Chapter 12

TEXTURE MAPPING

The shading equation contains several parameters referring to the optical properties of the surface interacting with the light, including k_a ambient, k_d diffuse, k_s specular, k_r reflective, k_t transmissive coefficients, ν index of refraction etc. These parameters are not necessarily constant over the surface, thus allowing surface details, called **textures**, to appear on computer generated images. Texture mapping requires the determination of the surface parameters each time the shading equation is calculated for a point on the surface. Recall that for ray tracing, the ray-object intersection calculation provides the visible surface point in the world coordinate system. For incremental methods, however, the surfaces are transformed to the screen coordinate system where the visibility problem is solved, and thus the surface points to be shaded are defined in screen coordinates.

Recall, too, that the shading equation is evaluated in the world coordinate system even if the visible surface points are generated in screen space, because the world-screen transformation usually modifies the angle vectors needed for the shading equation. For directional and ambient lightsource only models, the shading equation can also be evaluated in screen space, but the normal vectors defined in the world coordinate system must be used.

The varying optical parameters required by the shading equation, on the other hand, are usually defined and stored in a separate coordinate system, called **texture space**. The texture information can be represented by some data stored in an array or by a function that returns the value needed for the points of the texture space. In order for there to be a correspondence between texture space data and the points of the surface, a transformation is associated with the texture, which maps texture space onto the surface defined in its local coordinate system. This transformation is called **parameterization**.

Modeling transformation maps this local coordinate system point to the world coordinate system where the shading is calculated. In ray tracing the visibility problem is also solved here. Incremental shading models, however, need another transformation from world coordinates to screen space where the hidden surface elimination and simplified color computation take place. This latter mapping is regarded as **projection** in texture mapping (figure 12.1).

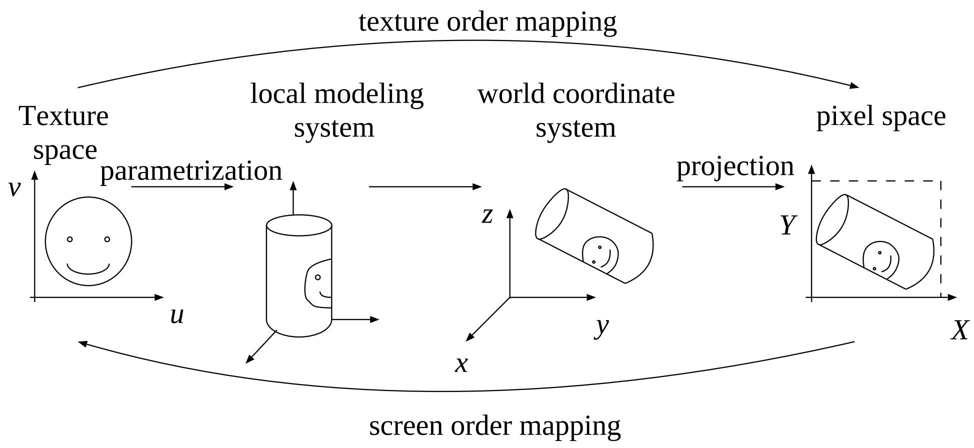

Figure 12.1: Survey of texture mapping

Since the parameters of the shading equation are required in screen space, but available only in texture space, the mapping between the two spaces must be evaluated for each pixel to be shaded.

Generally two major implementations are possible:

1. **Texture order** or **direct mapping** which scans the data in texture space and maps from texture space to screen space.

2. **Screen order** or **inverse mapping** which scans the pixels in screen space and uses the mapping from screen space to texture space.

Texture order mapping seems more effective, especially for large textures stored in files, since they access the texture sequentially. Unfortunately, there is no guarantee that, having transformed uniformly sampled data from texture space to screen space, all pixels belonging to a given surface will be produced. Holes and overlaps may occur on the image. The correct sampling of texture space, which produces all pixels needed, is a difficult problem if the transformation is not linear. Since texture order methods access texture elements one after the other they also process surfaces sequentially, making themselves similar to and appropriate for object precision hidden surface algorithms.

Image precision algorithms, on the other hand, evaluate pixels sequentially and thus require screen order techniques and random access of texture maps. A screen order mapping can be very slow if texture elements are stored on a sequential medium, and it needs the calculation of the inverse parameterization which can be rather difficult. Nevertheless, screen order is more popular, because it is appropriate for image precision hidden surface algorithms.

Interestingly, although the z-buffer method is an image precision technique, it is also suitable for texture order mapping, because it processes polygons sequentially.

The texture space can be either one-dimensional, two-dimensional or three-dimensional. A one-dimensional texture has been proposed, for example, to simulate the thin film interference produced on a soap bubble, oil and water [Wat89].

Two-dimensional textures can be generated from frame-grabbed or computer synthesized images and are glued or "wallpapered" onto a three-dimensional object surface. The "wallpapers" will certainly have to be distorted to meet topological requirements. The 2D texture space can generally be regarded as a unit square in the center of a u, v texture coordinate system. Two-dimensional texturing reflects our subjective concept of surface painting, and that is one of the main reasons why it is the most popular texturing method.

Three-dimensional textures, also called **solid textures**, neatly circumvent the parameterization problem, since they define the texture data in the 3D local coordinate system — that is in the same space where the geometry is defined — simplifying the parameterization into an identity transformation. The memory requirements of this approach may be prohibitive, however, and thus three-dimensional textures are commonly limited to functionally defined types only. Solid texturing is basically the equivalent of carving the object out of a block of material. It places the texture onto the object coherently, not producing discontinuities of texture where two faces meet, as 2D texturing does. The simulation of wood grain on a cube, for example, is only possible by solid texturing in order to avoid discontinuities of the grain along the edges of the cube.

12.1 Parameterization for two-dimensional textures

Parameterization connects the unit square of 2D texture space to the 3D object surface defined in the local modeling coordinate system.

12.1.1 Parameterization of parametric surfaces

The derivation of this transformation is straightforward if the surface is defined parametrically over the unit square by a positional vector function:

$$\vec{r}(u,v) = \begin{bmatrix} x(u,v) \\ y(u,v) \\ z(u,v) \end{bmatrix}. \tag{12.1}$$

Bezier and bicubic parametric patches fall into this category. For other parametric surfaces, such as B-spline surfaces, or in cases where only a portion of a Bezier surface is worked with, the definition is similar, but texture coordinates come from a rectangle instead of a unit square. These surfaces can also be easily parameterized, since only a linear mapping which transforms the rectangle onto the unit square before applying the parametric functions is required.

For texture order mapping, these formulae can readily be applied in order to obtain corresponding $\vec{r}(u,v)$ 3D points for u, v texture coordinates. For scan order mapping, however, the inverse of $\vec{r}(u,v)$ has to be determined, which requires the solution of a non-linear equation.

12.1.2 Parameterization of implicit surfaces

Parameterization of an implicitly defined surface means the derivation of an explicit equation for that surface. The ranges of the natural parameters may not fall into a unit square, thus making an additional linear mapping necessary. To explain this idea, the examples of the sphere and cylinder are taken.

Parameterization of a sphere

The implicit definition of a sphere around a point (x_c, y_c, z_c) with radius r is:

$$(x - x_c)^2 + (y - y_c)^2 + (z - z_c)^2 = r^2. \tag{12.2}$$

An appropriate parameterization can be derived using a spherical coordinate system with spherical coordinates ϕ and θ.

$$\begin{aligned} x(\phi, \theta) &= x_c + r \cdot \cos\theta \cdot \cos\phi, \\ y(\phi, \theta) &= y_c + r \cdot \cos\theta \cdot \sin\phi, \\ z(\phi, \theta) &= z_c + r \cdot \sin\theta. \end{aligned} \tag{12.3}$$

The spherical coordinate ϕ covers the range $[0..2\pi]$, and θ covers the range $[-\pi/2..\pi/2]$, thus, the appropriate (u, v) texture coordinates are derived as follows:

$$u = \frac{\phi}{2\pi}, \qquad v = \frac{(\theta + \pi/2)}{\pi}. \tag{12.4}$$

The complete transformation from texture space to modeling space is:

$$\begin{aligned} x(u, v) &= x_c + r \cdot \cos\pi(v - 0.5) \cdot \cos 2\pi u, \\ y(u, v) &= y_c + r \cdot \cos\pi(v - 0.5) \cdot \sin 2\pi u, \\ z(u, v) &= z_c + r \cdot \sin\pi(v - 0.5). \end{aligned} \tag{12.5}$$

For texture order mapping, the inverse transformation is:

$$\begin{aligned} u(x, y, z) &= \frac{1}{2\pi} \cdot \mathrm{arctan}^*(y - y_c, x - x_c), \\ v(x, y, z) &= \frac{1}{\pi} \cdot (\arcsin\frac{z - z_c}{r} + \pi/2), \end{aligned} \tag{12.6}$$

where $\mathrm{arctan}^*(a, b)$ is the extended arctan function, that is, it produces an angle ξ in $[0..2\pi]$ if $\sin\xi = a$ and $\cos\xi = b$.

