Chapter 7.

Randomization

In this chapter, we cover the different aspects of randomization: randomized analysis of randomized algorithms, randomization for sampling sequences, randomized data structures, and randomized optimization algorithms. The leitmotif is that randomization helps us design elegant and simple computational techniques for various tasks. In a sense, randomization simplifies sometimes drastically complex deterministic procedures.

7.1 Randomized Analysis of QuickSort

To get a quick overview of the randomization paradigm, we'll present the well-known *QuickSort* algorithm. The QuickSort algorithm is a *comparison-based sorting algorithm*, invented in 1961 by Hoare. QuickSort has worst-case running time quadratic, but performs in practice in optimal $\Theta(n \log n)$ time (to be defined later on in the chapter), for sorting a sequence of n elements. Even today, QuickSort is one of the fastest known sorting algorithms. A standard implementation can be invoked in C using the procedure `qsort` (described in the `<stdlib.h>` header file). The basic strategy of Quicksort is to partition an n-element array $\mathbf{S}$ into three subarrays, $\mathbf{S}_<$, $\mathbf{S}_=$, and $\mathbf{S}_>$, according to a *pivot* key $\mathbf{S}[k]$, as follows:

$$\begin{aligned}
\mathbf{S}_= &= \{e \in \mathbf{S} \mid e = \mathbf{S}[k]\},\\
\mathbf{S}_< &= \{e \in \mathbf{S} \mid e < \mathbf{S}[k]\},\\
\mathbf{S}_> &= \{e \in \mathbf{S} \mid e > \mathbf{S}[k]\}.
\end{aligned}$$

That is $\mathbf{S}_=$ is the subset of elements of $\mathbf{S}$ equal to $\mathbf{S}[k]$, $\mathbf{S}_<$ the subset of $\mathbf{S}$ of elements strictly inferior to $\mathbf{S}[k]$, and $\mathbf{S}_>$ the subset of elements of $\mathbf{S}$ strictly superior

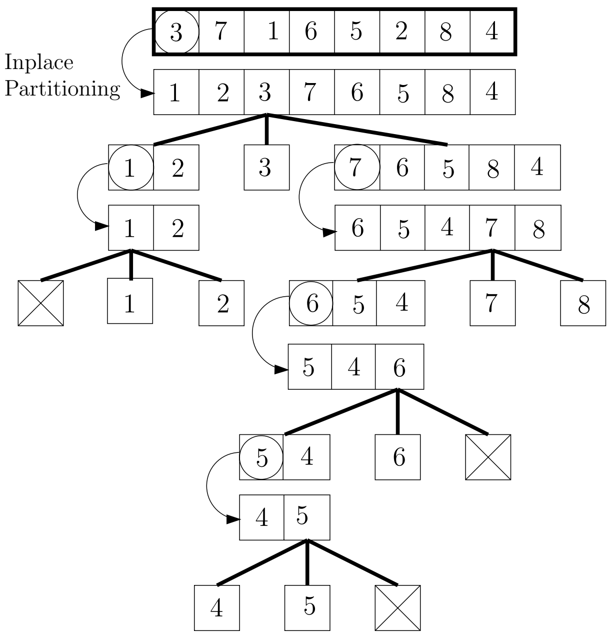


FIGURE 7.1 *Illustration of the recursive inplace partitioning principle of QuickSort.*

to $\mathbf{S}[k]$. Note that because all elements of $\mathbf{S}_=$ are identical to $\mathbf{S}[k]$, they do not need to be sorted. QuickSort sorts by calling recursively itself on strictly smaller size arrays. Namely, the recursion is called on the not yet sorted subarrays $\mathbf{S}_<$ and $\mathbf{S}_>$, that both satisfy $|\mathbf{S}_<| < n$ and $|\mathbf{S}_>| < n$. The recursion ends whenever array arguments are of unit size. In those cases, we do not need either to sort single-element arrays. That is, QuickSort sorts by recursively *partitioning* arrays. Figure 7.1 illustrates the tree structure obtained by the recursive calls of QuickSort on an 8-element integer array. We chose the pivot to be the first element of the array.

We'll analyze the time complexity of QuickSort. In a *best scenario*, each pivot $\mathbf{S}[k] \in \mathbf{S}$ ($k \in \{1, ..., n\}$) chosen by QuickSort splits the array in half, so that the time complexity $t(n)$ is given by the following recurrence equation:

$$t(n) = \underbrace{2t\left(\frac{n}{2}\right)}_{\text{Recursion}} + \underbrace{O(n)}_{\text{Cost for partitioning}} . \tag{7.1}$$

In the former equation, recall that $O(n)$ means that the running time for partitioning an array of n elements can be upper bounded by an, for some constant $a \in \mathbb{R}$, for all $n \geq n_0$, where n_0 is another constant. Solving this recurrence equation yields $t(n) = O(n \log n)$. We sketch the proof.

Proof. Let an denote the cost of partitioning an n-element array (using a pivot element $\mathbf{S}[k]$), and assume $t(k) \leq bk \log k$ for some constant b, and any $k < n$. The proof uses induction on n. Using the recurrence of Eq. 7.1, we have $t(n) = 2b\frac{n}{2} \log \frac{n}{2} + an$. That is, $t(n) = bn \log n + n(a - b)$. Thus, provided that $b \geq a$, we get $t(n) \leq bn \log n$. QED. □

However, in a *worst-case scenario*, each pivot of QuickSort is an extremal element (that is either $\mathbf{S}_<$ or $\mathbf{S}_>$ is empty). In such a case, the recurrence equation becomes:

$$t(n) = t(n-1) + O(n), \tag{7.2}$$

which solves as $t(n) = O(n^2)$, quadratic time.

Proof. Again, the proof proceeds by induction. Assume we have shown $t(k) \leq bk^2$ for $k < n$ and for some constant b (though different value from the previous proof). Using the recurrence formula Eq. 7.2, we get $t(n) = b(n-1)^2 + an = bn^2 + b + n(a - 2b)$. That is, $t(n) = bn^2 + n(a - 2b + \frac{b}{n}) \leq bn^2 + n(a - 2b + b) \leq bn^2$, provided that $b \geq a$. QED. □

The interesting analysis is when we ensure that at each step of the array partition, we end up with an α-partition. By definition, an *α-partition* is an array partition such that $\min\{|\mathbf{S}_<|, |\mathbf{S}_>|\} \leq \alpha|\mathbf{S}|$ and $\max\{|\mathbf{S}_<|, |\mathbf{S}_>|\} \leq (1 - \alpha)|\mathbf{S}|$ (with $0 < \alpha \leq \frac{1}{2}$).

Then the recurrence equation of QuickSort is written as:

$$t(n) \leq t(\alpha n) + t((1 - \alpha)n) + O(n). \tag{7.3}$$

This recurrence solves as $t(n) = O(n \log n)$ time, for any fixed $\frac{1}{2} \geq \alpha > 0$. For $\alpha = \frac{1}{2}$, the balanced partitioning case, we obtain an $O(n \log n)$ time (using obviously linear memory). Moreover, if there are only p distinct elements among the n elements, the running time of QuickSort improves to $O(n \log p)$. This bound matches another comparison-based sorting algorithm called *BucketSort*. BucketSort sorts n numbers represented using b bits in $O(bn)$ time. Since there are at most $p = 2^b$ distinct elements in $\mathbf{S}$, for very large $n \geq 2^p$, we get the performance of QuickSort as $O(n \log 2^b) = O(bn)$. That is, QuickSort runs in linear time when the universe size is bounded. There are many other variations and tedious analyses (without the big Oh-notation) of QuickSort that depend on the splitting pivot rule. We do not include them here, but we refer the reader to the Bibliographical Notes provided at the end of the chapter.

So far, we have analyzed the *worst-case*, *best*, and *balanced* running times of QuickSort. A first usage strategy consists in directly applying QuickSort on the array, and hoping that we are in a best or balanced time scenario. However, for any deterministic pivot rule there exists bad input arrays that require quadratic time to sort. We can define the *average running time* $\bar{t}(n)$ of QuickSort by averaging the running times on all permutations σ of the input:

$$\bar{t}(n) = \frac{1}{n!} \sum_{\text{all permutations } \sigma} t(\sigma(\mathbf{S})). \tag{7.4}$$

It can be shown that the average running time of QuickSort on an n-array element is $\bar{O}(n \log n)$, where $\bar{O}(\cdot)$ denotes the average time.[1] That is, the amortized analysis of QuickSort over all possible permutations yields the $\bar{O}(n \log n)$ time bound.

However, we cannot guarantee the average time since it depends on the input configuration. A much better strategy for sorting consists in first preprocessing the array, by applying a *random permutation*, and then applying QuickSort on that shuffled array. The difference is that, at a linear time expense for randomly permuting elements, we protect ourselves with some probability against almost already sorted sequences. Let's now look at this randomization aspect of QuickSort, the expected running time of its run. The expected time $\tilde{t}(n)$ is defined as the expectation of a function $t(\cdot)$ of a discrete random variable $\hat{\mathbf{S}}$:

$$\tilde{t}(n) = \mathbf{E}(t(\hat{\mathbf{S}})) = \sum_{\sigma_i \in \{\text{All possible permutations}\}} t(\sigma_i(\mathbf{S})) \times \mathbf{Pr}[\hat{\mathbf{S}} = \sigma_i]. \tag{7.5}$$

In the randomized QuickSort, all of the $n!$ permutations occur with the same probability:

$$\mathbf{Pr}[\hat{\mathbf{S}} = \sigma_i] = \frac{1}{n!}. \tag{7.6}$$

[1]Traditionally, the $O(\cdot)$ notation hides constant terms. In this randomization chapter, both the $\bar{O}(f(n))$ and $\tilde{O}(f(n))$ notations indicate the complexity analysis method that has been used to obtain the order $f(n)$ complexity: either average analysis ($\bar{O}(f(n))$) or randomized analysis ($\tilde{O}(f(n))$).

Computing a *uniformly random permutation* in linear time and in-place is easily done, as described by the following pseudocode:

```
RandomPermutation($\mathsf{S}$)
1. for $i \in 1$ to $n$
2.     do
3.         $\mathsf{S}[i] = i$
4.         $\triangleleft$ Draw a random number in $[\![1, i]\!]$ $\triangleright$
5.         $j = \mathrm{RandomNumber}(1, i)$
6.         swap($\mathsf{S}[j], \mathsf{S}[i]$)
```

The other common strategy for generating a uniformly random permutation is *permuting by sorting*: First, we draw n real numbers in an auxiliary array $\mathbf{R}$. Then we sort $\mathbf{S}$ according to the keys stored in $\mathbf{R}$, in $O(n \log n)$ time. This second permutation generation method is more costly in time and requires extra memory. Moreover, it requires sorting! We prefer to use permuting by swapping in-place.

For QuickSort, instead of randomly permuting the input set, we choose the pivot at each procedure call randomly. We summarize for completeness the QuickSort randomized algorithm, in pseudocode:

```
QuickSort($\mathbf{S}$)
1. $\triangleleft$ We only need to sort arrays of size strictly greater than one $\triangleright$
2. if $|\mathbf{S}| > 1$
3.     then $\triangleleft$ Partition in place the array into $\mathbf{S}_<$, $\mathbf{S}_>$ and $\mathbf{S}_=$ $\triangleright$
4.          Choose a random pivot index $k$
5.          $\mathbf{S}_<, \mathbf{S}_=, \mathbf{S}_> \leftarrow$ PartitionArrayInPlace($\mathbf{S}, k$)
6.          $\triangleleft$ Recursive calls $\triangleright$
7.          QuickSort($\mathbf{S}_<$)
8.          QuickSort($\mathbf{S}_>$)
```

In the above algorithm, partitioning in-place means that we rearrange all the elements of $\mathbf{S}$ without requiring extra memory. Therefore, at then end of QuickSort the input array is sorted in place.

