}^2 \rightarrow \cdots \rightarrow \mathbf{p}^\infty$, where $\mathbf{p}^\infty$ is the limit point. If $\mathbf{p}^k$ has n neighboring vertices, $\mathbf{p}_i^k$, $i \in \{0, 1, \ldots, n-1\}$, then we say that the *valence* of $\mathbf{p}^k$ is n. See Figure 17.44 for the notation described above. Also, a vertex that has valence 6 is called *regular* or *ordinary*. Otherwise it is called *irregular* or *extraordinary*.

Below, the subdivision rules for Loop's scheme are given, where the first formula is the rule for updating an existing vertex $\mathbf{p}^k$ into $\mathbf{p}^{k+1}$, and the second formula is for creating a new vertex, $\mathbf{p}_i^{k+1}$, between $\mathbf{p}^k$ and each of the $\mathbf{p}_i^k$. Again, n is the valence of $\mathbf{p}^k$:

$$\begin{aligned} \mathbf{p}^{k+1} &= (1 - n\beta)\mathbf{p}^k + \beta(\mathbf{p}_0^k + \cdots + \mathbf{p}_{n-1}^k), \\ \mathbf{p}_i^{k+1} &= \frac{3\mathbf{p}^k + 3\mathbf{p}_i^k + \mathbf{p}_{i-1}^k + \mathbf{p}_{i+1}^k}{8}, \ i = 0 \ldots n-1. \end{aligned} \tag{17.61}$$

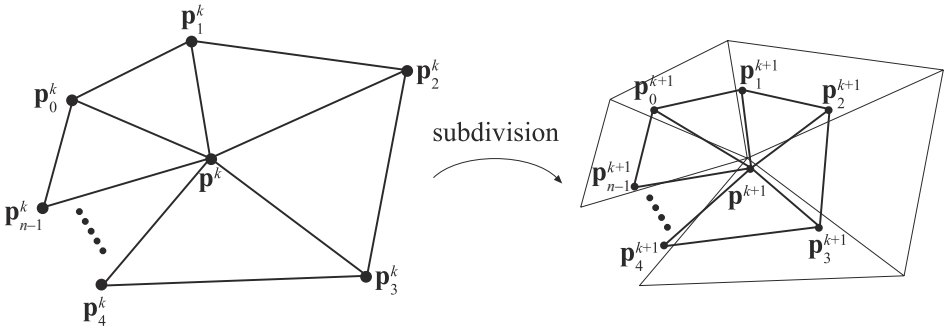

Figure 17.44. The notation used for Loop's subdivision scheme. The left neighborhood is subdivided into the neighborhood to the right. The center point $\mathbf{p}^k$ is updated and replaced by $\mathbf{p}^{k+1}$, and for each edge between $\mathbf{p}^k$ and $\mathbf{p}_i^k$, a new point is created ($\mathbf{p}_i^{k+1}$, $i \in 1, \ldots, n$).

Note that we assume that the indices are computed modulo n, so that if $i = n-1$, then for $i+1$, we use index 0, and likewise when $i = 0$, then for $i-1$, we use index $n-1$. These subdivision rules can easily be visualized as masks, also called stencils. See Figure 17.45. The major use of these is that they communicate almost an entire subdivision scheme using only a simple illustration. Note that the weights sum to one for both masks. This is a characteristic that is true for all subdivision schemes, and the rationale for this is that a new point should lie in the neighborhood of the weighted points. In Equation 17.61, the constant β is actually a function of n, and is given by

$$\beta(n) = \frac{1}{n}\left(\frac{5}{8} - \frac{(3+2\cos(2\pi/n))^2}{64}\right). \tag{17.62}$$

Loop's suggestion [1067] for the β function gives a surface of C^2 continuity at every regular vertex, and C^1 elsewhere [1976], that is, at all irregular vertices. As only regular vertices are created during subdivision, the surface is only C^1 at the

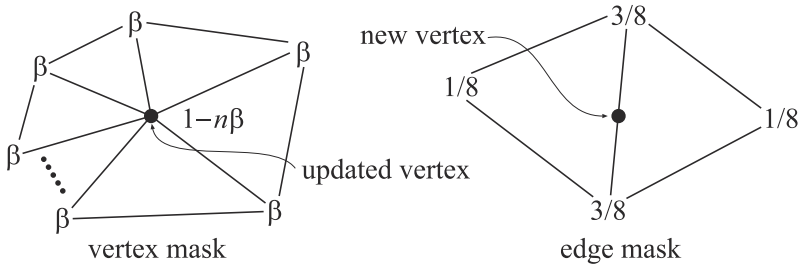

Figure 17.45. The masks for Loop's subdivision scheme (black circles indicate which vertex is updated/generated). A mask shows the weights for each involved vertex. For example, when updating an existing vertex, the weight $1 - n\beta$ is used for the existing vertex, and the weight β is used for all the neighboring vertices, called the 1-ring.

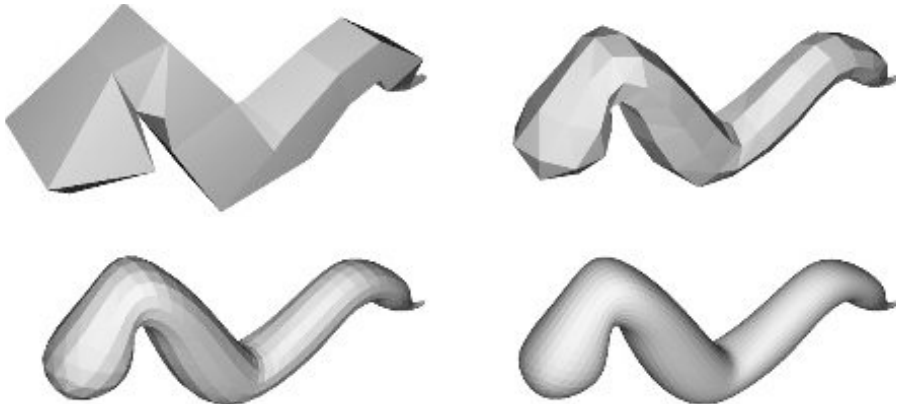

Figure 17.46. A worm subdivided three times with Loop's subdivision scheme.

places where we had irregular vertices in the control mesh. See Figure 17.46 for an example of a mesh subdivided with Loop's scheme. A variant of Equation 17.62, which avoids trigonometric functions, is given by Warren and Weimer [1847]:

$$\beta(n) = \frac{3}{n(n+2)}. \tag{17.63}$$

For regular valences, this gives a C^2 surface, and C^1 elsewhere. The resulting surface is hard to distinguish from a regular Loop surface. For a mesh that is not closed, we cannot use the presented subdivision rules. Instead, special rules have to be used for such boundaries. For Loop's scheme, the reflection rules of Equation 17.60 can be used. This is also treated in Section 17.5.3.

The surface after infinitely many subdivision steps is called the limit surface. Limit surface points and limit tangents can be computed using closed form expressions. The limit position of a vertex is computed [767, 1977] using the formula on the first row in Equation 17.61, by replacing $\beta(n)$ with

$$\gamma(n) = \frac{1}{n + \frac{3}{8\beta(n)}}. \tag{17.64}$$

Two limit tangents for a vertex $\mathbf{p}^k$ can be computed by weighting the immediate neighboring vertices, called the *1-ring* or *1-neighborhood*, as shown below [767, 1067]:

$$\mathbf{t}_u = \sum_{i=0}^{n-1} \cos(2\pi i/n)\mathbf{p}_i^k, \qquad \mathbf{t}_v = \sum_{i=0}^{n-1} \sin(2\pi i/n)\mathbf{p}_i^k. \tag{17.65}$$

The normal is then $\mathbf{n} = \mathbf{t}_u \times \mathbf{t}_v$. Note that this often is less expensive [1977] than the methods described in Section 16.3, which need to compute the normals of the neighboring triangles. More importantly, this gives the exact normal at the point.