Parameterization of a cylinder

A cylinder of height H located around the z axis has the following implicit equation:

$$X^2 + Y^2 = r^2, \qquad 0 \le z \le H. \tag{12.7}$$

The same cylinder can be conveniently expressed by cylindrical coordinates ($\theta \in [0..2\pi], h \in [0..H]$):

$$\begin{aligned} X(\theta, h) &= r \cdot \cos\theta, \\ Y(\theta, h) &= r \cdot \sin\theta, \\ Z(\theta, h) &= h. \end{aligned} \tag{12.8}$$

To produce an arbitrary cylinder, this is rotated and translated by an appropriate affine transformation:

$$[x(\theta,h), y(\theta,h), z(\theta,h), 1] = [X(\theta,h), Y(\theta,h), Z(\theta,h), 1] \cdot \begin{bmatrix} & 0 \\ \mathbf{A}_{3\times3} & 0 \\ & 0 \\ \mathbf{p}^{\mathbf{T}} & 1 \end{bmatrix} \tag{12.9}$$

where $\mathbf{A}$ must be an orthonormal matrix; that is, its row vectors must be unit vectors and must form a perpendicular triple. Matrices of this type do not alter the shape of the object and thus preserve cylinders.

Since cylindrical coordinates θ and h expand over the ranges $[0..2\pi]$ and $[0, H]$ respectively, the domain of cylindrical coordinates can thus be easily mapped onto a unit square:

$$u = \frac{\theta}{2\pi}, \qquad v = \frac{h}{H}. \tag{12.10}$$

The complete transformation from texture space to modeling space is:

$$[x(u,v), y(u,v), z(u,v), 1] = [r \cdot \cos 2\pi u, r \cdot \sin 2\pi u, v \cdot H, 1] \cdot \begin{bmatrix} & 0 \\ \mathbf{A}_{3\times3} & 0 \\ & 0 \\ \mathbf{p}^{\mathbf{T}} & 1 \end{bmatrix}. \tag{12.11}$$

The inverse transformation is:

$$[\alpha, \beta, h, 1] = [x(u,v), y(u,v), z(u,v), 1] \cdot \begin{bmatrix} & 0 \\ \mathbf{A}_{3\times 3} & 0 \\ & 0 \\ \mathbf{p^T} & 1 \end{bmatrix}^{-1}$$

$$u(x,y,z) = u(\alpha,\beta) = \frac{1}{2\pi} \cdot \arctan^*(\beta,\alpha), \qquad v(x,y,z) = \frac{h}{H} \qquad (12.12)$$

where $\arctan^*(a,b)$ is the extended arctan function as before.

12.1.3 Parameterization of polygons

Image generation algorithms, except in the case of ray tracing, suppose object surfaces to be broken down into polygons. This is why texturing and parameterization of polygons are so essential in computer graphics. The parameterization, as a transformation, must map a 2D polygon given by vertices $v_1(u,v), v_2(u,v), ..., v_n(u,v)$ onto a polygon in the 3D space, defined by vertex points $\vec{V}_1(x,y,z), \vec{V}_2(x,y,z), ..., \vec{V}_n(x,y,z)$. Let this transformation be $\mathcal{P}$. As stated, it must transform the vertices to the given points:

$$\vec{V}_1(x,y,z) = \mathcal{P}v_1(u,v), \; \vec{V}_2(x,y,z) = \mathcal{P}v_2(u,v), \ldots, \vec{V}_n(x,y,z) = \mathcal{P}v_n(u,v). \qquad (12.13)$$

Since each vertex $\vec{V}_i$ is represented by three coordinates, equation 12.13 consists of $3n$ equations. These equations, however, are not totally independent, since the polygon is assumed to be on a plane; that is, the plane equation defined by the first three non-collinear vertices

$$(\vec{r} - \vec{V}_1) \cdot (\vec{V}_3 - \vec{V}_1) \times (\vec{V}_2 - \vec{V}_1) = 0$$

should hold, where $\vec{r}$ is any of the other vertices, $\vec{V}_4$, $\vec{V}_5$,...,$\vec{V}_n$. The number of these equations is $n-3$. Thus, if $\mathcal{P}$ guarantees the preservation of polygons, it should have $3n - (n-3)$ free, independently controllable parameters in order to be able to map a 2D n-sided polygon onto an arbitrary 3D n-sided planar polygon. The number of independently controllable parameters is also called the **degree of freedom**.

Function $\mathcal{P}$ must also preserve lines, planes and polygons to be suitable for parameterization. The most simple transformation which meets this requirement is the linear mapping:

$$x = A_x \cdot u + B_x \cdot v + C_x, \quad y = A_y \cdot u + B_y \cdot v + C_y, \quad z = A_z \cdot u + B_z \cdot v + C_z. \qquad (12.14)$$

The degree of freedom (number of parameters) of this linear transformation is 9, requiring $3n - (n - 3) \leq 9$, or equivalently $n \leq 3$ to hold. Thus, only triangles ($n = 3$) can be parameterized by linear transformations.

Triangles

$$[x, y, z] = [u, v, 1] \cdot \begin{bmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{bmatrix} = [u, v, 1] \cdot \mathbf{P}. \qquad (12.15)$$

The unknown matrix elements can be derived from the solution of a 9×9 system of linear equations developed by putting the coordinates of $\vec{V}_1$, $\vec{V}_2$ and $\vec{V}_3$ into this equation.

For screen order, the inverse transformation is used:

$$[u, v, 1] = [x, y, z] \cdot \begin{bmatrix} A_x & A_y & A_z \\ B_x & B_y & B_z \\ C_x & C_y & C_z \end{bmatrix}^{-1} = [x, y, z] \cdot \mathbf{P}^{-1}. \qquad (12.16)$$

Quadrilaterals

As has been stated, a transformation mapping a quadrilateral from the 2D texture space to the 3D modeling space is generally non-linear, because the degree of freedom of a 2D to 3D linear transformation is less than is required by the placement of four arbitrary points. When looking for an appropriate non-linear transformation, however, the requirement stating that the transformation must preserve polygons has also to be kept in mind.

As for other cases, where the problem outgrows the capabilities of 3D space, 4D homogeneous representation can be relied on again [Hec86], since the number of free parameters is 12 for a linear 2D to 4D transformation, which is more than the necessary limit of 11 derived by inserting n=4 into formula $3n - (n - 3)$. Thus, a linear transformation can be established between a 2D and a 4D polygon.

From 4D space, however, we can get back to real 3D coordinates by a homogeneous division. Although this division reduces the degree of freedom by 1 — scalar multiples of homogeneous matrices are equivalent — the number of free parameters, 11, is still enough. The overall transformation consisting of a matrix multiplication and a homogeneous division has been proven to preserve polygons if the matrix multiplication maps no part of the quadrilateral onto the ideal points (see section 5.1). In order to avoid the wrap-around problem of projective transformations, convex quadrilaterals should not be mapped on concave ones and vice versa.