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

```
template <class item>
  int PartitionInPlace(item array[], int left, int key, int right)
{
item pivot;
int i,j;

Swap(array[right],array[key]);
pivot = array[right];
i = left-1;
j = right;
     for(;;)
    {
      while(array[++i] < pivot);
      while(array[--j] > pivot);
      if(i >= j) break;
      Swap(array[i], array[j]);
    }
    Swap(array[i],array[right]);

return i;
}

template <class item>
  void QuickSort(item array[], int left, int right)
{
  int e,key;

  if(right > left)
  {
  // Uniform random key
  key=left+(rand()%(right-left+1));
  e= PartitionInPlace(array,left,key,right);

  // Recursive calls
    QuickSort(array,left,e-1);
    QuickSort(array,e+1,right);
  }
}
```

The randomized analysis of QuickSort is done according to the random pivot choices, using *indicator variables.* For an event E, we define a corresponding indicator random variable $\hat{X}_E = I(E)$, such that:

$$I(E) = \begin{cases} 1 & \text{if event E occurs,} \\ 0 & \text{otherwise.} \end{cases} \tag{7.7}$$

Indicator variables are used to convert probabilities into expectations, as follows:

$$\mathbf{Pr}[E] = \mathbf{E}(\hat{X}_E). \tag{7.8}$$

Let us now prove that the randomized QuickSort algorithm runs in expected $t(n) = \tilde{O}(n \log n)$ time with high probability, where $\tilde{O}(\cdot)$ denotes the time bound expectation.[2]

Proof. For the purpose of the randomized analysis of QuickSort, we'll assume that all elements of array $\mathbf{S}$ are distinct, and order and denote them as:

$$s_1 < s_2 < ... < s_n. \tag{7.9}$$

Furthermore, let $\mathbf{S}_{ij} = \{s_i, s_{i+1}, ..., s_j\}$. The crucial observation is to notice that any two elements s_i and s_j of $\mathbf{S}$ are compared at most once. This yields the worst-case quadratic-time bound on the number of comparisons. Let $\hat{X}_{ij}$ be the indicator random variable of event E_{ij}, defined as $\hat{X}_{ij} = I(E_{ij})$, where:

$$E_{ij} : \text{Elements } s_i \text{ and } s_j \text{ are compared during Quicksort.} \tag{7.10}$$

Let $\hat{X}$ denote the random variable that counts the total number of comparisons performed by QuickSort:

$$\hat{X} = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \hat{X}_{ij}. \tag{7.11}$$

The expected number of comparisons of QuickSort is $\tilde{t}(n) = \mathbf{E}(\hat{X})$. Since the expectation of the sum of random variables is the sum of their individual expectations, we get:

$$\tilde{t}(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{E}(\hat{X}_{ij}), \tag{7.12}$$

$$\tilde{t}(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \mathbf{Pr}[s_i \text{ and } s_j \text{ are compared}]. \tag{7.13}$$

In order to be compared, s_i and s_j must be the first chosen pivot of $\mathbf{S}_{ij}$ (array of $j - i + 1$ elements). Otherwise, they will never be compared as they will be split into

[2] In some complexity analysis papers, the $\tilde{O}(\cdot)$ denotes the complexity up to polylogarithmic terms. That is, the $\tilde{O}(\cdot)$ notation hides polylogarithmic terms as opposed to constants. Here, the tilde notation indicates an upper-bound obtained by randomized analysis.

disjoint subsets. Thus, we have:

$$\mathbf{Pr}[s_i \text{ and } s_j \text{ are compared}] = \frac{1}{j-i+1} + \frac{1}{j-i+1} = \frac{2}{j-i+1}. \tag{7.14}$$

It follows that:

$$\tilde{t}(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} \frac{2}{j-i+1} \leq \sum_{i=1}^{n-1} \tilde{O}(\log n) \leq \tilde{O}(n \log n), \tag{7.15}$$

using the fact that $H_i = \sum_{i=1}^{n} \frac{1}{i} \leq \log n + O(1)$. H_n is called the nth harmonic number. QED. □

In fact, it can be shown that QuickSort runs in $\tilde{O}(n \log n)$ time with *high probability.* That is,

$$\mathbf{Pr}[\hat{t}(n) \geq c\mathbf{E}(\hat{t}(n))] \leq \frac{1}{n^d}, \tag{7.16}$$

where c and d are appropriate constants and $\hat{t}(n)$ denote $\hat{X}$. That is, the probability that the running time $\hat{t}(n)$ deviates from its expectation $\tilde{t}(n)$ by a factor c is below $\frac{1}{n^d}$:

$$\mathbf{Pr}[\hat{t}(n) \geq c\tilde{t}(n)] \leq \frac{1}{n^d}. \tag{7.17}$$

A *Las Vegas* algorithm is a randomized algorithm that always guarantees a correct output. Only the running time may change according to the random choices done by the algorithm. QuickSort is a typical classroom example of Las Vegas algorithms. Observe that we may have constant-time randomized algorithms that are not guaranteed to deterministically end up. Indeed, consider the algorithm that keeps tossing a two-sided coin (black color on one side, white on the other side) until it ends up with black face:

```
TossingACoin()
 1. repeat
 2.         Face ← Toss a black/white coin
 3.    until Face=White
```

The expected running time of this algorithm is:

$$\sum_{i=1}^{\infty} \frac{1}{2^i} i = 2. \tag{7.18}$$

But in the worst-case, we keep drawing the white face, infinitely many times.

This kind of random pivot element is useful in many other algorithms such as order statistics. *Order statistics* is a general framework for combining large size data into a small size set that reflects the overall characteristics of the data. The mean, standard deviation and moments of a collection of elements are typical order statistics. Let's consider another Las Vegas randomized algorithm: finding the median element of a set, or more generally finding the kth smallest element of an array. The median is defined as the halfway element of an array. For odd number of elements, the median is uniquely defined as the $\frac{n+1}{2}$th element. But for even number of elements, there are two medians, defined as the $\lfloor\frac{n+1}{2}\rfloor$th element and the $\lceil\frac{n+1}{2}\rceil$th element. We'll look at how this randomization paradigm can be used to design an efficient algorithm for that selection task. We follow the same spirit of QuickSort (random pivot element) for designing another Las Vegas procedure. Actually, historically, this SelectElement algorithm was also reported by Hoare when he described the Partition and QuickSort procedures (1961). SelectElement differs from QuickSort in the sense that the algorithm calls recursively itself only once, as described in the following pseudocode:

```
SelectElement($\mathbf{S}, k$)
 1.  ◁ Select the $k$th smallest element of an array $\mathbf{S}$ of $n$ elements ▷
 2.  if $|\mathbf{S}| = 1$
 3.     then return $\mathbf{S}[1]$
 4.     else  Choose a random pivot index $k \in [\![1, |\mathbf{S}|]\!]$
 5.           ◁ Partition inplace array $\mathbf{S}$ ▷
 6.           $\mathbf{S}_<, \mathbf{S}_=, \mathbf{S}_> \leftarrow$ PartitionArray($\mathbf{S}, k$)
 7.           if $k \leq |\mathbf{S}_<|$
 8.              then return SelectElement($\mathbf{S}_<, k$)
 9.              else  if $k > |\mathbf{S}_<| + |\mathbf{S}_=|$
10.                       then return SelectElement($\mathbf{S}_>, k - |\mathbf{S}_<| - |\mathbf{S}_=|$)
11.                       else  ◁ The $k$th smallest element of $\mathbf{S}$ is inside $\mathbf{S}_=$ ▷
12.                             ◁ $\mathbf{S}_=[1]$ is stored at $\mathbf{S}[k]$ (inplace partitioning) ▷
13.                             return $\mathbf{S}[k]$
```

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

```
template <class item>
   item SelectElement(item *array, int left, int right, int k)
{
int e,key;

if(right > left)
  {// Uniform random key
  key=left+(rand()%(right-left+1));
  e=PartitionInPlace(array,left,key,right);

  // Recursion
  if (k<=e) return SelectElement(array,left,e,k);
  else
    SelectElement(array,e+1,right,k);
  }
  else // right=left
    return array[left];
}
```

To perform the randomized analysis of SELECTELEMENT, we'll consider the following n events: $E_1, ..., E_n$ and corresponding indicator variables $\hat{X}_1, ..., \hat{X}_n$, such that event E_i is defined as:

$$E_i : \text{The selected pivot is the } i\text{th smallest element.} \tag{7.19}$$

Clearly, $\mathbf{E}(\hat{X}_i) = \mathbf{Pr}[E_i] = \frac{1}{n}$, by virtue of the equiprobability of the uniform sampling.

Let $\hat{t}(n)$ denote the running time random variable of the procedure SELECTELEMENT. We write $\hat{t}(n)$ using the following recurrence:

$$\hat{t}(n) \leq \sum_{i=1}^{n} \hat{X}_i \hat{t}(\max\{n-i, i-1\}) + \underbrace{O(n)}_{\text{In-place partition}} . \tag{7.20}$$

Thus, the expected running time $\tilde{t}(n) = \mathbf{E}(\hat{t}(n))$ is bounded by:

$$\tilde{t}(n) \leq \mathbf{E}\left(\sum_{i=1}^{n} \hat{X}_i \hat{t}(\max\{n-i, i-1\}) + O(n)\right), \tag{7.21}$$

$$\tilde{t}(n) \leq \sum_{i=1}^{n} \frac{1}{n} \mathbf{E}(\hat{t}(\max\{n-i, i-1\})) + O(n), \tag{7.22}$$

which solves as $\tilde{t}(n) = \tilde{O}(n)$.

Proof. We prove the bound by induction, using $\mathbf{E}(\hat{t}(k)) = \tilde{t}(k) \leq ck$, for $k < n$. Rewriting equation Eq. 7.22, we get:

$$\tilde{t}(n) \leq \frac{2}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} \tilde{t}(i) + an. \tag{7.23}$$

That is,

$$\begin{aligned}
\tilde{t}(n) &\leq \frac{2c}{n} \sum_{i=\lfloor \frac{n}{2} \rfloor}^{n-1} i + an, && (7.24)\\
&\leq \frac{2c}{n} \frac{(\lfloor \frac{n}{2} \rfloor + n - 1)(n - \lfloor \frac{n}{2} \rfloor)}{2} + an && (7.25)\\
&\leq \frac{2c}{n} \frac{\frac{3n}{2}\frac{n}{2}}{2} + an && (7.26)\\
&\leq (\frac{3}{4}c + a)n \leq cn, && (7.27)
\end{aligned}$$

provided that $\frac{3}{4}c + a \leq c$. That is, we choose any $c \geq 4a$. QED. □

This randomized analysis shows that by imposing a uniformly random distribution of the input, we prove that the running time of SelectElement is expected linear time, although its worst-case time is quadratic. Note that there exists a deterministic selection algorithm that matches the optimal linear time bound. However, this deterministic algorithm is more elaborate, as it uses the *prune-and-search* paradigm.

In summary, we have seen that the average-case analysis characterizes the average running time of deterministic algorithms, by assuming a random distribution of data. However, any deterministic algorithm has some input data (worst-case input) that elicits its worst-case time. On the contrary, randomized algorithms make (uniform) random choices that shuffle the data by a (uniform) random distribution. Thus, randomized algorithms have no *a priori* worst-case inputs. We compare randomized algorithms according to their expected times.

Finally, we'll mention that sorting n numbers by comparisons requires $\Omega(n \log n)$ time.[3] Indeed, *each* comparison of any sorting algorithm separates the set of permutations into two sets. Thus, to separate properly all of the $n!$ permutations and reach from every permutation the sorted sequence by performing comparisons, we need at least[4] $\log(n!) = O(n \log n)$ comparisons[5] (see also the Bibliographical Notes).

[3]The $\Omega(\cdot)$ notation denotes the intrinsic complexity of a problem. That is, a lower bound up to constant factors of *any* algorithm solving the given problem.

[4]Indeed, the best we can do at each comparison step is to halve the set of permutations.

[5]Using Stirling's approximation formula: $n! \sim n^n \exp(-n)\sqrt{2\pi n}$.

