CHAPTER 6

Texture

Texture is a phenomenon that is widespread, easy to recognise, and hard to define. Typically, whether an effect is referred to as texture or not depends on the scale at which it is viewed. A leaf that occupies most of an image is an object, but the foliage of a tree is a texture. Views of large numbers of small objects are often best thought of as textures. Examples include grass, foliage, brush, pebbles, and hair. Many surfaces are marked with orderly patterns that look like large numbers of small objects. Examples include the spots of animals such as leopards or cheetahs; the stripes of animals such as tigers or zebras; the patterns on bark, wood, and skin. Textures tend to show *repetition*: (roughly!) the same local patch appears again and again, though it may be distorted by a viewing transformation.

Texture is important, because texture appears to be a very strong cue to object identity. Most modern object recognition programs are built around texture representation machinery of one form or another. This may be because texture is also a strong cue to *material properties*: what the material that makes up an object is like. For example, texture cues can be used to tell tree bark (which is moderately hard and rough) from bare metal (which is hard, smooth, and shiny). People seem to be able to predict some mechanical properties of materials from their appearance. For example, often you can distinguish somewhat viscous materials, like hand cream, from highly viscous materials, like cream cheese, by eye (Adelson 2001). Material properties are correlated to the identity of objects, but they are not the same thing. For example, although hammers are commonly made of metal, a plastic hammer, a metal hammer, and a wooden hammer are all still hammers.

There are three main kinds of texture representation. *Local texture representations* encode the texture very close to a point in an image. These representations can't be comprehensive, because they look at a small piece of the image. However, they are very useful in image segmentation, where we must break an image into large, useful components, usually called *regions* (the details of what makes a region useful are deferred to Chapter 9). One reasonable requirement is that points inside a region look similar to one another, and different from points outside the region, and segmentation algorithms need a description of the appearance close to the point to impose this requirement. Local texture representations are described in Section 6.1.

Other problems require a description of the texture within an image domain. We refer to such representations as *pooled texture representations*. For example, **texture recognition** is the problem of determining what texture is represented by a patch in an image. Here we have a domain (the patch) and we want a representation of the overall texture in the domain. Similarly, in **material recognition**, one must decide what material is represented by a patch in the image. Section 6.2 describes methods for building pooled texture representations.

Data-driven texture representations model a texture by a procedure that can

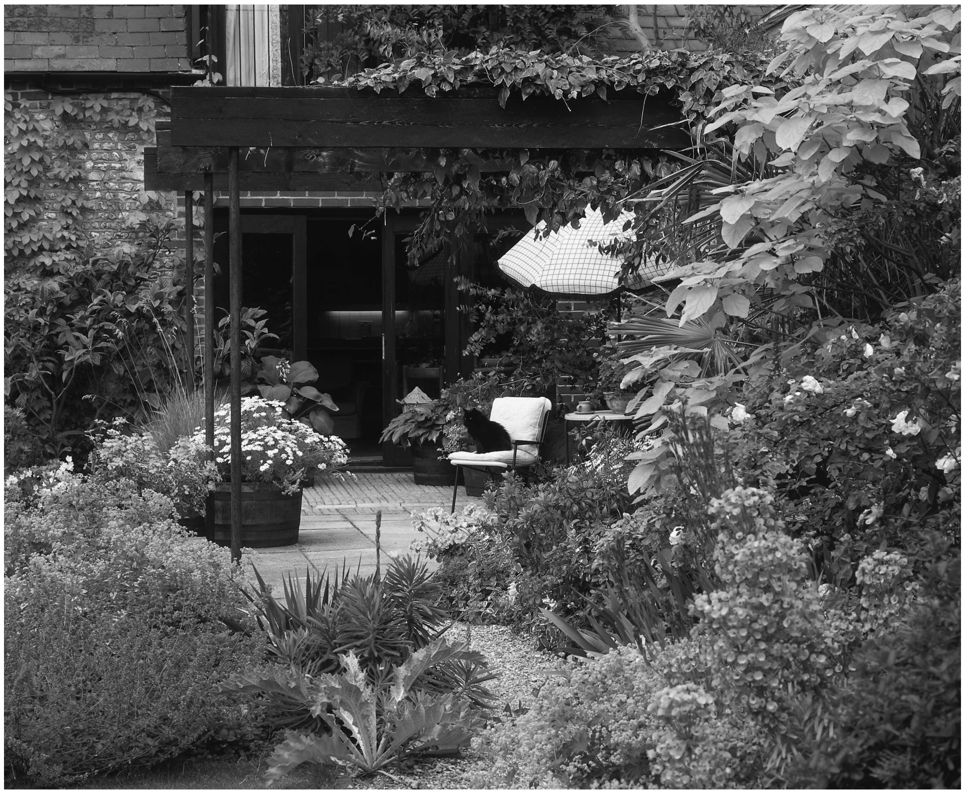

FIGURE 6.1: Although texture is difficult to define, it has some important and valuable properties. In this image, there are many repeated elements (some leaves form repeated "spots"; others, and branches, form "bars" at various scales; and so on). Our perception of the material is quite intimately related to the texture (what would the surface feel like if you ran your fingers over it? what is soggy? what is prickly? what is smooth?). Notice how much information you are getting about the type of plants, their shape, the shape of free space, and so on, from the textures. *Geoff Brightling © Dorling Kindersley, used with permission.*

generate a textured region from an example. These representations are not appropriate for segmentation or recognition applications, but are tremendously valuable for texture synthesis. In this problem, we must create regions of texture, for example, to fill holes in images (Section 6.3).

The texture on a surface can be a strong cue to its shape. If the texture is "the same" over the surface, then deformation of the texture from point to point can be a cue to the shape of the surface. For example, if we have a perspective view of an inclined plane with spots on it, the spots will be smaller closer to the horizon in the image. This can be used to recover the inclination of the plane. Similarly, on a curved surface, the foreshortening of texture elements gives some information about the local inclination of the surface. Recovering surface orientation or surface shape from an image texture is known as **shape from texture**; solutions to this

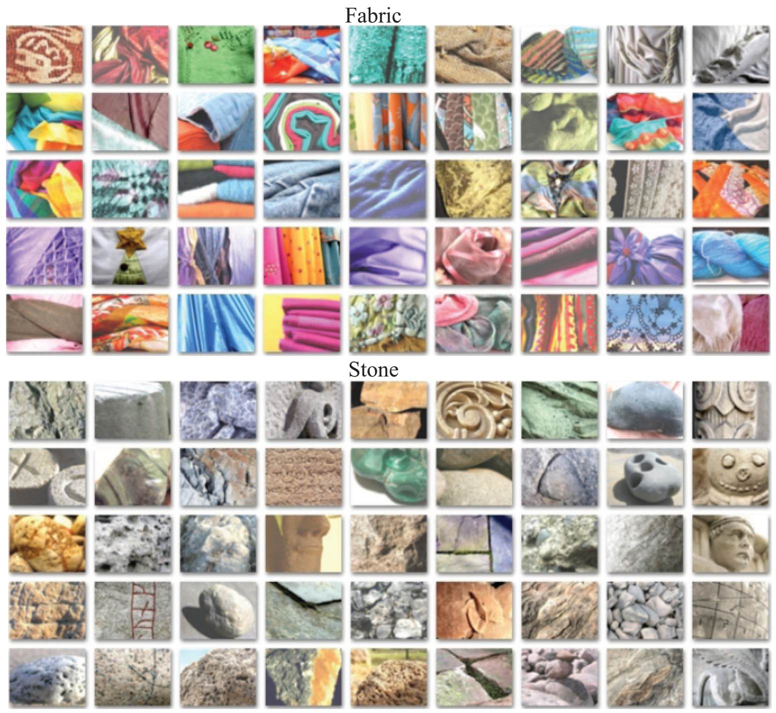

FIGURE 6.2: Typically, different materials display different image textures. These are example images from a collection of 1,000 material images, described in by Sharan *et al.* (2009); there are 100 images in each of the ten categories, including the two categories shown here (fabric and stone). Notice how (a) the textures vary widely, even within a material category; and (b) different materials seem to display quite different textures. *This figure shows elements of a database collected by C. Liu, L. Sharan, E. Adelson, and R. Rosenholtz, and published at* `http://people.csail.mit.edu/lavanya/research_sharan.html`. *Figure by kind permission of the collectors.*