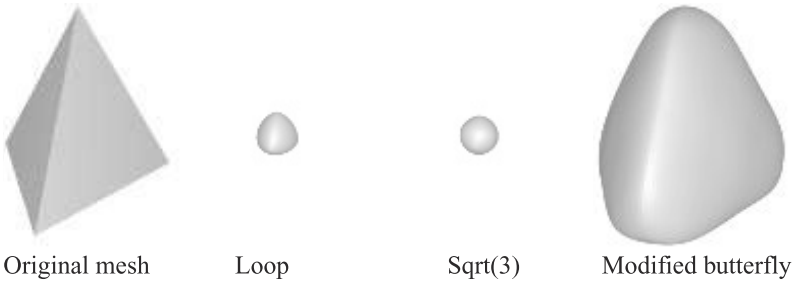

Figure 17.47. A tetrahedron is subdivided five times with Loop's, the $\sqrt{3}$, and the *modified butterfly* (MB) scheme [1975]. Loop's and the $\sqrt{3}$-scheme [915] are both approximating, while MB is interpolating, where the latter means that the initial vertices are located on the final surface. We cover only approximating schemes in this book due to their popularity in games and offline rendering.

A major advantage of approximating subdivision schemes is that the resulting surface tends to get fair. *Fairness* is, loosely speaking, related to how smoothly a curve or surface bends [1239]. A higher degree of fairness implies a smoother curve or surface. Another advantage is that approximating schemes converge faster than interpolating schemes. However, this means that the shapes often shrink. This is most notable for small, convex meshes, such as the tetrahedron shown in Figure 17.47. One way to decrease this effect is to use more vertices in the control mesh, i.e., care must be taken while modeling it. Maillot and Stam present a framework for combining subdivision schemes so that the shrinking can be controlled [1106]. A characteristic that can be used to great advantage at times is that a Loop surface is contained inside the convex hull of the original control points [1976].

The Loop subdivision scheme generates a generalized three-directional quartic box spline.[1] So, for a mesh consisting only of regular vertices, we could actually describe the surface as a type of spline surface. However, this description is not possible for irregular settings. Being able to generate smooth surfaces from any mesh of vertices is one of the great strengths of subdivision schemes. See also Sections 17.5.3 and 17.5.4 for different extensions to subdivision surfaces that use Loop's scheme.

17.5.2 Catmull-Clark Subdivision

The two most famous subdivision schemes that can handle polygonal meshes (rather than just triangles) are Catmull-Clark [239] and Doo-Sabin [370].[2] Here, we will only briefly present the former. Catmull-Clark surfaces have been used in Pixar's short film *Geri's Game* [347], *Toy Story 2*, and in all subsequent feature films from Pixar. This subdivision scheme is also commonly used for making models for games, and is probably the most popular one. As pointed out by DeRose et al. [347], Catmull-

[1] These spline surfaces are out of the scope of this book. Consult Warren's book [1847], the SIGGRAPH course [1977], or Loop's thesis [1067].

[2] Incidentally, both were presented in the same issue of the same journal.

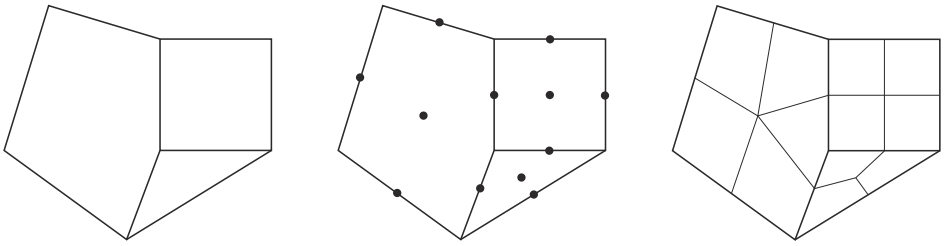

Figure 17.48. The basic idea of Catmull-Clark subdivision. Each polygon generates a new point, and each edge generates a new point. These are then connected as shown to the right. Weighting of the original points is not shown here.

Clark surfaces tend to generate more symmetrical surfaces. For example, an oblong box results in a symmetrical ellipsoid-like surface, which agrees with intuition. In contrast, a triangular-based subdivision scheme would treat each cube face as two triangles, and would hence generate different results depending on how the square is split.

The basic idea for Catmull-Clark surfaces is shown in Figure 17.48, and an actual example of Catmull-Clark subdivision is shown in Figure 17.41 on page 757. As can be seen, this scheme only generates faces with four vertices. In fact, after the first subdivision step, only vertices of valence 4 are generated, thus such vertices are called ordinary or regular (compared to valence 6 for triangular schemes).

Following the notation from Halstead et al. [655], let us focus on a vertex $\mathbf{v}^k$ with n surrounding edge points $\mathbf{e}_i^k$, where $i = 0 \ldots n-1$. See Figure 17.49. Now, for each face, a new face point $\mathbf{f}^{k+1}$ is computed as the face centroid, i.e., the mean of the points of the face. Given this, the subdivision rules are [239, 655, 1977]

$$\begin{aligned} \mathbf{v}^{k+1} &= \frac{n-2}{n}\mathbf{v}^k + \frac{1}{n^2}\sum_{j=0}^{n-1}\mathbf{e}_j^k + \frac{1}{n^2}\sum_{j=0}^{n-1}\mathbf{f}_j^{k+1}, \\ \mathbf{e}_j^{k+1} &= \frac{\mathbf{v}^k + \mathbf{e}_j^k + \mathbf{f}_{j-1}^{k+1} + \mathbf{f}_j^{k+1}}{4}. \end{aligned} \tag{17.66}$$

As can be seen, the vertex $\mathbf{v}^{k+1}$ is computed as weighting of the considered vertex, the average of the edge points, and the average of the newly created face points. On the other hand, new edge points are computed by the average of the considered vertex, the edge point, and the two newly created face points that have the edge as a neighbor.

The Catmull-Clark surface describes a generalized bicubic B-spline surface. So, for a mesh consisting only of regular vertices we could actually describe the surface as a bicubic B-spline surface (Section 17.2.6) [1977]. However, this is not possible for irregular mesh settings, and being able to handle these using subdivision surfaces is one of the scheme's strengths. Limit positions and tangents are also possible to

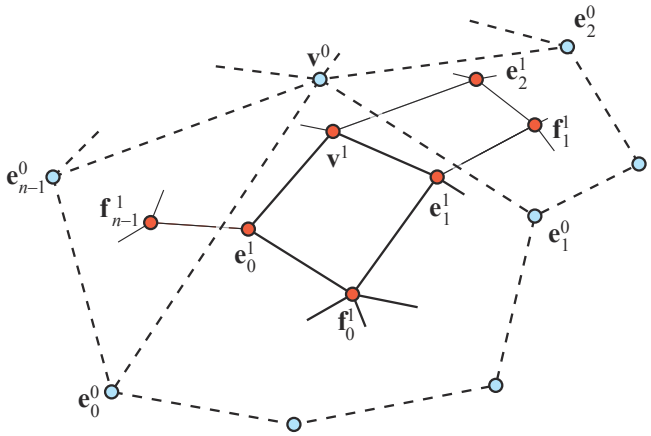

Figure 17.49. Before subdivision, we have the blue vertices and corresponding edges and faces. After one step of Catmull-Clark subdivision, we obtain the red vertices, and all new faces are quadrilaterals. *(Illustration after Halstead et al. [655].)*

compute, even at arbitrary parameter values using explicit formulae [1687]. Halstead et al. [655] describe a different approach to computing limit points and normals.

See Section 17.6.3 for a set of efficient techniques that can render Catmull-Clark subdivision surfaces using the GPU.

17.5.3 Piecewise Smooth Subdivision

In a sense, curved surfaces may be considered boring because they lack detail. Two ways to improve such surfaces are to use bump or displacement maps (Section 17.5.4). A third approach, *piecewise smooth subdivision*, is described here. The basic idea is to change the subdivision rules so that *darts*, *corners*, and *creases* can be used. This increases the range of different surfaces that can be modeled and represented. Hoppe et al. [767] first described this for Loop's subdivision surfaces. See Figure 17.50 for a comparison of a standard Loop subdivision surface, and one with piecewise smooth subdivision.

