Chapter 6

# Pixel-Oriented Rendering of Line Drawings

In Chaps. 4 and 5, several analytical approaches for creating line drawings were introduced. These methods allow the generation of a broad variety of drawing styles, and some applications of scientific and medical illustrations were given.

In the following, a pixel-based method is presented. A set of G-buffers is used for encoding visual and geometric properties of the models. G-buffers store information for each pixel of the image. These buffers are combined with other geometric data to form the line drawing.

In comparison to analytical solutions, a pixel-oriented approach has several advantages and also shortcomings. While working analytically, the results are resolution independent, which is not the case for pixel-oriented methods. An analytic hidden surface removal algorithm may generate more and other information about visible lines and surfaces than can be achieved by a pixel-oriented method. This makes analytic methods more flexible and also more general than pixel-based algorithms.

A pixel-oriented strategy, however, offers the general advantage of easily using graphics hardware which makes algorithms very fast. In addition, the methods are independent of the graphical primitives used to form the models. Everything that can be processed by the hardware can also be used to form the line drawing. Also, the algorithms can be parallelized easily, and in general they are much simpler to implement and to maintain than analytic methods. This makes pixel-based approaches preferentially applicable where results have to be generated quickly and where huge or complex data sets are to be handled.

After discussing previous work and presenting the ingredients of the method, some results are shown. A statue and a bust of Beethoven are used for

this purpose. The models have been chosen because of their slightly curved but complex geometry, which is also the case with most of the medical models shown in this book so far. In the last section some suggestions on combining analytic and pixel-based methods are given.

## 6.1 Previous Work

Pixel-based algorithms for solving visibility problems have a long tradition in computer graphics. CATMUL [Cat74] presented the idea of using a $z$-buffer, an array storing the color and depth for each pixel of the image. This buffer is used to compute visible parts of the given geometric data by comparing depth values of the surface pixels with the values already stored in the $z$-buffer. The amount of memory needed by this method can be reduced by using a scan line method which works only with one line of the $z$-buffer (cf. [Mye75]). ROSSIGNAC refined this method for Constructive Solid Geometry [RR86]; other work was done on processing of data stored in Binary Space Partitioning Trees. Hardware implementations, as described by BOOTH, FORSEY, and PAETH (cf. [BFP86]), of $z$-buffers can be found in nearly all graphics hardware.

SAITO and TAKAHASHI [ST90] applied image processing to the $z$-buffer and introduced other pixel buffers (they called them G-buffers) to enhance images in the case that the underlying geometry is given. Their work is stated to be one of the roots of non-photorealistic imaging.

Though extended in some ways, the real breakthrough for non-photorealistic imaging and especially for the use of line drawings took place in more recent years, motivated by the work of LANSDOWN and SCHOFIELD [LS95], STROTHOTTE et al. [SPR+94, SSRL96] and WINKENBACH, SALISBURY and SALESIN (cf. [SALS96, WS96, SWHS97]).

While SCHOFIELD mostly used pixel-based methods for non-photorealistic rendering (he invented G-buffers independently of SAITO and TAKAHASHI), STROTHOTTE et al. focused on generating line directions that are drawn by applying different line styles.

It was the idea of WINKENBACH et al. to use prioritized stroke textures for generating hatching lines. These textures are placed on the surface of objects to form the line drawing. Extensions like resolution dependent textures and orientable textures are given.

LEISTER [Lei94] introduced a special kind of ray tracing in combination with image processing operators for generating line drawings. His method can be seen as an application of volume texturing in order to visualize objects with an outlook similar to copper plates.

The method presented here is an extension of the work of SAITO and TAKAHASHI. Additional G-buffers and cross sectional information are used to form the typical outline of hatched line drawings. Half-toning on the basis of the hatching lines is used to approximate the intensities of a given image.

## 6.2 A Pixel-Oriented Graphics Pipeline

A pixel-based method for achieving line drawings is quite different from analytic approaches. First, some basic G-buffers are calculated. Image operators are applied to the basic buffers in order to generate additional buffers representing necessary information for the drawing process like structural lines or vector fields.