Using matrix notation, the parameterization transformation is:

$$[x \cdot h, y \cdot h, z \cdot h, h] = [u, v, 1] \cdot \begin{bmatrix} U_x & U_y & U_z & U_h \\ V_x & V_y & V_z & V_h \\ W_x & W_y & W_z & W_h \end{bmatrix} = [u, v, 1] \cdot \mathbf{P_{3\times 4}}. \qquad (12.17)$$

We arbitrarily choose $W_h = 1$ to select one matrix from the equivalent set. After homogeneous division, we get:

$$x(u,v) = \frac{U_x \cdot u + V_x \cdot v + W_x}{U_h \cdot u + V_h \cdot v + 1},$$
$$y(u,v) = \frac{U_y \cdot u + V_y \cdot v + W_y}{U_h \cdot u + V_h \cdot v + 1},$$
$$z(u,v) = \frac{U_z \cdot u + V_z \cdot v + W_z}{U_h \cdot u + V_h \cdot v + 1}. \qquad (12.18)$$

The inverse transformation, assuming $D_w = 1$, is:

$$[u \cdot w, v \cdot w, w] = [x, y, z, 1] \cdot \begin{bmatrix} A_u & A_v & A_w \\ B_u & B_v & B_w \\ C_u & C_v & C_w \\ D_u & D_v & D_w \end{bmatrix} = [x, y, z, 1] \cdot \mathbf{Q_{4\times 3}}, \qquad (12.19)$$

$$u(x,y,z) = \frac{A_u \cdot x + B_u \cdot y + C_u \cdot z + D_u}{A_w \cdot x + B_w \cdot y + C_w \cdot z + 1},$$
$$v(x,y,z) = \frac{A_v \cdot x + B_v \cdot y + C_v \cdot z + D_v}{A_w \cdot x + B_w \cdot y + C_w \cdot z + 1}. \qquad (12.20)$$

General polygons

The recommended method of parameterization of general polygons subdivides the polygon into triangles (or less probably into quadrilaterals) and generates a parameterization for the separate triangles by the previously discussed method. This method is pretty straightforward, although it maps line segments onto staggered lines, which may cause noticeable artifacts on the image. This effect can be greatly reduced by decreasing the size of the triangles composing the polygon.

Polygon mesh models

The natural way of reducing the dimensionality of a polygon mesh model from three to two is by unfolding it into a two-dimensional folding plane, having separated some of its adjacent faces along the common edges [SSW86]. If the faces are broken down into triangles and quadrilaterals, the texture space can easily be projected onto the folding space, taking into account which texture points should correspond to the vertices of the 2D unfolded object. The edges that must be separated to allow the unfolding of the polygon mesh model can be determined by topological considerations. The adjacency of the faces of a polyhedron can be defined by a graph where the nodes represent the faces or polygons, and the arcs of the graph represent the adjacency relationship of the two faces.

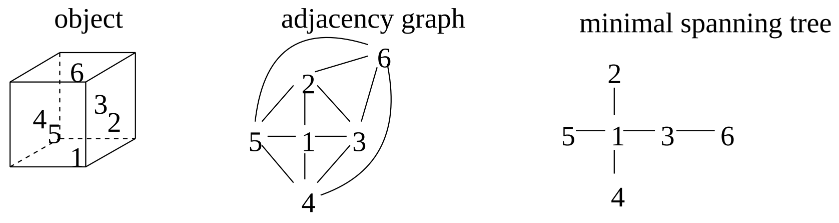

Figure 12.2: Face adjacency graph of a polyhedron

Polygon mesh models whose adjacency graphs are tree-graphs can obviously be unfolded. Thus, in order to prepare for the unfolding operation, those adjacent faces must be separated whose tearing will guarantee that the resulting adjacency graph is a tree. A graph usually has many spanning trees, thus the preprocessing step can have many solutions. By adding cost information to the various edges, an "optimal" unfolding can be achieved. There are several alternative ways to define the cost values:

- The user specifies which edges are to be preserved as the border of two adjacent polygons. These edges are given 0 unit cost, while the rest are given 1.
- The cost can be defined by the difference between the angle of the adjacent polygons and π. This approach aims to minimize the total rotations in order to keep the 2D unfolded model compact.

There are several straightforward algorithms which are suitable for the generation of a minimal total cost spanning tree of a general graph [Har69]. A possible algorithm builds up the graph incrementally and adds that new edge which has lowest cost from the remaining unused edges and does not cause a cycle to be generated in each step.

The unfolding operation starts at the root of the tree, and the polygons adjacent to it are pivoted about the common edge with the root polygon. When a polygon is rotated around the edge, all polygons adjacent to it must also be rotated (these polygons are the descendants of the given polygon in the tree). Having unfolded all the polygons adjacent to the root polygon, these polygons come to be regarded as roots and the algorithm is repeated for them recursively. The unfolding program is:

```
UnfoldPolygon( poly, edge, $\phi$ );
    Rotate poly and all its children around edge by $\pi - \phi$;
    for each child of poly
        UnfoldPolygon( child, edge(poly, child), angle(poly, child));
    endfor
end
Main program
    for each child of root
        UnfoldPolygon( child, edge(root, child), angle(root, child));
    endfor
end
```

The information regarding the orientation and position of polygons is usually stored in transformation matrices. A new rotation of a polygon means the concatenation of a new transformation matrix to the already developed matrix of the polygon. One of the serious limitations of this approach is that it partly destroys polygonal adjacency; that is, the unfolded surface will have a different topology from the original surface. A common edge of two polygons may be mapped onto two different locations of the texture space, causing discontinuities in texture along polygon boundaries.

This problem can be overcome by the method discussed in the subsequent section.

General surfaces

A general technique developed by Bier and Sloan [BS86] uses an intermediate surface to establish a mapping between the surface and the texture space. When mapping from the texture space to the surface, first the texture point is mapped onto the intermediate surface by its parameterization, then some "natural" projection is used to map the point onto the target surface. The texturing transformation is thus defined by a two-phase mapping.

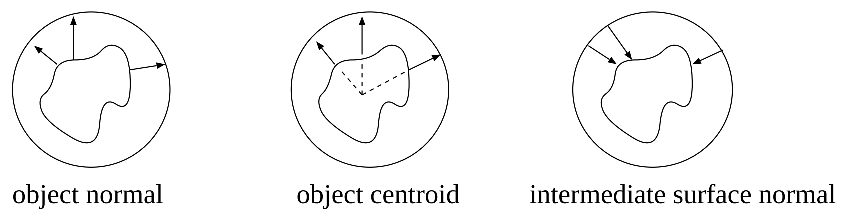

Figure 12.3: Natural projections

The intermediate surface must be easy to parameterize and therefore usually belongs to one of the following categories:

1. Planar polygon
2. Sphere
3. Cylinder
4. Faces of a cube.

The possibilities of the "natural" projection between the intermediate and target surfaces are shown in figure 12.3.

12.2 Texture mapping in ray tracing

Ray tracing determines which surface points must be shaded in the world coordinate system; that is, only the modeling transformation is needed to connect the shading space with the space where the textures are stored. Ray tracing is a typical image precision technique which generates the color of pixels sequentially, thus necessitating a scan order texture mapping method.

Having calculated the object visible along either a primary or secondary ray — that is, having found the nearest intersection of the ray and the objects — the shading equation must be evaluated, which may require the access of texture maps for varying parameters. The derivation of texture coordinates depends not only on the type of texturing, but also on the surface to be rendered.

For parametrically defined patches, such as Bezier and B-spline surfaces, the intersection calculation has already determined the parameters of the surface point to be shaded, thus this information is readily available to fetch the necessary values from 2D texture maps.

For other surface types and for solid texturing, the point should be transformed to local modeling space from the world coordinate system. Solid texture value is generated here usually by calling a function with the local coordinates which returns the required parameters. Two-dimensional textures require the inverse parameterization to be calculated, which can be done by any of the methods so far discussed.

12.3 Texture mapping for incremental shading models

In incremental shading the polygons are supposed to be in the screen coordinate system, and a surface point is thus represented by the (X, Y) pixel coordinates with the depth value Z which is only important for hidden surface elimination, but not necessarily for texture mapping since the definition of the surface, viewing, and the (X, Y) pair of pixel coordinates completely identify where the point is located on the surface. Incremental methods deal with polygon mesh approximations, never directly with the real equations of explicit or implicit surfaces. Texturing transformations, however, may refer either to the original surfaces or to polygons.