To actually be able to use such features on the surface the edges that we want to be sharp are first tagged, so we know where to subdivide differently. The number of sharp edges coming in at a vertex is denoted s. Then the vertices are classified into: smooth ($s = 0$), dart ($s = 1$), crease ($s = 2$), and corner ($s > 2$). Therefore, a crease is a curve on the surface, where the continuity across the curve is C^0. A dart is a non-boundary vertex where a crease ends and smoothly blends into the surface. Finally, a corner is a vertex where three or more creases come together. Boundaries can be defined by marking each boundary edge as sharp.

After classifying the various vertex types, Hoppe et al. use a table to determine which mask to use for the various combinations. They also show how to compute

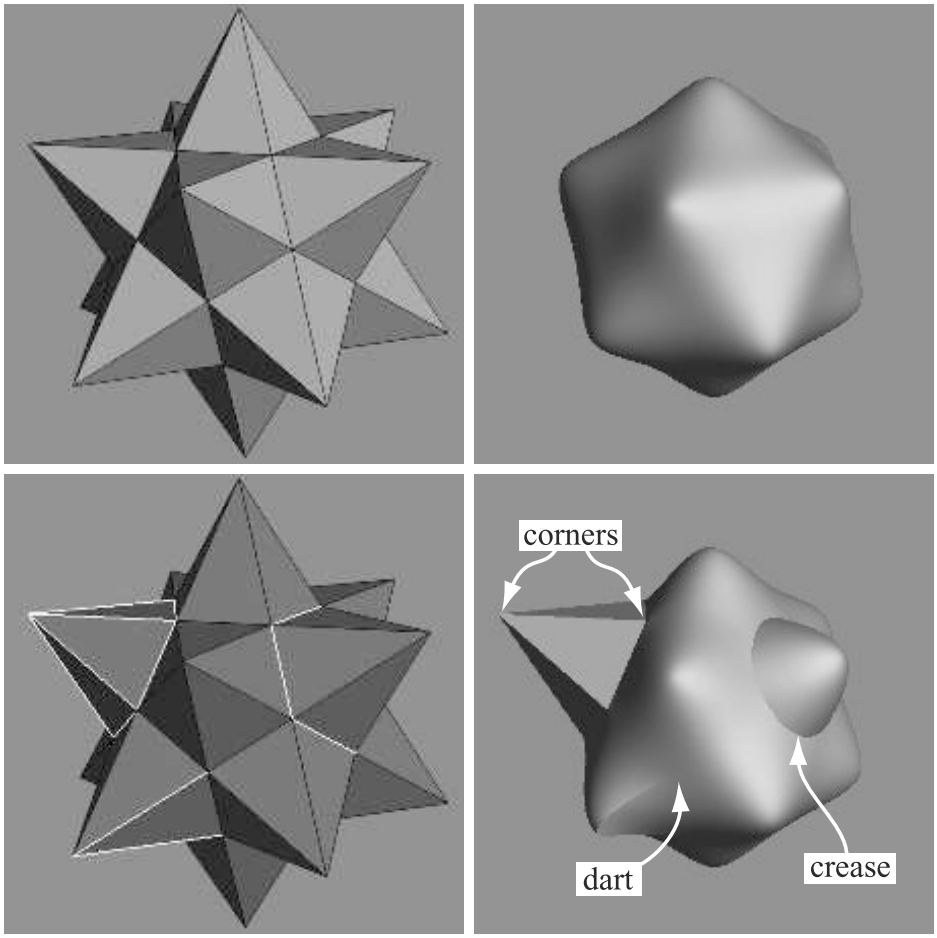

Figure 17.50. The top row shows a control mesh, and the limit surface using the standard Loop subdivision scheme. The bottom row shows piecewise smooth subdivision with Loop's scheme. The lower left image shows the control mesh with tagged edges (sharp) shown in a light gray. The resulting surface is shown to the lower right, with corners, darts, and creases marked. *(Image courtesy of Hugues Hoppe.)*

limit surface points and limit tangents. Biermann et al. [142] present several improved subdivision rules. For example, when extraordinary vertices are located on a boundary, the previous rules could result in gaps. This is avoided with the new rules. Also, their rules make it possible to specify a normal at a vertex, and the resulting surface will adapt to get that normal at that point. DeRose et al. [347] present a technique for creating soft creases. They allow an edge to first be subdivided as sharp a number of times (including fractions), and after that, standard subdivision is used.

17.5.4 Displaced Subdivision

Bump mapping (Section 6.7) is one way to add detail to otherwise smooth surfaces. However, this is just an illusionary trick that changes the normal or local occlusion at each pixel. The silhouette of an object looks the same with or without bump mapping. The natural extension of bump mapping is *displacement mapping* [287], where the surface is displaced. This is usually done along the direction of the normal. So, if the point of the surface is $\mathbf{p}$, and its normalized normal is $\mathbf{n}$, then the point on the displaced surface is

$$\mathbf{s}(u, v) = \mathbf{p}(u, v) + d(u, v)\mathbf{n}(u, v). \tag{17.67}$$

The scalar d is the displacement at the point $\mathbf{p}$. The displacement could also be vector-valued [938].

In this section, the *displaced subdivision surface* [1006] will be presented. The general idea is to describe a displaced surface as a coarse control mesh that is subdivided into a smooth surface that is then displaced along its normals using a scalar field. In the context of displaced subdivision surfaces, $\mathbf{p}$ in Equation 17.67 is the limit point on the subdivision surface (of the coarse control mesh), and $\mathbf{n}$ is the normalized normal at $\mathbf{p}$, computed as

$$\mathbf{n} = \frac{\mathbf{n}'}{||\mathbf{n}'||}, \quad \text{where} \quad \mathbf{n}' = \mathbf{p}_u \times \mathbf{p}_v, \tag{17.68}$$

In Equation 17.68, $\mathbf{p}_u$ and $\mathbf{p}_v$ are the first-order derivative of the subdivision surface. Thus, they describe two tangents at $\mathbf{p}$. Lee et al. [1006] use a Loop subdivision surface for the coarse control mesh, and its tangents can be computed using Equation 17.65. Note that the notation is slightly different here; we use $\mathbf{p}_u$ and $\mathbf{p}_v$ instead of $\mathbf{t}_u$ and $\mathbf{t}_v$. Equation 17.67 describes the displaced position of the resulting surface, but we also need a normal, $\mathbf{n}_s$, on the displaced subdivision surface in order to render it correctly. It is computed analytically as shown below [1006]:

$$\begin{aligned} &\mathbf{n}_s = \mathbf{s}_u \times \mathbf{s}_v, \quad \text{where} \\ &\mathbf{s}_u = \frac{\partial \mathbf{s}}{\partial u} = \mathbf{p}_u + d_u\mathbf{n} + d\mathbf{n}_u \quad \text{and} \quad \mathbf{s}_v = \frac{\partial \mathbf{s}}{\partial v} = \mathbf{p}_v + d_v\mathbf{n} + d\mathbf{n}_v. \end{aligned} \tag{17.69}$$

To simplify computations, Blinn [160] suggests that the third term can be ignored if the displacements are small. Otherwise, the following expressions can be used to compute $\mathbf{n}_u$ (and similarly $\mathbf{n}_v$) [1006]:

$$\begin{aligned} \bar{\mathbf{n}}_u &= \mathbf{p}_{uu} \times \mathbf{p}_v + \mathbf{p}_u \times \mathbf{p}_{uv}, \\ \mathbf{n}_u &= \frac{\bar{\mathbf{n}}_u - (\bar{\mathbf{n}}_u \cdot \mathbf{n})\mathbf{n}}{||\mathbf{n}'||}. \end{aligned} \tag{17.70}$$

Note that $\bar{\mathbf{n}}_u$ is not any new notation, it is merely a "temporary" variable in the computations. For an ordinary vertex (valence $n = 6$), the first and second