In the next subsection the set of G-buffers and corresponding image operators is presented, and some extensions to standard image operators are given. In Sect. 6.2.3, a half-toning scheme for short hatching lines is introduced. Their placement is controlled by an error diffusion algorithm.

The generation of long hatching lines (as needed especially for generating medical illustrations) is shown in Sect. 6.2.4. In this case the half-toning must control not the placement but the appearance of the lines; this is described in Sect. 6.2.5. Some results show the usability of the approach.

### 6.2.1 Basic G-Buffers

A set of buffers forms the basis of all subsequent operations (see Fig. 6.1). Each of them can be calculated efficiently by using standard graphics hardware. For simplification the domain of each pixel is assumed to be the set of integers.

The first G-buffer is the image itself. It is represented as the pixel array $I$ with values describing the light intensity for every point of the image. For the purpose of creating black and white line drawings, a gray-scale map suffices.

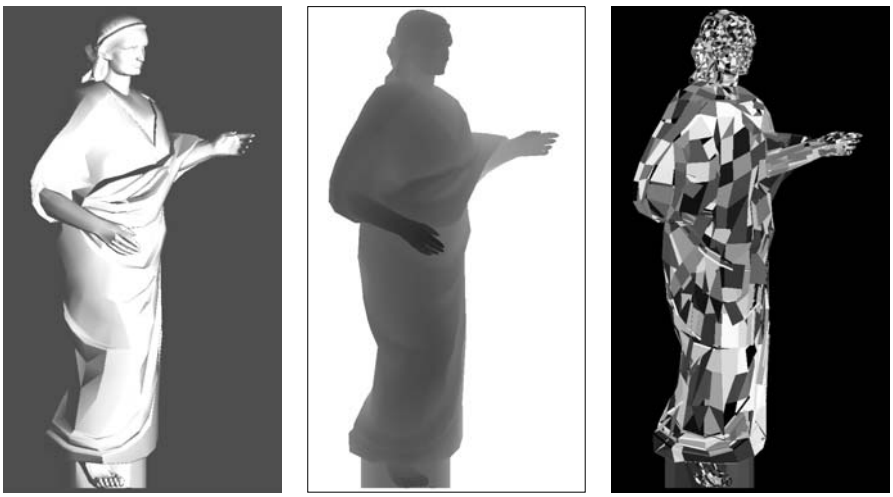


**Figure 6.1:** Basic G-buffers: image intensity (left), depth values (center), and primitive index (right)

The second G-buffer ($Z$) is called the depth buffer or $z$-buffer. The values determine the distance of the model to the viewing plane for each pixel. Later, some image operators will be applied to this buffer in order to generate other buffers.

Another useful buffer is the id-buffer, refered to as $ID$. The value (color or gray-scale value) of every pixel encodes the index of the visible geometric primitive at this point. For instance, this buffer can also be used for computing efficiently a high quality representation of the image $I$, as the hidden surface problem is already solved if an id-buffer is present.

The last basic G-buffer ($N$) stores normal vectors of the geometry. For each pixel the corresponding normal vector of the visible geometry is stored (if it is defined). Usually one needs three images to store the values, the pixels of each image storing one coordinate.

**Table 6.1:** Basic G-buffers

| Description | Symbol | Domain |
|---|---|---|
| Image | $I$ | $\mathbb{I}$ |
| Depth values | $Z$ | $\mathbb{I}$ |
| $id$-buffer | $ID$ | $\mathbb{I}$ |
| Normal vectors | $N$ | $\mathbb{R}^3$ |

It should be pointed out that numerical accuracy is often crucial for obtaining good results. Therefore, intermediate G-buffers might be stored by using floating point numbers or long integers.

### 6.2.2 G-Buffer Operators

In the following the set of image operators is shown. These operators are applied to the basic G-buffers to compute other G-buffers necessary for generating the line drawing. A functional notation is used for the operators. For example, the bitwise "and" of two images $I_1, I_2$ is denoted as "$And(I_1, I_2)$".

