Chapter 8.

Higher Dimensions for "3D"

Sometimes it happens that for processing inherently 3D data efficiently we should use algorithms in higher dimensions. The use of a higher dimensional space can be explicit like a lifting or linearization technique, or somewhat implicit like the auxiliary call to generic data structures that process data internally in higher dimensions. In this chapter, we provide a few examples that illustrate those notions either at the data-structure level (Section 8.1) or at the problem-solving level (Section 8.3.1).

8.1 Nearest Neighbors

In the nearest neighbor problem, we are given a set $\mathcal{P}$ of n d-dimensional points $\mathcal{P} = \{\mathbf{p}_1, ..., \mathbf{p}_n\}$, and a generic query point $\mathbf{q}$ for which we require to report efficiently the nearest point $N(\mathbf{q})$ of $\mathcal{P}$ to $\mathbf{q}$:

$$N(\mathbf{q}) = \mathbf{p} \in \mathcal{P} \text{ such that } ||\mathbf{qp}|| \leq ||\mathbf{qp}_i|| \; \forall i \in \{1, ..., n\}. \tag{8.1}$$

Point $\mathbf{p}$ is called the nearest neighbor of query point $\mathbf{q}$. We first illustrate the need for such efficient proximity data structures (and corresponding query algorithms) by introducing a simple but efficient 2D texture synthesis technique.

8.1.1 Application: 2D Texture Synthesis

Consider the problem of synthesizing a large picture $\mathbf{I}_t$ (of size $w_t \times h_t$) given a small texture sample $\mathbf{I}_s$ (of size $w_s \times h_s$), as shown in Figure 8.1. This *texture synthesis* problem has received a lot of attention in the computer graphics community, and various techniques based on spectral, statistical, per-pixel, per-patch, and per-tile methods have been designed. Here, we present a simple but powerful *per-pixel texture synthesis technique* that requires nearest neighbor queries. As usual, we give

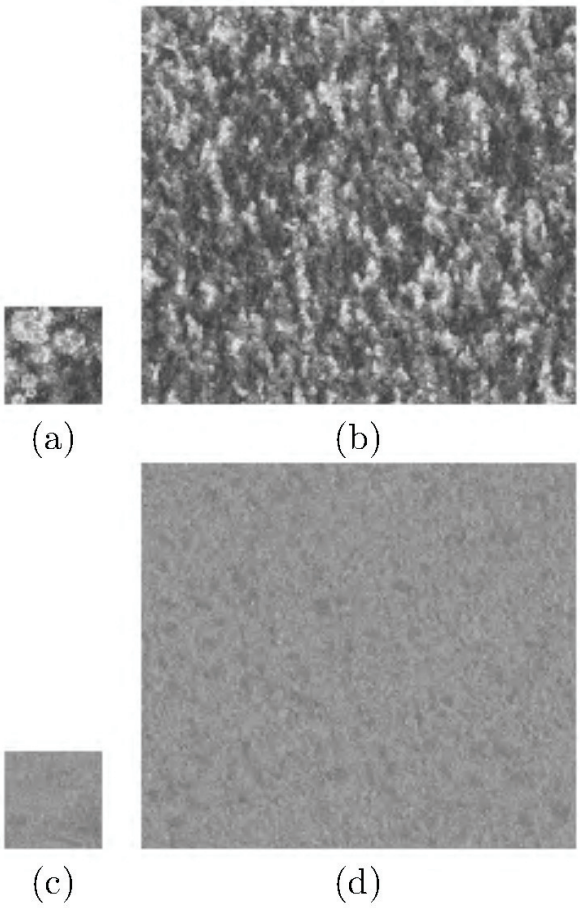

FIGURE 8.1 *Examples of texture synthesis. (a) and (c) are sample input textures. (b) and (d) are the corresponding synthesized texture output images.*

further references to relevant publications and materials in the Bibliographical Notes. Surprisingly, the synthesis technique we describe here was invented only recently by Efros and Leung (1999). Interestingly, their paper was presented in a computer vision conference.

The synthesis algorithm first starts by filling the target image by random-colored pixels, and then synthesizes the target image $\mathbf{I}_t$ by (re)assigning pixel colors, pixel by pixel, following the horizontal scanline order (i.e., lexicographic[1] (y, x) order), as depicted in Figure 8.2. For a given pixel position $(x, y) \in \mathbf{I}_t$, we consider a square window centered at (x, y) of side length $2s + 1$, where s denotes an integer parameter defining the neighborhood size and related to the texture synthesis quality. In that window, observe that $(2s + 1)s + s = 2s^2 + 2s$ pixels have already been synthesized

[1]The lexicographic order is a natural order of the Cartesian product of ordered sets. For two sets, consider $a, a' \in \mathcal{A}$ and $b, b' \in \mathcal{B}$, then for $(a, b), (a', b') \in \mathcal{A} \times \mathcal{B}$, we have $(a, b) \leq (a', b')$ if, and only if, $a < a'$, or $a = a'$ and $b \leq b'$.

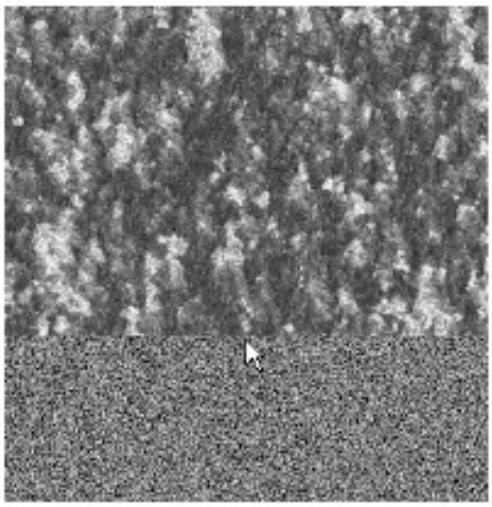

FIGURE 8.2 *Snapshot of in-progress scanline incremental 2D texture synthesis.*

(Figure 8.3): namely, those pixels (x', y') visited before by the scanline. That is, using the (y, x) lexicographic order of the horizontal scanline, the pixels (x', y') of the target image such that $(y', x') < (y, x)$. Those already synthesized pixels form an L-shape pattern, as depicted in Figure 8.3. Thus, at current scanline position (x_t, y_t), we are already given half of the window pixels synthesized, and we need to find an appropriate color for the current visited pixel (x_t, y_t).

For assigning the color of that pixel, we search for the *best match* of the current L-shaped window in the source image. The best match is defined as the matching position in $\mathbf{I}_s$ (possibly several places) that minimizes the sum of the square differences (SSD):

$$\mathrm{SSD}(x_s, y_s; x_t, y_t) = \sum_{l=-s}^{s} \sum_{c=-s}^{s} \mathrm{LShape}(l, c) \left(\mathbf{I}_s[x_s + c, y_s + l] - \mathbf{I}_t[x_t + c, y_t + l]\right)^2, \tag{8.2}$$

where

$$\mathrm{LShape}(l, c) = \begin{cases} 1 & \text{if } l < 0, \text{ or } (l = 0 \text{ and } c < 0), \\ 0 & \text{Otherwise.} \end{cases} \tag{8.3}$$

Using the argmin notation, we look for pixel coordinates (x_s, y_s) in the source image, such that:

$$(x_s, y_s) = \mathrm{argmin}_{(x,y)\in\mathbf{I}_s} \mathrm{SSD}(x, y; x_t, y_t). \tag{8.4}$$

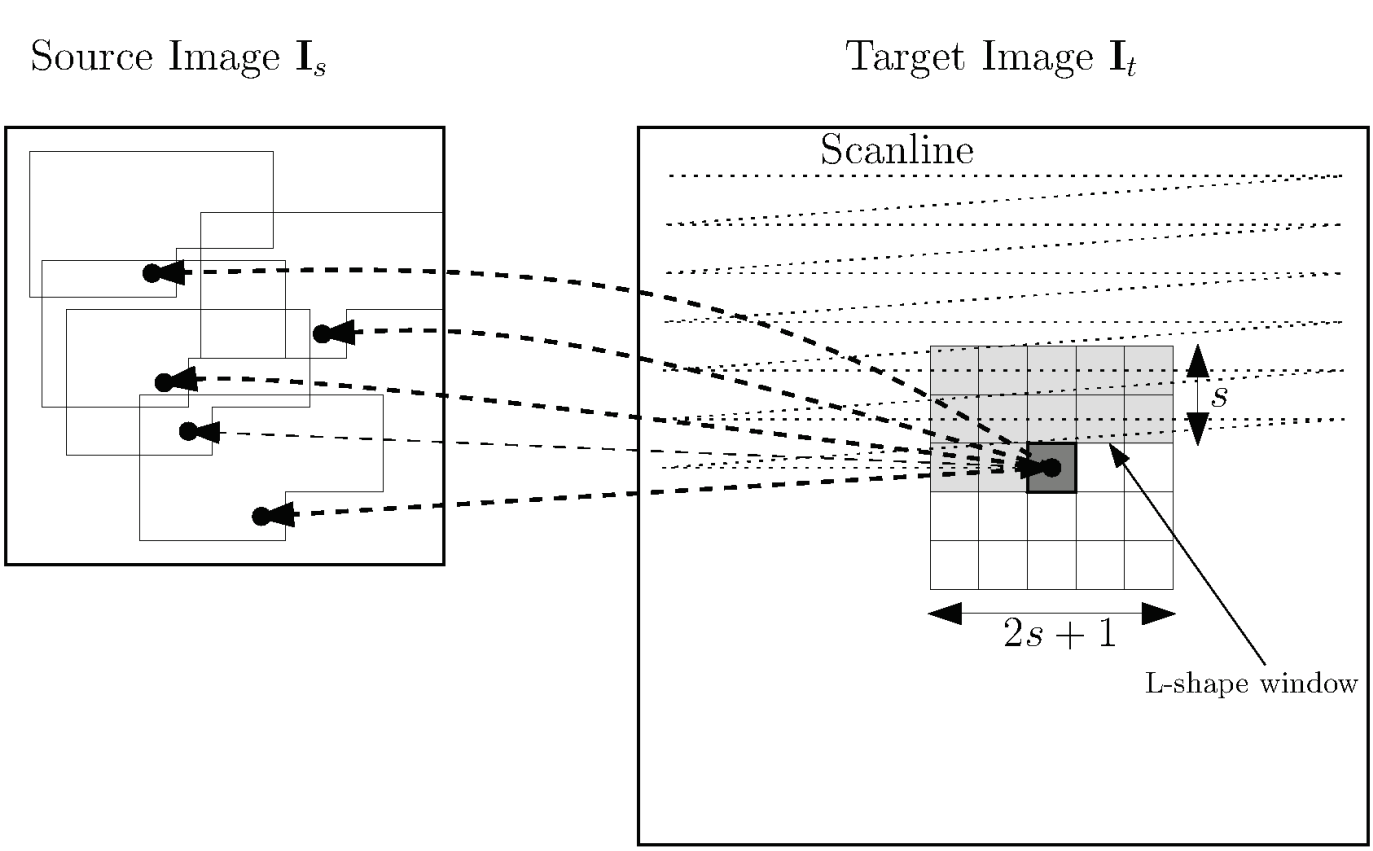

FIGURE 8.3 *Synthesis of a 2D texture image. Pixels are synthesized in horizontal scanline order. For a given position, we find the most similar L-shape window in the source image and write the current pixel position of the target image with the color pixel of the source image that produced the best match.*

We summarize the 2D texture synthesis algorithm in pseudocode below:

TEXTURESYNTHESIS($\mathbf{I}_s, \mathbf{I}_t$)
1. ◁ $\mathbf{I}_s$ is the input texture sample ▷
2. ◁ Create a large texture $\mathbf{I}_t$ ▷
3. Initialize a random color image $\mathbf{I}_t$
4. ◁ Synthesize pixels following the horizontal scanline order ▷
5. **for** $y \leftarrow 1$ **to** h_t
6. **do for** $x \leftarrow 1$ **to** w_t
7. **do** $(x_s, y_s) =$ BESTLSHAPEMATCH($\mathbf{I}_s, x, y$)
8. $\mathbf{I}_t[x, y] = \mathbf{I}_s[x_s, y_s]$

Because the synthesis algorithm is using locally an L-shape window, this technique is classified as a neighborhood-based texture synthesis method. There are other texture synthesis techniques that we mention in the Bibliographical Notes, provided at the end of the chapter. In practice, the size s of the neighborhood should match

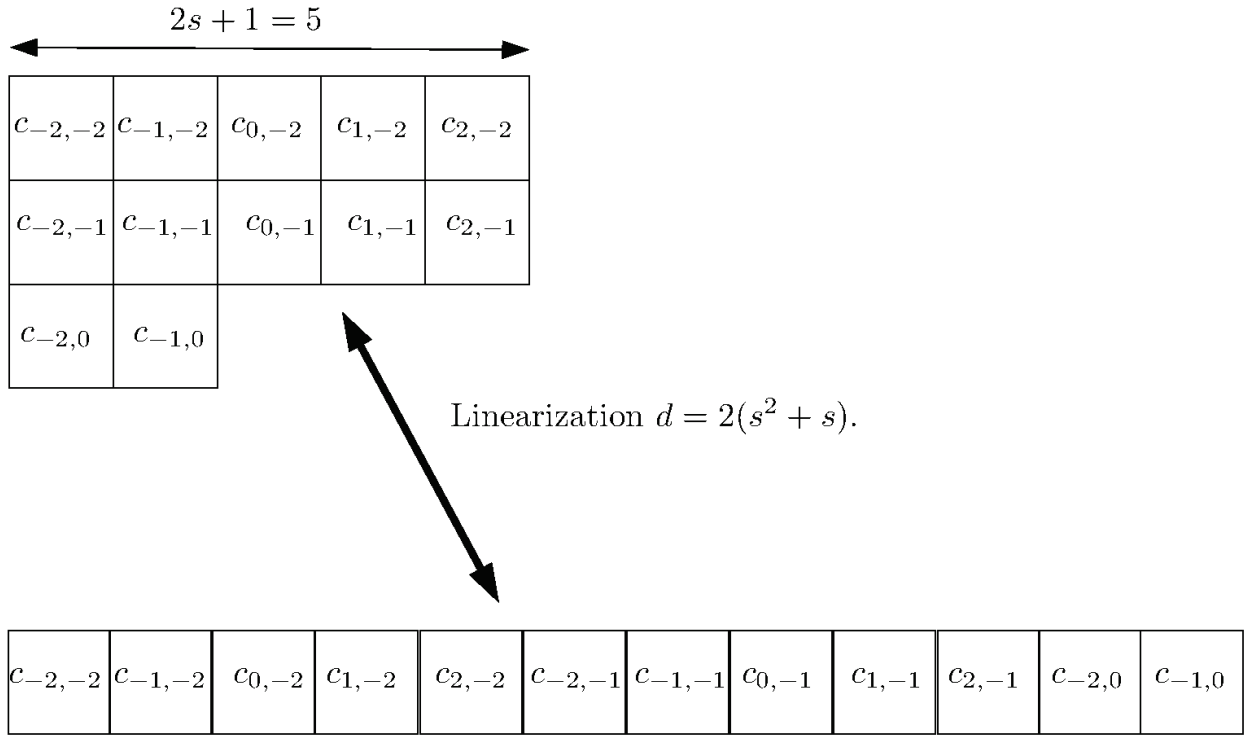

FIGURE 8.4 *Linearization of the 2D L-shape window to a high-dimensional dD vector, obtained by sequentially stacking coefficients.*

with the scale of the largest texture structure (for textures combining several granular elements), in order to preserve that largest pattern statistics on the synthesized image. Note that in the initialization, it is enough to write color *randomly* only for the first s lines of $\mathbf{I}_t$. An implementation would actually create a larger target image (of dimension $(w_t, h_t + s)$) so that its first s lines provide the random support for L-shape neighbor queries. Computing an SSD cross-correlation costs $O(s^2)$ time. Thus, a straightforward naive implementation of this algorithm runs in prohibitive $O(s^2 w_t h_t w_s h_s)$ time. Therefore, we need to devise a tailored data structure for efficiently computing the L-shape window scores (SSDs), or even better, to design a data structure that bypasses checking all possible $w_s \times h_s$ positions in the input texture for a given L-shape query.

The Wei-Levoy texture synthesis algorithm used a tree-structure vector quantization (TSVQ) technique to accelerate the neighborhood search of the original Efros and Leung's algorithm. Here, we present another practical acceleration method based on proximity queries. The idea is to map all source pixels (x_i, y_j) to corresponding points $\mathbf{n}(x_i, y_j)$ in large dimension $d = 2(s^2 + s) = 2s^2 + 2s$ using a process called *linearization*. Figure 8.4 illustrates the linearization process of pixel's L-shapes.