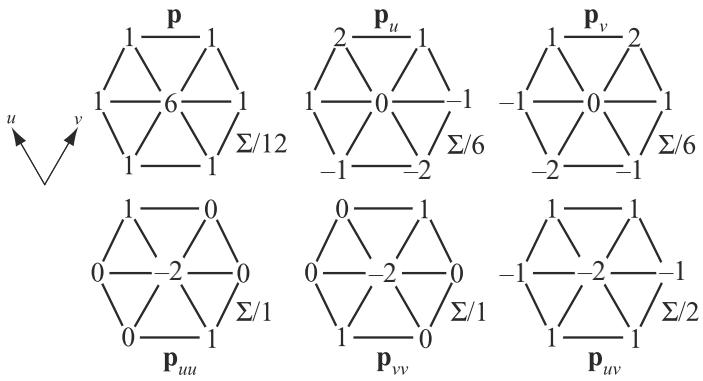

Figure 17.51. The masks for an ordinary vertex in Loop's subdivision scheme. Note that after using these masks, the resulting sum should be divided as shown. *(Illustration after Lee et al. [1006].)*

order derivatives are particularly simple. Their masks are shown in Figure 17.51. For an extraordinary vertex (valence $n \neq 6$), the third term in rows one and two in Equation 17.69 is omitted. An example using displacement mapping with Loop subdivision is shown in Figure 17.52.

When a displaced surface is far away from the viewer, standard bump mapping could be used to give the illusion of this displacement. Doing so saves on geometry processing. Some bump mapping schemes need a tangent-space coordinate system at the vertex, and the following can be used for that: $(\mathbf{b}, \mathbf{t}, \mathbf{n})$, where $\mathbf{t} = \mathbf{p}_u / ||\mathbf{p}_u||$ and $\mathbf{b} = \mathbf{n} \times \mathbf{t}$.

Nießner and Loop [1281] present a similar method as the one presented above by Lee et al., but they use Catmull-Clark surfaces and use direct evaluation of the derivatives on the displacement function, which is faster. They also use the hardware tessellation pipeline (Section 3.6) for rapid tessellation.

Figure 17.52. To the left is a coarse mesh. In the middle, it is subdivided using Loop's subdivision scheme. The right image shows the displaced subdivision surface. *(Image courtesy of Aaron Lee, Henry Moreton, and Hugues Hoppe.)*

17.5.5 Normal, Texture, and Color Interpolation

In this section, we will present different strategies for dealing with normals, texture coordinates and color per vertex.

As shown for Loop's scheme in Section 17.5.1, limit tangents, and thus, limit normals can be computed explicitly. This involves trigonometric functions that may be expensive to evaluate. Loop and Schaefer [1070] present an approximative technique where Catmull-Clark surfaces always are approximated by bicubic Bézier surfaces (Section 17.2.1). For normals, two tangent patches are derived, that is, one in the u-direction and one the v-direction. The normal is then found as the cross product between those vectors. In general, the derivatives of a Bézier patch are computed using Equation 17.35. However, since the derived Bézier patches approximate the Catmull-Clark surface, the tangent patches will not form a continuous normal field. Consult Loop and Schaefer's paper [1070] on how to overcome these problems. Alexa and Boubekeur [29] argue that it can be more efficient in terms of quality per computation to also subdivide the normals, which gives nicer continuity in the shading. We refer to their paper for the details on how to subdivide the normals. More types of approximations can also be found in Ni et al.'s SIGGRAPH course [1275].

Assume that each vertex in a mesh has a texture coordinate and a color. To be able to use these for subdivision surfaces, we also have to create colors and texture coordinates for each newly generated vertex, too. The most obvious way to do this is to use the same subdivision scheme as we used for subdividing the polygon mesh. For example, you can treat the colors as four-dimensional vectors (RGBA), and subdivide these to create new colors for the new vertices. This is a reasonable way to do it, since the color will have a continuous derivative (assuming the subdivision scheme is at least C^1), and thus abrupt changes in colors are avoided over the surface The same can certainly be done for texture coordinates [347]. However, care must be taken when there are boundaries in texture space. For example, assume we have two patches sharing an edge but with different texture coordinates along this edge. The geometry should be subdivided with the surface rules as usual, but the texture coordinates should be subdivided using boundary rules in this case.

A sophisticated scheme for texturing subdivision surfaces is given by Piponi and Borshukov [1419].

17.6 Efficient Tessellation

To display a curved surface in a real-time rendering context, we usually need to create a triangle mesh representation of the surface. This process is known as *tessellation*. The simplest form of tessellation is called *uniform tessellation*. Assume that we have a parametric Bézier patch, $\mathbf{p}(u, v)$, as described in Equation 17.32. We want to tessellate this patch by computing 11 points per patch side, resulting in $10 \times 10 \times 2 = 200$ triangles. The simplest way to do this is to sample the uv-space uniformly. Thus, we evaluate $\mathbf{p}(u, v)$ for all $(u_k, v_l) = (0.1k, 0.1l)$, where both k and l can be any

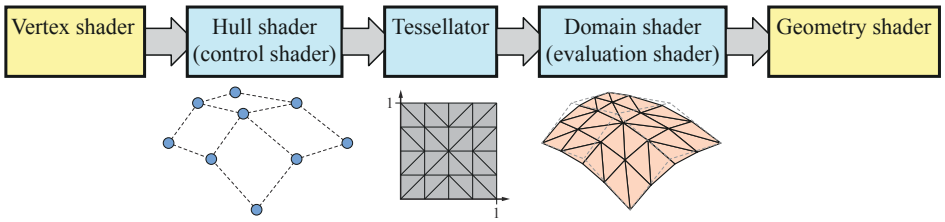

Figure 17.53. Pipeline with hardware tessellation, where the new stages are shown in the middle three (blue) boxes. We use the naming conventions from DirectX here, with the OpenGL counterparts in parenthesis. The hull shader computes new locations of control points and computes tessellation factors as well, which dictates how many triangles the subsequent step should generate. The tessellator generates points in the uv-space, in this case a unit square, and connects them into triangles. Finally, the domain shader computes the positions at each uv-coordinate using the control points.

integer from 0 to 10. This can be done with two nested `for`-loops. Two triangles can be created for the four surface points $\mathbf{p}(u_k, v_l)$, $\mathbf{p}(u_{k+1}, v_l)$, $\mathbf{p}(u_{k+1}, v_{l+1})$, and $\mathbf{p}(u_k, v_{l+1})$.

While this certainly is straightforward, there are faster ways to do it. Instead of sending tessellated surfaces, consisting of many triangles, over the bus from the CPU to the GPU, it makes more sense to send the curved surface representation to the GPU and let it handle the data expansion. Recall that the tessellation stage is described in Section 3.6. For a quick refresher, see Figure 17.53.

The tessellator may use a fractional tessellation technique, which is described in the following section. Then follows a section on adaptive tessellation, and finally, we describe how to render Catmull-Clark surfaces and displacement mapped surfaces with tessellation hardware.

17.6.1 Fractional Tessellation

To obtain smoother level of detail for parametric surfaces, Moreton introduced *fractional tessellation factors* [1240]. These factors enable a limited form of adaptive tessellation, since different tessellation factors can be used on different sides of the parametric surface. Here, an overview of how these techniques work will be presented.

In Figure 17.54, constant tessellation factors for rows and columns are shown on the left, and independent tessellation factors for all four edges on the right. Note that the tessellation factor of an edge is the number of points generated on that edge, minus one. In the patch on the right, the greater of the top and bottom factors is used in the interior for both of these edges, and similarly the greater of the left and right factors is used in the interior. Thus, the basic tessellation rate is 4×8. For the sides with smaller factors, triangles are filled in along the edges. Moreton [1240] describes this process in more detail.