12.3.1 Real surface oriented texturing

If the texture mapping has parameterized the original surfaces rather than their polygon mesh approximations, an applicable scan-order algorithm is very similar to that used for ray tracing. First the point is mapped from screen space to the local modeling coordinate system by the inverse of the composite transformation. Since the composite transformation is homogeneous (or in special cases linear if the projection is orthographic), its inverse is also homogeneous (or linear), as has been proven in section 5.1 on properties of homogeneous transformations. From the modeling space, any of the discussed methods can be used to inversely parameterize the surface and to obtain the corresponding texture space coordinates.

Unlike ray tracing, parametric surfaces may pose serious problems, when the inverse parameterization is calculated, since this requires the solution of a two-variate, non-linear equation $\vec{r}(u,v) = \vec{p}$, where $\vec{p}$ is that point in the modeling space which maps to the actual pixel.

To solve this problem, the extension of the iterative root searching method based on the refinement of the subdivision of the parametric patch into planar polygons can be used here.

Some degree of subdivision has also been necessary for polygon mesh approximation. **Subdivision** of a parametric patch means dividing it along its isoparametric curves. Selecting uniformly distributed u and v parameters, the subdivision yields a mesh of points at the intersection of these curves. Then a mesh of planar polygons (quadrilaterals) can be defined by these points, which will approximate the original surface at a level determined by how exact the isoparametric division was.

When it turns out that a point of a polygon approximating the parametric surface is mapped onto a pixel, a rough approximation of the u, v surface parameters can be derived by looking at which isoparametric lines defined this polygon. This does not necessitate the solution of the non-linear equation if the data of the original subdivision which defines the polygon vertices at the intersection of isoparametric lines are stored somewhere. The inverse problem for a quadrilateral requires the search for this data only. The search will provide the isoparametric values along the boundaries of the quadrilateral. In order to find the accurate u, v parameters for an inner point, the subdivision must be continued, but without altering or further refining the shape of the approximation of the surface, as proposed by Catmull [Cat74]. In his method, the refinement of the subdivision is a parallel procedure in parameter space and screen space. Each time the quadrilateral in screen space is divided into four similar polygons, the rectangle in the parameter space is also broken down into four rectangles. By a simple comparison it is decided which screen space quadrilateral maps onto the actual pixel, and the algorithm proceeds for the resulted quadrilateral and its corresponding texture space rectangle until the polygon coming from the subdivision covers a single pixel. When the subdivision terminates, the required texture value is obtained from the texture map. Note that it is not an accurate method, since the original surface and the viewing transformation are not linear, but the parallel subdivision used linear interpolation. However, if the original interpolation is not too inexact, then it is usually acceptable.

12.3.2 Polygon based texturing

When discussing parameterization, a correspondence was established between the texture space and the local modeling space. For polygons subdivided into triangles and quadrilaterals, the parameterization and its inverse can be expressed by a homogeneous transformation. The local modeling space, on the other hand, is mapped to the world coordinate system, then to the screen space by modeling and viewing transformations respectively. The concatenation of the modeling and viewing transformations is an affine mapping for orthographic projection and is a homogeneous transformation for perspective projection.

Since both the parameterization and the projection are given by homogeneous transformations, their composition directly connecting texture space with screen space will also be a homogeneous transformation. The matrix representation of this mapping for quadrilaterals and perspective transformation is derived as follows. The parameterization is:

$$[x \cdot h, y \cdot h, z \cdot h, h] = [u, v, 1] \cdot \mathbf{P_{3\times 4}}. \tag{12.21}$$

The composite modeling and viewing transformation is:

$$[X \cdot q, Y \cdot q, Z \cdot q, q] = [x \cdot h, y \cdot h, z \cdot h, h] \cdot \mathbf{T_{V(4\times4)}}. \qquad (12.22)$$

Projection will simply ignore the Z coordinate if it is executed in screen space, and thus it is not even worth computing in texture mapping. Since the third column of matrix $\mathbf{T_{V(4\times4)}}$ is responsible for generating Z, it can be removed from the matrix:

$$[X \cdot q, Y \cdot q, q] = [x \cdot h, y \cdot h, z \cdot h, h] \cdot \mathbf{T_{V(4\times3)}} = [u, v, 1] \cdot \mathbf{P_{3\times4}} \cdot \mathbf{T_{V(4\times3)}}. \qquad (12.23)$$

Denoting $\mathbf{P_{3\times4}} \cdot \mathbf{T_{V(4\times3)}}$ by $\mathbf{C_{3\times3}}$, the composition of parameterization and projection is:

$$[X \cdot q, Y \cdot q, q] = [u, v, 1] \cdot \mathbf{C_{3\times3}}. \qquad (12.24)$$

The inverse transformation for scan order mapping is:

$$[u \cdot w, v \cdot w, w] = [X, Y, 1] \cdot \mathbf{C^{-1}_{3\times3}}. \qquad (12.25)$$

Let the element of $\mathbf{C_{3\times3}}$ and $\mathbf{C^{-1}_{3\times3}}$ in ith row and in jth column be c_{ij} and C_{ij} respectively. Expressing the texture coordinates directly, we can conclude that u and v are quotients of linear expressions of X and Y, while X and Y have similar formulae containing u and v.

The texture order mapping is:

$$X(u,v) = \frac{c_{11} \cdot u + c_{21} \cdot v + c_{31}}{c_{13} \cdot u + c_{23} \cdot v + c_{33}}, \qquad Y(u,v) = \frac{c_{12} \cdot u + c_{22} \cdot v + c_{32}}{c_{13} \cdot u + c_{23} \cdot v + c_{33}}. \qquad (12.26)$$

The screen order mapping is:

$$u(X,Y) = \frac{C_{11} \cdot X + C_{21} \cdot Y + C_{31}}{C_{13} \cdot X + C_{23} \cdot Y + C_{33}}, \qquad v(X,Y) = \frac{C_{12} \cdot X + C_{22} \cdot Y + C_{32}}{C_{13} \cdot X + C_{23} \cdot Y + C_{33}}. \qquad (12.27)$$

If triangles are parameterized and orthographic projection is used, both transformations are linear requiring their composite texturing transformation to be linear also. Linearity means that:

$$c_{13}, c_{23} = 0, \qquad c_{33} = 1, \qquad C_{13}, C_{23} = 0, \qquad C_{33} = 1 \qquad (12.28)$$

Scan order polygon texturing

When pixel X, Y is shaded in screen space, its corresponding texture coordinates can be derived by evaluating a rational equation 12.27. This can be further optimized by the incremental concept. Let the numerator and the denominator of quotients defining $u(X)$ be $uw(X)$ and $w(X)$ respectively. Although the division cannot be eliminated, $u(X+1)$ can be calculated from $u(X)$ by two additions and a single division if $uw(X)$ and $w(X)$ are evaluated incrementally:

$$uw(X+1) = uw(X)+C_{11}, \quad w(X+1) = w(X)+C_{13}, \quad u(X+1) = \frac{uw(X+1)}{w(X+1)}. \tag{12.29}$$

A similar expression holds for the incremental calculation of v. In addition to the incremental calculation of texture coordinates inside the horizontal spans, the incremental concept can also be applied on the starting edge of the screen space triangle. Thus, the main loop of the polygon rendering of Phong shading is made appropriate for incremental texture mapping:

```
$X_{\text{start}} = X_1 + 0.5$; $X_{\text{end}} = X_1 + 0.5$; $\vec{N}_{\text{start}} = \vec{N}_1$;
$uw_s = uw_1$; $vw_s = vw_1$; $w_s = w_1$;
for $Y = Y_1$ to $Y_2$ do
    $uw = uw_s$; $vw = vw_s$; $w = w_s$;
    $\vec{N} = \vec{N}_{\text{start}}$;
    for $X = \text{Trunc}(X_{\text{start}})$ to $\text{Trunc}(X_{\text{end}})$ do
        $u = uw/w$; $v = vw/w$;
        $(R, G, B) = \text{ShadingModel}(\ \vec{N},\ u,\ v\ )$;
        write( $X, Y, \text{Trunc}(R), \text{Trunc}(G), \text{Trunc}(B)$ );
        $\vec{N}$ += $\delta\vec{N}_X$;
        $uw$ += $C_{11}$; $vw$ += $C_{12}$; $w$ += $C_{13}$;
    endfor
    $X_{\text{start}}$ += $\delta X_Y^s$; $X_{\text{end}}$ += $\delta X_Y^e$; $\vec{N}_{\text{start}}$ += $\delta\vec{N}_Y^s$;
    $uw_s$ += $\delta uw_Y^s$; $vw_s$ += $\delta vw_Y^s$; $w_s$ += $\delta w_Y^s$;
endfor
```