7.2 Random Sample Consensus

The *Random Sample Consensus* (RANSAC) paradigm is an extremely powerful selection method invented by Fischer and Bolles in 1981. RANSAC is used to improve matching techniques that occur in many pattern matching applications of computer vision and computational geometry. One of the key difficulties of matching two point sets, is to detect those which match from those that do not. The matching points are called the *inliers*, and the remaining points are called *outliers*. Such point sets can come from image processing feature detectors, such as corner detectors. For example, for stitching two pictures imaging a same planar surface, we may first detect respective point features in the two input images. A common method for detecting

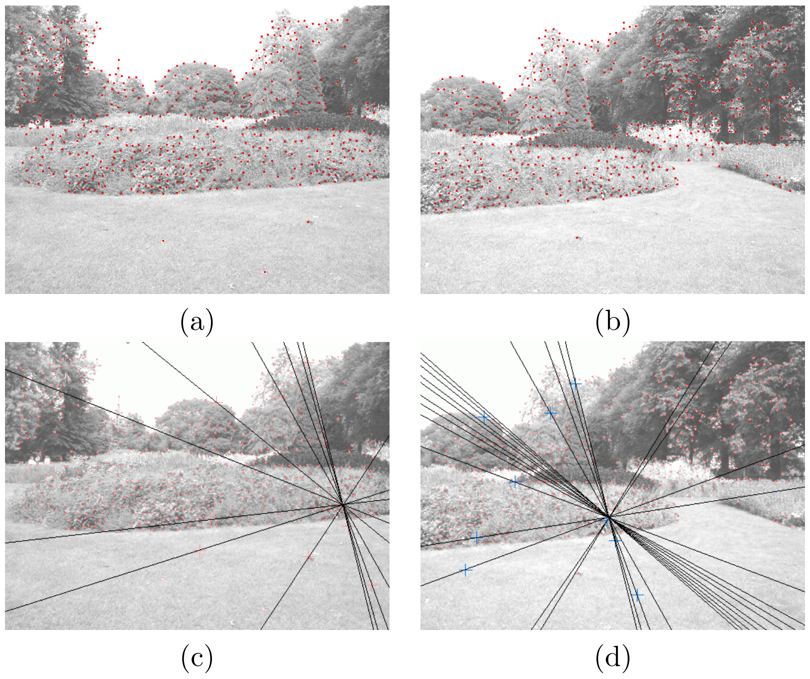


FIGURE 7.2 *Fully automatic computation of the epipolar geometry of two uncalibrated pinhole camera images. Pictures (a) and (b) display the two source images with their respective extracted Harris-Stephens point features. Pictures (c) and (d) shows the recovered epipolar geometry after a RANSAC procedure has been applied. Observe that the epipolar lines intersect respectively in their epipole.*

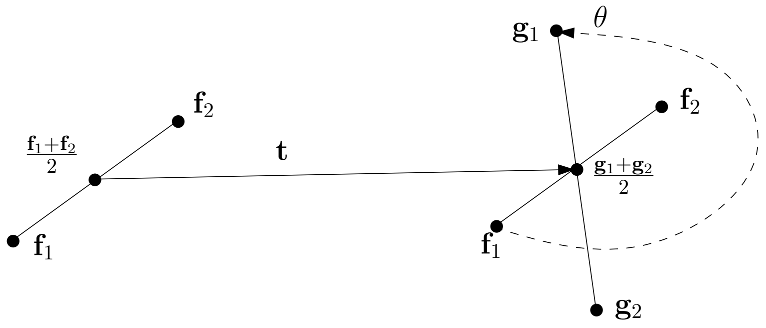


FIGURE 7.3 *A similitude transformation is unambiguously defined by a pair of matching features.*

feature points in images is to use the Harris-Stephens corner detector algorithm (see also Section 4.2.1). However, the Harris-Stephens corner sets contain both inliers and outliers.

Once a proper *labeling* of points into inliers and outliers is done, we perform classical numerical optimization on pairs of matching features. This inliers/outliers labeling problem often occurs in pattern matching and computer vision, where we have to deal with a large proportion of outliers. Figure 7.2 illustrates such an image matching problem, by considering the automatic computation of the epipolar geometry (fundamental matrix F) defined by two uncalibrated source images $\mathbf{I}_1$ and $\mathbf{I}_2$, respectively. Let $\mathcal{F} = \{\mathbf{f}_i\}_i$ and $\mathcal{G} = \{\mathbf{g}_i\}_i$ be two 2D point sets extracted by a feature detector algorithm from images $\mathbf{I}_1$ and $\mathbf{I}_2$.

For ease of presentation, we consider calculating from feature sets a similitude transformation M matching the two input images. That is, to find a rotation followed by a uniform scaling and a translation, that best matches those two point sets. The translation vector $\mathbf{t} = [t_x\ t_y]^T$, rotation angle θ and isotropic scaling s are called the *free parameters* of the matching problem (4 degrees of freedom).

First, observe that given two pairs of matching features $\mathbf{f}_1 \leftrightarrow \mathbf{g}_1$ and $\mathbf{f}_2 \leftrightarrow \mathbf{g}_2$, the transformation matrix M is fully determined. Indeed, the translation is computed as the difference of the midpoints of line segments $[\mathbf{f}_1\mathbf{f}_2]$ and $[\mathbf{g}_1\mathbf{g}_2]$: $\mathbf{t} = \frac{\mathbf{g}_1+\mathbf{g}_2}{2} - \frac{\mathbf{f}_1+\mathbf{f}_2}{2}$. The isotropic scaling is the ratio of the segment lengths: $s = \frac{||\mathbf{g}_1\mathbf{g}_2||}{||\mathbf{f}_1\mathbf{f}_2||}$. The rotation θ is then recovered using a simple arctan operation, as depicted in Figure 7.3, once the segment midpoints have been matched. Finally, the transformation matrix M is obtained by the following transformation pipeline (with associated matrix concatenation):

$$\mathsf{M} = \mathsf{M}[\mathbf{f}_1, \mathbf{f}_2; \mathbf{g}_1, \mathbf{g}_2] = \mathsf{T}_{\frac{\mathbf{g}_1+\mathbf{g}_2}{2}} \mathsf{R}_\theta \mathsf{S}_s \mathsf{T}_{-\frac{\mathbf{f}_1+\mathbf{f}_2}{2}}, \tag{7.28}$$

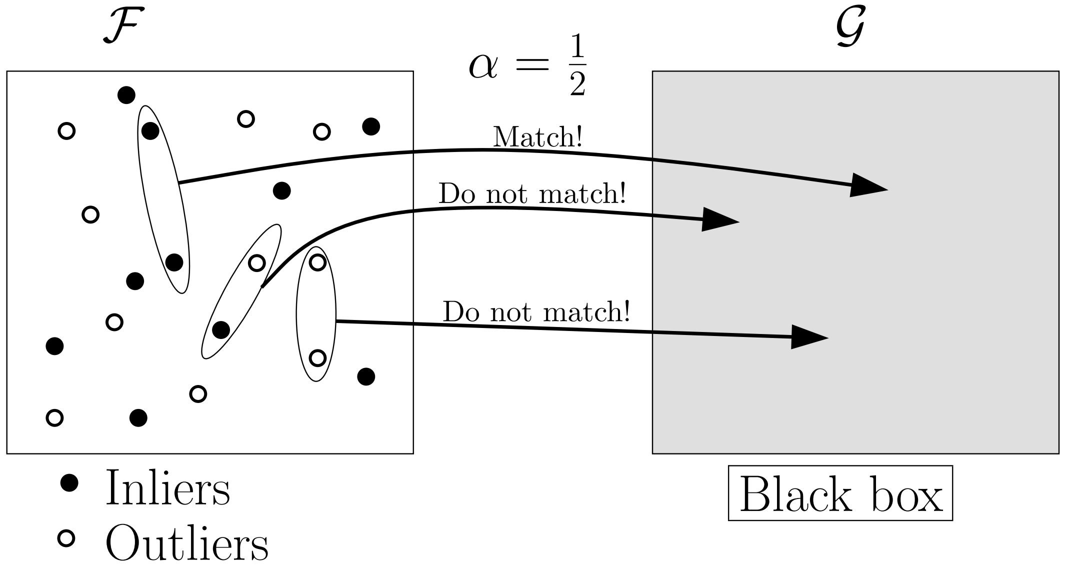


FIGURE 7.4 *Conceptual illustration of the RANSAC random sampling.*

$$\mathbf{M}[\mathbf{f}_1, \mathbf{f}_2; \mathbf{g}_1, \mathbf{g}_2] = \begin{bmatrix} \mathbf{I} & \frac{\mathbf{g}_1+\mathbf{g}_2}{2} \\ \mathbf{0}^T & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s & 0 & 0 \\ 0 & s & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{I} & -\frac{\mathbf{f}_1+\mathbf{f}_2}{2} \\ \mathbf{0}^T & 1 \end{bmatrix}. \tag{7.29}$$

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

Let's say that the point sets $\mathcal{F}$ and $\mathcal{G}$ match densely. That is, at least $\alpha \times 100$ percent of the points match. Since it should be clear that $0 \leq \alpha \leq 1$, we will loosely say α percent of points match, although that in fact it is $\alpha \times 100$ percent! In other words, α denotes the *ratio* of inliers. RANSAC provides a robust matching estimation method that simply consists in randomly drawing two elements $\{\mathbf{f}_1, \mathbf{f}_2\}$ of feature set $\mathcal{F}$: a *random sample*. The probability that these two elements match and give rise to two correspondence pairs with two other elements of $\mathcal{G}$ is α^2. Indeed, we have to be lucky twice. Saying it a different way, the probability that we failed to pick up a good subset is at most $1-\alpha^2$. Figure 7.4 gives a schematic illustration of the pair sampling process.

TABLE 7.1 *Number of rounds required by RANSAC to ensure a failure probability below 1%.*

Number of pairs / Application example		Outliers ratio $(1-\alpha)$		
		10%	30%	50%
2	Similitude	3	7	17
3	Affine	4	11	35
4	Homography	5	17	72
6	Trifocal tensor	7	37	293
7	Fundamental matrix	8	54	588

Observe that we are only using the combinatorial information of set $\mathcal{F}$, ignoring $\mathcal{G}$ (those sets are though related by α). To check whether a set of correspondence pairs yields an admissible transformation, we first compute $\mathbf{M}[\mathbf{f}_1, \mathbf{f}_2; \mathbf{g}_1, \mathbf{g}_2]$, and then we check that $|\mathbf{M}(\mathcal{F}) \cap \mathcal{G}| \geq \alpha n$, where $\mathbf{M}(\mathcal{F}) = \{\mathbf{M} \times \mathbf{e} \mid \mathbf{e} \in \mathcal{F}\}$. The matching set of inliers $\mathcal{I} = \mathbf{M}(\mathcal{F}) \cap \mathcal{G}$ defines the *consensus set* for the current sample. Again, for sake of simplicity, we didn't take into account some noise threshold ϵ into the analysis. RANSAC is known empirically to work well with noise; If Gaussian noise with zero mean is assumed then we can compute an optimal statistical threshold distance to determine whether two points match or not. If the current induced transformation does not produce a valid match with α percent of matched features, then we simply discard the two matching pairs and draw again another two-element sample from $\mathcal{F}$, and reiterate. After k *independent* sampling rounds, the probability of failure becomes $(1-\alpha^2)^k$. That is, as the number of draws k grows, the probability of failure tends to zero. More precisely, we can calculate the number of rounds k, so that the failure probability falls below a given threshold f: find the minimum k such that $(1-\alpha^2)^k \leq f$. That is,

$$e^{k \log(1-\alpha^2)} \leq f. \tag{7.30}$$

We get from Equation 7.30: $k = \left\lceil \frac{\log f}{\log(1-\alpha^2)} \right\rceil = \lceil \log_{1-\alpha^2} f \rceil$. In general, if we need s point correspondences to define the free parameters of our matching transform, we obtain the following similar bound:

$$k = \left\lceil \frac{\log f}{\log(1-\alpha^s)} \right\rceil = \lceil \log_{1-\alpha^s} f \rceil. \tag{7.31}$$

Table 7.1 provides some tabulated values for $f = 0.01$ (1% probability of failure) and various ratio values of α and integers s. RANSAC is useful for stitching images with homographies (require four pairs), computing the trifocal tensor (require six pairs), recovering the epipolar geometry (require seven pairs), pose estimation (the number of degrees of freedom depends on the template model) and tracking, etc.