The concept of fractional tessellation factors is shown for an edge in Figure 17.55. For an integer tessellation factor of n, $n + 1$ points are generated at k/n, where $k = 0, \ldots, n$. For a fractional tessellation factor, r, $\lceil r \rceil$ points are generated at k/r,

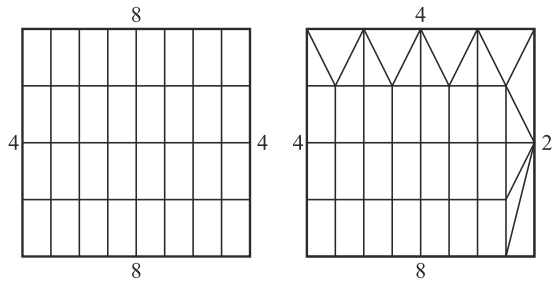

Figure 17.54. Left: normal tessellation—one factor is used for the rows, and another for the columns. Right: independent tessellation factors on all four edges. *(Illustration after Moreton [1240].)*

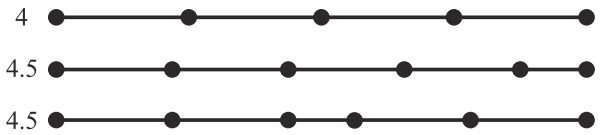

Figure 17.55. Top: integer tessellation. Middle: fractional tessellation, with the fraction to the right. Bottom: fractional tessellation with the fraction in the middle. This configuration avoids cracks between adjacent patches.

where $k = 0, \ldots, \lfloor r \rfloor$. Here, $\lceil r \rceil$ computes the *ceiling* of r, which is the closest integer toward $+\infty$, and $\lfloor r \rfloor$ computes the *floor*, which is the closest integer toward $-\infty$. Then, the rightmost point is just "snapped" to the rightmost endpoint. As can be seen in the middle illustration in Figure 17.55, this pattern in not symmetric. This leads to problems, since an adjacent patch may generate the points in the other direction, and so give cracks between the surfaces. Moreton solves this by creating a symmetric pattern of points, as shown at the bottom of Figure 17.55. See also Figure 17.56 for an example.

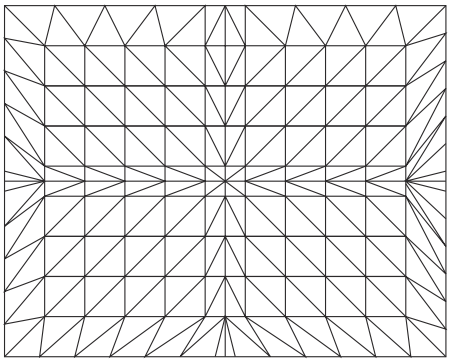

Figure 17.56. A patch with the rectangular domain fractionally tessellated. *(Illustration after Moreton [1240].)*

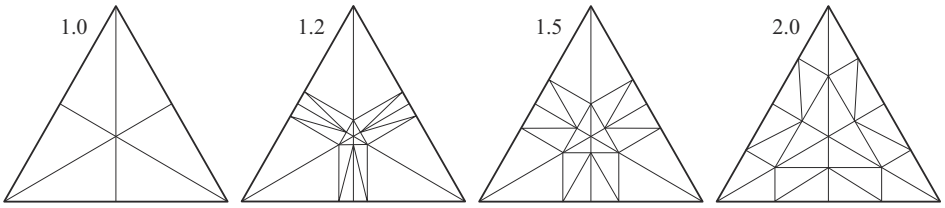

Figure 17.57. Fractional tessellation of a triangle, with tessellation factors shown. Note that the tessellation factors may not correspond exactly to those produced by actual tessellation hardware. (*Illustration after Tatarchuk [1745].*)

So far, we have seen methods for tessellating surfaces with a rectangular domain, e.g., Bézier patches. However, triangles can also be tessellated using fractions [1745], as shown in Figure 17.57. Like the quadrilaterals, it is also possible to specify an independent fractional tessellation rate per triangle edge. As mentioned, this enables adaptive tessellation (Section 17.6.2), as illustrated in Figure 17.58, where displacement mapped terrain is rendered. Once triangles or quadrilaterals have been created, they can be forwarded to the next step in the pipeline, which is treated in the next subsection.

17.6.2 Adaptive Tessellation

Uniform tessellation gives good results if the sampling rate is high enough. However, in some regions on a surface there may not be as great a need for high tessellation as in other regions. This may be because the surface bends more rapidly in some area and therefore may need higher tessellation there, while other parts of the surface are almost flat or far away, and only a few triangles are needed to approximate them. A solution to the problem of generating unnecessary triangles is *adaptive tessellation*, which refers to algorithms that adapt the tessellation rate depending on some measure (for example

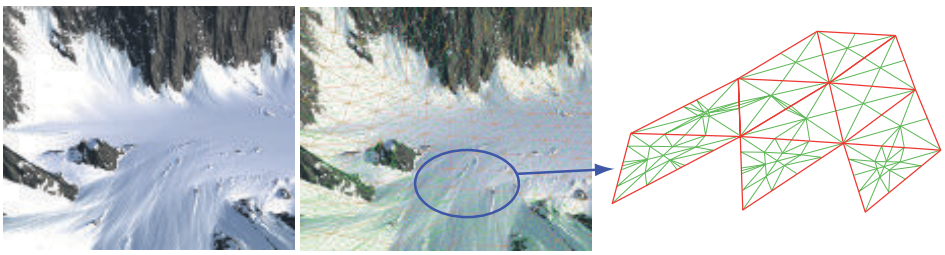

Figure 17.58. Displaced terrain rendering using adaptive fractional tessellation. As can be seen in the zoomed-in mesh to the right, independent fractional tessellation rates are used for the edges of the red triangles, which gives us adaptive tessellation. *(Images courtesy of Game Computing Applications Group, Advanced Micro Devices, Inc.)*

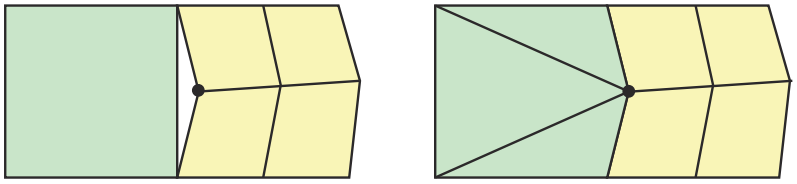

Figure 17.59. To the left, a crack is visible between the two regions. This is because the right has a higher tessellation rate than the left. The problem lies in that the right region has evaluated the surface where there is a black circle, and the left region has not. The standard solution is shown to the right.

curvature, triangle edge length, or some screen size measure). Figure 17.58 shows an example of adaptive tessellation for terrain.

Care must be taken to avoid cracks that can appear between different tessellated regions. See Figure 17.59. When using fractional tessellation, it is common to base the edge tessellation factors on information that comes from only the edge itself, since the edge data are all that is shared between the two connected patches. This is a good start, but due to floating point inaccuracies, there can still be cracks. Nießner et al. [1279] discuss how to make the computations fully watertight, e.g., by making sure that, for an edge, the exact same point is returned regardless of whether tessellation is done from $\mathbf{p}_0$ to $\mathbf{p}_1$, or vice versa.

In this section, we will present some general techniques that can be used to compute fractional tessellation rates, or to decide when to terminate further tessellation and when to split larger patches into a set of smaller ones.

Terminating Adaptive Tessellation

To provide adaptive tessellation, we need to determine when to stop the tessellation, or equivalently how to compute fractional tessellation factors. Either you can use only the information of an edge to determine if tessellation should be terminated, or use information from an entire triangle, or a combination.