If the texturing transformation is linear, that is if triangles are parameterized and orthographic projection is used, the denominator is always 1, thus simplifying the incremental formula to a single addition:

```
$X_{\text{start}} = X_1 + 0.5$; $X_{\text{end}} = X_1 + 0.5$; $\vec{N}_{\text{start}} = \vec{N}_1$;
$u_s = u_1$; $v_s = v_1$;
for $Y = Y_1$ to $Y_2$ do
    $u = u_s$; $v = v_s$;
    $\vec{N} = \vec{N}_{\text{start}}$;
    for $X = \text{Trunc}(X_{\text{start}})$ to $\text{Trunc}(X_{\text{end}})$ do
        $(R, G, B) = \text{ShadingModel}(\ \vec{N},\ u,\ v\ )$;
        write( $X, Y, \text{Trunc}(R), \text{Trunc}(G), \text{Trunc}(B)$ );
        $\vec{N}$ += $\delta\vec{N}_X$;
        $u$ += $C_{11}$; $v$ += $C_{12}$;
    endfor
    $X_{\text{start}}$ += $\delta X_Y^s$; $X_{\text{end}}$ += $\delta X_Y^e$; $\vec{N}_{\text{start}}$ += $\delta\vec{N}_Y^s$;
    $u_s$ += $\delta u_Y^s$; $v_s$ += $\delta v_Y^s$;
endfor
```

Texture order polygon texturing

The inherent problem of texture order methods — i.e., that the uniform sampling of texture space does not guarantee the uniform sampling of screen space, and therefore may leave holes in the image or may generate pixels redundantly in an unexpected way — does not exist if the entire texture mapping is linear. Thus, we shall consider the simplified case when triangles are parameterized and orthographic projection is used, producing a linear texturing transformation.

The isoparametric lines of the texture space may be rotated and scaled by the texturing transformation, requiring the shaded polygons to be filled by possibly diagonal lines. Suppose the texture is defined by an $N \times M$ array T of "texture space pixels" or **texels**, and the complete texturing transformation is linear and has the following form:

$$X(u,v) = c_{11} \cdot u + c_{21} \cdot v + c_{31}, \qquad Y(u,v) = c_{12} \cdot u + c_{22} \cdot v + c_{32}. \quad (12.30)$$

The triangle being textured is defined both in 2D texture space and in the 3D screen space (figure 12.4). In texture order methods those texels must be found which cover the texture space triangle, and the corresponding screen space pixels should be shaded using the parameters found in the given location of the texture array. The determination of the covering texels in the texture space triangle is, in fact, a two-dimensional polygon-fill problem that has straightforward algorithms. The direct application of these algorithms, however, cannot circumvent the problem of different texture and pixel sampling grids, and thus can produce holes and overlapping in screen space. The complete isoparametric line of v from $u = 0$ to $u = 1$ is

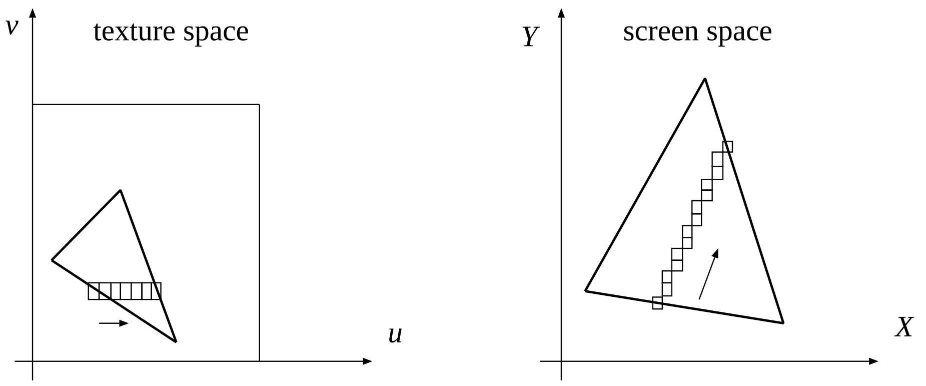

Figure 12.4: Texture order mapping

a digital line consisting of $d_u = \mathrm{Trunc}(\max[c_{11}, c_{12}])$ pixels in screen space. Thus, the N number of texture pixels should be re-distributed to d_u screen pixels, which is equivalent to obtaining the texture array in $\delta u = d_u/N$ steps. Note that δu is not necessarily an integer, requiring a non-integer U coordinate which accumulates the δu increments. The integer index to the texture array is defined by the integer part of this U value. Nevertheless, U and δu can be represented in a fixed point format and calculated only by fixed point additions. Note that the emerging algorithm is similar to incremental line drawing methods, and eventually the distribution of N texture pixels onto d_u screen pixels is equivalent to drawing an incremental line with a slope N/d_u.

The complete isoparametric lines of u from $v = 0$ to $v = 1$ are similarly $d_v = \text{Trunc}(\max[c_{21}, c_{22}])$ pixel long digital lines in screen space. The required increment of the v coordinate is $\delta v = d_v/M$ when the next pixel is generated on this digital line. Thus, a modified polygon-filling algorithm must be used in texture space to generate the texture values of the inner points, and it must be one which moves along the horizontal and vertical axes at d_u and d_v increments instead of jumping to integer points as in normal filling.

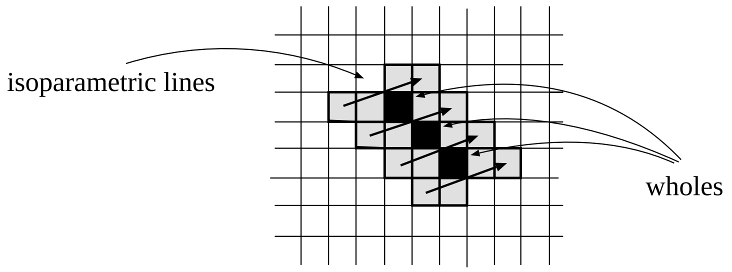

Figure 12.5: Holes between adjacent lines

This incremental technique must be combined with the filling of the screen space triangle. Since the isoparametric lines corresponding to constant v values are generally not horizontal but can have any slant, the necessary filling algorithm produces the internal pixels as a sequence of diagonal span lines. As in normal filling algorithms, two line generators are needed to produce the start and end points of the internal spans. A third line generator, on the other hand, generates the pixels between the start and end points.

The main problem of this approach is that using all pixels along the two parallel edges does not guarantee that all internal pixels will be covered by the connecting digital lines. Holes may appear between adjacent lines as shown in figure 12.5. Even linear texture transformation fails to avoid generating holes, but at least it does it in a well defined manner. Braccini and Marino [BG86] proposed drawing an extra pixel at each bend in the pixel space digital line to fill in any gaps that might be present. This is obviously a drastic approach and may result in redundancy, but it solves the inherent problem of texture order methods.

12.3.3 Texture mapping in the radiosity method

The general radiosity method consists of two steps: a view-independent radiosity calculation step, and a view-dependent rendering step where either an incremental shading technique is used, such as Gouraud shading, or else the shading is accomplished by ray-tracing. During the radiosity calculation step, the surfaces are broken down into planar elemental polygons which are assumed to have uniform radiosity, emission and diffuse coefficients. These assumptions can be made even if texture mapping is used, by calculating the "average" diffuse coefficient for each elemental surface, because the results of this approximation are usually acceptable. In the second, view-dependent step, however, textures can be handled as discussed for incremental shading and ray tracing.