We summarize in pseudocode the RANSAC procedure for automatically and robustly computing a homography between two point sets:

```
HomographyRANSAC($\mathcal{P}_1, \mathcal{P}_2, n, f, \alpha$)
 1. $\triangleleft$ $n$: number of points of $\mathcal{P}_1$ and $\mathcal{P}_2$ $\triangleright$
 2. $\triangleleft$ $f$: probability of failure $\triangleright$
 3. $\triangleleft$ $\alpha$: a priori inlier ratio $\triangleright$
 4. $k = \left\lceil \frac{\log f}{\log(1-\alpha^4)} \right\rceil$
 5. $\triangleleft$ $c_m$: maximum consensus set found by RANSAC $\triangleright$
 6. $c_m = 0$
 7. for $i \leftarrow 1$ to $k$
 8.     do Draw a random sample $\mathcal{S}_1$ of 4 elements from $\mathcal{P}_1$
 9.        Check with all other 4-elements of $\mathcal{S}_2$.
10.        For each correspondence sets, calculate the free parameters.
11.        Compute the consensus set size $c$
12.        if $c > c_m$
13.           then $c_m = c$
14.                Save current transformation and largest consensus set
15. $\triangleleft$ Final stage: compute the homography given correspondence pairs $\triangleright$
16. Estimate the homography with the largest found consensus sample (inliers)
```

From the viewpoint of time complexity of matching algorithms, RANSAC lets us save an $O(n^s)$ factor. Indeed, there are $\binom{n}{s} = O(n^s)$ possible s-matching sets in $\mathcal{F}$, for fixed s. Given two s-matching sets, we still need to get a unique labeling by specifying the sets point-to-point correspondences. That is, there are $s!\binom{n}{s} = O(n^s)$ possible labelings, for fixed s. Assuming that there is a nonzero $\alpha > 0$ ratio of inliers, RANSAC needs to check only $O(1)$ random samples, while guaranteeing a probability of failure as small as required, but fixed. A naive algorithm will check that many $\binom{n}{s}\binom{n}{s}s! = O_s(n^{2s})$ combinations. Thus, RANSAC let us save roughly a square root factor over the brute-force exploration algorithm.

In practice, we seldom know the *a priori* proportion α of inliers. We discover it *a posteriori*. Moreover, the proportion of inliers is input sensitive. It depends on the configuration of the input sets. Nevertheless, we can bootstrap RANSAC so that it becomes *adaptive* to the input configurations. This is because the samples are drawn independently from each other.

We provide the adaptive RANSAC pseudocode, essentially for comparison with HomographyRANSAC:

```
AdaptiveRANSAC($n, f$)
 1.  ◁ $n$: data set size ▷
 2.  ◁ $s$: number of correspondences required (free parameters) ▷
 3.  ◁ $f$: probability of failure ▷
 4.  ◁ Initialize $k$ to a very large number ▷
 5.  $k = \infty$
 6.  ◁ $d$: current number of independent random draws ▷
 7.  $d = 0$
 8.  ◁ $c_m$: maximum consensus set found so far by RANSAC ▷
 9.  $c_m = 0$
10.  while $k > d$
11.     do Draw a random sample
12.        Compute the consensus set size $c$
13.        if $c > c_m$
14.          then ◁ Update the proportion of inliers ▷
15.               $c_m = c$
16.               $\alpha = \frac{c}{n}$
17.               ◁ Lower the number of rounds ▷
18.               $k = \left\lceil \frac{\log f}{\log(1-\alpha^s)} \right\rceil$
19.               $d = d + 1$
```

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

RANSAC was somewhat a counterintuitive technique back in 1981. Indeed, at that time practitioners usually performed matching using as much data as possible: they were *smoothing* potential errors by taking large sets. In comparison, RANSAC is the inverse methodology that consists of finding a *sparse* matching random sample, and then extending it to its *dense* consensus set, eventually repeating this procedure at most a fixed number of times. RANSAC is not a Las Vegas-type of algorithm. It is classified as a *one-sided error* Monte Carlo algorithm. Indeed, if RANSAC finds a large consensus set, this means that the set really exists. Otherwise, we may have missed such a large consensus set, however this may happen with a small failure probability. We define a *two-sided error* Monte Carlo algorithm as a randomized procedure that has nonzero probability of doing an error in both ways: reporting a matching set that is not a matching set (so-called false "true" case), or failing to report an existing (large) matching set (so-called false "false" case). To conclude, we emphasize that Las Vegas randomized algorithms (such as QuickSort) do not make errors but have varying running time, while Monte Carlo algorithms (such as RANSAC) finish *on time* but may be prone to one-sided or two-sided errors.

7.3 Monte Carlo Samplings

Computers crunch numbers so fast[6] that one area of computing has been dedicated to simulating physical phenomena of the real world. For example, in computer graphics, we simulate light scattering phenomena using a Monte Carlo path-tracing algorithm. Or, in computer vision, we simulate image generation models to denoise and enhance captured raw data. And in computational geometry, we simulate flows and deforming geometries using the finite element method (FEM). All those simulations require to draw samples from probability density functions to effectively discretize domains. Those probability density functions describe the possible values for parameters and their likelihoods.

As a case-study, consider computing an approximation of the transcendental number π, an infinite nonrepetitive sequence of digits. We are going to simulate a simple physical process.

Draw uniformly random points on a square of side lengths $2r$ (for some given r), using two uniform and independent 1D probability density functions, for the x- and y-abscissae. Figure 7.5 illustrates this random sampling process. The probability of drawing a point inside the unit disk is written as:

$$\frac{\text{area(disk)}}{\text{area(square)}} = \frac{\pi r^2}{(2r)^2}. \tag{7.32}$$

That is, $\frac{\pi}{4}$ for any $r > 0$. Since that probability is independent of the radius r (provided it is strictly positive), we assume in the remainder that $r = 1$. Detecting whether a point $\mathbf{p}$ drawn by our Monte Carlo sampling process belongs to the disk is done by checking that its distance to the origin is less than one: $||\mathbf{p}|| \leq 1$. As the number n of drawn point samples increases, the empirical probability $\frac{\{||\mathbf{p}|| \leq 1 \mid \mathbf{p} \in \mathcal{P}\}}{n}$ tends to $\frac{\pi}{4}$. Therefore, an approximation of π is given by:

$$\pi \sim_{n \to \infty} 4 \frac{\{||\mathbf{p}|| \leq 1 \mid \mathbf{p} \in \mathcal{P}\}}{n}. \tag{7.33}$$

[6] At the time this book is published, we expect teraflops (TFLOPS) personal multicore chip workstations. The traditional interactive graphics rendering pipeline will further be complemented by real-time ray tracing and Monte Carlo rendering.

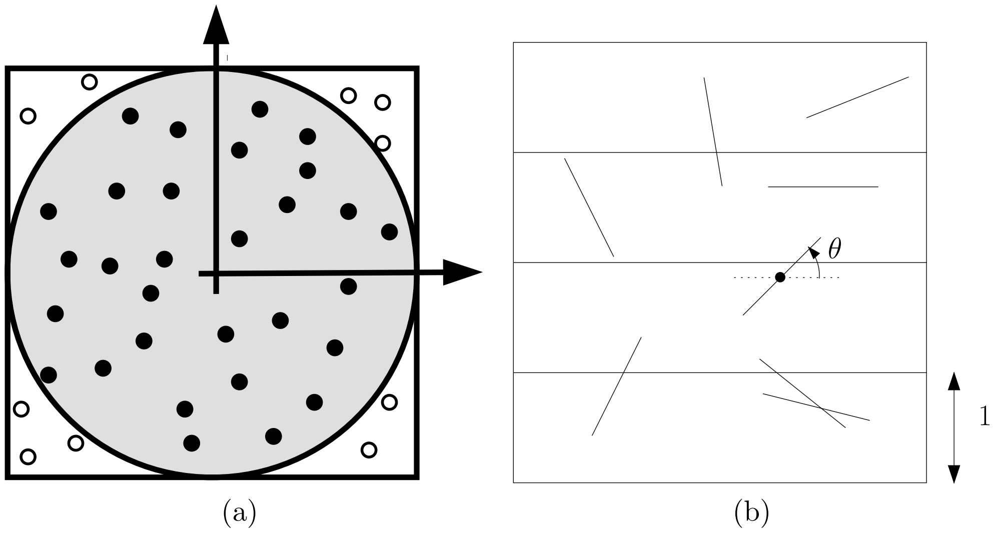


FIGURE 7.5 *Discrete Monte Carlo integrations: using computer simulations to calculate coarse approximations of π. (a) Point samples are drawn uniformly inside a square. π is approximated as four times the ratio of points lying inside the unit disk. (b) The Buffon's needle simulation (1733) consists in letting unit length needles fall randomly on a unit-spaced ruled paper. It can be shown that π is approximated as twice the inverse of the ratio of needles crossing any horizontal line.*

Unfortunately, this simple simulation heuristic does not converge quickly (see the variance discussion later on in the chapter). There is yet another Monte Carlo method to calculate π that is called the Buffon's needle: needles of unit length are randomly dropped onto a sheet of paper ruled with horizontal lines spaced one unit apart. More precisely, a new unit length needle is randomly drawn by first uniformly randomly choosing its segment midpoint inside the sheet of paper, and then sampling uniformly randomly an orientation $\theta \in [-\pi, \pi)$. The ratio of the number of needles that cross a line over the total number of needles approaches $\frac{2}{\pi}$. Figure 7.5 depicts this experimental process for approximating π. Note that there are also many more efficient ways of computing fine approximations of π.[7]

Consider another example: We are given a collection $\mathcal{O} = \{O_1, ..., O_n\}$ of n 3D

[7]See *http://numbers.computation.free.fr/Constants/PiProgram/pifast.html* if you want to compute billion digit approximations! We can also look for a simple closed formula that approximates π; for example, $\pi \sim \frac{\ln(640320^3+744)}{\sqrt{163}}$ approximates 30 digits. Machin's formula $\pi = 16 \arctan \frac{1}{5} - 4 \arctan \frac{1}{239}$ can also be used to approximate π using the arctangent serie: $\arctan x = \sum_{k=0}^{\infty} \frac{(-x)^{2k+1}}{2k+1}$.

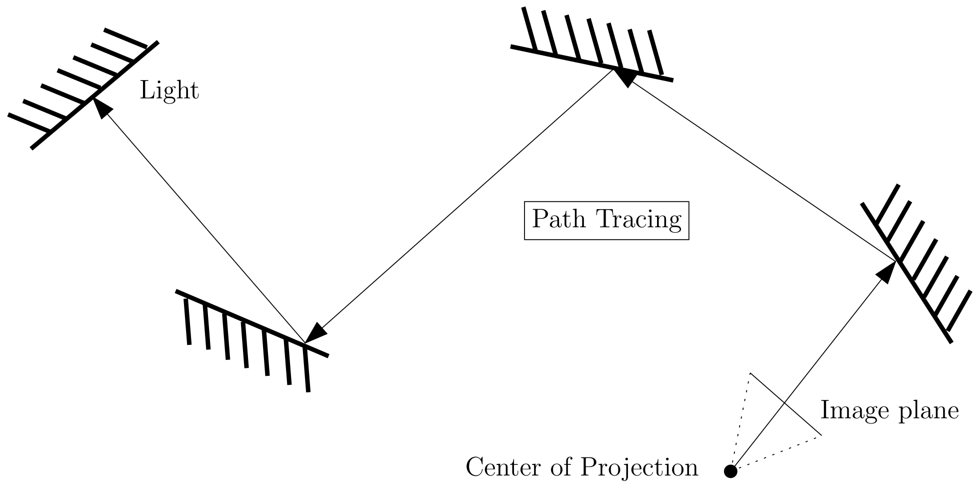


FIGURE 7.6 *Illustration of the path tracing algorithm.*

objects included in a 3D unit cube. We would like to calculate the volume of the union of these objects: $\text{Volume}(\mathcal{O}) = \text{Volume}(\bigcup_{i=1}^{n} O_i)$. The Monte Carlo sampling method provides us with an easy way to get a rough *estimate* of the occupancy volume: draw k points uniformly in the unit cube and report the ratio of points falling inside any object.