It should also be noted that with adaptive tessellation, one can get swimming or popping artifacts from frame to frame if the tessellation factors for a certain edge change too much from one frame to the next. This may be a factor to take into consideration when computing tessellation factors as well. Given an edge, $(\mathbf{a}, \mathbf{b})$, with an associated curve, i.e., a patch edge curve, we can try to estimate how flat the curve is between $\mathbf{a}$ and $\mathbf{b}$. See Figure 17.60. The midpoint in parametric space between $\mathbf{a}$ and $\mathbf{b}$ is found, and its three-dimensional counterpart, $\mathbf{c}$, is computed. Finally, the length, l, between $\mathbf{c}$ and its projection, $\mathbf{d}$, onto the line between $\mathbf{a}$ and $\mathbf{b}$, is computed. This length, l, is used to determine whether the curve segment on that edge if flat enough. If l is small enough, it is considered flat. Note that this method may falsely consider an S-shaped curve segment to be flat. A solution to this is to randomly perturb the parametric sample point [470]. An alternative to using just l is to use

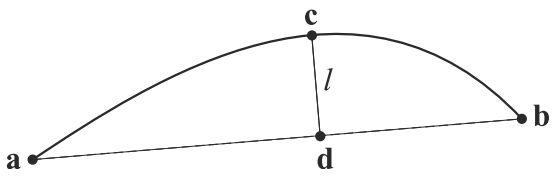

Figure 17.60. The points **a** and **b** have already been generated on this surface. The question is: Should a new point, that is **c**, be generated on the surface?

the ratio $l/||\mathbf{a}-\mathbf{b}||$, giving a relative measure [404]. Note that this technique can be extended to consider a triangle as well, where you simply compute the surface point in the middle of the triangle and use the distance from that point to the triangle's plane. To be certain that this type of algorithm terminates, it is common to set some upper limit on how many subdivisions can be made. When that limit is reached, the subdivision ends. For fractional tessellation, the vector from **c** to **d** can be projected onto the screen, and its (scaled) length used as a tessellation rate.

So far we have discussed how to determine the tessellation rate from only the shape of the surface. Other factors that typically are used for on-the-fly tessellation include whether the local neighborhood of a vertex is [769, 1935]:

1. Inside the view frustum.
2. Frontfacing.
3. Occupying a large area in screen space.
4. Close to the silhouette of the object.

These factors will be discussed in turn here. For view frustum culling, one can place a sphere to enclose the edge. This sphere is then tested against the view frustum. If it is outside, we do not subdivide that edge further.

For face culling, the normals at **a**, **b**, and possibly **c** can be computed from the surface description. These normals, together with **a**, **b**, and **c**, define three planes. If all are backfacing, it is likely that no further subdivision is needed for that edge.

There are many different ways to implement screen-space coverage (see also Section 19.9.2). All methods project some simple object onto the screen and estimate the length or area in screen space. A large area or length implies that tessellation should proceed. A fast estimation of the screen-space projection of a line segment from **a** to **b** is shown in Figure 17.61. First, the line segment is translated so that its midpoint is on the view ray. Then, the line segment is assumed to be parallel to the near plane, n, and the screen-space projection, s, is computed from this line segment. Using the points of the line segment $\mathbf{a}'$ and $\mathbf{b}'$ to the right in the illustration, the screen-space projection is then

$$s = \frac{\sqrt{(\mathbf{a}'-\mathbf{b}')\cdot(\mathbf{a}'-\mathbf{b}')}}{\mathbf{v}\cdot(\mathbf{a}'-\mathbf{e})}. \tag{17.71}$$

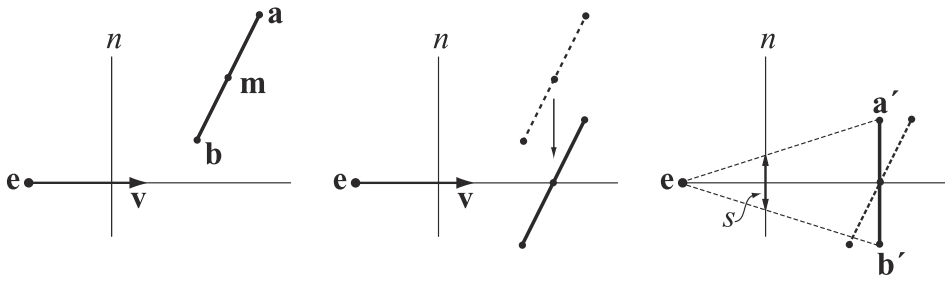

Figure 17.61. Estimation of the screen-space projection, s, of the line segment.

The numerator is simply the length of the line segment. This is divided by the distance from the eye, $\mathbf{e}$, to the line segment's midpoint. The computed screen-space projection, s, is then compared to a threshold, t, representing the maximum edge length in screen space. Rewriting the previous equation to avoid computing the square root, if the following condition is true, the tessellation should continue:

$$s > t \quad \Longleftrightarrow \quad (\mathbf{a}' - \mathbf{b}') \cdot (\mathbf{a}' - \mathbf{b}') > t^2(\mathbf{v} \cdot (\mathbf{a}' - \mathbf{e}))^2. \tag{17.72}$$

Note that t^2 is a constant and so can be precomputed. For fractional tessellation, s from Equation 17.71 can be used as the tessellation rate, possibly with a scaling factor applied. Another way to measure projected edge length is to place a sphere at the center of the edge, make the radius half the edge length, and then use the projection of the sphere as the edge tessellation factor [1283]. This test is proportional to the area, while the test above is proportional to edge length.

Increasing the tessellation rate for the silhouettes is important, since they play a prime role for the perceived quality of the object. Finding if a triangle is near a silhouette edge can be done by testing whether the dot product between the normal at $\mathbf{a}$ and the vector from the eye to $\mathbf{a}$ is close to zero. If this is true for any of $\mathbf{a}$, $\mathbf{b}$, or $\mathbf{c}$, further tessellation should be done.

For displaced subdivision, Nießner and Loop [1281] use one of the following factors for each base mesh vertex $\mathbf{v}$, which is connected to n edge vectors $\mathbf{e}_i$, $i \in \{0, 1, \ldots, n-1\}$:

$$\begin{aligned} f_1 &= k_1 \cdot ||\mathbf{c} - \mathbf{v}||, \\ f_2 &= k_2 \sqrt{\sum \mathbf{e}_i \times \mathbf{e}_{i+1}}, \\ f_3 &= k_3 \max\left(||\mathbf{e}_0||, ||\mathbf{e}_1||, \ldots, ||\mathbf{e}_{n-1}||\right), \end{aligned} \tag{17.73}$$

where the loop index, i, goes over all n edges $\mathbf{e}_i$ connected to $\mathbf{v}$, $\mathbf{c}$ is the position of the camera, and k_i are user-supplied constants. Here, f_1 is simply based on the distance to the vertex from the camera, f_2 computes the area of the quads connected to $\mathbf{v}$, and f_3 uses the largest edge length. The tessellation factor for a vertex is

then computed as the maximum of the edge's two base vertices' tessellation factors. The inner tessellation factors are computed as the maximum of the opposite edge's tessellation factor (in u and v). This method can be used with any of the edge tessellation factor methods presented in this section.

Notably, Nießner et al. [1279] recommend using a single global tessellation factor for characters, which depends on the distance to the character. The number of subdivisions is then $\lceil \log_2 f \rceil$, where f is the per-character tessellation factor, which can be computed using any of the methods above.

It is hard to say what methods will work in all applications. The best advice is to test several of the presented heuristics, and combinations of them.

Split and Dice Methods

Cook et al. [289] introduced a method called *split and dice*, with the goal being to tessellate surfaces so that each triangle becomes pixel-sized, to avoid geometrical aliasing. For real-time purposes, this tessellation threshold should be increased to what the GPU can handle. Each patch is first split recursively into a set of subpatches until it is estimated that if uniform tessellation is used for a certain subpatch, the triangles will have the desired size. Hence, this is also a type of adaptive tessellation.

Imagine a single large patch being used for a landscape. In general, there is no way that fractional tessellation can adapt so that the tessellation rate is higher closer to the camera and lower farther away, for example. The core of split and dice may therefore be useful for real-time rendering, even if the target tessellation rate is to have larger triangles than pixel-sized in our case.