12.4 Filtering of textures

Texture mapping establishes a correspondence between texture space and screen space. This mapping may magnify or shrink texture space regions when they are eventually projected onto pixels. Since raster based systems use regular sampling in pixel space, texture mapping can cause extremely uneven sampling of texture space, which inevitably results in strong aliasing artifacts. The methods discussed in chapter 11 (on sampling and quantization artifacts) must be applied to filter the texture in order to avoid aliasing. The applicable filtering techniques fall into two categories: pre-filtering, and post-filtering with supersampling.

12.4.1 Pre-filtering of textures

The difficulty of pre-filtering methods is that the mapping between texture and pixel spaces is usually non-linear, thus the shape of the convolution filter is distorted. The requirement for box filtering, for example, is the pre-image of a pixel, which is a curvilinear quadrilateral, and thus the texels lying inside this curvilinear quadrilateral must be summed to produce the pixel color. In order to ease the filtering computation, this curvilinear quadrilateral is approximated by some simpler geometric object; the possible alternatives are a square or a rectangle lying parallel to the texture coordinate axes, a normal quadrilateral or an ellipse (figure 12.6).

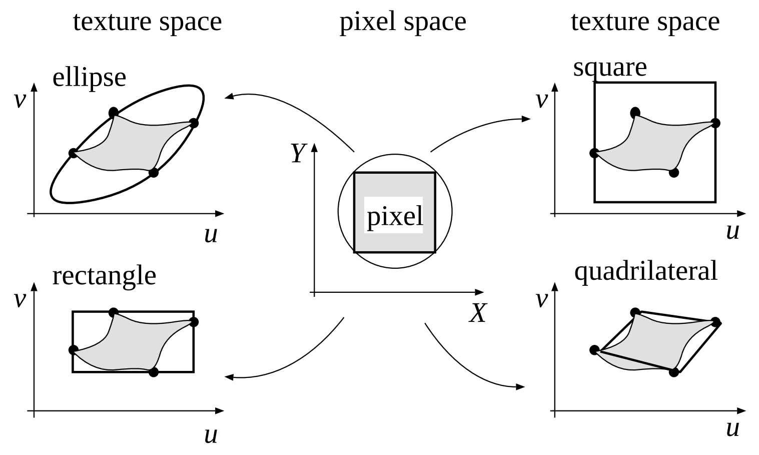

Figure 12.6: Approximations of the pre-image of the pixel rectangle

Rectangle approximation of the pre-image

The approximation by a rectangle is particularly suited to the Catmull texturing algorithm based on parallel subdivision in screen and texture space. Recall that in his method a patch is subdivided until the resulting polygon covers a single pixel. At the end of the process the corresponding texture domain subdivision takes the form of a square or a rectangle in texture space. Texels enclosed by this rectangle are added up, or the texture function is integrated here, to approximate a box filter. A conical or pyramidal filter can also be applied if the texels are weighted by a linear function increasing from the edges towards the center of the rectangle.

The calculation of the color of a single pixel requires integration or summation of the texels lying in its pre-image, and thus the computational burden is proportional to the size of the pre-image of the actual pixel. This can be disadvantageous if large textures are mapped onto a small area of the screen. Nevertheless, texture mapping can be speeded up, and this linear dependence on the size of the pixel's pre-image can be obviated if pre-integrated tables, or so-called **pyramids**, are used.

The image pyramid of pre-filtered data

Pyramids are multi-resolution data structures which contain the successively band-limited and subsampled versions of the same original image. These versions correspond to different sampling resolutions of the image. The resolution of the successive levels usually decrease by a factor of two. Conceptually, the subsequent images can be thought of as forming a pyramid, with the original image at the base and the crudest approximation at the apex, which provides some explanation of the name of this method. Versions are usually generated by box filtering and resampling the previous version of the image; that is, by averaging the color of four texels from one image, we arrive at the color of a texel for the subsequent level.

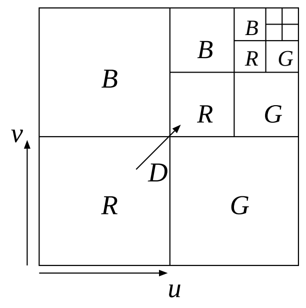

Figure 12.7: Mip-map organization of the memory

The collection of texture images can be organized into a **mip-map scheme**, as proposed by Williams [Wil83] (figure 12.7). ("mip" is an acronym of "multum in parvo" — "many things in a small space".) The mip-map scheme has a modest memory requirement. If the size of the original image is 1, then the cost of the mip-map organization is $1 + 2^{-2} + 2^{-4} + \ldots \approx 1.33$. The texture stored in a mip-map scheme is accessed using three indices: u, v texture coordinates and D for level of the pyramid. Looking up a texel defined by this three-index directory in the two-dimensional $M \times M$ mip-map array (MM) is a straightforward process:

$$\begin{aligned} R(u,v,D) &= MM[(1-2^{-D}) \cdot M + u \cdot 2^{-D}, (1-2^{-D}) \cdot M + v \cdot 2^{-D}], \\ G(u,v,D) &= MM[(1-2^{-(D+1)}) \cdot M + u \cdot 2^{-D}, (1-2^{-D}) \cdot M + v \cdot 2^{-D}], \\ B(u,v,D) &= MM[(1-2^{-D}) \cdot M + u \cdot 2^{-D}, (1-2^{-(D+1)}) \cdot M + v \cdot 2^{-D}]. \end{aligned} \tag{12.31}$$

The application of the mip-map organization makes it possible to calculate the filtered color of the pixel in constant time and independently of the number of texels involved if D is selected appropriately. Level parameter D must obviously be derived from the span of the pre-image of the pixel area, d, which can be approximated as follows:

$$\begin{aligned} d &= \max\{|[u(x+1), v(x+1)] - [u(x), v(x)]|, |[u(y+1), v(y+1)] - [u(y), v(y)]|\} \\ &\approx \max\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2}\}. \end{aligned} \qquad (12.32)$$

The appropriate image version is that which composes approximately d^2 pixels together, thus the required pyramid level is:

$$D = \log_2(\max\{d, 1\}). \qquad (12.33)$$

The minimum 1 value in the above equation is justified by the fact that if the inverse texture mapping maps a pixel onto a part of a texel, then no filtering is necessary. The resulting D parameter is a continuous value which must be made discrete in order to generate an index for accessing the mip-map array. Simple truncation or rounding might result in discontinuities where the span size of the pixel pre-image changes, and thus would require some inter-level blending or interpolation. Linear interpolation is suitable for this task, thus the final expression of color values is:

$$\begin{aligned} &R(u, v, \mathrm{Trunc}(D)) \cdot (1 - \mathrm{Fract}(D)) + R(u, v, \mathrm{Trunc}(D) + 1) \cdot \mathrm{Fract}(D), \\ &G(u, v, \mathrm{Trunc}(D)) \cdot (1 - \mathrm{Fract}(D)) + G(u, v, \mathrm{Trunc}(D) + 1) \cdot \mathrm{Fract}(D), \\ &B(u, v, \mathrm{Trunc}(D)) \cdot (1 - \mathrm{Fract}(D)) + B(u, v, \mathrm{Trunc}(D) + 1) \cdot \mathrm{Fract}(D). \end{aligned} \qquad (12.34)$$

The image pyramid relies on the assumption that the pre-image of the pixel can be approximated by a square in texture space. An alternative discussed by the next section, however, allows for rectangular areas oriented parallel to the coordinate axes.

Summed-area tables

A **summed-area table** (SA) is an array-like data structure which contains the running sum of colors as the image is scanned along successive scan-lines; that is, at $[i, j]$ position of this SA table there is a triple of R, G, B

values, each of them generated from the respective T texel array as follows:

$$SA_I[i,j] = \sum_{u=0}^{i} \sum_{v=0}^{j} T[u,v]_I \tag{12.35}$$

where the subscript I stands for any of R, G or B.