This kind of discrete integration technique is known as Monte Carlo integration. Monte Carlo samplings are randomized algorithms that guarantee termination but may yield incorrect results (one-sided or two-sided errors). In computer graphics rendering, the Monte Carlo sampling is used in ray path tracing, and for computing various integral approximations, fast. The Monte Carlo method allows us to produce stunning visual global illumination renderings. Using Monte Carlo sampling, we get fast estimates of pixel projected areas on surfaces or solid angles defined by complex shapes. Those integrals have usually many dimensions (one per parameter), including time. Moreover, the function to integrate, say, the rendering equation, has many discontinuities (occlusions, BRDF, cautics, etc.)

Monte Carlo principles are used in many computer graphics rendering procedures. For each pixel of the rendered image, we cast a fixed number of jittered rays. Those rays are then traced in the 3D scene. Whenever a ray hits a surface, it is either reflected or emits light according to the surface material.

We summarize a typical Monte Carlo *path-tracing* procedure in pseudocode (Figure 7.6):

```
MonteCarloRendering(I)
 1. ◁ Render the scene using path tracing ▷
 2. ◁ $w, h$: Image width and height ▷
 3. for $y \leftarrow 1$ to $h$
 4.     do for $x \leftarrow 1$ to $w$
 5.            do ◁ $s$: number of samples ▷
 6.               for $k \leftarrow 1$ to $s$
 7.                   do Initialize ray parameters $(x, y, t)$
 8.                      weight = 1
 9.                      Trace ray until it hits the nearest surface
10.                      ◁ Russian Roulette ▷
11.                      Decide whether this ray is reflected or emits light,
12.                      by comparing a uniform random number against a
13.                      threshold reflection probability characterizing the surface.
14.                      ◁ $c_e, c_r$: emitted and reflected transmitted coefficients ▷
15.                      if emitted
16.                         then ◁ Ray is absorbed and a new ray is created ▷
17.                              weight = $c_e \times$ weight
18.                              Delete old ray and create new ray
19.                              (with random direction)
20.                         else ◁ Reflection in a random direction ▷
21.                              weight = $c_r \times$ weight
22.                              Update the ray geometry
23.                              (random direction according to the BRDF)
```

Besides, Monte Carlo paradigm is also extremely useful for designing antialiasing procedures using jittering or stratified samplings. The main disadvantage of the Monte Carlo paradigm is that its calculated approximations converge slowly. This is because the variance decreases inversely proportional to the number of samples. Indeed, the basic discrete Monte Carlo integration technique approximates a continuous 1D integral by a 2D box area of width the domain size $(b - a)$ and height the average of the function evaluated at uniformly distributed random samples $\{x_1, ..., x_n\}$:

$$\int_a^b f(x)dx \sim (b-a)\frac{1}{n}\sum_{i=1}^{n} f(x_i). \tag{7.34}$$

In the limit case, we have equality:

$$\lim_{n\to\infty}(b-a)\frac{1}{n}\sum_{i=1}^{n} f(x_i) = \int_a^b f(x)dx. \tag{7.35}$$

The convergence rate is indicated by the *standard deviation* σ that measures the statistical dispersion. The standard deviation is the square root of the variance:

$$\mathrm{var}_n = \sigma_n^2 = \frac{1}{n}\sum_{i=1}^{n}\left(f(x_i) - \frac{1}{n}\sum_{i=1}^{n} f(x_i)\right)^2. \tag{7.36}$$

Thus, σ_n is of order $\frac{1}{\sqrt{n}}$. That is, to halve the amount of "imprecision" (σ), we have to quadruple the number of random samples. Typically, for a path tracing procedure, we cast 1,000 to 10,000 rays per pixel. That is, to render an image, it is customary to use billions to trillions of rays. There are various extensions of the Monte Carlo discrete sampling integration technique that tackle the variance reduction, such as adaptive sampling, stratified sampling or Metropolis sampling.

In practice, the difference between the exact rendering (the one we get if we could solve all integrations analytically) and its Monte Carlo discrete integral approximation rendering is perceived as "noise" in the image. The larger the variance, the bigger the amount of noise, and the greyer the synthesized image (see Figure 7.7). Figure 7.7 shows two images computed by a Monte Carlo path tracer using different sampling rates. Observe the large amount of noise in the undersampled image. There are yet some other biased and nonbiased variance-reduction techniques to further filter and partially remove some "noise."

7.4 Randomizing Incremental Algorithms

We have presented earlier (in Section 5.2.3) the edge-flip Delaunay triangulation algorithm that starts from any arbitrary triangulation of a point set, and flips edges of triangles that define nonempty Delaunay circles (see page 308) until the triangulation becomes Delaunay. This algorithm unfortunately has quadratic running time, in the worst case. Here, we present the *randomized incremental construction* of the Delaunay triangulation of a planar point set, using edge flips. Let $\mathcal{P} = \{\mathbf{p}_1, ..., \mathbf{p}_n\}$ be a planar point set in general position (no four points cocircular). For ease of presentation, we'll add three triangle points $\mathbf{t}_1$, $\mathbf{t}_2$, and $\mathbf{t}_3$, so that $\mathcal{P}$ is fully contained inside this bounding triangle $T_0 = \triangle\mathbf{t}_1\mathbf{t}_2\mathbf{t}_3$ (with $\mathcal{P} \subseteq T_0$). The incremental construction of the Delaunay triangulation adds a point at a time, so that at the ith stage, we have the Delaunay triangulation of $\mathcal{P}_i = \{\mathbf{t}_1, \mathbf{t}_2, \mathbf{t}_3, \mathbf{p}_1, ..., \mathbf{p}_i\}$. At the $(i+1)$-th stage, we add point $\mathbf{p}_{i+1} \in \mathcal{P}$. First, we find the triangle T_{i+1} of the current Delaunay triangulation

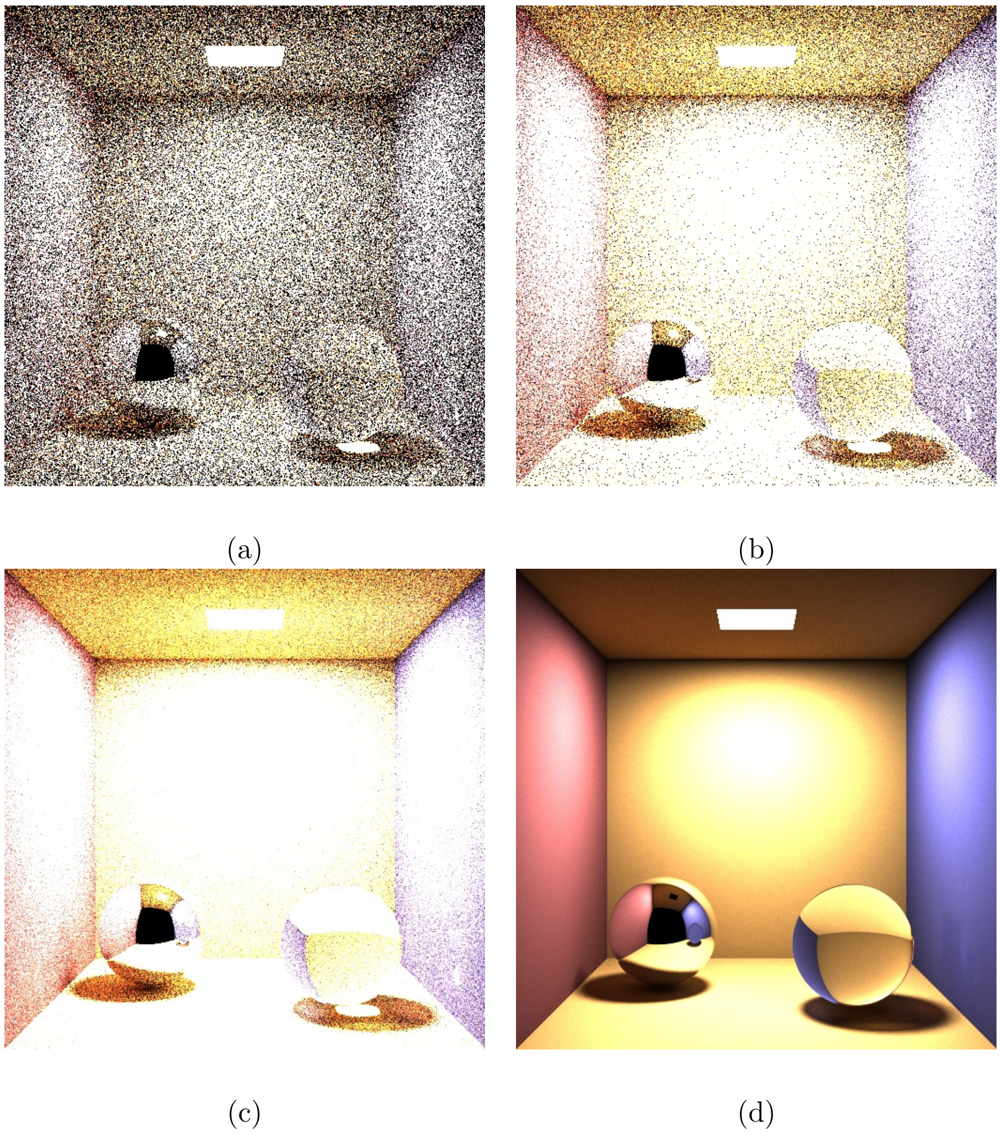


FIGURE 7.7 *Monte Carlo path tracing. Image (a) has been calculated by taking 100 jittered rays/pixel. Images (b) and (c) have been obtained by increasing the sampling rate to 400 and 1600 rays/pixel, respectively. Observe that undersampling is much visible in (a) as noise. Image (d) has been calculated using the bidirectional path-tracing method (1600 rays/pixel) that significantly improves the quality over the ordinary path-tracing Monte Carlo method.*

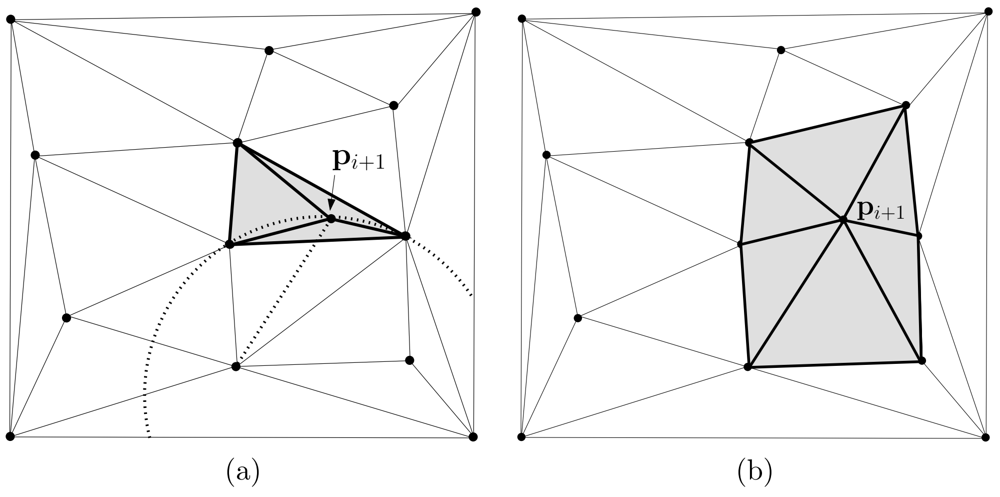


FIGURE 7.8 *Incremental Delaunay Triangulation. (a) The new inserted point* $\mathbf{p}_{i+1}$ *is first located on the current Delaunay triangulation and its triangle is split into three new triangles, yielding the initial star-shaped polygon. The dashed arc shows a nonempty disk defined by a triangle of the initial star-shaped polygon (grey polygon). This shows that the initial star-shaped polygon anchored at* $\mathbf{p}_{i+1}$ *needs to be updated since one of its triangles is not Delaunay (i.e., nonempty circle). (b) The updated star-shaped polygon and its boundary edges where edge flips have been performed. Here,* $6 - 3 = 3$ *edge flips have been called.*