Tables 6.2 and 6.3 list the image operators used for generating the line drawings. Most of them are known from standard image analysis literature (e.g., [RK82]). The implementation of the others can be found below.

**Difference Operators.** Calculating the value of first and second order differences from a pixel image is a classical operation in image analysis. In [ST90] the SOBEL operator (cf. [RK82]) is used. In the following, $G_{i,k}$ denotes the value of the pixel in line $i$ and row $k$ of G-buffer $G$.

$$d1(G_{i,k}) = \frac{1}{8}\begin{pmatrix} |G_{i-1,k-1} + 2G_{i,k-1} + G_{i+1,k-1} \\ -G_{i-1,k+1} - 2G_{i,k+1} - G_{i+1,k+1}| \\ +|G_{i+1,k-1} + 2G_{i+1,k} + G_{i+1,k+1} \\ -G_{i-1,k-1} - 2G_{i-1,k} - G_{i-1,k+1}| \end{pmatrix}$$

The purpose of this operator is to detect discontinuities in the basic G-buffers. These discontinuities can be used to form structural lines. Therefore the operator is normalized; $k_{d1}$ denotes the threshold, $d1_{min}$ and $d1_{max}$ the minimal and maximal differences.

$$Diff1(G_{i,k}) = \begin{cases} \dfrac{d1_{min} - d1(G_{i,k})}{d1_{max} - d1_{min}}, & \text{if } d1_{max} - d1_{min} > k_{d1} \\ \\ \dfrac{d1_{min} - d1(G_{i,k})}{k_{d1}}, & \text{if } d1_{max} - d1_{min} \leq k_{d1} \end{cases}$$

To detect discontinuities of second order, the following operator is used by SAITO and TAKAHASHI. It is also normalized to allow the generation of uniform lines:

$$d2(G_{i,k}) = \frac{1}{3} \begin{pmatrix} 8G_{i,k} - G_{i-1,k-1} - G_{i-1,k} \\ -G_{i-1,k+1} - G_{i,k-1} - G_{i,k+1} \\ -G_{i+1,k-1} - G_{i+1,k} - G_{i+1,k+1} \end{pmatrix}$$

**Table 6.2:** Unary G-buffer operators

| Description | Symbol | Domain |
|---|---|---|
| Bitwise operators | $Set()$ | $\mathbb{I} \to \mathbb{I}$ |
| | $Unset()$ | $\mathbb{I} \to \mathbb{I}$ |
| | $Round()$ | $\mathbb{R} \to \mathbb{I}$ |
| Value of first order difference | $Diff1()$ | $\mathbb{I}^2 \to \mathbb{R}$ |
| Value of second order difference | $Diff2()$ | $\mathbb{I}^2 \to \mathbb{R}$ |
| Direction of pixels with same value | $Iso()$ | $\mathbb{I}^2 \to \mathbb{R}^2$ |
| Integer conversion of direction vector (angle) | $A^{-1}()$ | $\mathbb{R}^2 \to \mathbb{I}$ |
| Retrieval of direction vector from integer value | $A()$ | $\mathbb{I} \to \mathbb{R}^2$ |

**Table 6.3:** Binary G-buffer operators

| Description | Symbol | Domain |
|---|---|---|
| Bitwise operations | $And()$ | $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$ |
| | $Or()$ | $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$ |
| Pixel-wise arithmetic operations | $Add()$ | $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$ |
| | $Sub()$ | $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$ |
| | $Max()$ | $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$ |
| | $Min()$ | $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$ |
| | $Mul()$ | $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$ |
| | $Div()$ | $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$ |
| Multiplication with scalar value | $Smult()$ | $\mathbb{I} \times \mathbb{R} \to \mathbb{R}$ |
| Threshold operation | $Tresh()$ | $\mathbb{I} \times \mathbb{I} \to \mathbb{I}$ |

$$Diff2(G_{i,k}) = \begin{cases} d2(G_{i,k}), & \text{if } d1_{max} \leq k_{d2} \\ \dfrac{d2(G_{i,k})}{(d1_{max}/k_{d2})^2}, & \text{if } d1_{max} > k_{d2} \end{cases}$$