We use the following linearization:

$$\mathbf{n}(x_i, y_j) = \begin{bmatrix} \mathbf{I}_s[x_{i-s}, y_{j-s}] \\ \vdots \\ \mathbf{I}_s[x_{i+s}, y_{j-s}] \\ \mathbf{I}_s[x_{i-s}, y_{j-s+1}] \\ \vdots \\ \mathbf{I}_s[x_{i+s}, y_{j-s+1}] \\ \vdots \\ \vdots \\ \vdots \\ \mathbf{I}_s[x_i - s, y_j] \\ \vdots \\ \mathbf{I}_s[x_i - 1, y_j] \end{bmatrix}. \tag{8.5}$$

That is, we stacked all elements on the L-shape window onto a dD vector (see also Section 3.5.7 where we introduced the Fröbenius norm of a matrix). We defined $\mathbf{n}(x_i, y_j)$ according to the source texture image $\mathbf{I}_s$. Similarly, we denote by $\mathbf{n}_{\mathbf{I}_t}(x_i, y_j)$ the mapping of L-shapes of the target image $\mathbf{I}_t$ to dD-dimensional points. Now, observe that the L_2-distance of two vectors $\mathbf{p} = \begin{bmatrix} p_1 & \dots & p_d \end{bmatrix}^T$ and $\mathbf{q} = \begin{bmatrix} q_1 & \dots & q_d \end{bmatrix}^T$ defined as:

$$||\mathbf{p} - \mathbf{q}|| = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}. \tag{8.6}$$

We defined in Equation 8.2:

$$\text{SSD}(x_s, y_s; x_t, y_t) = \sum_{l=-s}^{s} \sum_{c=-s}^{s} \text{LShape}(l, c)(\mathbf{I}_s[x_s + c, y_s + l] - \mathbf{I}_t[x_t + c, y_t + l])^2.$$

That is:

$$\text{SSD}(x_s, y_s; x_t, y_t) = ||\mathbf{n}(x_s, y_s) - \mathbf{n}(x_t, y_t)||^2. \tag{8.7}$$

Therefore, querying for a nearest neighbor of $\mathbf{n}(x_t, y_t)$ returns the pixel position that gives the best[2] L-shape window match (that is, the lowest SSD score). Note

[2]More precisely, distance order and SSD score order match. Let $\text{SSD}(x_s, y_s; x_t, y_t) = ||\mathbf{n}(x_s, y_s) - \mathbf{n}(x_t, y_t)||^2 = d_t^2$ and $\text{SSD}(x_s, y_s; x_u, y_u) = ||\mathbf{n}(x_s, y_s) - \mathbf{n}(x_u, y_u)||^2 = d_u^2$ (with $d_u > d_t$). Then, $0 \leq d_t < d_u$ implies that $d_t^2 < d_u d_t < d_u^2$. That is, $\text{SSD}(x_s, y_s; x_t, y_t) < \text{SSD}(x_s, y_s; x_u, y_u)$.

that the distance between any two distinct points $\mathbf{n}(x, y)$ is greater than one. For RGB color texture images, we need to calculate nearest neighbor queries in dimension $d = 3(2s^2 + 2s) = 6s(s + 1)$. That is, we further stack onto a vector all r, g, b color components of the L-shape window pixels. There are fortunately fast data structures for reporting or approximating nearest neighbors. We introduce such a data structure, called the kD-tree in Section 8.1.2.

The texture synthesis pseudocode algorithm becomes:

```
TEXTURESYNTHESIS(I_s, s, I_t)
 1. ◁ I_s: RGB color texture sample ▷
 2. ◁ s: defines half-square width ▷
 3. ◁ Create a large texture I_t ▷
 4. d = 3(2s^2 + 2s)
 5. Initialize a random image I_t
 6. ◁ Build a d-dimensional search tree ▷
 7. T = KDTREE(d, {n(x_i, y_j) | (i, j) ∈ w_s × h_s})
 8. ◁ Synthesize pixels in the scanline order ▷
 9. for y ← 1 to h_t
10.     do for x ← 1 to w_t
11.             do n(x_s, y_s) = T.NearestNeighbor(n_{I_t}(x, y))
12.                ◁ Points n(x_s, y_s) are associated with their index (x_s, y_s) ▷
13.                I_t[x, y] = I_s[x_s, y_s]
```

The complexity of this algorithm becomes:

$$O\left(\text{PreprocesskDTree}(w_s h_s, d) + w_t h_t \text{Query}(w_s h_s, d)\right), \tag{8.8}$$

instead of the former $O(w_s h_s w_t h_t s^2)$ time naive implementation. Computing the *exact* nearest neighbor turns out to be a delicate and computational intensive query in high dimensions. Here, we rely on the kD-tree described in Section 8.1.2. Under *some assumptions*, the query time for approximate nearest neighbor queries is $O(\log w_s h_s)$, with the constant hidden in the big Oh-notation depending (exponentially) on d and the approximation factor. Below, we give a C++ implementation of the texture synthesis algorithm that relies on the approximate nearest neighbor library (ANN) of Arya and Mount (refer to the Bibliographical Notes).

WWW Additional source code or supplemental material is provided on the book's Web site:
www.charlesriver.com/Books/BookDetail.aspx?productID=117120
File: `textureperpixel.cpp`

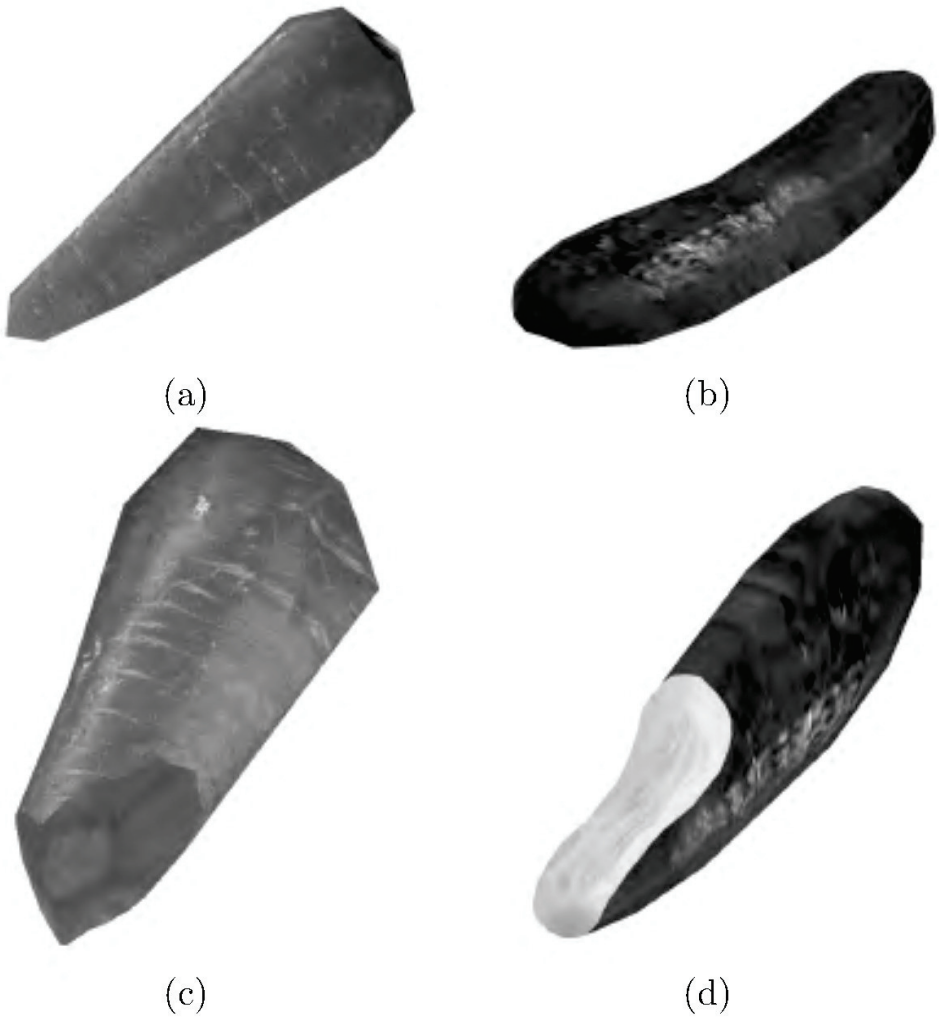

FIGURE 8.5 *Examples of volumetric illustrations created using the 2D distorted texture synthesis techniques of Shigeru Owada (University of Tokyo/Sony Computer Science Laboratories, Japan). The (a)* `carrot` *and (b)* `cucumber` *objects are faked volumetric objects. Every time the model is interactively cut, a different cross-sectional image, shown in (c) and (d), is generated on-the-fly using 2D* distorted *texture synthesis methods.*

```
1 class Image{
2 public:
3    int width, height;
4    unsigned char * raster;
5    ...
6    };
7
8 #define GETPIXELQUICK(img,x,y) &img->raster[((y)*img->width+(x))
```

```
      *3]
 9 #define GETPIXELLOOP(img,x,y) GETPIXELQUICK(img,(x+img->width)%img
      ->width,(y+img->height)%img->height)
10
11 void TextureSynthesis(Image* imgSrc, Image* imgDest, const int
      nsize_2)
12 {
13
14   const int nsize = nsize_2*2 + 1 ;
15
16   CAnnTree_Simple<unsigned char,CPoint> SearchTree ;
17   {
18     // Build search tree.
19
20     SearchTree.Setup( 3 * (nsize*nsize_2+nsize_2) ) ;
21
22     unsigned char* buf = new unsigned char[SearchTree.GetDim()] ;
23
24     for( int iy = nsize_2 ; iy < imgSrc->height ; ++iy ){
25       for( int ix = nsize_2 ; ix < imgSrc->width - nsize_2 ; ++ix
           ){
26         unsigned char* bp = buf ;
27
28         for( int diy=-nsize_2 ; diy<0 ; ++diy )
29           for( int dix=-nsize_2 ; dix<=nsize_2 ; ++dix,bp+=3 ){
30           memcpy( bp,GETPIXELQUICK(imgSrc, ix+dix,iy+diy ), 3);
31         }
32
33         int diy=0 ;
34         for( int dix=-nsize_2 ; dix<0 ; ++dix,bp+=3 ){
35           memcpy( bp,GETPIXELQUICK(imgSrc, ix+dix,iy+diy ), 3);
36         }
37
38         SearchTree.AddElement( buf,CPoint(ix,iy) ) ;
39       }
40     }
41
42     delete[] buf ;
43     SearchTree.ConstructSearchStructure() ;
44   }
```

```
 1   // Fill main L Shape
 2   {
 3     unsigned char* buf = new unsigned char[SearchTree.GetDim()] ;
 4
 5     for( int iy=0;iy<imgDest->height;++iy )
 6       for( int ix=0;ix<imgDest->width;++ix )
 7       {
 8       unsigned char* tgt_pxl = GETPIXELQUICK(imgDest,ix,iy) ;
 9       unsigned char* bp = buf ;
```

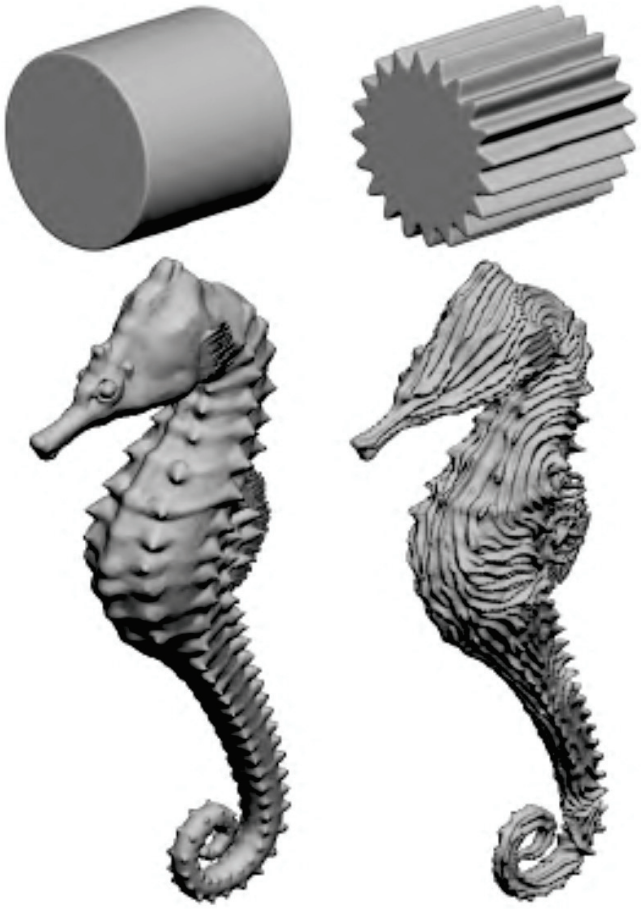

FIGURE 8.6 *Another example of neighborhood-based synthesis: geometry synthesis. The* `seahorse` *model on the right picture is obtained by analogies with the left picture. © Stephen Ingram (Emory University, USA) and Pravin Bhat (University of Washington, USA). Reprinted with permission.*

```
10
11        for( int diy=-nsize_2 ; diy<0 ; ++diy )
12          for( int dix=-nsize_2 ; dix<=nsize_2 ; ++dix,bp+=3 )
13            memcpy( bp,GETPIXELLOOP(imgDest, ix+dix,iy+diy ), 3 ) ;
14
15        int diy=0 ;
16        for( int dix=-nsize_2 ; dix<0 ; ++dix,bp+=3 )
17          memcpy( bp,GETPIXELLOOP(imgDest, ix+dix,iy+diy ), 3 );
18
19        CPoint& bestMatchLocation = SearchTree.FindMostSimilar( buf
              ) ;
20        memcpy( tgt_pxl,GETPIXELQUICK(imgSrc, bestMatchLocation.x,
              bestMatchLocation.y ), 3 ) ;
```

```
21      }
22
23      delete[] buf ;
24    }
25 }
```

Texture synthesis has many other marvelous applications in visual computing (e.g., image analogies, texturing 3D meshes or solid objects, geometry synthesis, etc.) For example, Figure 8.5 shows a volumetric illustration system developed by Owada and his colleagues that fakes textured solid objects by synthesizing *on-the-fly* 2D distorted textures every time a user cut a surface model. This kind of neighborhood-based synthesis can also be used for other attribute synthesis, such as geometry. Figure 8.6 presents some volumetric models obtained by analogies.

8.1.2 kD-Trees

A kD-tree is a generic multidimensional search tree based on partitioning the space recursively with axis-parallel line bisectors. Figure 8.7 illustrates such a construction for a toy set of eight points. Because kD-trees consider each axis independently, its construction is much more efficient than traditional quadtrees or octrees.