problem tend to use straightforward representations of texture together with strong constraints on the overall structure of the texture (Section 6.5).

6.1 LOCAL TEXTURE REPRESENTATIONS USING FILTERS

Image textures generally consist of repeated elements; an element is sometimes called a *texton*. For example, some of the fabric textures in Figure 6.2 consist of triangles of wool formed by the knit pattern. Similarly, some stone textures in that figure consist of numerous, near-circular, gray blobs. It is natural to represent a texture with some description of (a) what the textons are and (b) how they repeat. Notice that it is difficult to be precise about what a texton is, because if a large pattern repeats frequently, then so do its parts. This presents no major problems, because we do not need to extract textons accurately. Instead, what we need are representations that differ in ways that are easy to observe when two textures are significantly different. We can do this by assuming that all textons are made of

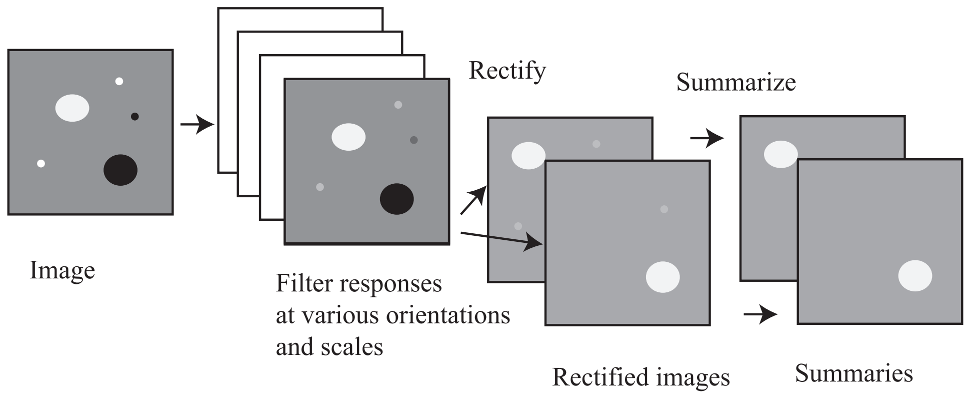

FIGURE 6.3: Local texture representations can be obtained by filtering an image with a set of filters at various scales, and then preparing a summary. Summaries ensure that, at a pixel, we have a representation of what texture appears near that pixel. The filters are typically spots and bars (see Figure 6.4). Filter outputs can be enhanced by rectifying them (so that positive and negative responses do not cancel), then computing a local summary of the rectified filter outputs. Rectifying by taking the absolute value means that we do not distinguish between light spots on a dark background and dark spots on a light background; the alternative, half-wave rectification (described in the text), preserves this distinction at the cost of a fuller representation. One can summarize either by smoothing (which will tend to suppress noise, as in the schematic example above) or by taking the maximum over a neighborhood. Compare this figure to Figure 6.7, which shows a representation for a real image.

generic subelements, such as spots and bars. We find subelements with filters, then represent each point in the image with a summary of the pattern of subelements nearby. This will work because the parts of a texton repeat in the same way that the texton does.

This suggests representing image textures in terms of the response of a collection of filters. Each filter is a detector for a subelement. The collection of different filters would represent subelements—spots and bars are usual—at a collection of scales (to identify bigger or smaller subelements). We can now represent each point in an image by the vector of filter outputs at that point. This vector gives a sense of how much the neighborhood around that point looks like each subelement at each scale (Figure 6.3).

6.1.1 Spots and Bars

But what filters should we use? There is no canonical answer. A variety of answers have been tried. By analogy with the human visual cortex, one could use some spot filters, some oriented edge filters, and some oriented bar filters at different orientations and scales (Figure 6.4). This seems like a natural choice, because these are in some sense “minimal” subelements. It would be hard to have subelements of patterns with less structure than a spot, and it would be hard to have oriented subelements with less structure than an edge.

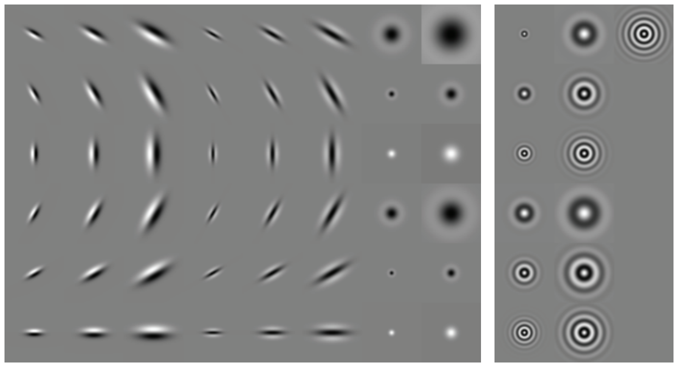

FIGURE 6.4: **Left** shows a set of 48 oriented filters used for expanding images into a series of responses for texture representation. Each filter is shown on its own scale, with zero represented by a mid-gray level, lighter values being positive, and darker values being negative. The left three columns represent edges at three scales and six orientations; the center three columns represent stripes; and the right two represent two classes of spots (with and without contrast at the boundary) at different scales. This is the set of filters used by Leung and Malik (2001). **Right** shows a set of orientation-independent filters, used by Schmid (2001), using the same representation (there are only 13 filters in this set, so there are five empty slots in the image). The orientation-independence property means that these filters look like complicated spots.

In some applications, we would like texture recognition performance to be unaffected if the texture sample is rotated. This is difficult to achieve with oriented filters, because one might need to sample the orientations very finely. An alternative to using oriented filters is to use filters that are orientation-independent, all of which must look like complicated spots (Figure 6.4).

6.1.2 From Filter Outputs to Texture Representation

Assume we have an image $\mathcal{I}$. A set of filter output maps (which would have the form $\mathcal{F}_i * * \mathcal{I}$ for different filters $\mathcal{F}_i$) is not, in itself, a representation of texture. The representation tells us what the window around a pixel looks like; but in a texture, what counts is not only what's at a pixel, but also what's nearby. For example, a field of yellow flowers may consist of many small yellow spots with some vertical green bars. What's important is not just the fact that a particular pixel looks a lot like a spot, but also that near the pixel there are no other spots, but some bars. This means the texture representation at a point should involve some kind of summary of nearby filter outputs, rather than just the filter outputs themselves.