Next, we describe the general method for split and dice in a real-time graphics scenario. Assume that a rectangular patch is used. Then start a recursive routine with the entire parametric domain, i.e., the square from $(0, 0)$ to $(1, 1)$. Using the adaptive termination criteria just described, test whether the surface is tessellated enough. If it is, then terminate tessellation. Otherwise, split this domain into four different equally large squares and call the routine recursively for each of the four subsquares. Continue recursively until the surface is tessellated enough or a predefined recursion level is reached. The nature of this algorithm implies that a quadtree is recursively created during tessellation. However, this will give cracks if adjacent subsquares are tessellated to different levels. The standard solution is to ensure that two adjacent subsquares only differ in one level at most. This is called a *restricted quadtree*. Then the technique shown to the right in Figure 17.59 is used to fill in the cracks. A disadvantage with this method is that the bookkeeping is more involved.

Liktor et al. [1044] present a variant of split and dice for the GPU. The problem is to avoiding swimming artifacts and popping effects when suddenly deciding to split one more time due to, e.g., the camera having moved closer to a surface. To solve that, they use a fractional split method, inspired by fractional tessellation. This is illustrated in Figure 17.62. Since the split is smoothly introduced from one side toward the center of the curve, or toward the center of the patch side, swimming and popping artifacts are avoided. When the termination criteria for the adaptive tessellation have

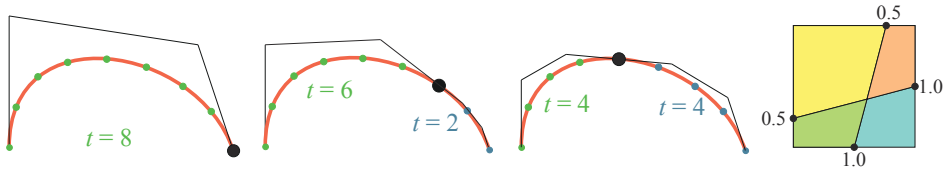

Figure 17.62. Fractional splitting is applied to a cubic Bézier curve. The tessellation rate t is shown for each curve. The split point is the large black circle, which is moved in from the right side of the curve toward the center of the curve. To fractionally split the cubic curve, the black point is smoothly moved toward the curve's center and the original curve is replaced with two cubic Bézier segments that together generate the original curve. To the right, the same concept is illustrated for a patch, which has been split into four smaller subpatches, where 1.0 indicates that the split point is on the center point of the edge, and 0.0 means that it is at the patch corner. (*Illustration after Liktor et al. [1044].*)

been reached, each remaining subpatch is tessellated by the GPU using fractional tessellation as well.

17.6.3 Fast Catmull-Clark Tessellation

Catmull-Clark surfaces (Section 17.5.2) are frequently used in modeling software and in feature film rendering, and hence it is attractive to be able to render these efficiently using graphics hardware as well. Fast tessellation methods for Catmull-Clark surfaces have been an active research field over recent years. Here, we will present a handful of these methods.

Approximating Approaches

Loop and Schaefer [1070] present a technique to convert Catmull-Clark surfaces into a representation that can be evaluated quickly in the domain shader, without the need to know the polygons' neighbors.

As mentioned in Section 17.5.2, the Catmull-Clark surface can be described as many small B-spline surfaces when all vertices are ordinary. Loop and Schaefer convert a quadrilateral (quad) polygon in the original Catmull-Clark subdivision mesh to a bi-cubic Bézier surface (Section 17.2.1). This is not possible for non-quadrilaterals, and so we assume that there are no such polygons (recall that after the first step of subdivision there are only quadrilateral polygons). When a vertex has a valence different from four, it is not possible to create a bicubic Bézier patch that is identical to the Catmull-Clark surface. Hence, an approximative representation is proposed, which is exact for quads with valence-four vertices, and close to the Catmull-Clark surface elsewhere. To this end, both *geometry patches* and *tangent patches* are used, which will be described next.

The geometry patch is simply a bicubic Bézier patch with 4×4 control points. We will describe how these control points are computed. Once this is done, the patch can be tessellated and the domain shader can evaluate the Bézier patch quickly at

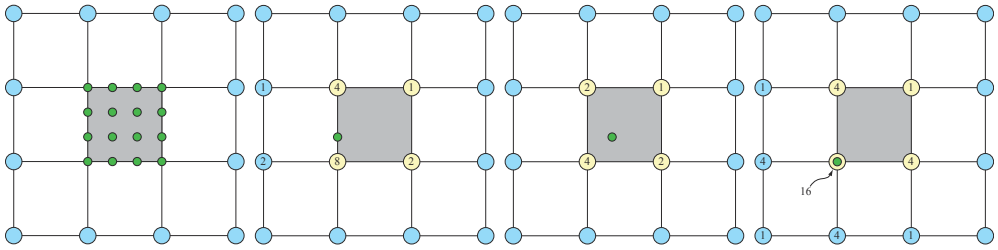

Figure 17.63. Left: part of a quad mesh, where we want to compute a Bézier patch for the gray quad. Note that the gray quad has vertices with only valence four. The blue vertices are neighboring quads' vertices, and the green circles are the control points of the Bézier patch. The following three illustrations show the different masks used to compute the green control points. For example, to compute one of the inner control points, the middle right mask is used, and the vertices of the quad are weighted with the weight shown in the mask.

any parametric coordinate (u, v). So, assuming we have a mesh consisting of only quads with vertices of valence four, we want to compute the control points of the corresponding Bézier patch for a certain quad in the mesh. To that end, the neighborhood around the quad is needed. The standard way of doing this is illustrated in Figure 17.63, where three different masks are shown. These can be rotated and reflected in order to create all the 16 control points. Note that in an implementation the weights for the masks should sum to one, a process that is omitted here for clarity.

The above technique computes a Bézier patch for the ordinary case. When there is at least one extraordinary vertex, we compute an extraordinary patch [1070]. The masks for this are shown in Figure 17.64, where the lower left vertex in the gray quad is an extraordinary vertex.

Note that this results in a patch that approximates the Catmull-Clark subdivision surface, and, it is only C^0 along edges with an extraordinary vertex. This often looks distracting when shading is added, and hence a similar trick as used for N-patches (Section 17.2.4) is suggested. However, to reduce the computational complexity, two

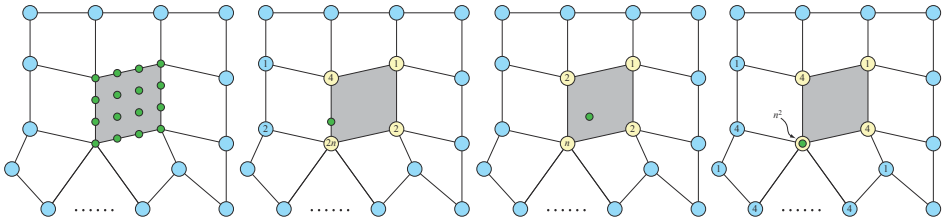

Figure 17.64. Left: a Bézier patch for the gray quad of the mesh is to be generated. The lower left vertex in the gray quad is extraordinary, since its valence is $n \neq 4$. The blue vertices are neighboring quads' vertices, and the green circles are the control points of the Bézier patch. The following three illustrations show the different masks used to compute the green control points.

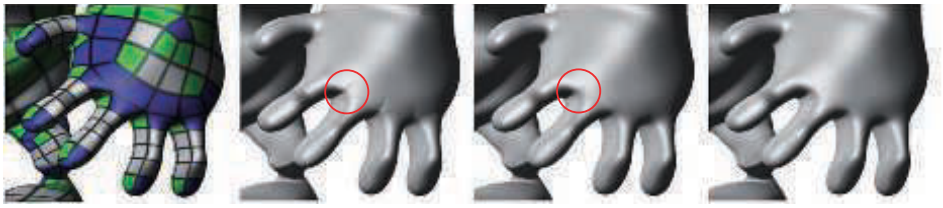

Figure 17.65. Left: quad structure of a mesh. White quads are ordinary, green have one extraordinary vertex, and blue have more than one. Middle left: geometry patch approximation. Middle right: geometry patches with tangent patches. Note that the obvious (red circle) shading artifacts disappeared. Right: true Catmull-Clark surface. *(Image courtesy of Charles Loop and Scott Schaefer, reprinted with permission from Microsoft Corporation.)*

tangent patches are derived: one in the u-direction, and one the v-direction. The normal is then found as the cross product between those vectors. In general, the derivatives of a Bézier patch are computed using Equation 17.35. However, since the derived Bézier patches approximate the Catmull-Clark surface, the tangent patches will not form a continuous normal field. Consult Loop and Schaefer's paper [1070] on how to overcome these problems. Figure 17.65 shows an example of the types of artifacts that can occur.