In 2D, a kD-tree for a set of n points uses linear storage and can be constructed using the divide-and-conquer paradigm, in $O(n \log n)$ time. The pseudocode below explains the recursive construction of a kD-tree by splitting regions alternatively by horizontal or vertical lines, according to the level parity (see Figure 8.7):

```
kDTree(P, l)
 1. ◁ Build a 2D kD-tree ▷
 2. ◁ l denote the level. Initially, l = 0 ▷
 3. if |P| = 1
 4.    then return Leaf(P)
 5.    else if Even(l)
 6.            then ◁ Compute the median x-abscissa (vertical split) ▷
 7.                 x_l = MedianX(P)
 8.                 P_left = {p ∈ P | x(p) ≤ x_l}
 9.                 P_right = {p ∈ P | x(p) > x_l}
10.                 return Tree(x_l, kDTree(P_left, l + 1), kDTree(P_right, l + 1));
11.            else ◁ Compute the median y-abscissa (horizontal split) ▷
12.                 y_l = MedianY(P)
13.                 P_bottom = {p ∈ P | y(p) ≤ y_l}
14.                 P_top = {p ∈ P | y(p) > y_l}
15.                 return Tree(y_l, kDTree(P_bottom, l + 1), kDTree(P_top, l + 1));
```

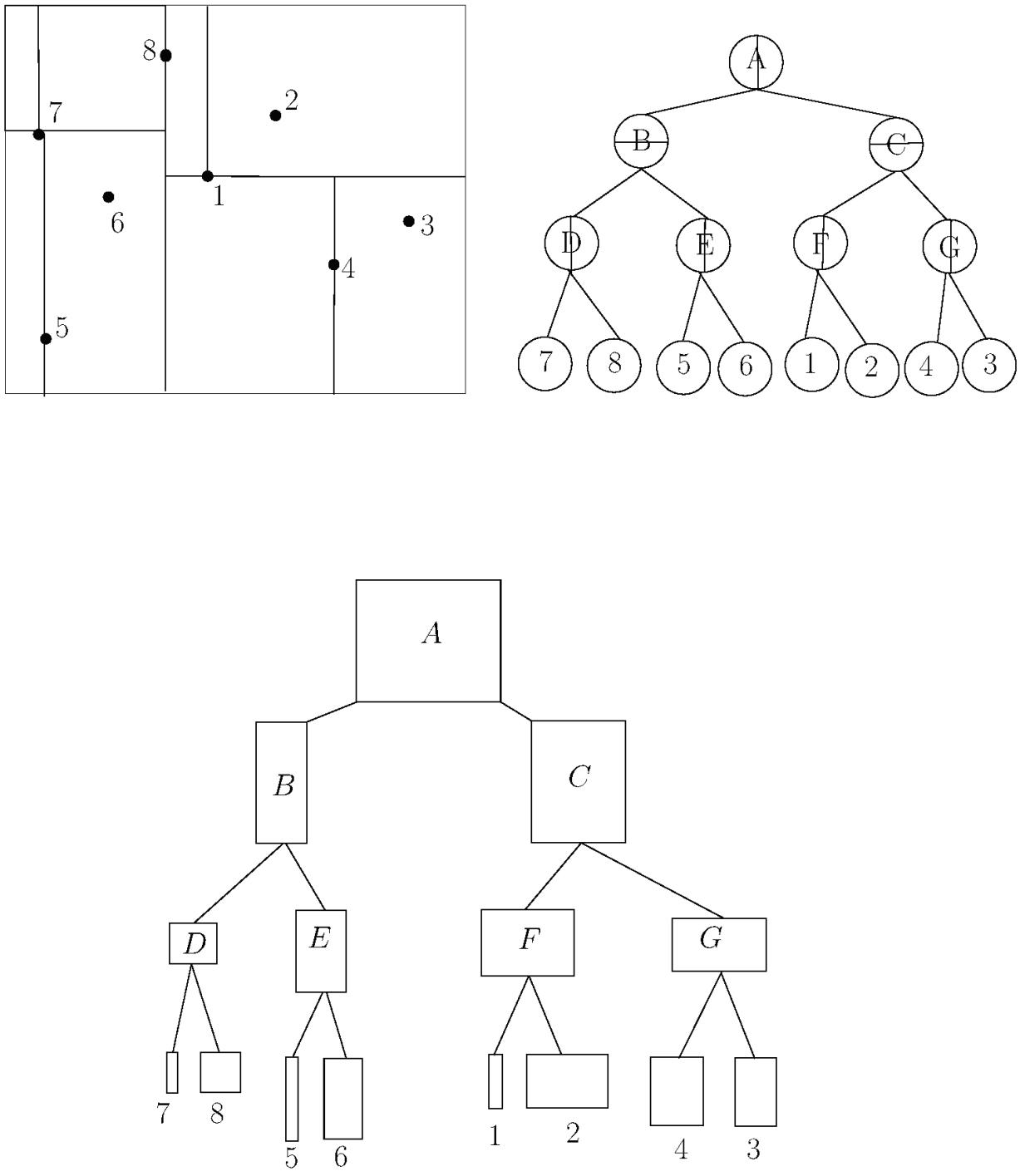

FIGURE 8.7 *A 2D kD-tree that illustrates the recursive splitting of isothetic rectangles by either horizontal or vertical line bisectors. The bottom tree shows the rectangular domain associated to each node of the kD-tree.*

kD-trees are specially tailored data structures to answer orthogonal range queries. An *orthogonal range query* consists in reporting all points of $\mathcal{P}$ contained in an axis-parallel domain $R : [a_1, b_1] \times [a_2, b_2]$. Domain R is also called an isothetic range box. The geometric domain R is equivalently defined as:

$$R = \{(x, y) \mid a_1 \leq x \leq b_1 \text{ and } a_2 \leq y \leq b_2\}. \tag{8.9}$$

Orthogonal range searching is particularly well suited for managing databases. In dimension d, a range query is written similarly as the product of d 1D intervals:

$$R : \prod_{i=1}^{d} [a_i, b_i]. \tag{8.10}$$

The recursive construction of kD-trees straightforwardly generalizes to arbitrary dimension. Orientation of line bisectors are chosen according $l \bmod d \in \{0, ..., d-1\}$ (instead of $l \bmod 2$), as suggested in Figure 8.7. In d-dimensional Euclidean space, a kD-tree on n points can be built in $O(dn \log n)$-time using $O(dn)$ storage. The query time for an orthogonal range query R is $O(dn^{1-\frac{1}{d}} + k)$, where k is the number of points belonging to the query domain. That is $k = |R \cap \mathcal{P}|$. Let's describe the algorithm for answering proximity queries.

Using a kD-Tree, we find the nearest neighbor of a query point by traversing the structure recursively. Let each node V of the kD-tree store the following five records:

- **axis**: the splitting axis dimension (from 1 to d)
- **value**: the splitting value ($x_{\text{axis}} = \text{value}$)
- **left**, **right**: the pointers to the left and right kD-Trees rooted at the current node
- **point**: a pointer to a point if both left and right records are null pointers (that is, current node is a leaf)

Given a query point $\mathbf{q}$, let dist denote the distance to the so-far nearest point found in the kD-Tree. Initially, dist is set to infinity: dist $= +\infty$. To determine at a node V whether to stop or proceed on the recursive exploration of the kD-tree, observe that if, and only if, $\mathbf{q}_{V.\text{axis}} - \text{dist} \leq V.\text{value}$ we need to explore the left kD-subtree (see Figure 8.8). Similarly, if, and only if, $\mathbf{q}_{V.\text{axis}} + \text{dist} \geq V.\text{value}$ we need to explore the right kD-subtree.

Thus, the nearest neighbor query algorithm on a kD-tree T is written as follows:

```
SearchNNinkDTree(q, V; p, dist)
 1.  ◁ Input: ▷
 2.  ◁ V: a kD-Tree node ▷
 3.  ◁ q: a query point ▷
 4.  ◁ Output: ▷
 5.  ◁ p: nearest neighbor point ▷
 6.  ◁ dist: distance to the nearest neighbor ▷
 7.  if V.left = V.right = NULL
 8.     then ◁ Leaf of a kD-Tree ▷
 9.          dist' = ||q − V.point||
10.          if dist' < dist
11.             then dist = dist'
12.                  p = V.point
13.     else if q_{V.axis} ≤ V.value
14.             then ◁ Search on the left subtree first ▷
15.                  SearchNNinkDTree(q, V.left; p, dist)
16.                  if q_{V.axis} + dist > V.value
17.                     then SearchNNinkDTree(q, V.right; p, dist)
18.             else ◁ Search on the right subtree first ▷
19.                  SearchNNinkDTree(q, V.right; p, dist)
20.                  if q_{V.axis} − dist ≤ V.value
21.                     then SearchNNinkDTree(q, V.left; p, dist)
```

Initially, we call the procedure as follows: SearchNNinkDTree(**q**, root; **p**, $+\infty$), where root is the root of the kD-tree, and **q** the query point. The nearest neighbor is reported in **p** with its distance to **q**: $||\mathbf{pq}|| = \text{dist}$.

Arya and Fu designed a particular kD-tree splitting-rule and studied its expected time performance of finding not exact, but rather approximate nearest neighbors. A point **p** is a $(1+\epsilon)$-approximate nearest neighbor of **q** if:

$$||\mathbf{qp}|| \leq (1+\epsilon)||\mathbf{q}N(\mathbf{q})||, \tag{8.11}$$

where $N(\mathbf{q})$ denotes the exact nearest neighbor. For arbitrary d-dimensional point sets belonging to the unit hypercube $H : \prod_{i=1}^{d}[0,1] = [0,1]^d$ and approximate nearest neighbor queries sampled from the *uniform distribution* in the same hypercube H, it can be shown that the expected query time is $\tilde{O}(\frac{1}{\epsilon^d}\log n)$. That is, expected logarithmic time but exponentially dependent on the dimension. Interestingly, this running time is *invariant* from the distribution of the point set. The only assumption is that queries are *randomly* and *uniformly* drawn from the unit hypercube.

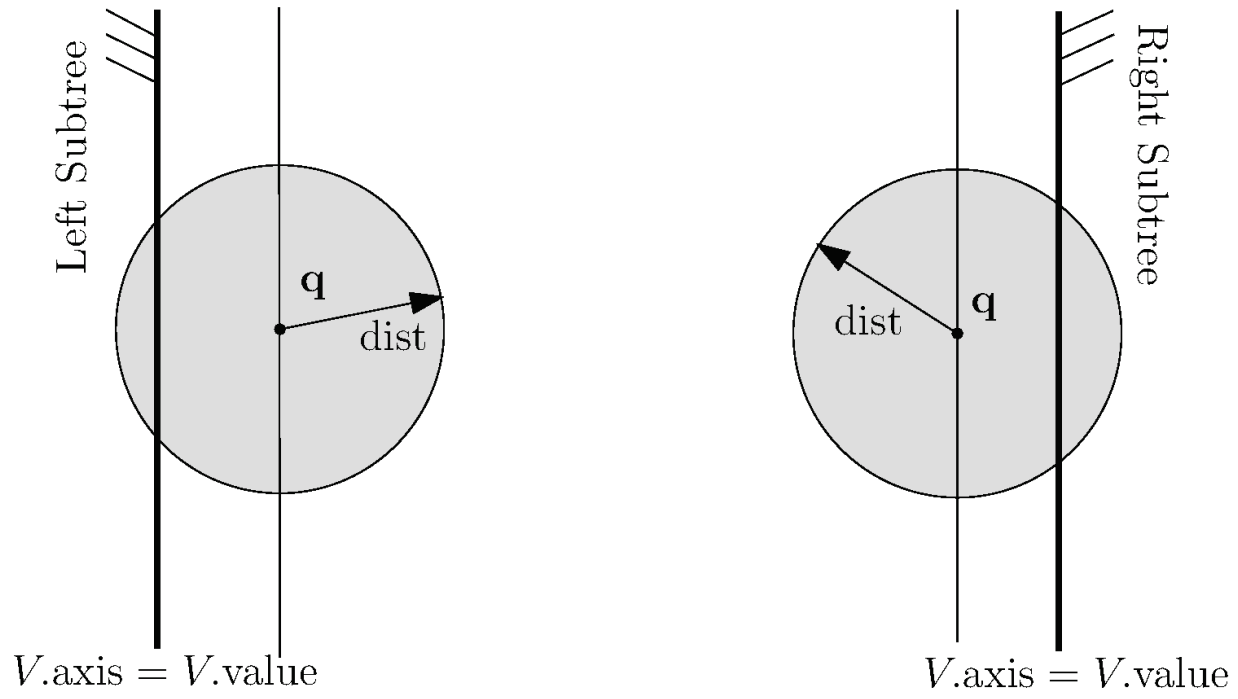

FIGURE 8.8 *Using a kD-tree to perform nearest neighbor queries. Given a current distance* dist *to the nearest neighbor of* $\mathbf{q}$*, the algorithm decides to explore on the kD-subtrees depending on whether the ball centered at* $\mathbf{q}$ *and radius* dist *intersects the splitting plane or not.*

Finally, let us mention another way of using orthogonal range queries for finding nearest (or approximate) neighbors of a query point $\mathbf{q}$. The idea is to use a dichotomy search on the side length s of orthogonal *square* queries R centered at $\mathbf{q}$. That is, we recursively expand or shrink the orthogonal square query R based on whether some points of $\mathcal{P}$ belong to R or not. Figure 8.9 illustrates the use of orthogonal range queries to answer exact or approximate nearest neighbor queries. We spend logarithmic time to report that no points of $\mathcal{P}$ are in R, and we only need to report one point to confirm that the nearest neighbor $N(\mathbf{q})$ is inside R (logarithmic cost). Even if the distance is defined using the L_2 metric, it is enough to consider the L_∞ metric (orthogonal ranges) to probe for the presence of the nearest neighbor.

kD-trees are quite expensive data structures for orthogonal range queries whenever the number of points to report is not large, since the overhead $O(n^{1-\frac{1}{d}})$ becomes prohibitively expensive. There exists an alternative optimized geometric data structure, called *layered range trees*, that bypasses the $n^{1-\frac{1}{d}}$ complexity bottleneck. A d-dimensional set of n points is first preprocessed in a layered range tree in $O(n \log^{d-1} n)$ time, using $O(n \log^{d-1} n)$ space. Then, each orthogonal range query R is answered in $O(k + \log^{d-1} n)$ time using a technique called *fractional cascading*, where $k = |R \cap \mathcal{P}|$.

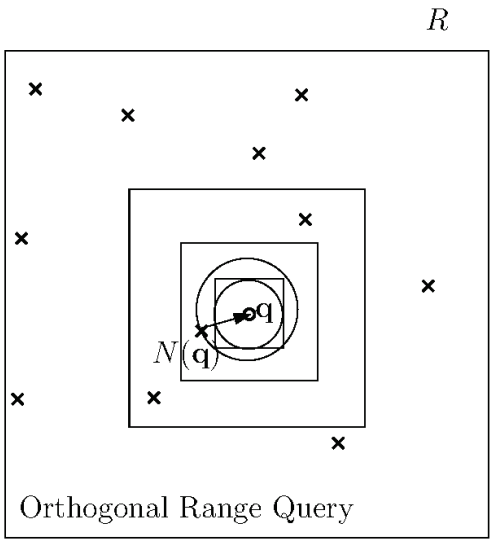

FIGURE 8.9 *Using orthogonal range queries (squares) to probe for the approximate or exact nearest neighbor $N(\mathbf{q})$ of query point $\mathbf{q}$.*

8.2 Clustering

Clustering is the problem of grouping similar elements of a data set into corresponding clusters. Finding the right number of clusters and measuring the quality of a clustering are key challenges, especially for noisy input. Clustering is one of the most common techniques used in data organization and data mining. But clustering also finds many other (indirect) applications in visual computing. For example, in computer vision, clustering allows us to significantly reduce the input size to work on classified categories rather than directly on data elements. We first start by describing a popular application of clustering: color quantization of images. We then present certainly the most famous clustering algorithm: kMeans.

8.2.1 Application: Color Quantization

Color quantization is the problem of choosing among the space of continuous colors (or very large discretized space such as the 24-bit true color space that contains 16,777,216 distinct colors) a given number of representative colors. Those representative colors are arranged in a *color palette*, and the pixel colors of the image are then indexed according to that palette. Color quantization is an important ingredient of image compression as it allows us to reduce the numbers of distinct pixel colors used in images. For example, a 24-bit true-color image may be first converted into a 256-color image before being saved. Eventually, dithering techniques such as Floyd-Steinberg error diffusion (see Section 4.4.2) may further be applied to generate the

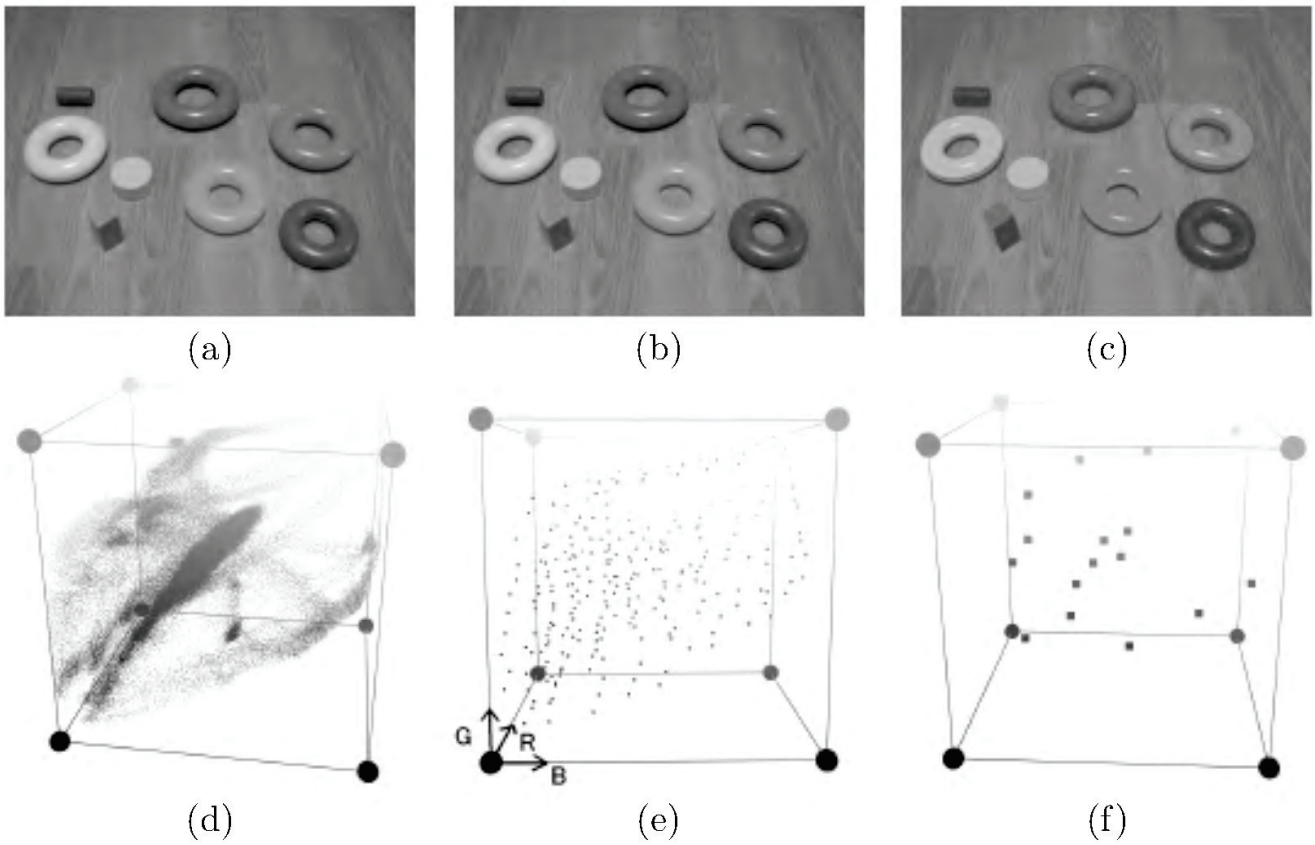

FIGURE 8.10 *Color quantizations. Given a true-color input image (a) of resolution* 1024×768 *(with 16,777,216 distinct colors), picture (b) shows a 256-color downconverted result, and a 16-color version is shown in (c). The corresponding color pixels (color palettes in (b) and (c)) are shown respectively in the RGB color cube in (d), (e), and (f). See* COLOR PLATE XV.