$\mathcal{D}_i$ that contains the newly inserted point $\mathbf{p}_{i+1}$. This triangle necessarily exists because we added the three "bounding" vertices of T_0. We then split triangle T_{i+1} into three triangles using point $\mathbf{p}_{i+1}$, and perform edge flips, so that the triangulation remains Delaunay. It turns out that vertex $\mathbf{p}_{i+1}$ in the triangulation defines a star-shaped polygon (see Section 5.1), as depicted in Figure 7.8. The boundary of the star-shaped polygon consists in all edges of triangles incident to $\mathbf{p}_{i+1}$ but not sharing and edge endpoint with $\mathbf{p}_{i+1}$. Moreover, every new edge-flipped triangle necessarily contains vertex $\mathbf{p}_{i+1}$ at one of its apex. Indeed, each edge flip replaces an edge with a newly created edge having extremity $\mathbf{p}_{i+1}$. Thus, the number of edge flips is exactly the difference of the two star-shaped polygon number of edges. That is, the difference of the number of edges of the star-shaped polygon after having finished updating the Delaunay triangulation minus the number of edges of the initial star-shaped polygon (the three edges of triangle T_{i+1}). This is also equal to the degree of $\mathbf{p}_{i+1}$ minus three in the updated Delaunay triangulation $\mathcal{D}_{i+1}$. Let us now analyze the total number of edge flips when building the incremental Delaunay triangulation of $\mathcal{P}$. Consider *any* random permutation of the points in $\mathcal{P}$, and let us look at all possible combinatorial positions of the last inserted point $\mathbf{p}_n$. The sum of the degrees of $\mathbf{p}_n$ for all possible configurations is the same as the sum of the degrees of all points in the final Delaunay triangulation. This is simply because $\mathbf{p}_n$ could be chosen as any of the points of the final Delaunay triangulation. The sum of the degrees of the vertices of *any* Delaunay triangulation is equal to twice the number of its edges. Since each point insertion creates three new edges, we have $3n$ edges in a Delaunay triangulation, and the sum of the degrees of vertices of a Delaunay triangulation is $2 \times 3n = 6n$. Thus, the total number of edge flips for all configurations of $\mathbf{p}_n$ is at most $6n - 3n = 3n$. It follows that the overall number of edge flips $e(n)$ for all permutations of the point set $\mathcal{P}$ is obtained by using the following recurrence equation:

$$e(n) \leq \underbrace{n}_{\text{Possible choices for last point } \mathbf{p}_n} \times e(n-1) + \underbrace{3n}_{\text{Total \# edge flips}} \leq 3n \times n!. \tag{7.37}$$

To get the *amortized average* number of flips $\bar{e}(n)$, we now have to divide by the number $n!$ of possible permutations:

$$\bar{e}(n) \leq 3n \times n! \frac{1}{n!} \leq 3n. \tag{7.38}$$

The randomized incremental Delaunay construction proceeds by inserting *randomly* one vertex at a time. Thus, the overall *expected* number of flips $\tilde{e}(n)$ matches $\bar{e}(n)$:

$$\tilde{e}(n) \leq 3n. \tag{7.39}$$

This kind of complexity analysis is called the *backward analysis*. In the algorithm complexity analysis, we didn't describe how we efficiently localize the inserted points into the current Delaunay triangulations. That is, to find efficiently triangles $\{T_i\}_i$. This step can be done using a history direct acyclic graph (DAG) that stores all events (triangle splittings and edge flips) in a hierarchical way. An amortized randomized analysis further shows that locations of inserted points into Delaunay triangulations are done in *expected* logarithmic time. (There are many other algorithmic methods for this localization step mentioned in the Bibliographical Notes.) We conclude that the incremental randomized Delaunay triangulation of an n-point set can be computed in expected $\tilde{\Theta}(n \log n)$ time, using expected linear memory. The $\tilde{\Theta}(t)$ notation means that the algorithm runs in expected $\tilde{O}(t)$ time, and that t is also a lower bound for the problem too (mathematically written using the notation $\Omega(t)$).

We summarize the incremental construction of the Delaunay triangulation below:

```
IncrementalDelaunayTriangulation($\mathcal{P} = \{\mathbf{p}_1, ..., \mathbf{p}_n\}$)
 1.  ◁ Start with a bounding triangle $T_0 = \triangle \mathbf{t}_1\mathbf{t}_2\mathbf{t}_3$ ▷
 2.  ◁ $\mathcal{D}_i$: Delaunay triangulation at stage $i$ ▷
 3.  $\mathcal{D}_0 = \triangle \mathbf{t}_1\mathbf{t}_2\mathbf{t}_3$
 4.  for $i \leftarrow 1$ to $n$
 5.      do $T_i$ = LocalizeTriangleInTriangulation($\mathbf{p}_i, \mathcal{D}_{i-1}$)
 6.          SplitTriangle1To3($T_i, \mathbf{p}_i$)
 7.          $\mathcal{S}$ = InitializeStarShapedPolygon($\mathbf{p}_i$)
 8.          while there exists a non-Delaunay edge $\mathbf{e}$ of $\mathcal{S}$
 9.              do
10.                  ◁ Flip $\mathbf{e}$ with the other diagonal ▷
11.                  ◁ (See quad-edge data structure of Section 5.3.4) ▷
12.                  EdgeFlip($\mathbf{e}$)
13.          ◁ $\mathcal{D}_i$ denotes the current Delaunay triangulation of $\mathcal{P}_i$ ▷
```

7.5 Randomized Incremental Optimization

The randomization analysis can also yield simple yet powerful combinatorial optimization algorithms. We consider the *smallest enclosing ball* problem that is a core primitive in many collision-detection algorithms. Indeed, the smallest enclosing ball (or a fast approximation of it) is often used in computer graphics for collision-detection algorithms that use bounding sphere hierarchies[8] (Figure 7.9). A *bounding sphere hierarchy* is a hierarchical covering of the 3D model by balls. For checking whether two 3D models collide or not, we first check whether their sphere coverings intersect or not. If not, clearly, the included models obviously do not intersect. Otherwise, if the current sphere coverings intersect then we refine the model sphere coverings. That is, we go down one level in the multiresolution hierarchy, if possible. At the bottommost level, we need to check whether the two 3D models intersect or not, a computationally expensive operation. Fortunately, this last stage model-model intersection check is seldom performed. Thus, the sphere covering of a 3D model provides a geometric proxy of its shape. Besides, the smallest enclosing ball is also used for hierarchical frustum culling as well. For ease of presentation, we consider the planar case. However, we use the term "ball" as the algorithm generalizes well to arbitrary dimension (say, up to dimension 30; see Section 8.3.2).

[8]The bounding sphere hierarchy is a particular case of *bounding volume hierarchies* (BVHs).

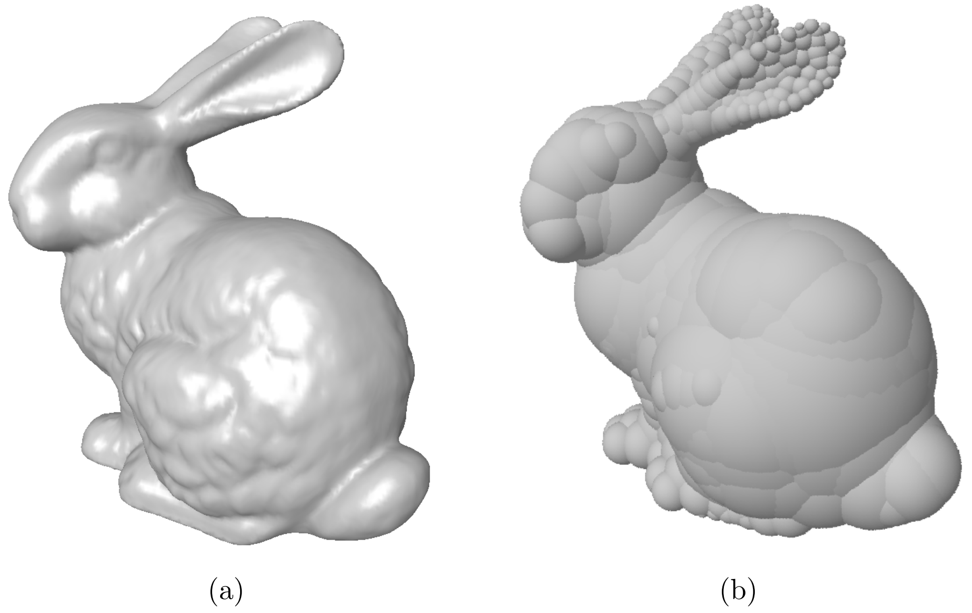


FIGURE 7.9 *Example of a bounding sphere hierarchy of a 3D* `bunny` *model. (a) shows the 3D model, and (b) a bounding sphere hierarchy used as a geometric proxy for efficient collision detection. The 3D* `bunny` *model is courtesy of © The Stanford 3D scanning repository, Marc Levoy. Used with permission.*

Let Ball($\mathbf{p}, r$) denote the ball of center $\mathbf{p}$ and radius r:

$$\mathrm{Ball}(\mathbf{p}, r) = \{\mathbf{x} \in \mathbb{R}^2 \mid ||\mathbf{px}|| \leq r\}. \tag{7.40}$$

Denote by $\mathcal{P}$ the point set $\mathcal{P} = \{\mathbf{p}_1, ..., \mathbf{p}_n\}$. The smallest enclosing ball of $\mathcal{P}$ is the *unique* ball $B^* = \mathrm{Ball}(\mathbf{c}^*, r^*)$, fully enclosing $\mathcal{P}$ ($\mathcal{P} \subseteq \mathrm{Ball}(\mathbf{c}^*, r^*)$), and of minimum radius r^*. Historically, Sylvester first studied the 2D instance in 1857. Since then, there have been many types of algorithms, mostly numerical though. A breakthrough was the exact combinatorial randomized algorithm of Welzl in 1991. We present the framework in 2D, although the approach extends to arbitrary dimensions and minimum enclosing ellipsoids as well. The smallest enclosing ball is defined by its *combinatorial basis*, in 2D, a set of *at most* three points lying on the boundary ∂B^* of the smallest enclosing ball B^*. Note that many more points can possibly lie on ∂B^*, in case of cocircular degeneracies ($\geq$ 4 points lying on a same circle). Figure 7.10 illustrates those different configurations. Figure 7.10(a) shows an instance where the basis is of size two (a pair of diametrically opposed points). Figure 7.10(b) and (c) respectively shows an instance of basis three in nondegenerated and degenerated cases.

Welzl proposed a simple combinatorial optimization algorithm whose randomized analysis yields an optimal linear expected time bound. The algorithm cleverly proceeds by determining a basis for the smallest enclosing ball.