The first step in building a reasonable summary is to notice that the summary must represent a neighborhood around a pixel that is rather bigger than the scale of the filter. To know whether the neighborhood around a pixel is "spotty," it is not enough to know that there is one strong spot in it; there should be many spots, each quite small compared to the size of the patch. However, it is important not to look at too large a neighborhood, or else the representation will not change much as we move across the image (because the neighborhoods overlap). The particular

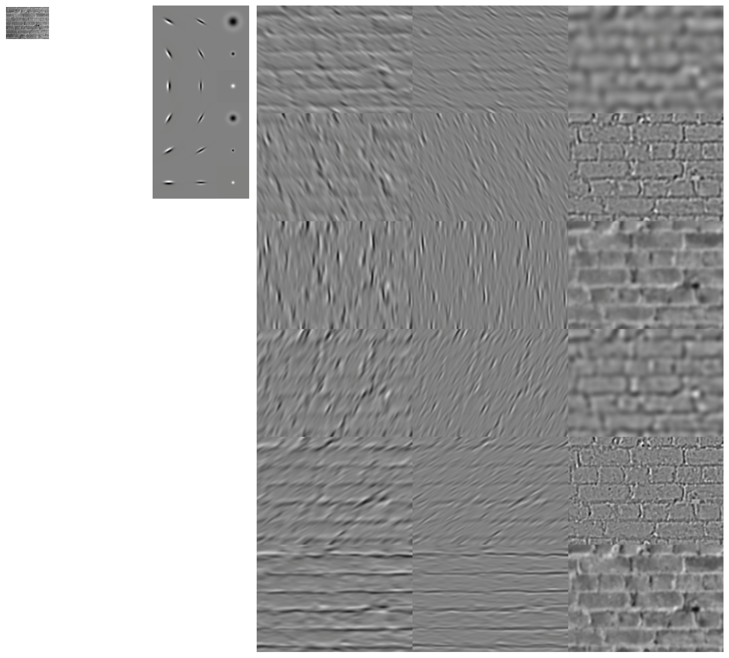

FIGURE 6.5: Filter responses for the oriented filters of Figure 6.4, applied to an image of a wall. At the **center**, we show the filters for reference (but not to scale, because they would be too small to resolve). The responses are laid out in the same way that the filters are (i.e., the response map on the top left corresponds to the filter on the top left, and so on). For reference, we show the image at the **left**. The image of the wall is small, so that the filters respond to structures that are relatively large; compare with Figure 6.6, which shows responses to a larger image of the wall, where the filters respond to smaller structures. These are filters of a fixed size, applied to a small version of the image, and so are equivalent to large-scale filters applied to the original version. Notice the strong response to the vertical and horizontal lines of mortar between the bricks, which are at about the scale of the bar filters. All response values are shown on the same intensity scale: lighter is positive, darker is negative, and mid-gray is zero.

arrangement of these spots within a neighborhood doesn't matter all that much, because the patch is small. This suggests that some form of average could give a fair description of what is going on; an alternative is to take the strongest response. We must process the responses before we summarize them. For example, a light spot filter will give a positive response to a light spot on a dark background, and a negative response to a dark spot on a light background. As a result, if we simply average filter responses over a patch, then a patch containing dark and light spots might record the same near-zero average as a patch containing no spots. This would be misleading.

We could compute the absolute value for each output map, to get $|\mathcal{F}_i * *\mathcal{I}|$. This does not distinguish between light spots on a dark background and dark spots on a light background. An alternative, which does preserve this distinction, is to report both $\max(0, \mathcal{F}_i * *\mathcal{I}(x, y))$ and $\max(0, -\mathcal{F}_i * *\mathcal{I}(x, y))$ (this is half-wave rectifi-

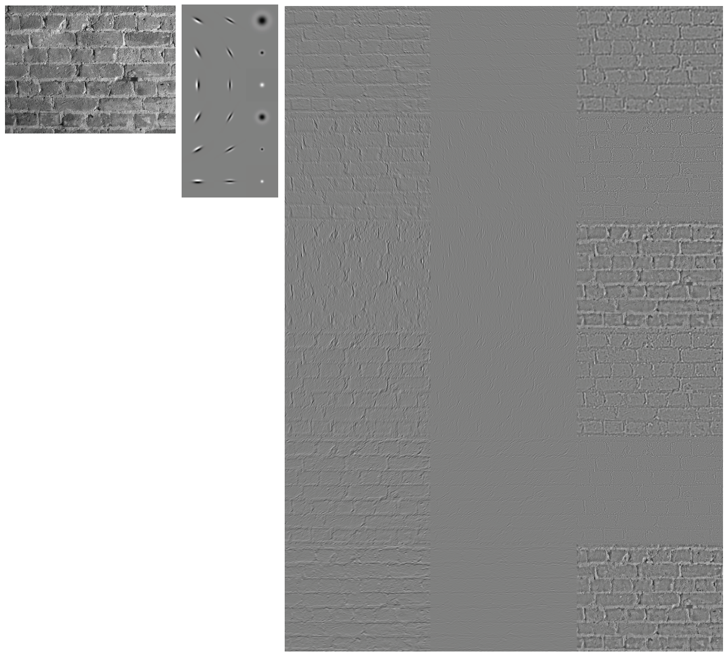

FIGURE 6.6: Filter responses for the oriented filters of Figure 6.4, applied to an image of a wall. At the **center**, we show the filters for reference (not to scale). The responses are laid out in the same way that the filters are (i.e., the response map on the top left corresponds to the filter on the top left, and so on). For reference, we show the image at the **left**. Although there is some response to the vertical and horizontal lines of mortar between the bricks, it is not as strong as the coarse scale (Figure 6.5); there are also quite strong responses to texture on individual bricks. All response values are shown on the same intensity scale: lighter is positive, darker is negative, and mid-gray is zero.

cation), which yields two maps per filter. We can now summarize the neighborhood around a pixel by computing a Gaussian weighted average (equivalently, convolving with a Gaussian). The scale of this Gaussian depends on the scale of the filter for the map; typically, it is around twice the scale of the filter.

6.1.3 Local Texture Representations in Practice

Several different sets of filters have been used for texture representation. The Visual Geometry Group at Oxford publishes code for different sets of filters, written by Manik Varma and by Jan-Mark Guesebroek, at `http://www.robots.ox.ac.uk/~vgg/research/texclass/filters.html`; this is part of an excellent web page on texture classification (`http://www.robots.ox.ac.uk/~vgg/research/texclass/index.html`). One important part of filtering an image with a large number of filters is doing so quickly; recent code for this purpose, by Jan-Mark Guesebroek, can be found at `http://www.science.uva.nl/research/publications/2003/GeusebroekTIP2003/`. Some sets of oriented filters allow fast, efficient representations and have good translation and rotation properties. One such set is