downconverted image. Figure 8.10 illustrates the color quantization results on a given source image. The distribution of colors of respective 24-bit, 8-bit, and 4-bit color images are shown in the RGB color cube. Observe that the perceptual difference between the true-color picture of Figure 8.10(a) and the 8-bit color image of Figure 8.10(b) is less noticeable than the perceptual gap from Figure 8.10(a) to the 4-bit color image of Figure 8.10(c). To find a *good* color palette, we need to cluster the color distribution of the true-color image into a fixed number of clusters (say, $256 = 2^8$ as shown in Figure 8.10(e), or $16 = 2^4$ as shown in Figure 8.10(f)). Once a clustering of the 24-bit color distribution is calculated, we select a representative color of each cluster (like its centroid) to create the color palette of the downconverted image. Let us now describe a simple method to perform this clustering task: the kMeans algorithm.

8.2.2 Clustering by kMeans

kMeans is a clustering technique that was first developed for vector quantization (VQ) applications by Lloyd in 1957. In a *vector quantization* problem, we are given a set of vectors, which we would like to group into a few representative vectors to reduce the overall number of bit occupancy of data. The set of representative vectors form a *codebook*. Color quantization is a typical example of vector quantization, where the color palette is namely the color codebook. Measuring the goodness of clusters, and therefore the performance of different clustering algorithms, can be done in various ways. We consider the mean square error (MSE) defined as the total within-cluster variance. Let $\mathcal{P} = \{\mathbf{p}_1, ..., \mathbf{p}_n\}$ denote the input vector set, a set of n d-dimensional points. Let $\mathcal{C} = \{\mathbf{c}_1, ..., \mathbf{c}_k\}$ be the k codebook vectors (cluster centers). Then the mean square error is defined by the following error measures:

$$\text{MSE}(\mathcal{P}, \mathcal{C}) = \sum_{i=1}^{k} \sum_{j=1}^{n} w(j,i) ||\mathbf{p}_j - \mathbf{c}_i||^2, \tag{8.12}$$

where $w(j,i)$ is the point membership of $\mathbf{p}_j$ with respect to all the cluster centers, normalized so that the weights of each point add up to one:

$$w(j,i) \geq 0, \qquad \sum_{i=1}^{k} w(j,i) = 1. \tag{8.13}$$

For the kMeans, a point $\mathbf{p} \in \mathcal{P}$ is allowed to belong to exactly one cluster. That is $w(j,i) = 1$ if, and only if, $\mathbf{p}_j$ belongs to the i-th cluster: $||\mathbf{p}_j\mathbf{c}_i|| \leq ||\mathbf{p}_j\mathbf{c}_l||$ for all $l \in \{1, .., k\}$. (In case of ties, we choose the cluster center with the lowest index.) We say that kMeans clusters using the *hard memberships* of points (see Equation 8.14). If a point may belong to several clusters, we need to take into account its contribution to each cluster, and we look for a *soft membership* clustering algorithm. In that case the weights of a point form a membership distribution. The kMeans algorithm is a hard-membership iterative local optimization technique that clusters the data into a given number k of clusters. kMeans is proven to minimize the MSE but can be trapped into local minima. Each input vector is attributed to the Voronoi cell of the Voronoi diagram of the codebook (Figure 8.11):

$$\text{MSE}(\mathcal{P}, \mathcal{C}) = \sum_{i=1}^{n} \min_{j=1}^{k} ||\mathbf{p}_i - \mathbf{c}_j||^2. \tag{8.14}$$

Each iteration of the kMeans algorithm works into two substeps: (1) allocate points to their clusters, and (2) refine cluster centroids. Initialization of the clusters is essential since the kMeans is a *local* optimization procedure. A common initialization

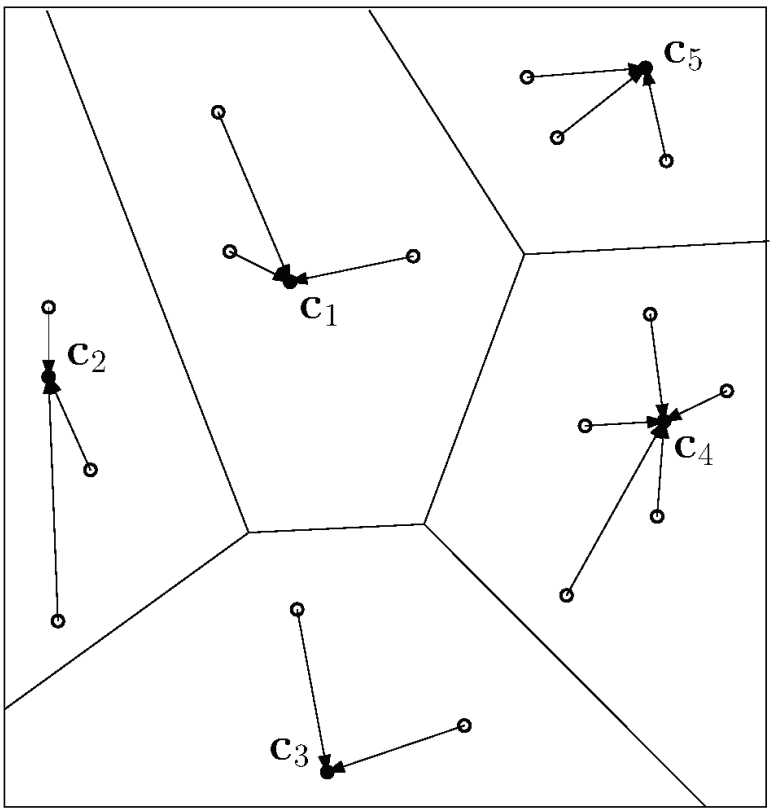

FIGURE 8.11 *Each point is associated to its nearest cluster center. The cluster centers form a Voronoi partition of the plane that defines the hard membership of points.*

heuristic is to randomly draw k points from the point set $\mathcal{P}$. This is the Forgy classic initialization routine.

We summarize the kMeans algorithm in pseudocode:

kMeans$(\mathcal{P}, \epsilon)$
1. ◁ Cluster points of $\mathcal{P}$ using kMeans ▷
2. ◁ ϵ: threshold criterion to decide whether to stop or not ▷
3. Initialize centroids $\mathcal{C}$
4. **while** Total centroid displacements is less than threshold ϵ
5. **do** ◁ Allocate points to clusters (hard membership) ▷
6. **for** $i \leftarrow 1$ **to** n
7. **do** $C(\mathbf{p}_i) = \text{argmin}_{j=1}^{k} ||\mathbf{p}_i\mathbf{c}_j||$
8. **for** $i \leftarrow 1$ **to** k
9. **do** ◁ Update centroids to the center of mass of clusters ▷
10. $\mathcal{C}(\mathbf{c}_i) = \{\mathbf{p} \in \mathcal{P} \mid C(\mathbf{p}) = i\}$
11. $\mathbf{c}_i = \text{CenterOfMass}(\mathcal{C}(\mathbf{c}_i))$

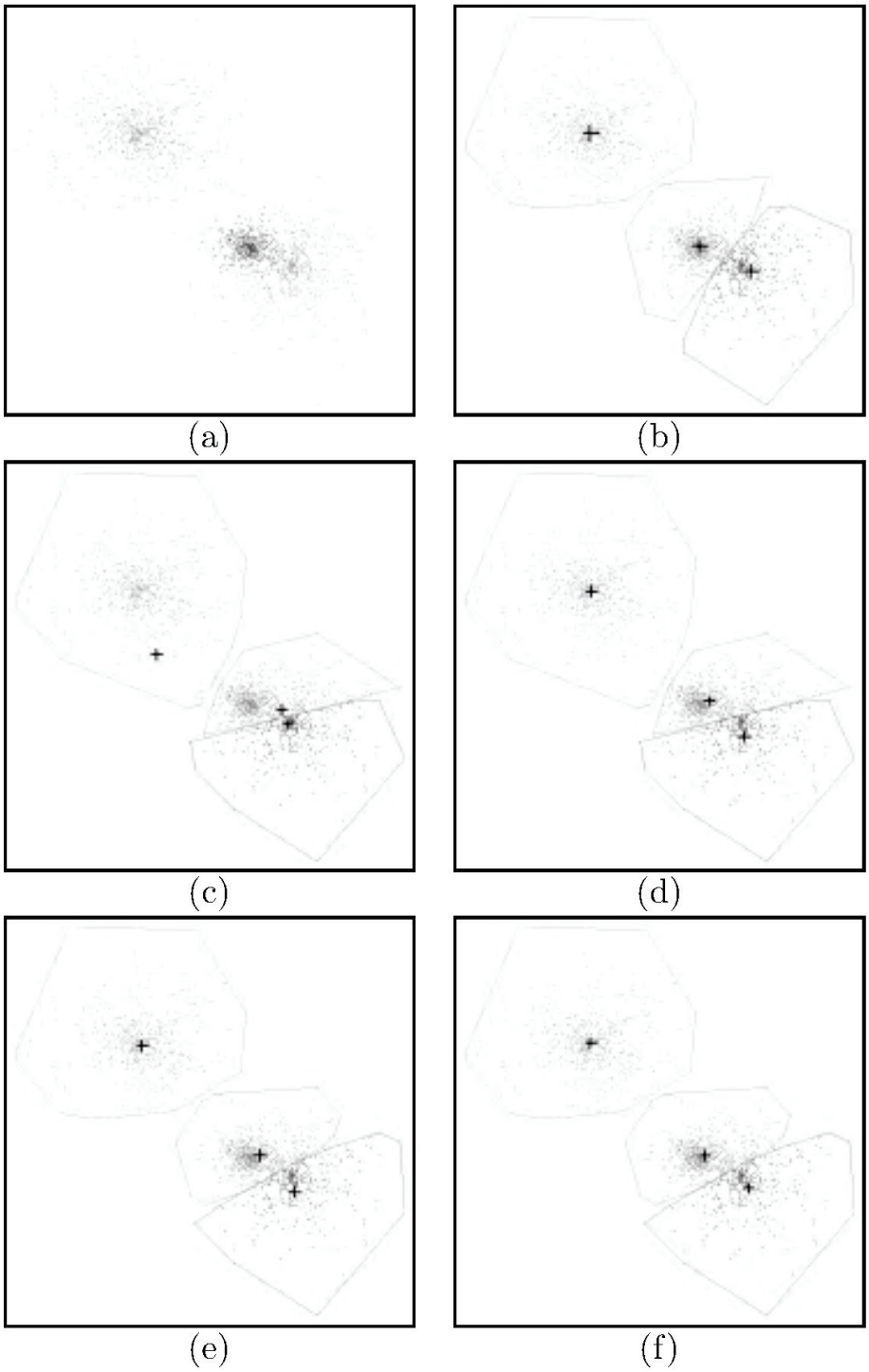

FIGURE 8.12 *Example of clustering. (a) The initial 1500-point set is drawn from three Gaussians (500 points each). The initial three cluster centers have been randomly initialized. (b) depicts the result of the kMeans clustering after 50 iterations. (c) & (d) illustrate the first round of kMeans (assigning points to clusters and updating the cluster centroids). (e) & (f) show the configuration for the second round. For each cluster, we show the convex hull of the points belonging to it.*

WWW Additional source code or supplemental material is provided on the book's Web site:
www.charlesriver.com/Books/BookDetail.aspx?productID=117120
File: `kmeans.cpp`

Figure 8.12 shows an example of clustering using kMeans. In practice, kMeans is known to be very sensitive to the initialization of the cluster centers. kMeans belongs to a larger class of center-based clustering algorithms. kMeans can be applied on very high-dimensional data sets, such as a set of images (dimension is the image width times the image height). This is particularly useful for compressing a sequence of images using a common codebook. Another popular clustering algorithm is the expectation-maximization (EM), which assumes a linear mixing of Gaussian distributions. In an EM-clustering, points are clustered using soft memberships. That is, each point belongs potentially to *all* clusters. The weights of each point represent the membership distribution.

In computational geometry, we state the problem of clustering point sets by defining a mathematical *global* criterion. For example, the k-center problem is stated as follows: given an n-point set $\mathcal{P}$ on the plane, minimize r and find corresponding k positions $\mathbf{c}_1, ..., \mathbf{c}_k$ such that $\mathcal{P}$ can be covered by k disks of radius r centered at the $\mathbf{c}_i$'s:

$$\mathcal{P} \subseteq \bigcup_{i=1}^{k} \text{Disk}(\mathbf{c}_i, r). \tag{8.15}$$

Yet another related optimization problem is the k-median of a point set: given an n-point set $\mathcal{P}$ on the plane, minimize the sum of the radii $\sum_{i=1}^{k} r_i$ and find corresponding k positions $\mathbf{c}_1, ..., \mathbf{c}_k$ such that $\mathcal{P}$ can be covered by k disks (disks have potentially different radii) centered at the $\mathbf{c}_i$'s:

$$\mathcal{P} \subseteq \bigcup_{i=1}^{k} \text{Disk}(\mathbf{c}_i, r_i). \tag{8.16}$$

Unfortunately, both those clustering problems have been proven NP-complete. These problems are therefore intractable in practice (unless P=NP). Since the point set can be considered as the geometric embedding of a complete graph (a clique), those k-median and k-center problems on points are special instances of more general graph-theoretic problems that have been motivated by facility locations problems, and for which many heuristics have been designed.