We summarize this recursive MiniBall algorithm for the planar case, in pseudocode:

```
MiniBall($\mathcal{P} = \{\mathbf{p}_1, ..., \mathbf{p}_n\}, \mathcal{B}$)
 1.  ◁ Initially the basis $\mathcal{B}$ is empty ▷
 2.  ◁ Output: $\mathcal{B}$ contains a basis solution ▷
 3.  ◁ That is, two or three points defining the minimum enclosing ball ▷
 4.  ◁ Function MiniBall returns the smallest enclosing ball $B^*$ ▷
 5.  if $|\mathcal{B}| = 3$
 6.     then return $B$ = SolveBasis($\mathcal{B}$)
 7.     else
 8.          if $|\mathcal{P} \cup \mathcal{B}| \leq 3$
 9.             then return $B$ = SolveBasis($\mathcal{P} \cup \mathcal{B}$)
10.             else
11.                  Select at random point $\mathbf{p} \in \mathcal{P}$
12.                  $B$ = MiniBall($\mathcal{P} \backslash \{\mathbf{p}\}, \mathcal{B}$)
13.                  if $\mathbf{p} \notin B$
14.                     then ◁ Then $\mathbf{p}$ belongs to the boundary of $B^*(\mathcal{P})$ ▷
15.                          return $B$ = MiniBall($\mathcal{P} \backslash \{\mathbf{p}\}, \mathcal{B} \cup \{\mathbf{p}\}$)
```

The SolveBasis procedure has to determine the circle passing through either one (trivial case), two basis points $\mathbf{p}_1$ and $\mathbf{p}_2$, or three basis points $\mathbf{p}_1$, $\mathbf{p}_2$, and $\mathbf{p}_3$ (see Figure 7.10). For two points, because the basis pair lies diametrically opposed on the boundary of the smallest enclosing ball, its circumcenter is $\mathbf{c}^* = \frac{\mathbf{p}_1 + \mathbf{p}_2}{2}$. For three basis points, the circumcenter is the intersection of the three perpendicular bisectors defined by the triangle $\triangle \mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$. Thus, $\mathbf{c}^*$ necessarily lies inside[9] the triangle $\triangle \mathbf{p}_1\mathbf{p}_2\mathbf{p}_3$. Another way, to compute the circumcenter of the circle passing through three points is to use the determinant approach, as detailed in Section 8.3.1, or to solve a linear program (see 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: `MiniBall.cpp`

[9] In arbitrary dimension, it can be shown that the circumcenter of the smallest enclosing ball lies necessarily inside the convex hull of its basis points.

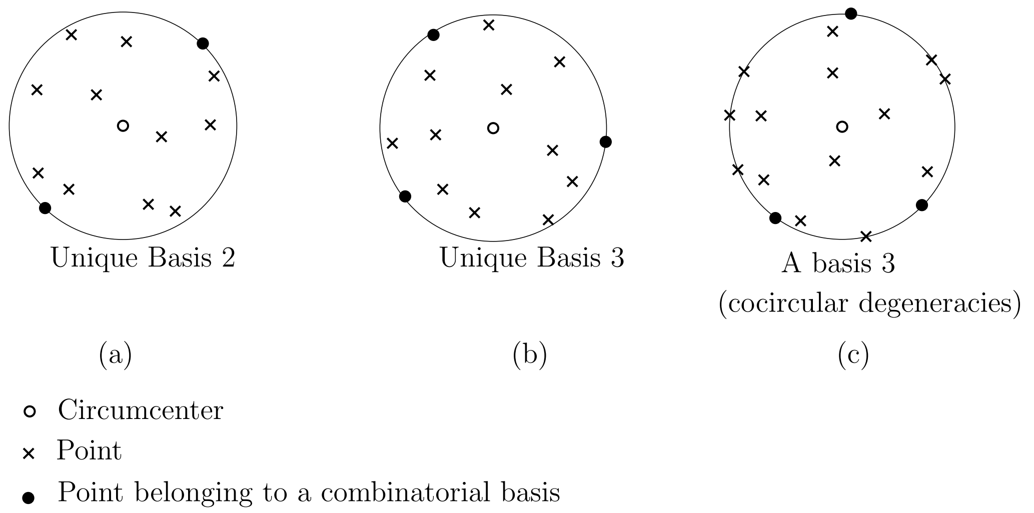

Unique Basis 2
Unique Basis 3
A basis 3
(cocircular degeneracies)
(a)
(b)
(c)
Circumcenter
Point
Point belonging to a combinatorial basis

FIGURE 7.10 *Smallest enclosing disk. The smallest enclosing disk B^* is defined by its combinatorial basis, of size 2 (a) or 3 (b) in the planar case. The basis is unique for point sets without cocircularities (a) and (b), and chosen arbitrarily otherwise (c).*

We give below a complete C++ implementation of the MiniBall code:

```
void MiniDisc(point* set, int n, point* basis, int b,
                              point& center, number& rad)
{
point cl; number rl;
int k;

// Terminal cases
if (b==3) SolveDisc3(basis[0], basis[1], basis[2], cl, rl);

if ((n==1)&&(b==0)) {rl=0.0; cl=set[0];}
if ((n==2)&&(b==0))  SolveDisc2(set[0],set[1], cl, rl);
if ((n==0)&&(b==2))  SolveDisc2(basis[0],basis[1], cl, rl);
if ((n==1)&&(b==1))  SolveDisc2(basis[0], set[0], cl, rl);

// General case
if ((b<3)&&(n+b>2))
  {
  // Randomization: choosing a pivot
  k=rand()%n; // between 0 and n-1
  if (k!=0) SwapPoint(set[0],set[k]);

  MiniDisc(&set[1],n-1,basis,b,cl,rl);

  if (cl.Distance(set[0])>rl)
    {
    // Then set[0] necessarily belongs to the basis.
      basis[b++]=set[0];
      MiniDisc(&set[1],n-1,basis,b,cl,rl);
    }
  }
  center=cl;rad=rl;
}
```

The C++ primitives for computing the disk passing through either two or three points is written as follows:

```
// The unique circle passing through exactly three non-collinear
    points p1, p2 ,p3
void SolveDisc3(point p1, point p2, point p3, point & center,
    number &rad)
{
number a = p2.x - p1.x;
number b = p2.y - p1.y;
number c = p3.x - p1.x;
number d = p3.y - p1.y;
number e = a*(p2.x + p1.x)*0.5 + b*(p2.y + p1.y)*0.5;
number f = (c*(p3.x + p1.x)*0.5) + (d*(p3.y + p1.y)*0.5);
number det = a*d - b*c;

center.x = (d*e - b*f)/det;
center.y = (-c*e + a*f)/det;

rad =p1.Distance(center);
}

// The smallest circle passing through two points
void SolveDisc2(point p1, point p2, point & center, number &rad)
{
center.x = 0.5*(p1.x+p2.x);
center.y = 0.5*(p1.y+p2.y);

rad =p1.Distance(center);
}
```

To analyze the time complexity of the procedure MiniBall, we proceed by upper-bounding the number of times we have some point $\mathbf{p} \notin B$ (since this event triggers a new recursive call).

Let $c_b(n)$ denote the number of recursive calls to MiniBall with an n-point set $\mathcal{P}$ and a b-point set basis $\mathcal{B}$. Trivially, we define $c_b(0) = 0$, for $2 \leq b \leq 3$. The probability that MiniBall$(\mathcal{P}, \mathcal{B}) \neq$MiniBall$(\mathcal{P}\backslash\{\mathbf{p}\}, \mathcal{B} \cup \{\mathbf{p}\})$ is bounded by at most $\frac{3-b}{n}$ (with $\frac{3-b}{n} \leq \frac{3}{n}$), since at most $3-b$ points belonging to the boundary of an enclosing ball are enough to define uniquely that ball already constrained to pass through b points. This yields the following simple recursion equation:

$$c_b(n) \leq \underbrace{c_b(n-1)}_{\text{MiniBall without } \mathbf{p}} + \underbrace{1}_{\text{current call}} + \underbrace{\frac{3-b}{n} c_{b+1}(n-1)}_{\text{Balls are different case}} . \qquad (7.41)$$

Solving this recursive equation yields to $c_3(n) \leq n$, $c_2(n) \leq 3n$, and $c_1(n) \leq 10n$. Therefore, the above randomized incremental combinatorial optimization procedure runs in expected $\tilde{O}(c_1(n) + c_2(n) + c_3(n)) = \tilde{\Theta}(n)$ optimal time and memory.

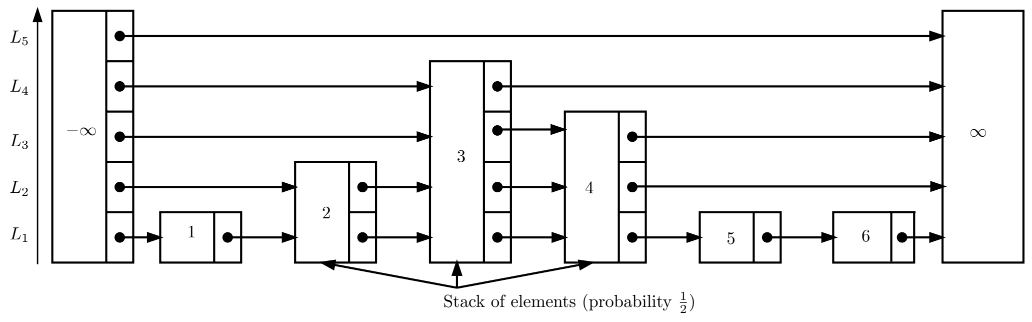
7.6 Skip Lists

The *skip list* is one of the data structures that has been designed to overcome the complexity of balanced binary search trees, such as red-black trees[10] or Adelson-Velskii-Landis trees,[11] commonly called AVL trees. A skip list is a *probabilistic data structure* pioneered by Pugh, in 1990. Consider a set $\mathcal{S}$ of n scalar elements $\{s_1, ..., s_n\}$ with $s_i \leq s_{i+1}$ for all $1 \leq i < n$ (ordered elements). Let us add two extrema: elements $-\infty$ and ∞, such that $-\infty < s_1$ and $s_n < \infty$ in $\mathcal{S}$. A skip list is a sparse bidimensional list, where cells store elements organized into levels. To build a skip list on $\mathcal{S}$, we first start by building the first level L_1 of the skip list as a singly connected list on all elements of $\mathcal{S}$. Then, we build a next level L_{i+1} by independently tossing a coin for each list element of L_i. For one face of the coin, we let the element survive to that level L_{i+1}, otherwise we discard it for *all* the remaining layers L_j, with $j \geq i+1$. We add the two extremal elements to all layers, and we link each cell of layer L_{i+1} with its corresponding cell at layer L_i. Because those corresponding cells store identical elements, we rather store the element once. That is, for each element, we create a *tower structure* storing only once the element value and we associate to that tower an array of links. For each layer of the tower, we point to the immediate successor tower of that level, as depicted in Figure 7.11. We reiterate until no more elements are available.

Let $L_1, ..., L_k$ be the k levels of the skip list (see Figure 7.11). A skip list can be interpreted as an interval search tree data structure (but not necessarily binary tree), where levels successively represent coarser partitions of the real

[10]Red-black trees are binary search trees such that every node is colored either in red or black, and stores a value. The root node is colored black. Red-black trees satisfy the following properties: (1) for each node, its value is greater than the value of its left child and less than the value of its right child. (2) Every path from the root to a leaf contains the same number of black nodes. (3) Every red node has only black children. It can be shown that red-black trees have logarithmic depths.

[11]AVL trees are binary search trees such that for each node, the heights of its right and its left subtrees differ by at most one. The height of leaves is one. AVL trees have logarithmic depths.