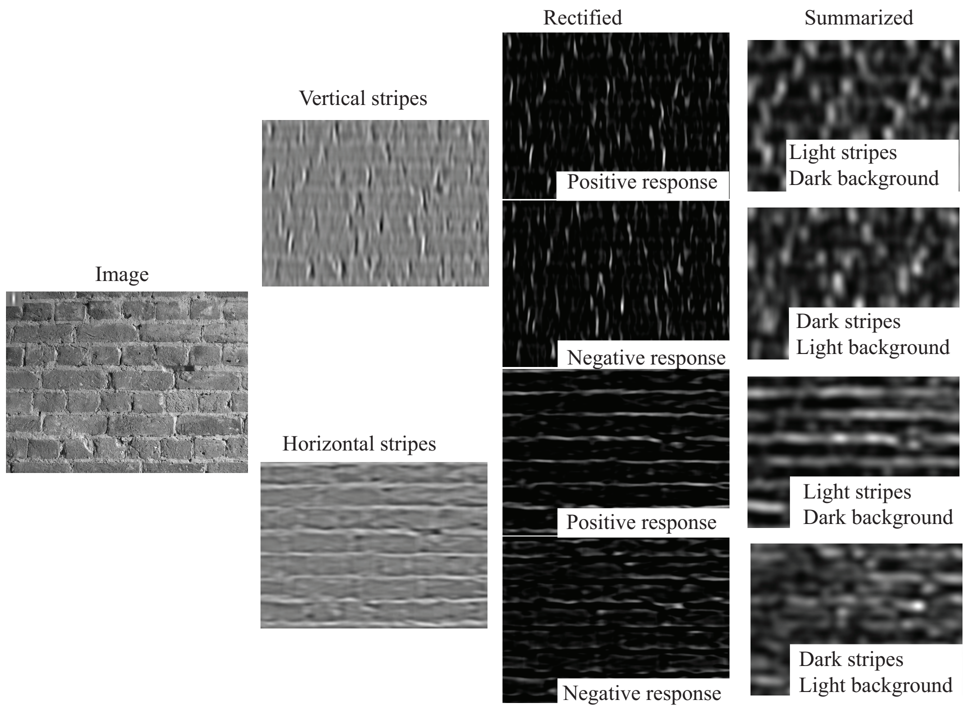

FIGURE 6.7: Filter-based texture representations look for pattern subelements such as oriented bars. The brick image on the **left** is filtered with an oriented bar filter (shown as a tiny inset on the top left of the image at full scale) to detect bars, yielding stripe responses (**center left**; negative is dark, positive is light, mid-gray is zero). These are rectified (here we use half-wave rectification) to yield response maps (**center right**; dark is zero, light is positive). In turn, these are summarized (here we smoothed over a neighborhood twice the filter width) to yield the texture representation on the **right**. In this, pixels that have strong vertical bars nearby are light, and others are dark; there is not much difference between the dark and light vertical structure for this image, but there is a real difference between dark and light horizontal structure.

the steerable pyramid of Simoncelli and Freeman (1995a). Code for these filters is available at `http://www.cns.nyu.edu/~eero/steerpyr/`.

6.2 POOLED TEXTURE REPRESENTATIONS BY DISCOVERING TEXTONS

A texture is a set of textons that repeat in some way. We could find these textons by looking for image patches that are common. An alternative is to find sets of texton subelements—that is, vectors of filter outputs—that are common (if textons are repeated, then so are their subelements). There are two important difficulties in finding image patches or vectors of filter outputs that commonly occur together. First, these representations of the image are continuous. We cannot simply count how many times a particular pattern occurs, because each vector is slightly different. Second, the representation is high dimensional in either case. A patch around a

```
Obtain a set of $n$ filters representing subelements, at multiple scales
Apply each filter $\mathcal{F}_i$ to the image
For each filter response map $\mathcal{F}_i ** \mathcal{I}$, compute
    $\max(0, \mathcal{F}_i ** \mathcal{I}(x,y))$ and $\max(0, -\mathcal{F}_i ** \mathcal{I}(x,y))$
For each of the $2n$ rectified maps, compute local summaries
    either by convolving with a Gaussian of scale approximately twice the
    scale of the base filter, or by taking the maximum value over that radius.
```

Algorithm 6.1: Local Texture Representation Using Filters.

pixel might need hundreds of pixels to represent it well; similarly, hundreds of different filters might be needed to represent the image at a pixel. This means we cannot build a histogram directly, either, because it will have an unmanageable number of cells.

6.2.1 Vector Quantization and Textons

Vector quantization is a strategy to deal with these difficulties. Vector quantization is a way of representing vectors in a continuous space with numbers from a set of fixed size. We first build a set of clusters out of a training set of vectors; this set of clusters is often thought of as a dictionary. We now replace any new vector with the cluster center closest to that vector. This strategy applies to vectors quite generally, though we will use it for texture representation. Many different clusterers can be used for vector quantization, but it is most common to use k-means or one of its variants. For concreteness, we describe this algorithm in Section 6.2.2, but the other clusterers of Chapter 9 would apply.

We can now represent a collection of vectors as a histogram of cluster centers. This general recipe can be applied to texture representation by describing each pixel in the domain with some vector, then vector quantizing and describing the domain with the histogram of cluster centers. Natural vectors to use are: the local summary representation described in Section 6.1; a vector of unprocessed filter outputs, using filters appropriate for a local texture representation (Figure 6.9); or even just a vector obtained by reshaping the pixels from a fixed-size patch around the image pixel (Figure 6.10). In each case, we are building a representation in terms of commonly repeated pattern elements.

6.2.2 K-means Clustering for Vector Quantization

We could use any clustering method to vector quantize (Chapter 9 describes a number of different clustering methods in the context of segmentation). However, by far the most common method used is *k-means* clustering. Assume we have a set of data items that we wish to cluster. We now assume that we know how many clusters there are in the data, which we write k. This is equivalent to fixing the number of values we wish to quantize to. Each cluster is assumed to have a center; we write the center of the ith cluster as $\boldsymbol{c}_i$. The jth data item to be clustered is described by a feature vector $\boldsymbol{x}_j$. In our case, these items are vectors of filter