A related clustering technique is the *centroidal Voronoi diagram*, that is also used to initialize the cluster seeds of other clusterings. The idea of a centroidal Voronoi

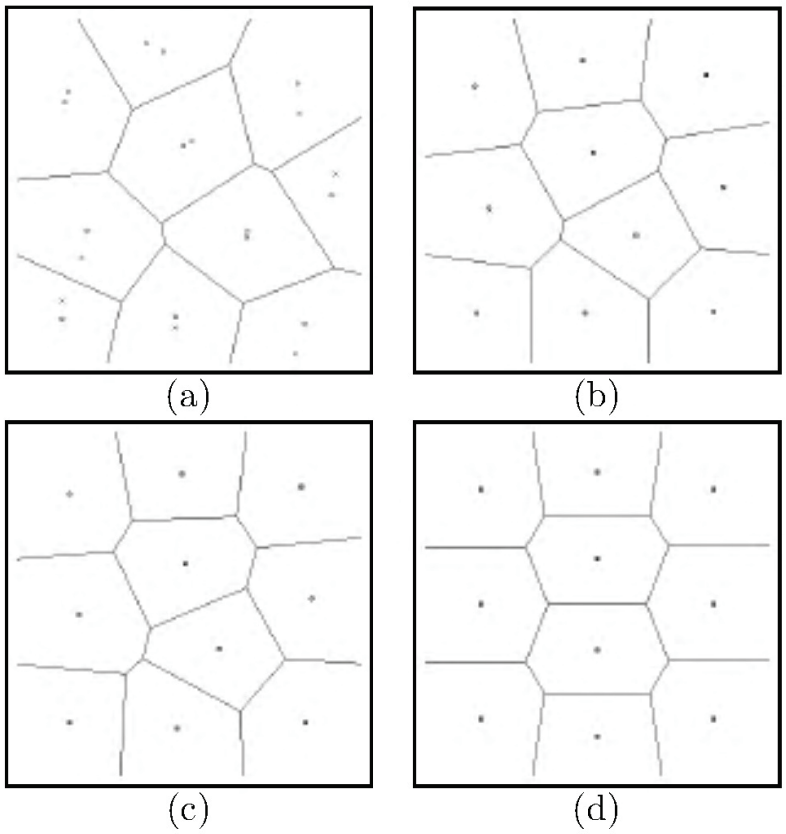

FIGURE 8.13 *Initial configuration of the generators (a), and centroidal Voronoi diagrams after 5 iterations (b), 10 iterations (c), and 100 iterations (d). Especially visible in (a), we show the generators using crosses and the centers of mass of the Voronoi cells as circles.*

diagram is to distribute a given set of k points evenly on an irregular continous domain (for example, a surface mesh). Thus, in a centroidal Voronoi diagram, the point density of sites, also called generators, tends to be the same. That is, the areas of the Voronoi cells of generators tend to be identical. To compute a centroidal Voronoi diagram, we relax the kMeans algorithm as follows (see Figure 8.13):

CentroidalVoronoi($\mathcal{C}, \epsilon$)
1. $\triangleleft$ Compute k points evenly distributed on a spatial domain $\triangleright$
2. $\triangleleft$ ϵ: threshold criterion to decide whether to stop or not $\triangleright$
3. Initialize centroids $\mathcal{C}$
4. **while** Total centroid displacements less than ϵ
5. **do** Compute Voronoi diagram of $\mathcal{C}$
6. Allocate each $\mathbf{c}_i$ to the center of mass of its Voronoi cell

That is, we iteratively compute Voronoi diagrams. At each iteration, we move

the current sites (generators) to the centers of mass of their Voronoi cells. Thus, the difference with the former kMeans algorithm is that the centroidal Voronoi diagram considers the continuous space (Voronoi cells), and not a finite (discrete) space of points. This difference plays a theoretically important role, because no convergence of the centroidal Voronoi diagram is proven guaranteed in the plane. (Remember that kMeans is formally proven to converge to a local minimum, as *each* stage of the algorithm reduces the mean square error.) More precisely, convergence of centroidal Voronoi diagrams have been shown in 1D, and experimentally observed in other dimensions, but lack a mathematical proof. Centroidal Voronoi diagrams have been used successfully in many visual computing applications, such as for efficiently remeshing 3D models, half-toning images, Monte Carlo samplings, or creating non-photo-realistic renderings (stipplings).

We now give an overview of Voronoi diagrams that are omnipresent structures in visual computing. In an *ordinary Voronoi diagram*, each bounded or unbounded region is defined by a convex *Thiessen polygon*. Boundaries of Voronoi cells are obtained from the *bisectors* of generators. Bisectors represent lines of equilibrium of generators. That is, bisectors are straight lines perpendicular to the lines connecting the generators, and intersecting them exactly half-way. Two nonparallel bisectors meet in a point.

Voronoi diagrams can be interpreted in two different ways: (1) static point assignment, or (2) dynamic generator growth.

Static (Point→Generator). Each point of $\mathbb{R}^2$ is assigned to its closest generator. This is an assignment task, where distance may be thought as a friction function.

Dynamic (Generator→Point). All generators grow their regions, starting at the same time from their seed positions with the same growing rate in all directions. This dynamic interpretation of Voronoi diagram models the physical properties of crystal growth.

There exists many other regionalization variants of the ordinary Voronoi diagram. The measure function that describes the path length from any point to a generator can further be taken as an arbitrary function $f(\cdot)$, thus relaxing the traditional Euclidean distance function. Further, attributes can be attached to generators for computing the "distance" from points to generators.

The Voronoi cell of generator $\mathbf{p}_i \in \mathcal{P}$ is defined as:

$$V(\mathbf{p}_i) = \{\mathbf{x} \mid f(\mathbf{p}_i) \leq f(\mathbf{p}_j),\ \forall j \in \{1, ..., n\}\}\,. \tag{8.17}$$

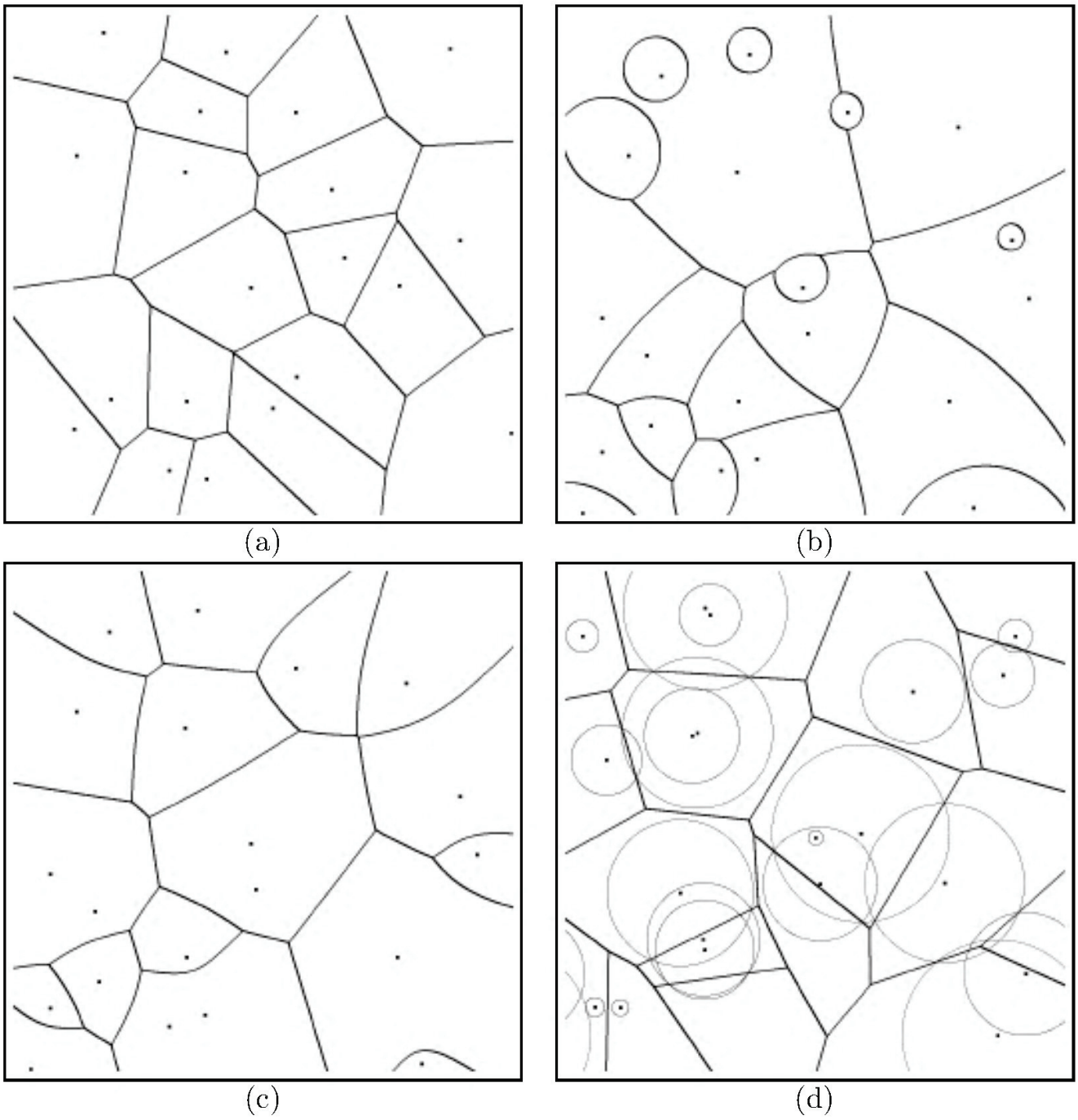

FIGURE 8.14 *Generalizations of Voronoi diagrams. (a) ordinary Voronoi diagram. (b) multiplicatively weighted Voronoi diagram. (c) additively weighted Voronoi diagram. (d) power diagram (as known as Laguerre diagram).*

Let us describe a few examples with corresponding diagrams illustrated in Figure 8.14:

Additively weighted Voronoi diagram. Each generator $\mathbf{p}_i$ is associated with a weight w_i. The Voronoi cell $V(\mathbf{p}_i)$ of generator $\mathbf{p}_i$ is defined as:

$$V(\mathbf{p}_i) = \{\mathbf{x} \mid ||\mathbf{x}\mathbf{p}_i|| + w_i \leq ||\mathbf{x}\mathbf{p}_j|| + w_j,\ \forall j \in \{1, ..., n\}\}. \tag{8.18}$$

This means that generators grow at the *same rate* but start their growing process at time indicated by their *respective* additive weights. Bisectors are no longer straight lines but arcs of hyperboles (Figure 8.14(c)).

Multiplicatively weighted Voronoi diagram. Instead of adding weights, we can choose to multiply weights to the distance function. The Voronoi cell of generator $\mathbf{p}_i$ is then defined as:

$$V(\mathbf{p}_i) = \left\{\mathbf{x} \;\middle|\; \frac{||\mathbf{x}\mathbf{p}_i||}{w_i} \leq \frac{||\mathbf{x}\mathbf{p}_j||}{w_j},\ \forall j \in \{1, ..., n\}\right\}. \tag{8.19}$$

Intuitively, this means that generators start growing at the same time, but with *different growth rates*. The multiplicative weights can also be interpreted as the different friction coefficients. Bisectors are portions of Appolonius circles[3] and twice intersect the lines connecting the generators $\mathbf{p}_i$ and $\mathbf{p}_j$, at relative positions $\frac{w_i}{w_i+w_j}$ and $\frac{w_j}{w_i+w_j}$. The highest weight region is unbounded and surrounds all other bounded regions. Cells are not necessarily anymore convex (see Figure 8.14(b)).

Multiplicatively and additively weighted Voronoi diagram. This diagram is a compound of the previous two diagrams. The Voronoi cell of generator $\mathbf{p}_i$ is defined as:

$$V(\mathbf{p}_i) = \left\{\mathbf{x} \;\middle|\; \frac{||\mathbf{x}\mathbf{p}_i||}{w_i} + w_i' \leq \frac{||\mathbf{x}\mathbf{p}_j||}{w_j} + w_j',\ \forall j \in \{1, ..., n\}\right\}. \tag{8.20}$$

Power diagram. Instead of taking the regular distance function, we consider a *power function* of the distance. The Voronoi region associated to $\mathbf{p}_i$ is defined as:

$$V(\mathbf{p}_i) = \{\mathbf{x} \mid ||\mathbf{x}\mathbf{p}_i||^p - w_i \leq ||\mathbf{x}\mathbf{p}_j||^p - w_j,\ \forall j \in \{1, ..., n\}\}, \tag{8.21}$$

for some given p.

[3] An Appolonius circle is defined as the set of points whose distances from two fixed points are in a constant ratio.

In particular, the power of a point $\mathbf{p}$ to a disk $\text{Disk}(\mathbf{c}, r)$ of circumcenter $\mathbf{c}$ and radius r is defined as $||\mathbf{pc}||^2 - r^2$. Power diagrams are also known as *Laguerre diagrams* or *Dirichlet cell complexes*. Each cell is convex and bounded by straight line edges. Some cells may be empty of generators whilst others may contain several. The bisector of two nonconcentric disks is a line perpendicular to the line joining these two disk centers. Observe that setting $r = 0$ for all generators yields back the ordinary Voronoi diagram. Indeed, writing $||\mathbf{xp_i}||^p \leq ||\mathbf{xp}_j||^p$ as $\exp(p \ln ||\mathbf{xp}_i||) \leq \exp(p \ln ||\mathbf{xp}_j||)$, we conclude that $||\mathbf{xp}_i|| \leq ||\mathbf{xp}_j||$ for any $p > 0$. Power diagrams are used in computational geometry to efficiently compute the union of disks or for the reconstruction of 3D shapes from point clouds.

WWW Additional source code or supplemental material is provided on the book's Web site:
www.charlesriver.com/Books/BookDetail.aspx?productID=117120
File: `DiscreteVoronoi.cpp`

Moreover, we can extend the partition principle of Voronoi diagrams to *higher-order Voronoi diagrams*. Figure 8.15 depicts some higher-order Voronoi diagrams. An order-k Voronoi diagram considers all k-tuples of a given point set $\mathcal{P}$ to partition the space into cells $V(\mathcal{T})$ such that:

$$|\mathcal{T}| = k,$$
$$V(\mathcal{T}) = \{\mathbf{x} \mid \forall \mathbf{t} \in \mathcal{T}\ \forall \mathbf{p} \in \mathcal{P} \backslash \mathcal{T},\ ||\mathbf{xt}|| \leq ||\mathbf{xp}||\}.$$

In other words, the Voronoi cell of k-tuple $\mathcal{T}$ is the locus of points closer to all points of $\mathcal{T}$ than to any other point of $\mathcal{P}\backslash\mathcal{T}$. Higher-order Voronoi diagrams may contain some cells empty of generators, as shown in Figure 8.15(a) and (b). The number of possible k-order Voronoi cells is naively upperbounded by $\binom{n}{k}$. Fortunately, all higher-order Voronoi diagrams for a given planar point set is much less. Namely, the complexity of all higher-order diagrams for a plane n-point set is upperbounded by $O(n^3)$. k-Order Voronoi diagrams find useful applications in proximity location problems. Indeed, finding the k nearest neighbors to any given query point $\mathbf{q}$ amounts to localize $\mathbf{q}$ in the k-order Voronoi diagram of $\mathcal{P}$.

Yet another useful type of Voronoi diagram is the *farthest point Voronoi diagram*, defined by reversing the distance order to define cells. The farthest point Voronoi diagram of a point set $\mathcal{P}$ partitions the space into farthest regions such that the Voronoi cell of $\mathbf{p}$ is defined as:

$$V(\mathbf{p}) = \{\mathbf{x} \mid ||\mathbf{xp}|| \geq ||\mathbf{xp}'||,\ \forall \mathbf{p}' \in \mathcal{P}\}. \tag{8.22}$$

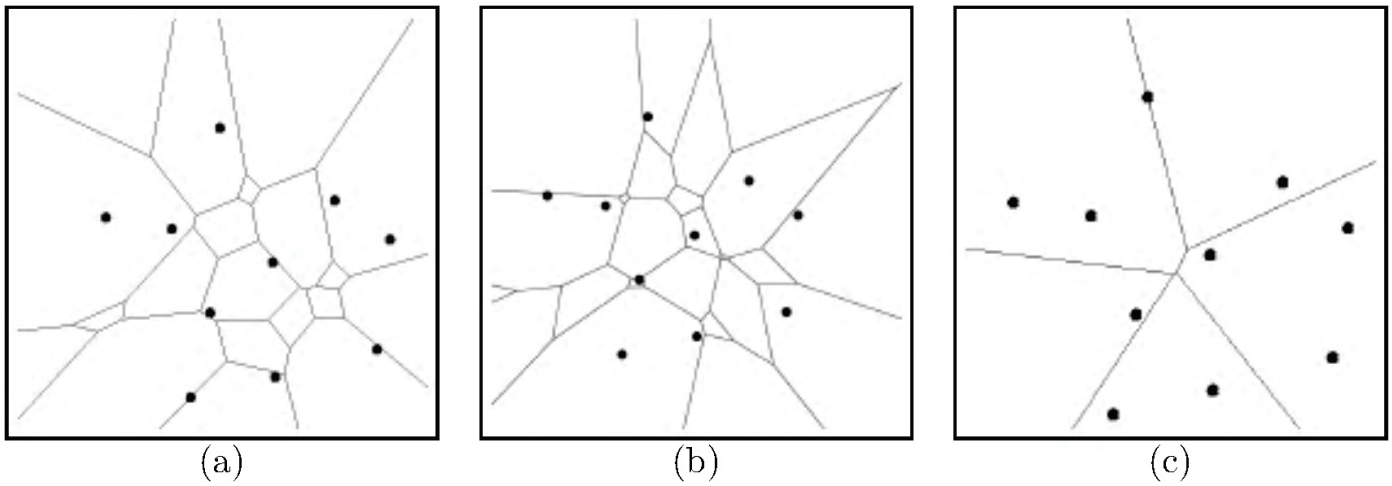

FIGURE 8.15 *Higher-order Voronoi diagrams. (a) order-2 Voronoi diagram. (b) order-3 Voronoi diagram. (c) farthest Voronoi diagram (order-$(n-1)$).*

Interestingly, observe that the set of points whose k nearest neighbors are k-tuple $\mathcal{T}$ is also the region whose $n-k$ farthest neighbors are the $(n-k)$-tuple $\mathcal{P}\backslash\mathcal{T}$. That is, an order-$k$ (nearest) Voronoi diagram is an order-$(n-k)$ farthest Voronoi diagram. In particular, we have the farthest Voronoi diagram that coincides exactly with the order-$(n-1)$ nearest Voronoi diagram. In summary, we have:

$$\text{Order-}k \text{ nearest Voronoi diagram} = \text{Order-}(n-k) \text{ farthest Voronoi diagram.} \quad (8.23)$$

8.3 Mathematical Techniques

8.3.1 Linearization

Linearization is a general mathematical technique for obtaining linear equations by adding extra slack variables. Thus, linearization increases the dimension of the parameter space. Linearization is also called *lifting*, as we embed our initial space into a higher-dimensional space. We'll illustrate this technique with a typical problem in computational geometry.