This data structure makes it possible to calculate the box filtered or area summed value of any rectangle oriented parallel to the axes, since the sum of pixel colors in a rectangle given by corner points $[u_0, v_0; u_1, v_1]$ is:

$$I([u_0, v_0; u_1, v_1]) = \sum_{u=u_0}^{u_1} \sum_{v=v_0}^{v_1} T[u,v]_I =$$

$$SA_I[u_1, v_1] - SA_I[u_1, v_0] - SA_I[u_0, v_1] + SA_I[u_0, v_0]. \tag{12.36}$$

Image pyramids and summed-area tables allow for constant time filtering, but require set-up overhead to build the data structure. Thus they are suitable for textures which are used many times.

Rectangle approximation of the pixel's pre-image

Deeming the curvilinear region to be a normal quadrilateral provides a more accurate method [Cat74]. Theoretically the generation of internal texels poses no problem, because polygon filling algorithms are effective tools for this, but the implementation is not so simple as for a square parallel to the axes. Pyramidal filters are also available if an appropriate weighting function is applied [BN76].

EWA — Elliptical Weighted Average

Gangnet, Perny and Coueignoux [GPC82] came up with an interesting idea which considers pixels as circles rather than squares. The pre-image of a pixel is then an ellipse even for homogeneous transformations, or else can be quite well approximated by an ellipse even for arbitrary transformations. Ellipses thus form a uniform class of pixel pre-images, which can conveniently be represented by few parameters. The idea has been further refined by Greene and Heckbert [GH86] who proposed distorting the filter kernel according to the resulting ellipse in texture space.

Let us consider a circle with radius r at the origin of pixel space. Assuming the center of the texture coordinate system to have been translated to the inverse projection of the pixel center, the pre-image can be approximated by the following ellipse:

$$F(r) = Au^2 + Buv + Cv^2. \tag{12.37}$$

Applying Taylor's approximation for the functions $u(x, y)$ and $v(x, y)$, we get:

$$\begin{aligned} u(x,y) &\approx \frac{\partial u}{\partial x} \cdot x + \frac{\partial u}{\partial y} \cdot y = u_x \cdot x + u_y \cdot y, \\ v(x,y) &\approx \frac{\partial v}{\partial x} \cdot x + \frac{\partial v}{\partial y} \cdot y = v_x \cdot x + v_y \cdot y. \end{aligned} \tag{12.38}$$

Substituting these terms into the equation of the ellipse, then:

$$\begin{aligned} F = x^2 \cdot (u_x^2 A + u_x v_x B + v_x^2 C) + y^2 \cdot (u_y^2 A + u_y v_y B + v_y^2 C) + \\ xy \cdot (2u_x u_y A + (u_x v_y + u_y v_x) B + 2 v_x v_y C). \end{aligned} \tag{12.39}$$

The original points in pixel space are known to be on a circle; that is the coordinates must satisfy the equation $x^2 + y^2 = r^2$. Comparing this to the previous equation, a linear system of equations can be established for A, B, C and F respectively. To solve these equations, one solution from the possible ones differing in a constant multiplicative factor is:

$$\begin{aligned} A &= v_x^2 + v_y^2, \\ B &= -2(u_x v_x + u_y v_y), \\ C &= u_x^2 + u_y^2, \\ F &= (u_x v_y - u_y v_x)^2 \cdot r^2. \end{aligned} \tag{12.40}$$

Once these parameters are determined, they can be used to test for point-inclusion in the ellipse by incrementally computing

$$f(u, v) = A \cdot u^2 + B \cdot u \cdot v + C \cdot v^2$$

and deciding whether its absolute value is less than F. If a point satisfies the $f(u, v) \leq F$ inequality — that is, it is located inside the ellipse — then the actual $f(u, v)$ shows how close the point is to the center of the pixel, or which concentric circle corresponds to this point as shown by the above expression of F. Thus, the $f(u,v)$ value can be directly used to generate the weighting factor of the selected filter. For a cone filter, for example, the weighting function is:

$$w(f) = \frac{\sqrt{f(u,v)}}{|u_x v_y - u_y v_x|}. \tag{12.41}$$

The square root compensates for the square of r in the expression of F. Apart from for cone filters, almost any kind of filter kernel (Gaussian, B-spline, sinc etc.) can be realized in this way, making the EWA approach a versatile and effective technique.

12.4.2 Post-filtering of textures

Post-filtering combined with supersampling means calculating the image at a higher resolution. The pixel colors are then computed by averaging the colors of subpixels belonging to any given pixel. The determination of the necessary subpixel resolution poses a critical problem when texture mapped surfaces are present, because it depends on both the texel resolution and the size of the areas mapped onto a single pixel, that is on the level of compression of the texturing, modeling and viewing transformations. By breaking down a pixel into a given number of subpixels, it is still not guaranteed that the corresponding texture space sampling will meet, at least approximately, the requirements of the sampling theorem. Clearly, the level of pixel subdivision must be determined by examination of all the factors involved. An interesting solution is given to this problem in the *REYES Image Rendering System* [CCC87] (REYES stands for "Renders Everything You Ever Saw"), where supersampling has been combined with stochastic sampling. In REYES surfaces are broken down into so-called **micropolygons** such that a micropolygon will have the size of about half a pixel when it goes through the transformations of image synthesis. Micropolygons have constant color determined by evaluation of the shading equation for coefficients coming from pre-filtered textures. Thus, at micropolygon level, the system applies a pre-filtering strategy. Constant color micropolygons are projected onto the pixel space, where at each pixel several subpixels are placed randomly, and the pixel color is computed by averaging those subpixels. Hence, on surface level a stochastic supersampling approach is used with post-filtering. The adaptivity of the whole method is provided by the subdivision criterion of micropolygons; that is, they must have approximately half a pixel size after projection. The half pixel size is in fact the Nyquist limit of sampling on the pixel grid.

12.5 Bump mapping

Examining the formulae of shading equations we can see that the surface normal plays a crucial part in the computation of the color of the surface. Bumpy surfaces, such as the moon with its craters, have darker and brighter patches on them, since the modified normal of bumps can turn towards or away from the lightsources. Suppose that the image of a slightly bumpy surface has to be generated, where the height of the bumps is considerably smaller than the size of the object. The development of a geometric model to represent the surface and its bumps would be an algebraic nightmare, not to mention the difficulties of the generation of its image. Fortunately, we can apply a deft and convenient approximation method called **bump mapping**. The geometry used in transformations and visibility calculations is not intended to take the bumps into account — the moon, for example, is assumed to be a sphere — but during shading calculations a perturbed normal vector, taking into account the geometry and the bumps as well, is used in the shading equation. The necessary perturbation function is stored in texture maps, called bump maps, making bump mapping a special type of texture mapping. An appropriate perturbation of the normal vector gives the illusion of small valleys, providing the expected image without the computational burden of the geometry of the bumps. Now the derivation of the perturbations of the normal vectors is discussed, based on the work of Blinn [Bli78].

Suppose that the surface incorporating bumps is defined by a function $\vec{r}(u,v)$, and its smooth approximation is defined by $\vec{s}(u,v)$, that is, $\vec{r}(u,v)$ can be expressed by adding a small displacement $d(u,v)$ to the surface $\vec{s}(u,v)$ in the direction of its surface normal (figure 12.8). Since the surface normal $\vec{n}_s$ of $\vec{s}(u,v)$ can be expressed as the cross product of the partial derivatives $(\vec{s}_u, \vec{s}_v)$ of the surface in two parameter directions, we can write:

$$\vec{r}(u,v) = \vec{s}(u,v) + d(u,v) \cdot [\vec{s}_u(u,v) \times \vec{s}_v(u,v)]^0 = \vec{s}(u,v) + d(u,v) \cdot \vec{n}_s^0 \quad (12.42)$$

(the 0 superscript stands for unit vectors).