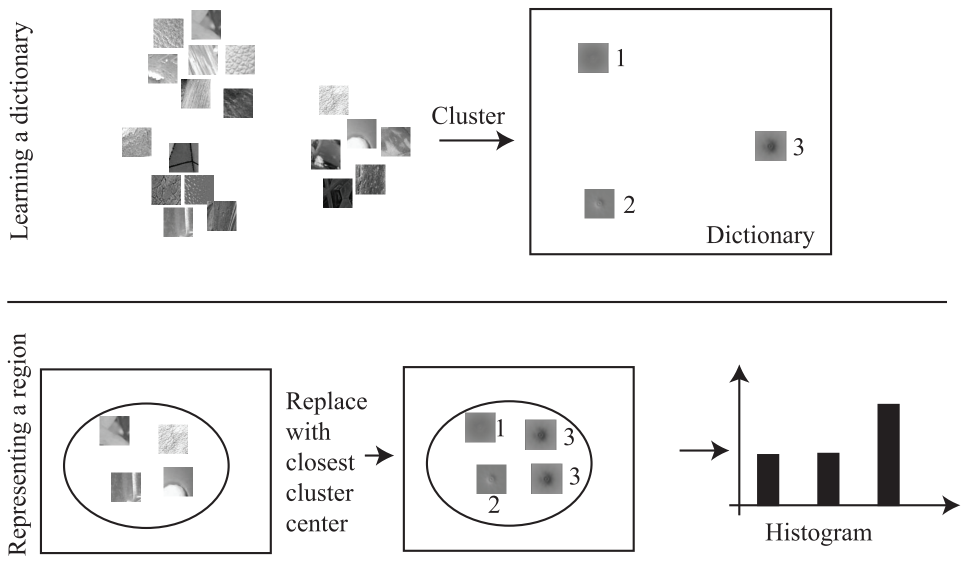

FIGURE 6.8: There are two steps to building a pooled texture representation for a texture in an image domain. First, one builds a dictionary representing the range of possible pattern elements, using a large number of texture patches. This is usually done in advance, using a training data set of some form. Second, one takes the patches inside the domain, vector quantizes them by identifying the number of the closest cluster center, then computes a histogram of the different cluster center numbers that occur within a region. This histogram might appear to contain no spatial information, but this is a misperception. Some frequent elements in the histogram are likely to be textons, but others describe common ways in which textons lie close to one another; this is a rough spatial cue. *This figure shows elements of a database collected by C. Liu, L. Sharan, E. Adelson, and R. Rosenholtz, and published at `http://people.csail.mit.edu/lavanya/research_sharan.html`. Figure by kind permission of the collectors.*

```
Build a dictionary:
    Collect many training example textures
    Construct the vectors $\boldsymbol{x}$ for relevant pixels; these could be
      a reshaping of a patch around the pixel, a vector of filter outputs
      computed at the pixel, or the representation of Section 6.1.
    Obtain $k$ cluster centers $\boldsymbol{c}$ for these examples

Represent an image domain:
    For each relevant pixel $i$ in the image
        Compute the vector representation $\boldsymbol{x}_i$ of that pixel
        Obtain $j$, the index of the cluster center $\boldsymbol{c}_j$ closest to that pixel
        Insert $j$ into a histogram for that domain
```

Algorithm 6.2: Texture Representation Using Vector Quantization.

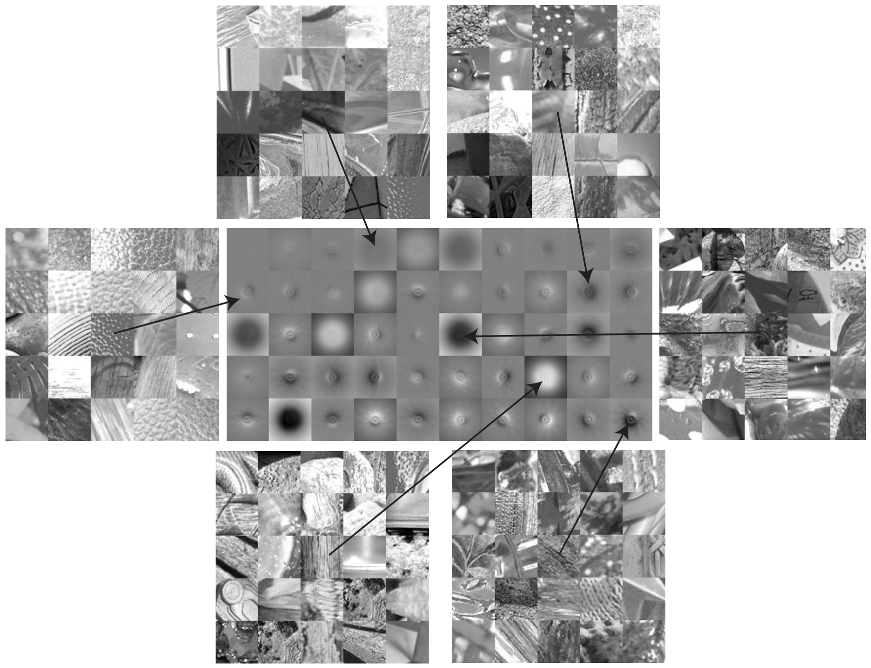

FIGURE 6.9: Pattern elements can be identified by vector quantizing vectors of filter outputs, using k-means. Here we show the top 50 pattern elements (or textons), obtained from all 1,000 images of the collection of material images described in Figure 6.2. These were filtered with the complete set of oriented filters from Figure 6.4. Each subimage here illustrates a cluster center. For each cluster center, we show the linear combination of filter kernels that would result in the set of filter responses represented by the cluster center. For some cluster centers, we show the 25 image patches in the training set whose filter representation is closest to the cluster center. *This figure shows elements of a database collected by C. Liu, L. Sharan, E. Adelson, and R. Rosenholtz, and published at `http://people.csail.mit.edu/lavanya/research_sharan.html`. Figure by kind permission of the collectors.*

responses observed at image locations.

Because pattern elements repeat, and so are common, we can assume that most data items are close to the center of their cluster. This suggests that we cluster the data by minimizing the the objective function

$$\Phi(\text{clusters}, \text{data}) = \sum_{i \in \text{clusters}} \left\{ \sum_{j \in i\text{th cluster}} (\boldsymbol{x}_j - \boldsymbol{c}_i)^T (\boldsymbol{x}_j - \boldsymbol{c}_i) \right\}.$$

Notice that if we know the center for each cluster, it is easy to determine which cluster is the best choice for each point. Similarly, if the allocation of points to clusters is known, it is easy to compute the best center for each cluster. However, there are far too many possible allocations of points to clusters to search this space for a minimum. Instead, we define an algorithm that iterates through two activities:

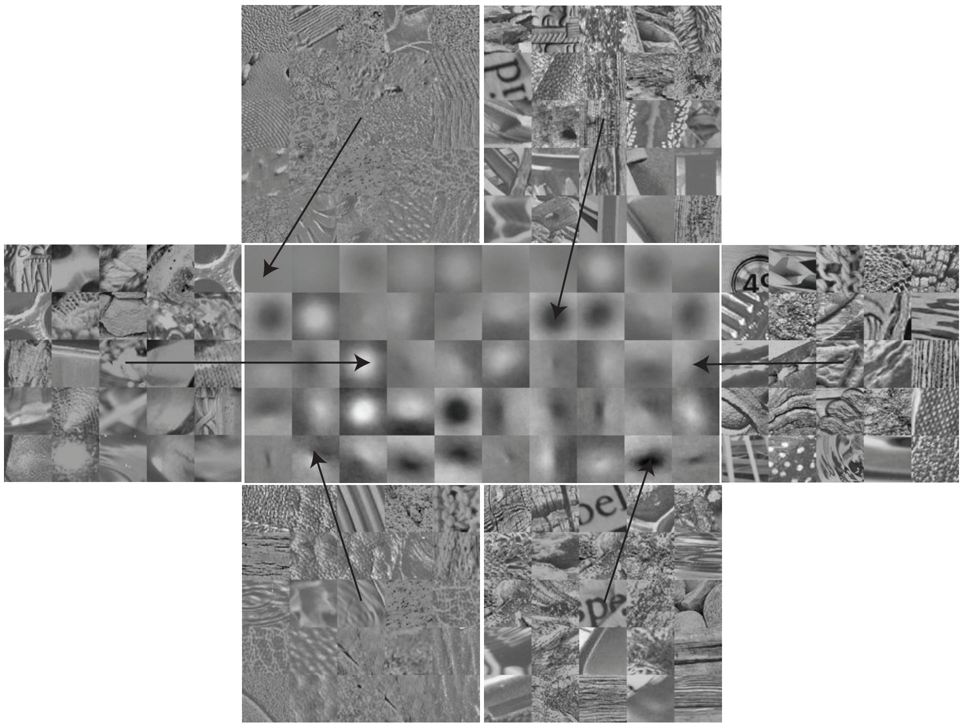

FIGURE 6.10: Pattern elements can also be identified by vector quantizing vectors obtained by reshaping an image window centered on each pixel. Here we show the top 50 pattern elements (or textons), obtained using this strategy from all 1,000 images of the collection of material images described in Figure 6.2. Each subimage here illustrates a cluster center. For some cluster centers, we show the closest 25 image patches. To measure distance, we first subtracted the average image intensity, and we weighted by a Gaussian to ensure that pixels close to the center of the patch were weighted higher than those far from the center. *This figure shows elements of a database collected by C. Liu, L. Sharan, E. Adelson, and R. Rosenholtz, and published at* `http://people.csail.mit.edu/lavanya/research_sharan.html`*. Figure by kind permission of the collectors.*

- Assume the cluster centers are known and, allocate each point to the closest cluster center.
- Assume the allocation is known, and choose a new set of cluster centers. Each center is the mean of the points allocated to that cluster.