Application: Closest Pair of Points

The closest pair of points problem on the plane asks to compute the minimum interdistance of a set of distinct points. It is related to the broader all nearest neighbors problem that requires us to compute for each point $\mathbf{p}_i \in \mathcal{P} = \{\mathbf{p}_1, ..., \mathbf{p}_n\}$, its closest neighbor $N(\mathbf{p}_i)$. If there is more than one "closest" neighbor point, we arbitrary pick

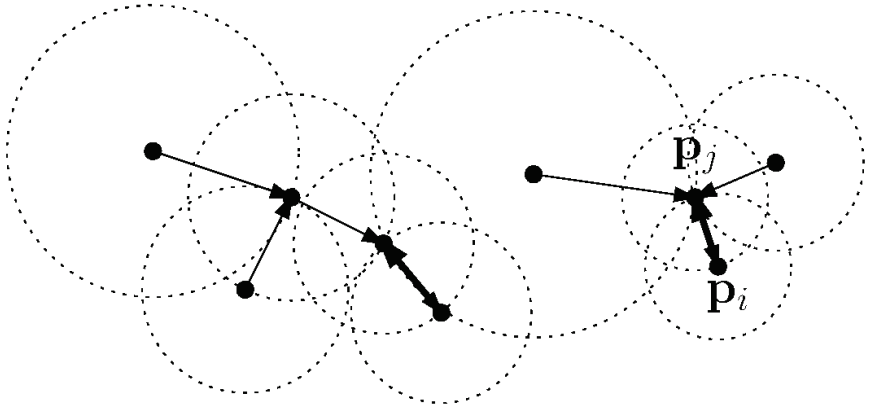

FIGURE 8.16 *All nearest neighbor graph. For each point, we draw the corresponding empty circle (dotted circles). The closest pair of points belongs among the double-oriented edges.*

one of the nearest points. Figure 8.16 depicts the geometric graph obtained by drawing all oriented edges $\mathbf{p}_i N(\mathbf{p}_i)$. Each pair $\mathbf{p}_i N(\mathbf{p}_i)$ defines a circle of circumcenter $\mathbf{p}_i$ and radius $||\mathbf{p}_i N(\mathbf{p}_i)||$, empty of other points of $\mathcal{P}$, except possibly at the boundary. In this figure, $\mathbf{p}_i$ and $\mathbf{p}_j$ are *mutual nearest neighbors*, and are visualized as such by a double arrow edge in the geometric graph. Clearly, the closest pair of points of $\mathcal{P}$ is found from the nearest neighbor graph, by simply inspecting pairs of mutually nearest neighbors (doubly oriented edges, as shown in Figure 8.16). How many and how easily can we check those mutually nearest neighbor pairs? First, let us prove that for any $\mathbf{p}_i$, $\mathbf{p}_i N(\mathbf{p}_i)$ is an edge of the Delaunay triangulation of $\mathcal{P}$. Consider the Voronoi diagram of $\mathcal{P}$ (see Section 5.2.3 and Section 8.2.2) and the Voronoi cells of sites $\mathbf{p}_i$ and $N(\mathbf{p}_i)$. Clearly, those Voronoi regions are adjacent and this means that the edge $[\mathbf{p}_i N(\mathbf{p}_i)]$ is necessarily a Delaunay edge (by taking the dual graph representation, as explained in Section 5.2.3). Figure 8.17 illustrates this property.

Thus, it is enough to check all Delaunay edges to find the closest pair of point set $\mathcal{P}$. Now, let's define a *lifting* to the 3D unit paraboloid of revolution to show that the Delaunay edges can be obtained from the edges of the convex hull of a "special" 3D point set. First, we'll write the equation of the empty circle C_i of center $\mathbf{p}_i = [x_i \; y_i]^T$ and radius $\mathbf{r}_i = ||\mathbf{p}_i N(\mathbf{p}_i)||$.

$$C_i : (x - x_i)^2 + (y - y_i)^2 = r_i^2. \tag{8.24}$$

We rewrite equivalently the equation as:

$$C_i : x^2 + y^2 = 2x_i x + 2y_i y + r_i^2 - (x_i^2 + y_i^2). \tag{8.25}$$

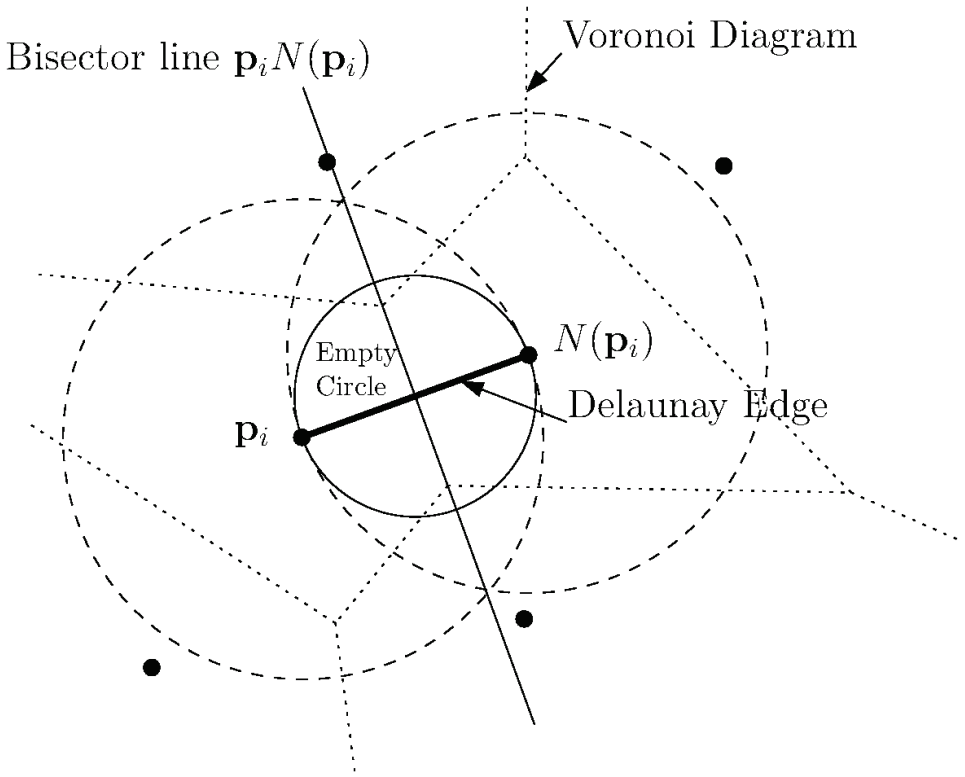

FIGURE 8.17 *Any mutual nearest neighbor pair of a finite point set is a Delaunay edge because their corresponding Voronoi cells are necessarily adjacent.*

Let us *linearize* the latter equation by introducing an extra slack variable:

$$z = x^2 + y^2. \tag{8.26}$$

Thus, we get:

$$C_i : \underbrace{z = 2x_i x + 2y_i y + r_i^2 - (x_i^2 + y_i^2)}_{\Pi_i}, \text{ with } z = x^2 + y^2. \tag{8.27}$$

Let z denote the vertical axis in the augmented 3D space. Such a linearization provides for each circle C_i a linear equation of a nonvertical 3D plane $\prod_i = \prod(C_i)$. The second equation $z = x^2 + y^2$, which represents the constraint related to the slack variable z, is *fixed* (independent of point set $\mathcal{P}$) and represents the *paraboloid of revolution* (a parabola curve $z = x^2$ that is rotated around the z-axis). Visually speaking, the circle C_i is interpreted as the vertical projection of the intersection of the plane $\prod_i$ with the paraboloid of revolution. In other words, the vertical lifting of a circle to the paraboloid yields an ellipsis supported by the plane $\prod_i$. Points are considered as circles of null radii.

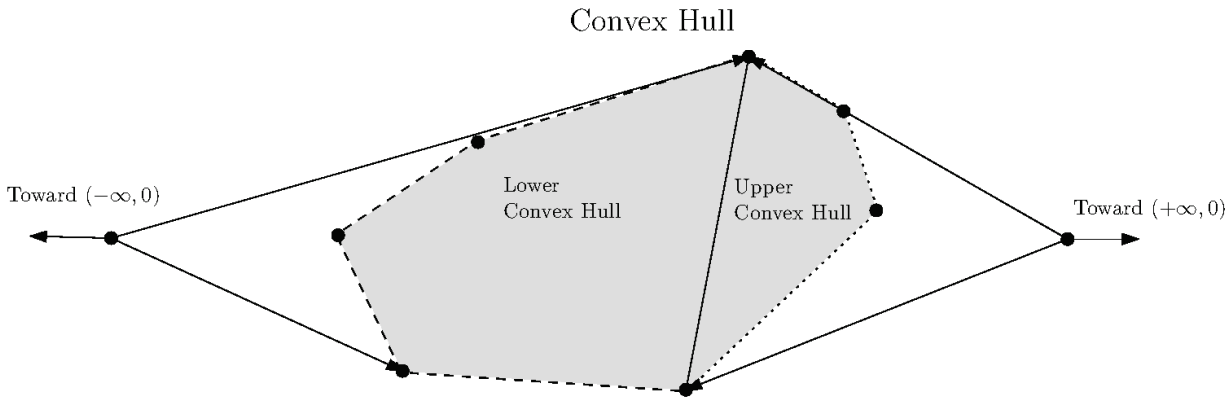

FIGURE 8.18 *Decomposing a convex hull in its "lower" and "upper" part.*

Let $f(x, y) = (x, y, x^2 + y^2)$ denote the lifting function from 2D to 3D, and $\mathcal{P}'$ represent the lifted point set $\mathcal{P}' = \{\mathbf{p}'_i = f(\mathbf{p_i}) \mid \mathbf{p_i} \in \mathcal{P}\}$. Now, consider any circle C of the plane, then C is an empty circle (no points in $\mathcal{P}$ in its interior) if, and only if, all points of $\mathcal{P}'$ are strictly above the 3D plane $\prod(C)$. Indeed, a point $\mathbf{p} = [x\ y]^T$ is outside a circle defined by its circumcenter $\mathbf{p}_i = [x_i\ y_i]^T$ and radius r_i, if and only if:

$$(x_i - x)^2 + (y_i - y)^2 - r_i^2 > 0. \tag{8.28}$$

That is,

$$z > 2x_i x + 2y_i y + r_i^2 - (x_i^2 + y_i^2). \tag{8.29}$$

This latter equation means that point $\mathbf{p}$ is above the 3D plane $\prod_i$ (see Eq. 8.27). Furthermore, consider a pair of points $\mathbf{p}_i$ and $\mathbf{p}_j$. This diametrically opposed pair of points defines an empty circle C if all lifted points of $\mathcal{P}$ are on or above the plane $\prod(C)$. Since $\prod(C)$ necessarily passes through $\mathbf{p}'_i$ and $\mathbf{p}'_j$, this means that $\mathbf{p}'_i$ and $\mathbf{p}'_j$ defines an edge of the lower convex hull of $\mathcal{P}'$. By definition, any 3D plane passing through $\mathbf{p}'_i$ and $\mathbf{p}'_j$ with all other points of $\mathcal{P}'$ above it is called a *supporting plane* of the convex hull. The *lower convex hull* is defined as the part of the convex hull visible from $(0, 0, -\infty)$. Another way to define the lower convex hull of a point set is to compute the convex hull of the point set to which we further add the extra point $(0, 0, +\infty)$. Similarly, the upper convex hull is defined as $\text{conv}(\mathcal{P} \cup \{(0, 0, -\infty)\})$ (see Figure 8.18). The convex hull is the intersection of the lower convex hull with the upper convex hull. Another related property is that a triangle $\triangle \mathbf{p}_i\mathbf{p}_j\mathbf{p}_k$ is a triangle of the Delaunay triangulation of $\mathcal{P}$ if, and only if, $\triangle \mathbf{p}'_i\mathbf{p}'_j\mathbf{p}'_k$ is a triangle (facet) of the lower convex hull of the corresponding lifted point set. Figure 8.19 illustrates the lifting process and those 2D/3D relationships.

Thus, by linearization, the algorithm for computing all the nearest neighbors becomes straightforward: Compute the lower convex hull of $\mathcal{P}'$. Then for each vertex $\mathbf{p_i}'$ inspect all adjacent edges of the lower convex hull, and associate to $\mathbf{p}_i$ its closest

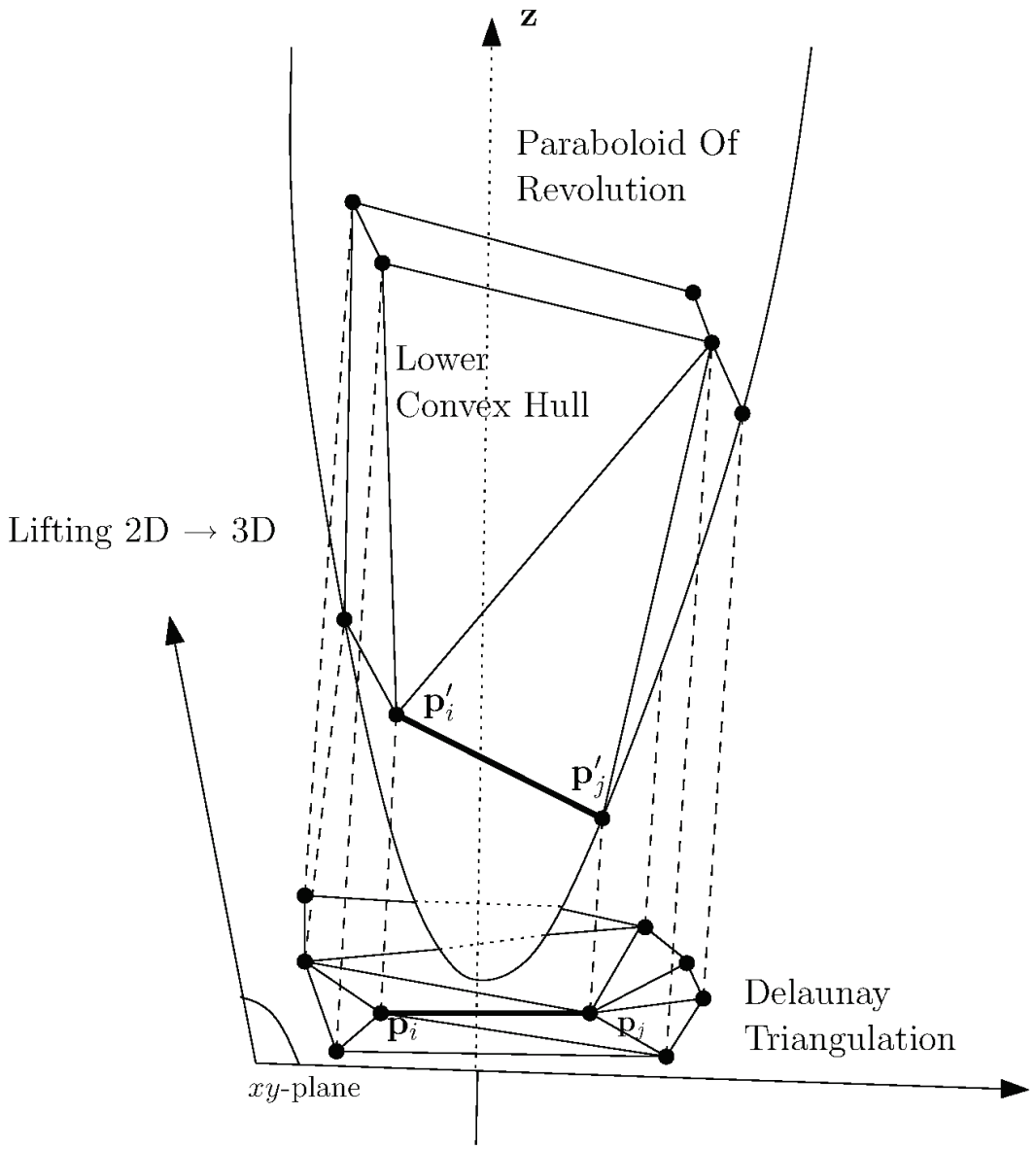

FIGURE 8.19 *Lifting a 2D point set onto the 3D paraboloid of revolution. Edges of the 2D Delaunay triangulation directly come from the xy-projection of the 3D edges of the lower convex hull.*

neighbor $N(\mathbf{p_i})$. This is enough, since the point that has the minimum distance from $\mathbf{p}_i$ is found among the edges of the lower convex hull departing from $\mathbf{p}'_i$.