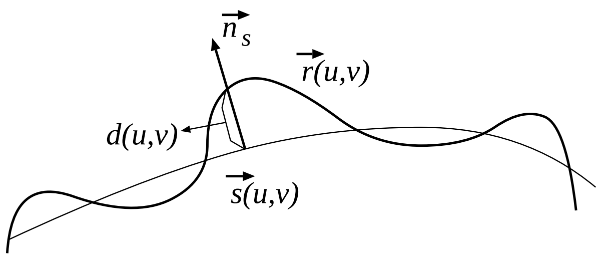

Figure 12.8: Description of bumps

The partial derivatives of $\vec{r}(u,v)$ are:

$$\vec{r}_u = \vec{s}_u + d_u \cdot \vec{n}_s^0 + d \cdot \frac{\partial \vec{n}_s^0}{\partial u},$$

$$\vec{r}_v = \vec{s}_v + d_v \cdot \vec{n}_s^0 + d \cdot \frac{\partial \vec{n}_s^0}{\partial v}. \tag{12.43}$$

The last terms can be ignored, since the normal vector variation is small for smooth surfaces, as is the $d(u,v)$ bump displacement, thus:

$$\vec{r}_u \approx \vec{s}_u + d_u \cdot \vec{n}_s^0,$$

$$\vec{r}_v \approx \vec{s}_v + d_v \cdot \vec{n}_s^0. \tag{12.44}$$

The surface normal of $r(u,v)$, that is the perturbed normal, is then:

$$\vec{n}_r = \vec{r}_u \times \vec{r}_v = \vec{s}_u \times \vec{s}_v + d_u \cdot \vec{n}_s^0 \times \vec{s}_v + d_v \cdot \vec{s}_u \times \vec{n}_s^0 + d_u d_v \cdot \vec{n}_s^0 \times \vec{n}_s^0. \tag{12.45}$$

Since the last term of this expression is identically zero because of the axioms of the vector product, and

$$\vec{n}_s = \vec{s}_u \times \vec{s}_v, \qquad \vec{s}_u \times \vec{n}_s^0 = -\vec{n}_s^0 \times \vec{s}_u,$$

thus we get:

$$\vec{n}_r = \vec{n}_s + d_u \cdot \vec{n}_s^0 \times \vec{s}_v - d_v \cdot \vec{n}_s^0 \times \vec{s}_u. \tag{12.46}$$

This formula allows for the computation of the perturbed normal using the derivatives of the displacement function. The displacement function $d(u,v)$ must be defined by similar techniques as for texture maps; they can be given either by functions or by pre-computed arrays called **bump maps**. Each time a normal vector is needed, the (u, v) coordinates have to be determined, and then the derivatives of the displacement function must be evaluated and substituted into the formula of the perturbation vector.

The formula of the perturbated vector requires the derivatives of the bump displacement function, not its value. Thus, we can either store the derivatives of the displacement function in two bump maps, or calculate them each time using finite differences. Suppose the displacement function is defined by an $N \times N$ bump map array, B. The calculated derivatives are then:

```
$U = \mathrm{Trunc}(u * N);$        $V = \mathrm{Trunc}(v * N);$
**if** $U < 1$ **then** $U = 1;$ **if** $U > N - 2$ **then** $U = N - 2;$
**if** $V < 1$ **then** $V = 1;$ **if** $V > N - 2$ **then** $V = N - 2;$
$d_u(u, v) = (B[U + 1, V] - B[U - 1, V]) \cdot N/2;$
$d_v(u, v) = (B[U, V + 1] - B[U, V - 1]) \cdot N/2;$
```

The displacement function, $d(u, v)$, can be derived from frame-grabbed photos or hand-drawn digital pictures generated by painting programs, assuming color information to be depth values, or from z-buffer memory values of computer synthesized images. With the latter method, an arbitrary arrangement of 3D objects can be used for definition of the displacement of the bump-mapped surface.

Blinn [Bli78] has noticed that bump maps defined in this way are not invariant to scaling of the object. Suppose two differently scaled objects with the same bump map are displayed on the screen. One might expect the bigger object to have bigger wrinkles proportionally to the size of the object, but that will not be the case, since

$$\frac{|\vec{n}_r - \vec{n}_s|}{|\vec{n}_s|} = \frac{|d_u \cdot \vec{n}_s^0 \times \vec{s}_v - d_v \cdot \vec{n}_s^0 \times \vec{s}_u|}{|\vec{s}_u \times \vec{s}_v|}$$

is not invariant with the scaling of $\vec{s}(u, v)$ and consequently of $\vec{n}_s$, but it is actually inversely proportional to it. If it generates unwanted effects, then a compensation is needed for eliminating this dependence.

12.5.1 Filtering for bump mapping

The same problems may arise in the context of bump mapping as for texture mapping if point sampling is used for image generation. The applicable solution methods are also similar and include pre-filtering and post-filtering with supersampling. The pre-filtering technique — that is, the averaging of displacement values stored in the bump map — contains, however, the theoretical difficulty that the dependence of surface colors on bump displacements is strongly non-linear. In effect, pre-filtering will tend to smooth out not only high-frequency aliases, but also bumps.

12.6 Reflection mapping

Reading the section on bump mapping, we can see that texture mapping is a tool which can re-introduce features which were previously eliminated because of algorithmic complexity considerations. The description of bumps by their true geometric characteristics is prohibitively expensive computationally, but this special type of texture mapping, bump mapping, can provide nearly the same effect without increasing the geometric complexity of the model. Thus it is no surprise that attempts have been made to deal with other otherwise complex phenomena within the framework of texture mapping. The most important class of these approaches addresses the problem of coherent reflection which could otherwise be solved only by expensive ray tracing.

A reflective object reflects the image of its environment into the direction of the camera. Thus, the pre-computed image visible from the center of the reflective object can be used later, when the color visible from the camera is calculated. These pre-computed images with respect to reflective objects are called **reflection maps** [BN76]. Originally Blinn and Newell proposed a sphere as an intermediate object onto which the environment is projected. Cubes, however, are more convenient [MH84], since to generate the pictures seen through the six sides of the cube, the same methods can be used as for computing the normal images from the camera.

Having generated the images visible from the center of reflective objects, a normal image synthesis algorithm can be started using the real camera. Reflective surfaces are treated as texture mapped ones with the texture

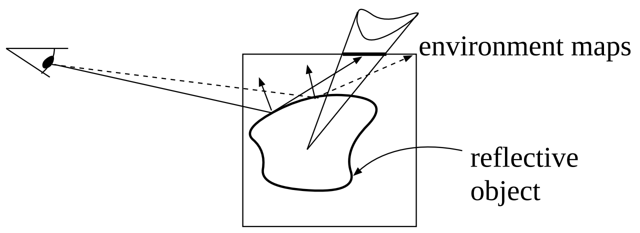

Figure 12.9: Reflection mapping

coordinates calculated by taking into account not only the surface point, but the viewing direction and the surface normal as well. By reflecting the viewing direction — that is, the vector pointing to the surface point from the camera — onto the surface normal, the reflection direction vector is derived, which unambiguously defines a point on the intermediate cube (or sphere), which must be used to provide the color of the reflection direction. This generally requires a cube-ray intersection. However, if the cube is significantly greater than the object itself, the dependence on the surface point can be ignored, which allows for the access of the environment map by the coordinates of the reflection vector only.

Suppose the cube is oriented parallel to the coordinate axes and the images visible from the center of the object through its six sides are stored in texture maps $R[0,u,v], R[1,u,v], \ldots, R[5,u,v]$. Let the reflected view vector be $\vec{V}_r = [V_x, V_y, V_z]$.

The *color* of the light coming from the reflection direction is then:

```
if $|V_x| = \max\{|V_x|, |V_y|, |V_z|\}$ then
    if $V_x > 0$ then color = $R[0, 0.5 + V_y/V_x, 0.5 + V_z/V_x]$;
    else              color = $R[3, 0.5 + V_y/V_x, 0.5 + V_z/V_x]$;
if $|V_y| = \max\{|V_x|, |V_y|, |V_z|\}$ then
    if $V_y > 0$ then color = $R[1, 0.5 + V_x/V_y, 0.5 + V_z/V_y]$;
    else              color = $R[4, 0.5 + V_x/V_y, 0.5 + V_z/V_y]$;
if $|V_z| = \max\{|V_x|, |V_y|, |V_z|\}$ then
    if $V_z > 0$ then color = $R[2, 0.5 + V_x/V_z, 0.5 + V_y/V_z]$;
    else              color = $R[5, 0.5 + V_x/V_z, 0.5 + V_y/V_z]$;
```