We then choose a start point by randomly choosing cluster centers, and then iterate these stages alternately. This process eventually converges to a local minimum of the objective function (the value either goes down or is fixed at each step, and it is bounded below). It is not guaranteed to converge to the global minimum of the objective function, however. It is also not guaranteed to produce k clusters, unless we modify the allocation phase to ensure that each cluster has some nonzero number of points. This algorithm is usually referred to as *k-means* (summarized in

Algorithm 6.3). It is possible to search for an appropriate number of clusters by applying k-means for different values of k and comparing the results; we defer a discussion of this issue until Section 10.7.

```
Choose $k$ data points to act as cluster centers
Until the cluster centers change very little
    Allocate each data point to cluster whose center is nearest.
    Now ensure that every cluster has at least
      one data point; one way to do this is by
      supplying empty clusters with a point chosen at random from
      points far from their cluster center.
    Replace the cluster centers with the mean of the elements
      in their clusters.
end
```

Algorithm 6.3: Clustering by K-Means.

6.3 SYNTHESIZING TEXTURES AND FILLING HOLES IN IMAGES

Many different kinds of user want to remove things from images or from video. Art directors might like to remove unattractive telephone wires; restorers might want to remove scratches or marks; there's a long history of government officials removing people with embarrassing politics from publicity pictures (see the fascinating pictures in King (1997)); and home users might wish to remove a relative they dislike from a family picture. All these users must then find something to put in place of the pixels that were removed. Ideally, a program would create regions of texture that fit in and look convincing, using either other parts of the original image, or other images.

There are other important applications for such a program. One is to produce large quantities of texture for digital artists to apply to object models. We know that good textures make models look more realistic (it's worth thinking about why this should be true). Tiling small texture images tends to work poorly, because it can be hard to obtain images that tile well. The borders have to line up properly, and even when they do, the resulting periodic structure can be annoying.

6.3.1 Synthesis by Sampling Local Models

As Efros and Leung (1999) point out, an example texture can serve as a probability model for texture synthesis (Figure 6.11). Assume for the moment that we know every pixel in the synthesized image, except one. To obtain a probability model for the value of that pixel, we could match a neighborhood of the pixel to the example image. Every matching neighborhood in the example image has a possible value for the pixel of interest. This collection of values is a conditional histogram for the pixel of interest. By drawing a sample uniformly and at random from this collection, we obtain the value that is consistent with the example image.

We must now take some form of neighborhood around the pixel of interest, compare it to neighborhoods in the example image, and select some of these to form a set of example values. The size and shape of this neighborhood is significant, because it codes the range over which pixels can affect one another's values directly (see Figure 6.12). Efros *et al.* use a square neighborhood, centered at the pixel of interest.

```
Choose a small square of pixels at random from the example image
Insert this square of values into the image to be synthesized
Until each location in the image to be synthesized has a value
  For each unsynthesized location on
    the boundary of the block of synthesized values
    Match the neighborhood of this location to the
      example image, ignoring unsynthesized
      locations in computing the matching score
    Choose a value for this location uniformly and at random
      from the set of values of the corresponding locations in the
      matching neighborhoods
  end
end
```

Algorithm 6.4: Non-parametric Texture Synthesis.

The neighborhoods we select will be similar to the image example in some sense. A good measure of similarity between two image neighborhoods can be measured by forming the *sum of squared differences* (or *ssd*) of corresponding pixel values. We assume that the missing pixel is at the center of the patch to be synthesized, which we write $\mathcal{S}$. We assume the patch is square, and adjust the indexes of the patch to run from $-n$ to n in each direction. The sum of squared differences between this patch and an image patch $\mathcal{P}$ of the same size is given by

$$\sum_{(i,j)\in\text{patch},(i,j)\neq(0,0)} (\mathcal{A}_{ij} - \mathcal{B}_{ij})^2.$$

The notation implies that because we don't know the value of the pixel to be synthesized (which is at $(0,0)$), we don't count it in the sum of squared differences. This similarity value is small when the neighborhoods are similar, and large when they are different (it is essentially the length of the difference vector). However, this measure places the same weight on pixels close to the unknown value as it does on distant pixels. Better results are usually obtained by weighting up nearby pixels and weighting down distant pixels. We can do so using Gaussian weights, yielding

$$\sum_{(i,j)\in\text{patch},(i,j)\neq(0,0)} (\mathcal{A}_{ij} - \mathcal{B}_{ij})^2 \exp\left(\frac{-(i^2+j^2)}{2\sigma^2}\right).$$

Now we know how to obtain the value of a single missing pixel: choose uniformly and at random amongst the values of pixels in the example image whose neighborhoods match the neighborhood of our pixel. We cannot choose those matching

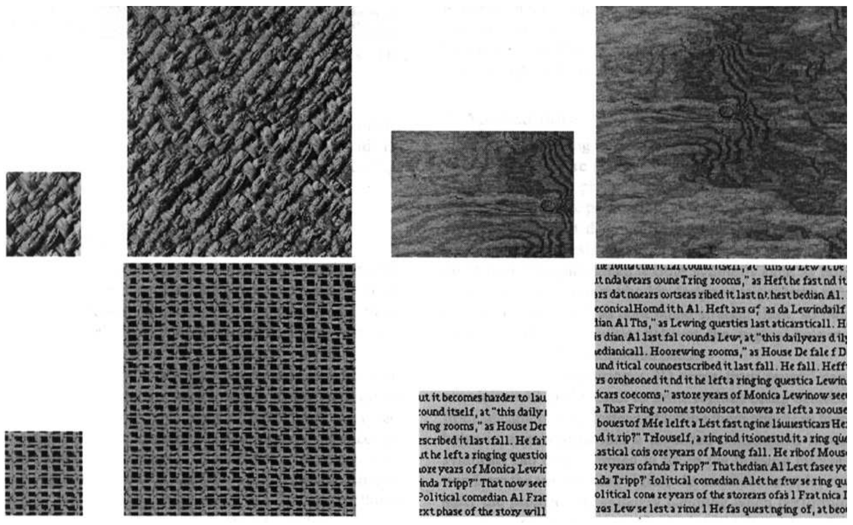

FIGURE 6.11: Efros and Leung (1999) synthesize textures by matching neighborhoods of the image being synthesized to the example image, and then choosing at random amongst the possible values reported by matching neighborhoods (Algorithm 6.4). This means that the algorithm can reproduce complex spatial structures, as these examples indicate. The small block on the **left** is the example texture; the algorithm synthesizes the block on the **right**. Note that the synthesized text looks like text: it appears to be constructed of words of varying lengths that are spaced like text, and each word looks as though it is composed of letters (though this illusion fails as one looks closely). *This figure was originally published as Figure 3 of "Texture Synthesis by Non-parametric Sampling," A. Efros and T.K. Leung, Proc. IEEE ICCV, 1999 © IEEE, 1999.*

neighborhoods by just setting a threshold on the similarity function, because we might not have any matches. A better strategy to find matching neighborhoods is to select all whose similarity value is less than $(1+\epsilon)s_{min}$, where s_{min} is the similarity function of the closest neighborhood and ϵ is a parameter.