Let us now analyze the time complexity: lifting requires linear time. Computing the convex hull of a 3D n-point set is done in $O(n \log n)$ time. Finally, traversing all the lower convex hull edges of all points of $\mathcal{P}'$ costs linear time (twice the number of edges). Therefore, we conclude that computing *all* the nearest neighbors of an n-point set is done in $O(n \log n)$ time.

We summarize the overall procedure in pseudocode below:

AllNearestNeighbors($\mathcal{P} = \{\mathbf{p_1}, ..., \mathbf{p_n}\}$)
1. ◁ Use linearization and 3D lower convex hull edges ▷
2. $\mathcal{P}'$=Lift all points of $\mathcal{P}$ to the paraboloid of revolution.
3. $\mathcal{CH}$ = LowerConvexHull3D($\mathcal{P}'$)
4. $\mathcal{E}$ = All edges of $\mathcal{CH}$ projected on the xy-plane
5. Traverse $\mathcal{E}$ to determine for each point its nearest neighbor

Although we considered the 2D to 3D lifting, this circle-to-plane lifting technique generalizes to *arbitrary dimension*. To conclude, let us mention that in Section 3.5.3, we have concisely described using the polarity that the convex hull problem is equivalent to the problem of computing a dual bounded intersection of half-spaces. Moreover, here we showed how to obtain the Voronoi diagram from the intersection of half-spaces via lifting onto the unit paraboloid. Finally, the Delaunay triangulation is obtained from the Voronoi diagram using the duality of their face incidence graph. Thus, a generic arbitary dimension convex hull algorithm allows us to compute Voronoi/Delaunay triangulations (see Bibliographical Notes).

Lifting for Designing Geometric Predicates

Let us first describe the orientation predicate for a sequence of points. In the plane, the orientation predicate Orient2D for a sequence of points $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{r}$ is defined as the sign of the following 3×3 determinant:

$$\text{Orient2D}(\mathbf{p}, \mathbf{q}, \mathbf{r}) = \text{sign}\det \begin{bmatrix} 1 & 1 & 1 \\ \mathbf{p} & \mathbf{q} & \mathbf{r} \end{bmatrix}, \tag{8.30}$$

which is mathematically rewritten after developing the determinant by the first row as the sign of a 2×2 determinant:

$$\text{Orient2D}(\mathbf{p}, \mathbf{q}, \mathbf{r}) = \text{sign}\det \begin{bmatrix} x_q - x_p & x_r - x_p \\ y_q - y_p & y_r - y_p \end{bmatrix}. \tag{8.31}$$

Geometrically, the absolute value of the determinant corresponds to twice the area of the triangle $\triangle\mathbf{pqr}$. The sign of the determinant indicates whether the points are oriented clockwise, counterclockwise, or aligned (Figure 8.20). Note that since $\det \mathbf{M} = \det \mathbf{M}^T$, we can rewrite the orientation predicate equivalently as follows:

$$\text{Orient2D}(\mathbf{p}, \mathbf{q}, \mathbf{r}) = \text{sign}\det \begin{bmatrix} x_q - x_p & y_q - y_p \\ x_r - x_p & y_r - y_p \end{bmatrix}. \tag{8.32}$$

Another interpretation of the determinant consists in embedding the 2D points $\mathbf{p}$, $\mathbf{q}$, and $\mathbf{r}$ in 3D, by adding an extra coordinate z set to zero. Let $\mathbf{p}'$, $\mathbf{q}'$, and $\mathbf{r}'$ be those

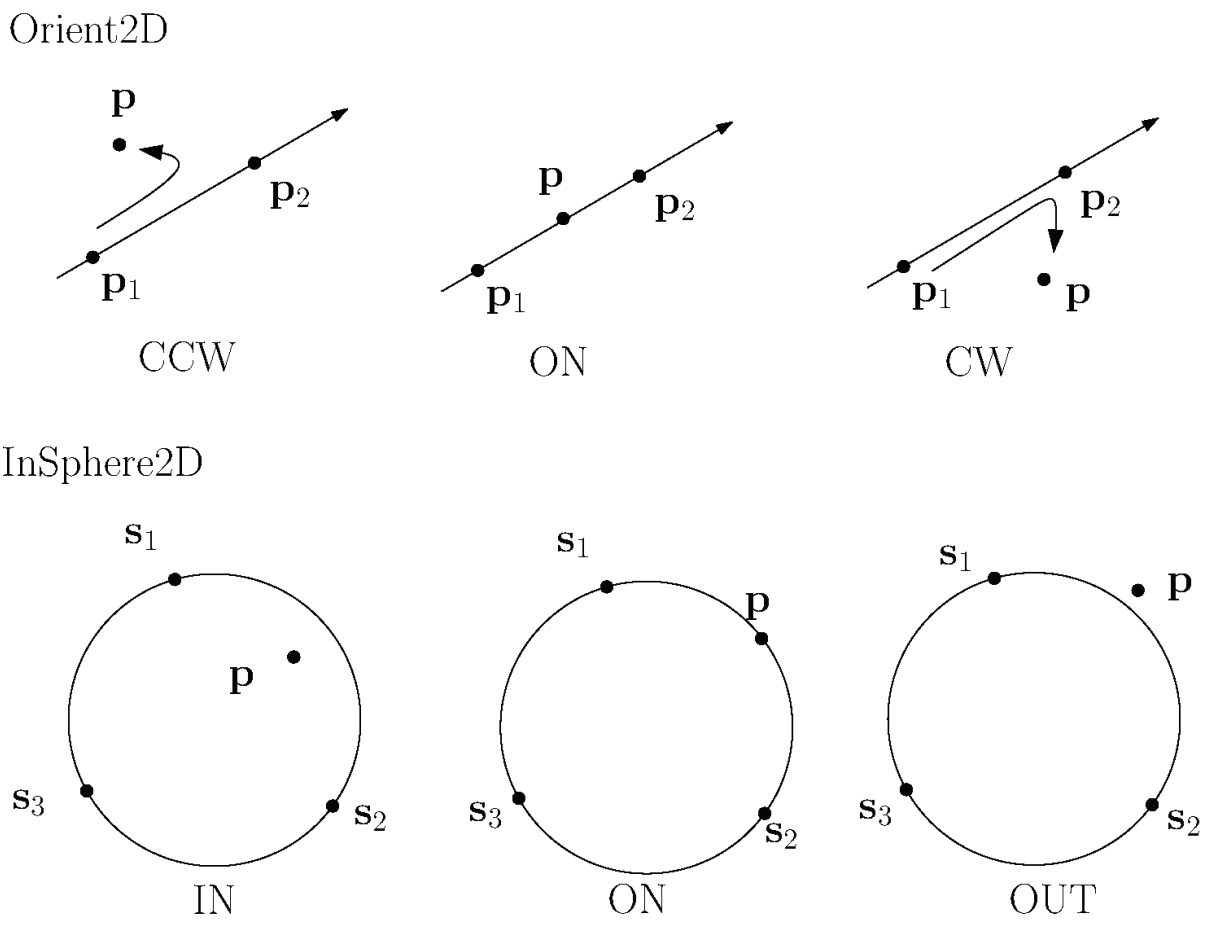

FIGURE 8.20 *Standard geometric predicates.* `OrientdD` *and* `InSpheredD` *illustrated on the plane.*

3D points lying on the 2D xy-plane: $z = 0$. Then, the determinant value corresponds to the z-coordinate of the cross product vector $\mathbf{n}$:

$$\mathbf{n} = (\mathbf{p}' - \mathbf{q}') \times (\mathbf{p}' - \mathbf{r}'). \tag{8.33}$$

Those $\mathbf{p}' - \mathbf{q}'$, $\mathbf{p}' - \mathbf{r}'$, and $\mathbf{n}$ vectors define either a left-handed or right-handed 3D coordinate system (see Section 3.2.2) according to the sign of the determinant of Equation 8.31.

WWW	Additional source code or supplemental material is provided on the book's Web site: *www.charlesriver.com/Books/BookDetail.aspx?productID=117120* File: `Orient2D` (demo interface in OpenGL)

In C++, such an orientation predicate can be implemented as follows:

```
1 // Orientation test: 2x2 determinant sign
2 // This is not the best bound but enough for this screen demo.
3 #define ERR_THRESHOLD 1.0e-6
4
5 int Orient2D(const Point2D& p, const Point2D& q, const Point2D& r)
6   {
7     if ((q.x-p.x)*(r.y-p.y) > (r.x-p.x)*(q.y-p.y)+ERR_THRESHOLD)
          return CCW;
8     if ((q.x-p.x)*(r.y-p.y) < (r.x-p.x)*(q.y-p.y)-ERR_THRESHOLD)
          return CW;
9    return ON;
10  }
```

In 3D, Orient3D is defined similarly as the sign of the following 4×4 determinant:

$$\text{Orient3D}(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{s}) = \text{sign}\det \begin{bmatrix} 1 & 1 & 1 & 1 \\ \mathbf{p} & \mathbf{q} & \mathbf{r} & \mathbf{s} \end{bmatrix}. \tag{8.34}$$

Note that changing the row position of the vector whose coordinates are all 1 may reverse the sign of the determinant (the sign depends on the row parity where $[1\ 1\ 1\ 1]^T$ is located).

A geometric interpretation of the determinant using the cross product and the dot product is given as:

$$\text{Orient3D}(\mathbf{p}, \mathbf{q}, \mathbf{r}, \mathbf{s}) = ((\mathbf{q} - \mathbf{p}) \times (\mathbf{r} - \mathbf{p})) \cdot (\mathbf{s} - \mathbf{p}). \tag{8.35}$$

Orientation tests are thus expressed by the signs of determinants. These determinants amount to compute the volume of simplices. Simplices are convex bodies defined by the convex hull of *affinely independent* vertices. A k-dimensional simplex, defined by $k+1$ affinely independent point, has $\frac{(k+1)k}{2}$ edges, $\frac{(k+1)k(k-1)}{3!}$ 2-faces, ..., and $k+1$ facets.[4] To test whether a simplex is degenerated or not, we need to check whether its volume is zero or not. In general, the volume of a k-dimensional simplex S_k sitting in $\mathbb{R}^d$ can be computed in several ways depending on whether we use vertex coordinates or edge lengths:

Gram determinant. Let $\mathbf{p}_1, ..., \mathbf{p}_{k+1}$ be the $k+1$ vertices of simplex S_k. Further, let $\mathbf{p}'_i = \mathbf{p}_i - \mathbf{p}_1$ be the k associated vectors, with $1 < i \leq k+1$. Denote by $\mathbf{D}$ the $k \times d$ matrix whose rows are the vectors $\mathbf{p}'_i$ $(i > 1)$. Then, the volume of simplex S_k is given as:

$$\text{Volume}(S_k) = \frac{\sqrt{|\det \mathbf{D}\mathbf{D}^T|}}{k!}. \tag{8.36}$$

[4] A facet is by definition a $(k-1)$-dimensional face.

Cayley-Menger determinant. If only edge lengths $d_{i,j} = ||\mathbf{p}_i\mathbf{p}_j||$ are known (but not the vertex coordinates), consider the following $(k+2) \times (k+2)$ matrix $\mathbf{D}$:

$$\mathbf{D} = \begin{bmatrix} 0 & 1 & 1 & \dots & 1 \\ 1 & 0 & d_{1,2}^2 & \dots & d_{1,k+1}^2 \\ \vdots & & \dots & \dots & \dots \\ 1 & d_{k+1,1}^2 & d_{k+1,2}^2 & \dots & 0 \end{bmatrix}. \tag{8.37}$$

Then, the volume of simplex S_k is given as:

$$\text{Volume}(S_k) = \frac{\sqrt{|\det \mathbf{D}|}}{2^{\frac{k}{2}} k!}. \tag{8.38}$$

This Cayley-Menger determinant generalizes the Heron's formula which computes the area of a triangle using its edge lengths (see Section 9.2).

There is also a recent[5] determinant formula (due to Klebaner, Sudbury, and Watterson in 1988) which computes the volume of d-dimensional simplices given as the intersection of $d+1$ halfspace equations.

We are now ready to describe how to compute another fundamental geometric predicate using a lifting procedure. In dimension d, consider the sphere defined by (at most) $d+1$ points on its boundary: $\mathbf{s}_1, ..., \mathbf{s}_{d+1}$. We often need to answer whether a query point $\mathbf{p}$ is inside, on, or outside the ball defined by the basis points $\mathbf{s}_1, ...\mathbf{s}_{d+1}$. That is, we need to return the sign of $||\mathbf{pc}|| - r$, where $\mathbf{c}$ is the circumcenter of the sphere defined by $\mathbf{s}_1, ..., \mathbf{s}_{d+1}$, and r its radius. In computational geometry, this is known as the d-dimensional predicate InSpheredD (Figure 8.20). Using the linearization technique, the sign of $||\mathbf{pc}|| - r$ amounts to finding the sign of a particular determinant. The idea is that $d+1$ points are coplanar in dimension d if their corresponding matrix determinant is zero (degenerated matrix said rank deficient). That is, we would like to report the mutual position of a point $\mathbf{p}$ with d other points $\mathbf{p}_1, ..., \mathbf{p}_d$ defining an hyperplane. We'll call this orientation predicate OrientdD. We write predicate OrientdD using the sign of the following determinant:

$$\text{OrientdD}(\mathbf{p}_1, ..., \mathbf{p}_d, \mathbf{p}) = \text{sign}\det \begin{bmatrix} \mathbf{p}_1^T & 1 \\ \mathbf{p}_2^T & 1 \\ \vdots & 1 \\ \mathbf{p}_d^T & 1 \\ \mathbf{p}^T & 1 \end{bmatrix}. \tag{8.39}$$

[5]Readers interested in those results should also check the Brunn-Minkowski's theory of mixed volume.