Kovacs et al. [931] describe how the method above can be extended to also handle creases and corners (Section 17.5.3), and implement these extensions in Valve's Source engine.

Feature Adaptive Subdivision and OpenSubdiv

Pixar presented an open-source system called OpenSubdiv that implements a set of techniques called *feature adaptive subdivision* (FAS) [1279, 1280, 1282]. The basic approach is rather different from the previous technique just discussed. The foundation of this work lies in that subdivision is equivalent to bicubic B-spline patches (Section 17.2.6) for regular faces, i.e., quads where each vertex is regular, which means that the vertex has valence four. So, subdivision continues recursively only for non-regular faces, up until some maximum subdivision level is reached. This is illustrated on the left in Figure 17.66. FAS can also handle creases and semi-smooth creases [347], and the FAS algorithm also needs to subdivide around such creases, which is illustrated on the right in Figure 17.66. The bicubic B-spline patches can be rendered directly using the tessellation pipeline.

The method starts by creating a table using the CPU. This table encodes indices to vertices that need to be accessed during subdivision up to a specified level. As such, the base mesh can be animated since the indices are independent of the vertex positions. As soon as a bicubic B-spline patch is generated, recursion needs not to be continued, which means that the table usually becomes relatively small. The base mesh and the table with indices and additional valence and crease data are uploaded to the GPU once.

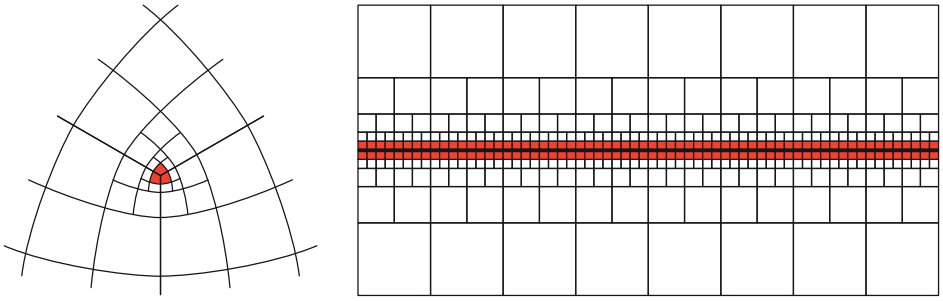

Figure 17.66. Left: recursive subdivision around an extraordinary vertex, i.e., the middle vertex has three edges. As recursion continues, it leaves a band of regular patches (with four vertices, each with four incoming edges) behind. Right: subdivision around a smooth crease indicated by the bold line in the middle. (*Illustration after Nießner et al. [1279].*)

To subdivide the mesh one step, new face points are computed first, followed by new edge points, and finally the vertices are updated, and one compute shader is used for each of these types. For rendering, a distinction between full patches and transition patches is made. A full patch (FP) only shares edges with patches of the same subdivision level, and a regular FP is directly rendered as a bicubic B-spline patch using the GPU tessellation pipeline. Subdivision continues otherwise. The adaptive subdivision process ensures that there is at most a difference of one subdivision level between neighboring patches. A transition patch (TP) has a difference in subdivision level against at least one neighbor. To get crack-free renderings, each TP is split into several subpatches as shown in Figure 17.67. In this way, tessellated vertices match along both sides of each edge. Each type of subpatch is rendered using different hull and domain shaders that implement interpolation variants. For example, the leftmost case in Figure 17.67 is rendered as three triangular B-spline patches. Around extraordinary vertices, another domain shader is used where limit positions and limit normals are computed using the method of Halstead et al. [655]. An example of Catmull-Clark surface rendering using OpenSubdiv is shown in Figure 17.68.

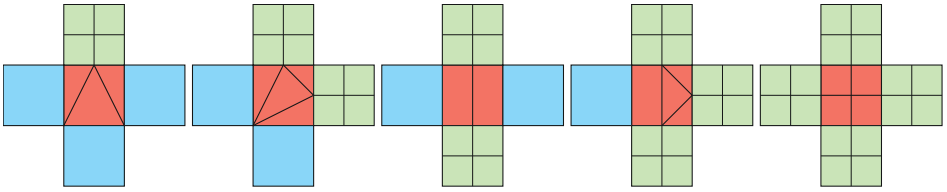

Figure 17.67. Red squares are transition patches, and each has four immediate neighbors, which are either blue (current subdivision level) or green (next subdivision level). This illustration shows the five configurations that can occur, and how they are stitched together. (*Illustration after Nießner et al. [1279].*)

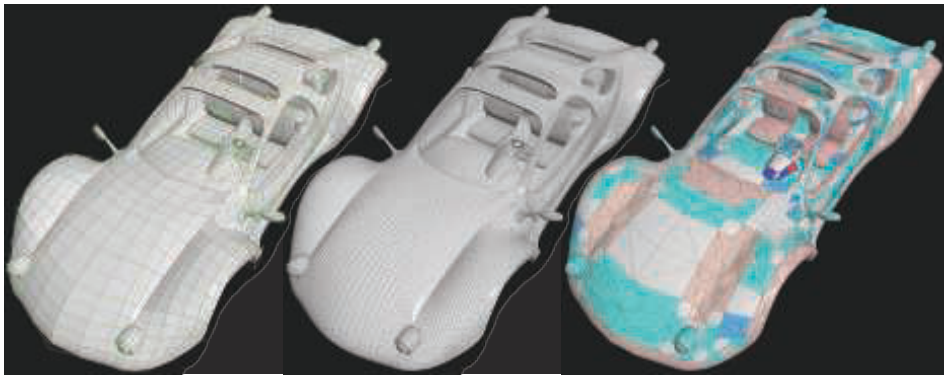

Figure 17.68. Left: the control mesh in green and red lines with the gray surface (8k vertices) generated using one subdivision step. Middle: the mesh subdivided an additional two steps (102k vertices). Right: the surface generated using adaptive tessellation (28k vertices). (*Images generated using OpenSubdiv's dxViewer.*)

The FAS algorithm handles creases, semi-smooth creases, hierarchical details, and adaptive level of detail. We refer to the FAS paper [1279] and Nießner's PhD thesis [1282] for more details. Schäfer et al. [1547] presents a variant of FAS, called DFAS, which is even faster.

Adaptive Quadtrees

Brainerd et al. [190] present a method called *adaptive quadtrees*. It is similar to the approximating scheme by Loop and Schaefer [1070] in that a single tessellated primitive is submitted per quad of the original base mesh. In addition, it precomputes a subdivision plan, which is a quadtree that encodes the hierarchical subdivision (similar to feature adaptive subdivision) from an input face, down to some maximum subdivision level. The subdivision plan also contains a list of stencil masks of the control points needed by the subdivided faces.

During rendering, the quadtree is traversed, making it possible to map (u, v)-coordinates to a patch in the subdivision hierarchy, which can be directly evaluated. A quadtree leaf is a subregion of the domain of the original face, and the surface in this subregion can be directly evaluated using the control points in the stencil. An iterative loop is used to traverse the quadtree in the domain shader, whose input is a parametric (u, v)-coordinate. Traversal needs to continue until a leaf node is reached in which the (u, v)-coordinates are located. Depending on the type of node reached in the quadtree, different actions are taken. For example, when reaching a subregion that can be evaluated directly, the 16 control points for its corresponding bicubic B-spline patch are retrieved, and the shader continues with evaluating that patch.

See Figure 17.1 on page 718 for an example rendered using this technique. This method is the fastest to date that renders Catmull-Clark subdivision surfaces exactly,