Generally, we need to synthesize more than just one pixel. Usually, the values of some pixels in the neighborhood of the pixel to be synthesized are not known; these pixels need to be synthesized too. One way to obtain a collection of examples for the pixel of interest is to count only the known values in computing the sum of squared differences, and scale the similarity to take into account the number of known pixels. Write $\mathcal{K}$ for the set of pixels around a point whose values are known, and $\sharp\mathcal{K}$ for the size of this set. We now have, for the similarity function,

$$\frac{1}{\sharp\mathcal{K}} \sum_{(i,j)\in\mathcal{K}} (\mathcal{A}_{ij} - \mathcal{B}_{ij})^2 \exp\left(\frac{-(i^2+j^2)}{2\sigma^2}\right).$$

The synthesis process can be started by choosing a block of pixels at random from the example image, yielding Algorithm 6.4.

Filling in Patches

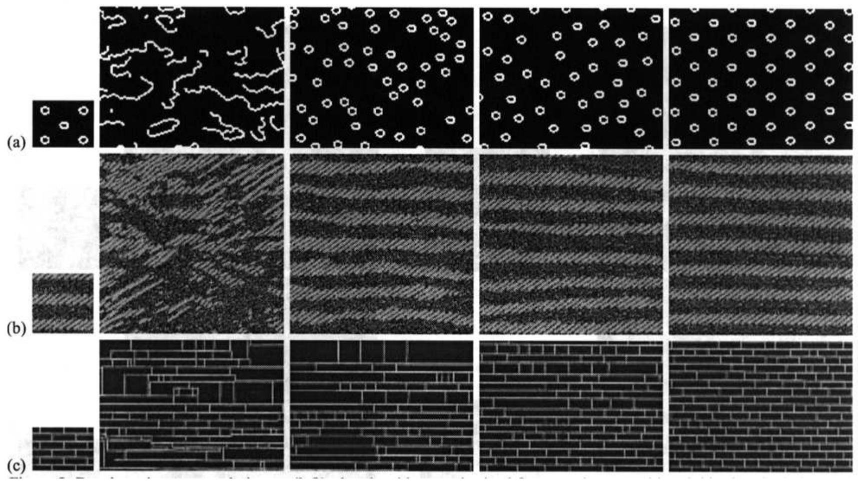

FIGURE 6.12: The size of the image neighborhood to be matched makes a significant difference in Algorithm 6.4. In the figure, the textures at the right are synthesized from the small blocks on the **left**, using neighborhoods that are increasingly large as one moves to the **right**. If very small neighborhoods are matched, then the algorithm cannot capture large-scale effects easily. For example, in the case of the spotty texture, if the neighborhood is too small to capture the spot structure (and so sees only pieces of curve), the algorithm synthesizes a texture consisting of curve segments. As the neighborhood gets larger, the algorithm can capture the spot structure, but not the even spacing. With very large neighborhoods, the spacing is captured as well. *This figure was originally published as Figure 2 of "Texture Synthesis by Non-parametric Sampling," A. Efros and T.K. Leung, Proc. IEEE ICCV, 1999 © IEEE, 1999.*

Synthesizing a large texture in terms of individual pixels will be unnecessarily slow. Because textures repeat, we expect that whole blocks of pixels also should repeat. This suggests synthesizing a texture in terms of image patches, rather than just pixels. Most of the mechanics of the procedure follow those for pixels: to synthesize a texture patch at a location, we find patches likely to fit (because they have pixels that match the boundary at that location), then choose uniformly and at random from among them. However, when we place down the new patch, we must deal with the fact that some (ideally, many) of its pixels overlap with pixels that have already been synthesized. This problem is typically solved by image segmentation methods, and we defer that discussion to Chapter 9.

6.3.2 Filling in Holes in Images

There are four approaches we can use to fill a hole in an image. **Matching methods** find another image patch that looks a lot like the boundary of the hole, place that patch over the hole, and blend the patch and the image together. The patch might well be found in the image (for example, Figure 6.13). If we have a very large set of images, we could find a patch by looking for another image that matches the image with a hole in it. Hays and Efros (2007) show this strategy can be extremely

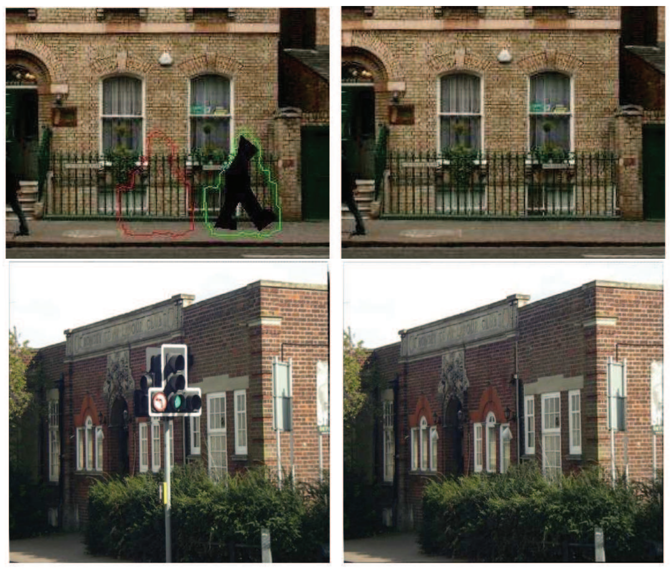

FIGURE 6.13: If an image contains repeated structure, we have a good chance of finding examples to fill a hole by searching for patches that are compatible with its boundaries. **Top left:** An image with a hole in it (black pixels in a rough pedestrian shape). The pixels on the region outside the hole, but inside the boundary marked on the image, match pixels near the other curve, which represents a potentially good source of hole-filling pixels. **Top right:** The hole filled by placing the patch over the hole, then using a segmentation method (Chapter 9) to choose the right boundary between patch and image. This procedure can work for apparently unpromising images, such as the one on the **bottom left**, an image of the facade of a house, seen at a significant slant. This slant means that distant parts of the facade are severely foreshortened. However, if we rectify the facade using methods from Section 1.3, then there are matching patches. On the **bottom right**, the hole has been filled in using a patch from the rectified image, that is then slanted again. *This figure was originally published as Figures 3 and 6 of "Hole Filling through Photomontage," by M. Wilczkowiak, G. Brostow, B. Tordoff, and R. Cipolla, Proc. BMVC, 2005 and is reproduced by kind permission of the authors.*

successful. Blending is typically achieved using methods also used for image segmentation (Section 9.4.3 describes one method that can be used for blending).