Note that the equation of the hyperplane H passing through d points $\mathbf{p}_1, ..., \mathbf{p}_d$ is found as:

$$H : \text{OrientdD}(\mathbf{p}_1, ..., \mathbf{p}_d, \mathbf{x}) = 0. \tag{8.40}$$

This latter determinant can be simplified further by lowering the matrix size by one dimension, as follows (see Eq. 8.32):

$$\text{OrientdD}(\mathbf{p}_1, ..., \mathbf{p}_d, \mathbf{p}) = \text{sign}\det \begin{bmatrix} (\mathbf{p}_1 - \mathbf{p})^T \\ (\mathbf{p}_2 - \mathbf{p})^T \\ \vdots \\ (\mathbf{p}_d - \mathbf{p})^T \end{bmatrix}. \tag{8.41}$$

Thus, the strategy for defining InSpheredD consists in first lifting both the sphere basis points $\mathbf{s}_i \rightarrow \mathbf{s}'_i$ and the query point $\mathbf{p}$ to the paraboloid of revolution. Let InSphere_d and Orient_d denote these generic predicates in arbitray dimension d. Then, we check for the sign of the following determinant:

$$\text{InSphere}_d(\mathbf{s}_1, ..., \mathbf{s}_{d+1}, \mathbf{p}) = \text{Orient}_{d+1}(\mathbf{s}'_1, ..., \mathbf{s}'_{d+1}, \mathbf{p}'). \tag{8.42}$$

Thus, it comes that:

$$\text{InSpheredD}(\mathbf{s}_1, ..., \mathbf{s}_{d+1}, \mathbf{p}) = \text{sign}\det \begin{bmatrix} \mathbf{s}'^T_1 & 1 \\ \mathbf{s}'^T_2 & 1 \\ \vdots & 1 \\ \mathbf{s}'^T_{d+1} & 1 \\ \mathbf{p}'^T & 1 \end{bmatrix}. \tag{8.43}$$

Using the fact that:

$$x_{d+1}(\mathbf{s}) = \sum_{i=1}^{d} x_i(\mathbf{s})^2 = \mathbf{s} \cdot \mathbf{s}, \tag{8.44}$$

we rewrite this determinant as:

$$\text{InSpheredD}(\mathbf{s}_1, ..., \mathbf{s}_{d+1}, \mathbf{p}) = \text{sign}\det \begin{bmatrix} \mathbf{s}^T_1 & \mathbf{s}_1 \cdot \mathbf{s}_1 & 1 \\ \mathbf{s}^T_2 & \mathbf{s}_2 \cdot \mathbf{s}_2 & 1 \\ \vdots & \vdots & 1 \\ \mathbf{s}^T_{d+1} & \mathbf{s}_{d+1} \cdot \mathbf{s}_{d+1} & 1 \\ \mathbf{p}^T & \mathbf{p} \cdot \mathbf{p} & 1 \end{bmatrix}. \tag{8.45}$$

8.3.2 Approximating Distances in Large Dimensions

When the dimension becomes large, computing the Euclidean distance between two points becomes an expensive operation in itself. Indeed, the distance of two points $\mathbf{p}$ and $\mathbf{q}$ of the d-dimensional Euclidean space is defined as:

$$d_2(\mathbf{p}, \mathbf{q}) = ||\mathbf{pq}|| = \sqrt{\sum_{i=1}^{d} (p_i - q_i)^2}. \tag{8.46}$$

Thus, computing the distance between two points costs $O(d)$ time. The distance can be conveniently rewritten mathematically using the dot product of vectors $\mathbf{p}$ and $\mathbf{q}$, as follows:

$$||\mathbf{pq}|| = \sqrt{(\sum_{i=1}^{d} p_i^2) + (\sum_{i=1}^{d} q_i^2) - 2 \sum_{i=1}^{d} p_i q_i}, \tag{8.47}$$

$$||\mathbf{pq}|| = \sqrt{||\mathbf{p}||^2 + ||\mathbf{q}||^2 - 2\ \mathbf{p} \cdot \mathbf{q}}. \tag{8.48}$$

Now, using the Cauchy-Schwarz inequality, which states that the absolute value of the dot product of vectors is less than or equal to the product of its norms,

$$|\mathbf{p} \cdot \mathbf{q}| \leq ||\mathbf{p}||\ ||\mathbf{q}||, \tag{8.49}$$

we obtain the following inequality:

$$||\mathbf{p}||^2 + ||\mathbf{q}||^2 - 2\ ||\mathbf{p}||\ ||\mathbf{q}|| \leq ||\mathbf{pq}||^2 \leq ||\mathbf{p}||^2 + ||\mathbf{q}||^2 + 2\ ||\mathbf{p}||\ ||\mathbf{q}||. \tag{8.50}$$

Suppose $||\mathbf{p}||$ and $||\mathbf{q}||$ are already computed, then we get in constant time (independent of the dimension) a lower and an upper bound estimate on the distance $||\mathbf{pq}||$, using the inequalities of Eq. 8.50.

Let's consider a particular case of clustering: the smallest enclosing ball of a high-dimensional point set $\mathcal{P} = \{\mathbf{p}_1, ...\mathbf{p}_n\}$. The smallest enclosing ball is also called the *Euclidean 1-center* in the operations research community. We have already presented some algorithms for small dimensions, in Section 7.5. Here, we present a simple guaranteed approximation algorithm that uses Cauchy-Schwarz filtering: we compute a $(1+\epsilon)$-approximation of the minimum enclosing ball. That is, an enclosing ball with radius $r \leq (1+\epsilon)r^*$, where r^* is the radius of the smallest[6] enclosing ball.

First, we initialize the circumenter $\mathbf{c}_1$ to any arbitrary input point of $\mathcal{P}$, say $\mathbf{p}_1$. The approximation algorithm proceeds iteratively. At the ith iteration, given the current

[6]The smallest enclosing ball is unique (proof by contradiction).

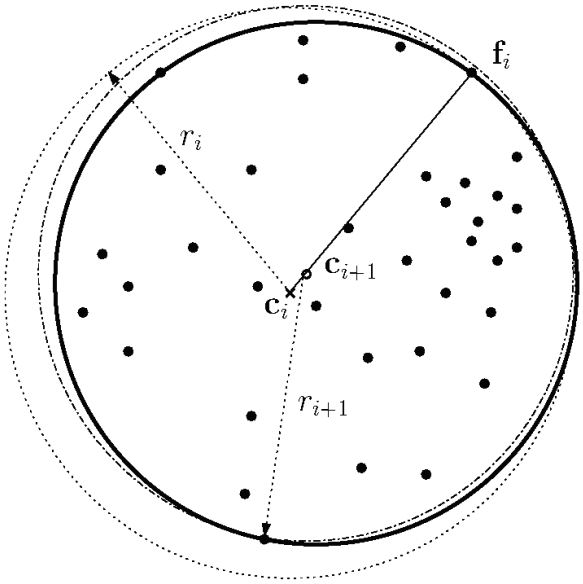

FIGURE 8.21 *Approximating the smallest enclosing ball of a 2D planar point set using a local iterative updating of circumcenters. The drawing explains the configuration at the i-th iteration. The thick circle depicts the smallest enclosing ball.*

circumcenter $\mathbf{c}_i$, we seek the *farthest* point of $\mathcal{P}$. Let $\mathbf{f}_i \in \mathcal{P}$ be that point (in case of ties, again we choose any arbitary one). We then update the current circumcenter $\mathbf{c}_i$ in direction of $\mathbf{f}_i$ using a harmonic weighting rule:

$$\mathbf{c}_{i+1} = \mathbf{c}_i + \frac{1}{i+1}\mathbf{c}_i\mathbf{f}_i. \tag{8.51}$$

Figure 8.21 illustrates those operations between two successive iterations. The beauty of this approximation algorithm is that the algorithm computes a $(1+\epsilon)$-approximation of the smallest enclosing ball after at most $\frac{1}{\epsilon^2}$ iterations.

We summarize the smallest enclosing ball approximation algorithm in pseudocode:

SMALLENCLOSINGBALL($\mathbf{p_1}, ..., \mathbf{p_n}, \epsilon$)
1. ◁ Compute a $(1+\epsilon)$-approximation of the smallest enclosing ball ▷
2. ◁ Return the circumcenter of a small enclosing ball ▷
3. $\mathbf{c} \leftarrow \mathbf{p_1}$
4. **for** $i \leftarrow 1$ **to** $\lceil \frac{1}{\epsilon^2} \rceil$
5. **do** ◁ Furthest point is $\mathbf{f}_i = \mathbf{p}_j$ ▷
6. $j = \text{argmax}_{i=1}^{n} ||\mathbf{cp_i}||$
7. $\mathbf{c} \leftarrow \mathbf{c} + \frac{1}{i+1}\mathbf{cp}_j$
8. **return c**

A straightforward implementation in C++ follows:

```
1 void SmallEnclosingBall(Point *set, int n, Point &center, double &
      radius, double epsilon)
2 {
3 int i,iter, nbiter, winner;
4 double dist;
5
6 nbiter=(int)ceil(1.0/(epsilon*epsilon));
7 center=set[0];
8
9 for(iter=0;iter<nbiter;iter++)
10   {
11   winner=0;
12   radius=Distance(center,set[0]);
13
14   // Farthest point query
15   for(i=1;i<n;i++)
16     {
17     dist=Distance(center,set[i]);
18     if (dist>radius) {winner=i;radius=dist;}
19   }
20
21   // Update circumcenter
22   for(i=0;i<DIMENSION;i++)
23   {center.coord[i]+=(1.0/(iter+1.0))*(set[winner].coord[i]-center.
         coord[i]);}
24   }
25 }
```

This heuristic unfortunately does not scale well with ϵ, but is useful for, say $\epsilon \geq 0.01$ (at most 10,000 iterations for guaranteeing an enclosing ball with radius within 1% of the optimal ball). A straightforward implementation of the algorithm yields running time $O(\frac{dn}{\epsilon^2})$. Computing the farthest points in high dimensions is clearly the bottleneck of that method. However, all over the iterations, our input point set $\mathcal{P}$ is fixed. At each iteration, we query for the farthest point of the current approximate circumcenter. Therefore, we preprocess $\mathcal{P}$ by computing all distances to the origin $||\mathbf{p}_i||$, for a preprocessing total cost $O(dn)$. Then, at a given iteration, we compute the current circumcenter point norm $||\mathbf{c}||$ (in $O(d)$ time), and look for the farthest point of $\mathcal{P}$ from $\mathbf{c}$ using the Cauchy-Schwarz-based filtering inequality, described in Eq. 8.50. Assume that using the Cauchy-Schwarz filtering inequality, we skip α percent (normalized to a ratio: $0 \leq \alpha \leq 1$) of distance computations, then the running time of this simple approximation algorithm becomes $O(\frac{1}{\epsilon^2}(\alpha n + (1-\alpha)dn + d))$. For uniform or Gaussian distributions, we *empirically* observe that $\alpha \rightarrow 1$. In those cases, the complexity of the algorithm becomes a much faster $O(\frac{1}{\epsilon^2}(n+d))$.

The pseudocode below gives details on implementing the search for a farthest point using the Cauchy-Schwarz upper bound:

```
FarthestPoint(P = {p_1, ..., p_n}, normP, c)
 1. ◁ Find the farthest point of P to c using Cauchy-Schwarz filtering ▷
 2. ◁ normP: array of norms (normP[i] = ||p_i||) ▷
 3. ◁ A better implementation should consider squared distances ▷
 4. ◁ c: current circumcenter for which we seek the farthest point ▷
 5. maxdist = 0.0
 6. ◁ j: index of the furthest point of P (argmax) ▷
 7. j = −1
 8. ◁ Calculate ||c|| ▷
 9. normc = Distance(c, o)
10. for i ← 1 to |P|
11.     do ◁ Filtering distance computations ▷
12.        dist = sqrt(normP[i]^2 + normc^2 + 2 × normP[i] × normc)
13.        if dist > maxdist
14.           then ◁ Compute the distance in linear time ▷
15.                pc = Distance(p[i], c)
16.                if pc > maxdist
17.                   then maxdist = pc
18.                        j = i
19. return j
```


Additional source code or supplemental material is provided on the book's Web site:
www.charlesriver.com/Books/BookDetail.aspx?productID=117120
File `smallenclosingball.cpp`

8.4 Bibliographical Notes

The seminal paper on 2D texture synthesis is due to Efros and Leung [109]. This synthesis algorithm was further sped up by Wei and Levoy [339]. The texture synthesis method was later extended to arbitrary manifolds [340], and solid textures [338].

The kD-Tree data structure was invented by Bentley [29] in 1975, as a major improvement of the quadtree search structure. Arya and Mount [13] and Arya and Fu [12] developed tailored proximity data structures for answering approximate nearest neighbors in high dimensions. They provide a C++ package, available online

at *http://www.cs.umd.edu/~mount/ANN/*. There is also another software package called RANGER that can be downloaded at *http://www.cs.sunysb.edu/~algorith/implement/ranger/implement.shtml*. Alternatives to the kD-trees for nearest neighbor queries include Orchard's algorithm [252] and its variants (annulus [168], double annulus [175]), and the principal component partitioning [354] (PCP). Yet another recent method is the vantage point tree (VP-Tree) proposed by Yianilos [351, 352]. VP-Trees use a partitioning scheme based on relative distances and are designed for handling multidimensional nearest neighbor queries. The range tree data structure is described in most computational geometry textbooks [271, 91]. Historically, in the 1970s, orthogonal range searching was a hot topic. This explains why range trees have been independently discovered by several researchers [30, 194, 215]. Using a technique called fractional cascading [76, 76], range tree queries can be further sped up by using additional pointers. A rectangular query is performed in $O(k + \log^{d-1} n)$ time in dimension d, saving a logarithmic factor over the traditional structure.

Nearest neighbor queries are also used in many other applications of computer graphics, such as image analogies [164], or geometry synthesis [34]. Alternative methods of texture synthesis consist in patch-based graph cuts [188] or fast tiling [83]. Recently, Zelinka and Garland [355, 356] developed a fast interactive texture synthesis method based on preprocessing efficiently the source textures into so-called jump maps. Their seminal method was later extended to texture arbitrary surface manifolds. Finally, the system of Owada et al. [258] demonstrates cutable objects by synthesizing on-the-fly 2D distorted textures on cross sections.

Clustering is an essential technique in machine learning and data mining. The seminal kMeans algorithm dates back to 1957 by a technical report by Lloyd [206]. kMeans is by essence a local iterative methodology that is therefore trapped into local minima. The paper by Kanungo et al. [180] showed that a local heuristic guarantees a global clustering. Kumar et al. [187] present a fast linear-time $(1 + \epsilon)$-approximation algorithm for the kMeans algorithm. Their random sampling-based method work in linear $O_{\epsilon,k}(dn)$ time and can handle outliers. Unfortunately the constant hidden in the big-Oh notation is exponential in $\frac{k}{\epsilon}$. Har-Peled and Sadri [157] further present results on the polynomial convergence of kMeans and describe two simple variants with guaranteed performance. Their convergence bounds depend on the *spread* of a point set, where the spread is defined as the ratio of the maximum interdistance over the minimum interdistance. The soft clustering performed by the expectation maximization was first thoroughly analyzed by Dempster et al. [95]. Another recent center-based clustering algorithm is the k-Harmonic means that uses the harmonic averages of the distances from each data point to the centers as its performance function [358, 359]. The advantage of k-Harmonic means compared to traditional kMeans is that it is experimentally observed less sensitive to the cluster initialization. Furthermore, k-Harmonic means uses a dynamic weighting technique [357] of the

data to escape some local optima. This weighting technique is further investigated by Nock and Nielsen [250], proving some analogy with a well-known algorithm in computational learning: boosting [287]. Recently, there has been an abstraction of kMeans-type/EM-type of algorithms using the concept of Bregman divergences [22, 250].

Burkardt gives an implementation of the centroidal Voronoi diagram, available online at *http://www.csit.fsu.edu/~burkardt/*. Ju et al. [176] provides a fast probabilistic method to implement in parallel the centroidal Voronoi diagram of a point set. Clustering is often used in visual computing. For example, the paper by Cohen-Steiner et al. [84] presents a mesh simplification method based on a centroidal Voronoi clutering method.

Yao and Yao showed how the linearization techniques [350] help in designing geometric searching data structures. They show how any algebraic range query can be linearized to halfspace range searching. Mehlhorn et al. [223] describe a geometric kernel on rational coordinates that includes a robust implementation of high-dimensional orientation predicates. The closest-pair problem of an n-point set was first solved in $O(n \log n)$-time by Hoey and Shamos [299]. In arbitary fixed dimension, the closest pair can be computed in $O(n \log n)$ time by a seminal algorithm by Bentley and Shamos [32]. Golin et al. [136] further proved that the use of the floor function with randomization allows us to find the closest pair within expected linear time. We presented a simple lifting method that required us to compute the convex hull [270] of 3D points in $O(n \log n)$ time. The `Qhull` library (*http://www.qhull.org/*) allows us to robustly compute convex hulls in arbitary dimension, and hence Voronoi/Delaunay structures. Lifting is often used in computational geometry. For example, the convex hull of balls in $\mathbb{R}^d$ can be computed from the convex hull of lifted points in $\mathbb{R}^{d+1}$ [41]. The union of n balls of $\mathbb{R}^d$ can be computed from the intersection of a polytope of $\mathbb{R}^{d+1}$ with the paraboloid of revolution of $\mathbb{R}^{d+1}$ [105]. In $\mathbb{R}^3$, the complexity of the union of unit balls can be quadratic, even if all unit balls contain the origin [58]. This result contrasts with the fact that the combinatorial complexity of the intersection of unit balls of $\mathbb{R}^3$ is linear. The power diagram [16] of spheres of $\mathbb{R}^d$ is a cell complex of $\mathbb{R}^{d+1}$. Similarly, the multiplicative weighted Voronoi diagram [101] of a point set (a Voronoi diagram with a non-Euclidean metrics) in $\mathbb{R}^d$ can be solved by embedding the d-dimensional space into $\mathbb{R}^{d+2}$. More details on the unit paraboloid lifting is found in the book by Edelsbrunner [104]. Reitsma et al. [279] investigate the use of multiplicatively weighted Voronoi diagram for information visualization tasks. Namely, they define the inverse weighted Voronoi diagram as the following reconstruction problem: given relative area sizes, find a corresponding weight distribution of generators whose regionalization best approaches the given relative areas.

Computing the smallest enclosing ball in high dimensions is not known to be

strongly polynomial. However, there are practical heuristics that seem to solve the exact smallest enclosing ball (of points) in very large dimensions ($d > 10,000$). The two-line approximation code of the smallest enclosing ball in arbitrary dimension was reported in the paper by Bădoiu and Clarkson [62]. Combining this simple iterative approximation algorithm with distance filtering using Cauchy-Schwarz inequalities is presented in [248].