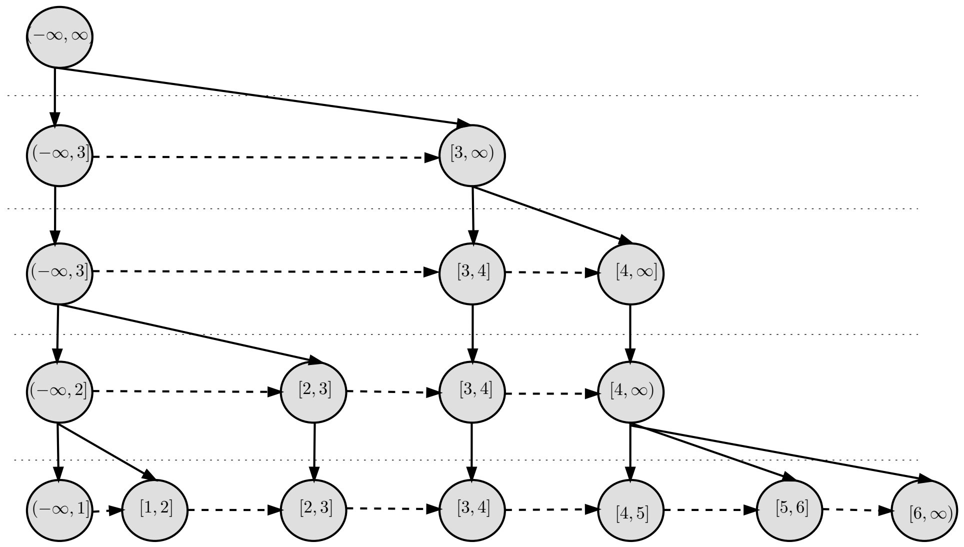


FIGURE 7.11 *Example of a skip list. The top figure explains the skip list construction. The bottom figure is a visual interpretation of the* implicit *interval tree representation of the above skip list. Observe that the implicit tree is not binary (degree* 3 *of the node* $[4,\infty)$*).*

interval $(-\infty,\infty)$ (Figure 7.11). First, we create for each layer a node for each element present at that level. We then define the interval range of a node: each node has an associated range beginning from its element and ending to the following element of the same layer. Then, two nodes of consecutive levels are linked if, and only if, their corresponding intervals intersect. Observe that the degree of nodes in the interval tree can be arbitrarily high. (Fortunately, the randomization construction yields expected degree two, on average.)

A Find operation localizes an element s by starting from the highest level and successively querying for its position on the singly connected list of that level. Once the node interval where s lies in has been identified, we branch to the lower level of the appropriate child, and so on until the element is found, or proved not to belong to the skip list.

The overall cost of a FIND query is proportional to the number of visited nodes. At a given level L_i, it is proportional to the number of cells visited on the list, which is probabilistically bounded by the *expected* number of children of *any* cell at level L_{i+1} (the parent node where we come from). Thus, the time complexity t of a FIND procedure is written as:

$$\tilde{t} \leq O\left(\sum_{i=1}^{k}(1+\mathrm{ExpectedVisitedChildren(L_i)})\right). \tag{7.42}$$

It turns out that a randomization analysis shows that $\tilde{t}$ is logarithmic. That is $t = \tilde{O}(\log n)$. Similarly, INSERT and DELETE query operations can be defined.

Basically, the running time of query primitives on skip lists depend on the number of layers k (see Eq. 7.42). With *high probability*, let us prove that $k = \tilde{O}(\log n)$. Indeed, consider any element s and its stack of cells from level L_1 to level L_l with $l = \mathrm{level}(s)$. All levels $\mathrm{level}(s_i)$ ($s_i \in \mathcal{S}$) are interpreted as *independent and identically distributed* (iid) random variables $\hat{S}_i$ distributed geometrically with parameter $\frac{1}{2}$ (one of the two sides of a coin). Thus, we have:

$$\mathbf{Pr}[\max_i \hat{S}_i > l] \leq \frac{n}{2^l}. \tag{7.43}$$

Plugging $l = c\log n$ and setting $k = \max_i \hat{S}_i$, we get:

$$\mathbf{Pr}[k > c\log n] \leq \frac{n}{2^{c\log n}} \leq \frac{1}{n^{c-1}}. \tag{7.44}$$

That is, the maximum level of elements of $\mathcal{S}$ is of order $\log n$ with high probability. A similar result holds for the expected level, and the expected memory requirement is also shown linear.

Now, we are ready to prove with high probability that *any* FIND query is performed in $\tilde{O}(\log n)$ expected time. Since we have already shown with high probability that the number of levels of the skip list is $\tilde{O}(\log n)$, it remains to bound the time spent at each level, as written in Eq. 7.42. The key observation is to notice that $\mathrm{ExpectedVisitedChildren} = \frac{1}{\frac{1}{2}} = 2$. More precisely, it is $\frac{1}{p}$ if we consider to let survive an element to the next level with probability p. Indeed, the expected number of siblings at a node is $\frac{1}{p}$ because for a node to be a sibling it must have failed to advance to the next level of the skip list. Thus, ExpectedVisitedChildren is the same as asking the number of times we need to flip a coin to get a heads: $\mathrm{ExpectedVisitedChildren} = \frac{1}{p}$. Plugging this result in Eq. 7.42, we prove that any FIND query costs $\tilde{O}(\log n)$ time, with high probability (and for any fixed p). Similarly, Insert and Delete primitives require only expected logarithmic time. To conclude, randomized skip lists provide an elegant and efficient alternative to deterministic balanced search trees. Moreover, skip lists can be made deterministic too (see the Bibliographical Notes).

7.7 Bibliographical Notes

Randomization is a subarea of computer science that has its own devoted textbooks. The book by Motwani and Raghavan [233] surveys with a wide scope the many different aspects of randomization. Randomization in computational geometry is detailed in the book by Mulmuley [236]. The paper by Clarkson and Shor [82] further explains the random sampling in computational geometry.

The QuickSort sorting algorithm is due to Hoare [165]. Sedgewick wrote a note [290] on implementing QuickSort. We recommend the book by Flajolet and Sedgewick [291] for rigorous analysis of algorithms. The book by Skiena [313] explains how to generate a random permutation of an array of elements. We chose the QuickSort algorithm to define the Las Vegas randomized algorithms because of its simplicity, and its widespread use in practice. The randomized SelectElement algorithm was described in 1961 by Hoare [166]. The first deterministic linear-time algorithm for selection is due to Floyd et al. [37]. Readers interested in order statistics may check the paper by Cunto and Munro [90] that describes a better method for choosing pivots when selecting the k-th smallest element of an array. They achieve a remarkable $n+k+O(1)$ comparisons on average for the selection algorithm. Proving lower bounds of problems is a difficult task. Ben-Or [26] describes a general topological method for obtaining lower bounds for the height of algebraic decision trees. In randomized analysis, we often need to approximate probabilities of the tails or spreads of general random variables by inequalities. See the inequalities explained in [236] by Markov, Chebychev and Chernoff [277]. For example, Chernoff bound allows us to prove that QuickSort runs in $\tilde{O}(n \log n)$ with high probability.

The random sample consensus was first explained by Fisher and Bolles, in their seminal paper [115]. The technique has further been combinatorically improved by Irani et al. [171]. Nielsen [245] also describes how random samples can be efficiently geometrically filtered. Since its inception in the early 1980s, many other robust estimators [316] have been studied in computer vision: M-estimator [191], least median of squares [234] (LMedS), iterative closest point [33] (ICP), and so on. Brown and Lowe [61] showed how to use Scale Invariant Feature Transform (SIFT) features [212] with RANSAC for automatically recognising and creating panoramas from a large set of unordered pictures. A demo program `autostitch` is available online at *http://www.cs.ubc.ca/~mbrown/autostitch/autostitch.html*. Automatic stitching has numerous applications for stabilizing, summarizing, compressing or pulling out alpha mattes of videos.

The Monte Carlo method is used a lot in computer graphics where complex integrals need to be approximated fast. The seminal ideas appeared in the "Rendering Equation" paper [177] by Kajiya. The Ph.D. theses by Veach [332] and Lafortune [190] also give plenty details of the Monte Carlo principles in computer graphics. Finally, a state of the art tutorial was given at the ACM SIGGRAPH 2004 [103]. Let us mention the bidirectional path-tracing [190] and the Metropolis light transport [332] as two main extensions of the ordinary Monte Carlo method. We recommend the book by Pharr and Humphreys [262] for an introduction to physically based rendering, and the book by Jensen on photon mapping [174]. The recent paper by Ostromoukhov et al. [256] provides a fast hierarchical importance Monte Carlo sampling algorithm with applications to computer graphics. Chazelle's book [75] on discrepancy is a must for anyone interested in knowing more about distribution irregularities that provide some hints on the discrete/continuous sampling gap.

The incremental construction of the Delaunay triangulation is due to Guibas et al. [152]. For ease of presentation, we added three auxiliary bounding triangle vertices $\mathbf{t}_1$, $\mathbf{t}_2$, and $\mathbf{t}_3$ at the initialization step of the incremental Delaunay triangulation. A better alternative approach consists in adding a single point at infinity (like in the CGAL triangulation package[12]). In practice, the history DAG has been superseded by other localization methods, such as walk, jump and walk, and the Delaunay hierarchy structures. Devillers et al. [99] further explain those methods and their relatives. Historically, Boissonnat and Teillaud invented in 1986 the Delaunay tree [47, 48], a randomized data structure to compute and update the Delaunay triangulation that supports queries. (See also the related Influence DAG (I-DAG) structure and its randomized analysis [43].) The incremental randomized Delaunay triangulation has further been investigated and refined by Devillers in [97]. Su and Drysdale provide detailed comparisons on several Delaunay algorithms [320]. The *backward analysis* of randomization algorithms is due to Seidel [292]. Chew [79] gave a simple algorithm for the randomized construction of the Voronoi diagram and (dual) Delaunay triangulation of a convex polygon, in expected linear time. Further, the Voronoi diagrams of a few selected generators can be constructed in an output-sensitive way [244]. Another appropriate choice for illustrating randomization in computational geometry would have been to consider the randomized autopartition constructions [126, 127] of binary space partitions useful in hidden surface removal [260] (painter's algorithm) or CSG modeling [243].

The seminal MiniBall ingenious algorithm of Welzl [343] extends to arbitrary dimension and ellipsoids as well. The running time generalizes to $O(b!bn)$, where n is the input size and b is the maximum basis size. In d-dimensional space, we have $b = d + 1$ for balls, and $b = \frac{d(d+3)}{2}$ for ellipsoids. Gärtner explained in his

[12]Visit the Web site *http://www.cgal.org*.

implementation [133] how to use a linear program to solve for the circumcenter of the smallest enclosing ball tangent to exactly k balls in arbitrary dimension ($2 \leq k \leq d+1$). Sharir and Welzl extended this kind of combinatorial optimization paradigm into the class of LP-type [220], where LP is a shorthand for linear programming. Fischer and Gärtner [114] further investigated the d-dimensional case for ball sets for small instances $n = O(d)$, and designed a new LP-type algorithm. Nielsen and Nock [248] classified the heuristics for solving or approximating the smallest enclosing ball into three categories: combinatorial, numerical, and hybrid combinatorial/numerical. A fast and robust implementation of the smallest enclosing ball is available at *http://www.inf.ethz.ch/personal/gaertner/miniball.html* or in the CGAL library. Applications of the smallest enclosing balls are found in computer graphics for collision detection based on bounding sphere hierarchies [173, 53]. A toolkit for building bounding sphere hierarchies of 3D models is available online at *http://isg.cs.tcd.ie/spheretree/*. Guibas et al. [2] recently provided the first subquadratic algorithm for collision detection using so-called deformable necklaces (that are special sphere coverings) in arbitrary fixed dimension.

The skip list is a probabilistic data structure [273] that was invented by Pugh in 1990. It was later made deterministic by Munro et al. [240]. Another useful randomized data structure is the random treap [293], which has also the merit of having a very short code for simulating common operations found on complex implementations (and therefore more prone to bugs) of balanced binary search trees. Note that the CGAL Delaunay hierarchy [42] makes use of a few levels of intermediate Delaunay triangulations for efficient localization, a technique similar in spirit to the skip lists.

Sometimes, it happens that randomization yields algorithms that theoretically or practically perform better. We then would like to get the same benefits without having to rely on randomness. This process is called *derandomization* [55] and is tackled using the discrepancy theory [75].

Unfortunately, computers do not create pure random numbers but only *calculate* pseudorandom numbers. Pseudorandom numbers are only almost random, but not perfectly random. A good introduction to random number generations is provided by Knuth [184].