As you would expect, matching methods work very well when a good match is available, and poorly otherwise. If the hole is in a region of relatively regular texture, then a good match should be easy to find. If the texture is less strongly structured, it might be hard to find a good match. In cases like this, it makes sense to try and synthesize the texture over the region of the hole, using the rest of the image as an example. Making such **texture synthesis methods** work well requires considerable care, because the order in which pixels are synthesized has a strong effect on the results. Texture synthesis tends to work better for patches when most of their neighbors are known, because the match is more constrained. As a result, one wants to synthesize patches at the boundary of the hole. It is also important to extend edges at the boundary of the hole into the interior (for

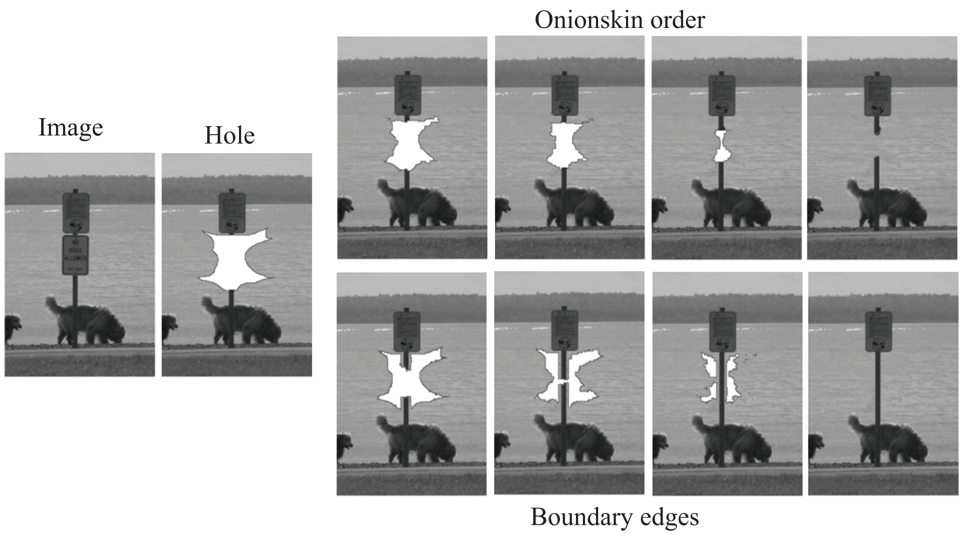

FIGURE 6.14: Texture synthesis methods can fill in holes accurately, but the order in which pixels are synthesized is important. In this figure, we wish to remove the sign, while preserving the signpost. Generally, we want to fill in pixels where most of the neighbors are known first. This yields better matching patches. One way to do so is to fill in from the boundary. However, if we simply work our way inwards (onionskin filling), long scale image structures tend to disappear. It is better to fill in patches close to edges first. *This figure was originally published as Figure 11 of "Region Filling and Object Removal by Exemplar-Based Image Inpainting," by A. Criminisi, P. Perez, and K. Toyama, IEEE Transactions on Image Processing, 2004 © IEEE, 2004.*

example, see Figure 6.14); in practice, this means that it is important to synthesize patches at edges on the boundary before one fills in other patches. It is possible to capture both requirements in a priority function ((Criminisi *et al.* 2004)), which specifies where to synthesize next.

If we choose an image patch at (i, j) as an example to fill in location (u, v) in the hole, then image patches near (i, j) are likely to be good for filling in points near (u, v). This observation is the core of **coherence methods**, which apply this constraint to texture synthesis. Finally, some holes in images are not really texture holes; for example, we might have a hole in a smoothly shaded region. Texture synthesis and matching methods tend to work poorly on such holes, because the intensity structure on the boundary is not that distinctive. As a result, we may find many matching patches, some of which have jarring interiors. **Variational methods** apply in these cases. Typically, we try to extend the level curves of the image into the hole in a smooth way. Modern hole-filling methods use a combination of these approaches, and can perform very well on quite demanding tasks (Figure 6.15).

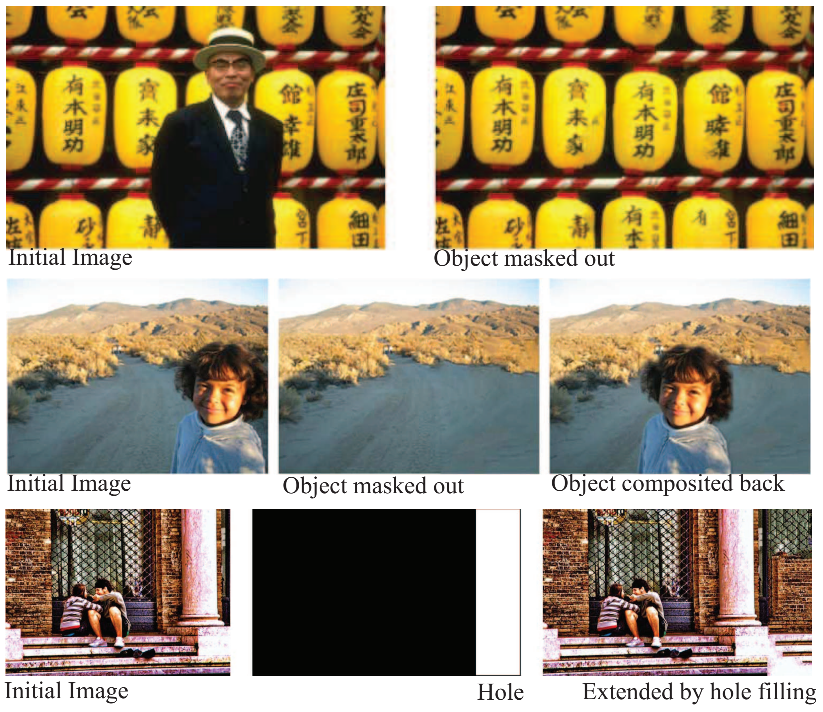

FIGURE 6.15: Modern hole-filling methods get very good results using a combination of texture synthesis, coherence, and smoothing. Notice the complex, long-scale structure in the background texture for the example on the **top** row. The **center** row shows an example where a subject was removed from the image and replaced in a different place. Finally, the **bottom** row shows the use of hole-filling to resize an image. The white block in the center mask image is the "hole" (i.e., unknown pixels whose values are required to resize the image). This block is filled with a plausible texture. *This figure was originally published as Figures 9 and 15 of "A Comprehensive Framework for Image Inpainting," by A. Bugeau, M. Bertalmío, V. Caselles, and G. Sapiro, Proc. IEEE Transactions on Image Processing, 2010 © IEEE, 2010.*

6.4 IMAGE DENOISING

This section addresses the problem of reconstructing an image given the noisy observations gathered by a digital camera sensor. Today, with advances in sensor design, the signal is relatively clean for digital SLRs at low sensitivities, but it remains noisy for consumer-grade and mobile-phone cameras at high sensitivities (low-light and/or high-speed conditions). Adding to the demands of consumer and professional photography those of astronomy, biology, and medical imaging, it is thus clear that image restoration is still of acute and in fact growing importance. Working with noisy images recorded by digital cameras is difficult because different devices produce different kinds of noise, and introduce different types of artifacts and spatial correlations in the noise as a result of internal post-processing (demosaicking, white balance, etc.).