Although these operators allow to extract discontinuities in a desirable way, a second operator for detecting first order differences was designed to generate "lighter" lines which are needed for high-quality pixel–vector conversion (as needed below). The difference operator *Diff1* generates lines, as can be seen in the center part of Fig. 6.2. Vertical and horizontal lines are generated as required, but diagonal lines are too fat. Application of the operator *Diff1a* leads to a better (lighter) result.

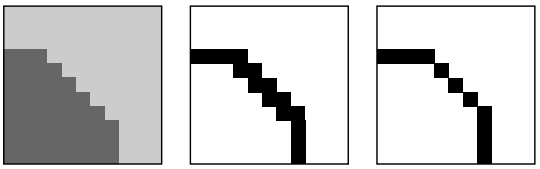


**Figure 6.2:** Results of difference operators: Original image, after applying *Diff1*, and after applying *Diff1a*

The operator *Diff1a* is also normalized, i.e., it delivers binary values and no information about the level of discontinuity is provided.

The operator has a procedural implementation that works in two steps: First, the image is scanned line by line. If the horizontal differences are above the threshold $k_{d1}$, the pixel is set. Second, the image is processed row by row. In this step a pixel is set only if either the upper or right neighbor is not set at this time.

```
proc Diff1a()
  for k := 0 to height − 1) do
      Unset(G_{i,0});
      for i := 1 to width − 1) do
          if |G_{i−1,k} − G_{i,k}| > k_{d1}
            then Set(G_{i,k}) else Unset(G_{i,k}) fi
      od
  od
  for i := 1 to (width − 1) do
      for k := 0 to (height − 1) do
          if |G_{i,k+1} − G_{i,k}| > k_{d1} ∧ (G_{i,k+1} = 0 ∨ G_{i−1,k} = 0)
            then begin
                    Set(G_{i,k});
                    if G_{i,k−1} > 0 ∧ G_{i−1,k} > 0 then Unset(G_{i,k}) fi
                 end fi
      od
  od
end
```

An application of the difference operators to G-buffers is shown in Fig. 6.3 (center) and 6.3 (right), where structure lines are generated using the image and the $z$-buffer by application of difference operators. These results can be seen as first sketches of the given statue.

**Figure 6.3:** Application of difference operators to the image and the $z$-buffer: $\mathit{Diff1}(I)$, $\mathit{Tresh}(\mathit{Diff1}(I), 30)$, $\mathit{Tresh}(\mathit{Diff1}(Z), 30)$ (from left to right)

**The Iso-Operator.** During the generation of hatching lines it is sometimes necessary to draw lines along pixels with the same depth value. This is done by using another G-buffer that stores the directions of pixels with the same value (if this direction is unique).

This buffer can be generated by applying the *Iso*-operator to a G-buffer. The *Iso*-operator is defined by the perpendicular vector of the pixel-wise gradient direction:

$$\begin{aligned} \mathit{Grad}_x(G_{i,k}) &= -G_{i-1,k-1} + G_{i+1,k-1} - G_{i-1,k} \\ &\quad +G_{i+1,k} - G_{i-1,k+1} + G_{i+1,k+1} \\ \mathit{Grad}_y(G_{i,k}) &= -G_{i-1,k-1} + G_{i-1,k+1} - G_{i,k-1} \\ &\quad +G_{i,k+1} - G_{i+1,k-1} + G_{i+1,k+1} \\ \\ \mathit{Iso}(G_{i,k}) &= \big(\mathit{Grad}_y(G_{i,k}), -\mathit{Grad}_x(G_{i,k})\big)^T \end{aligned}$$

In Fig. 6.4 the direction of the vectors is visualized. The vectors are stored by their direction angle using the functions $A()$ and $A^{-1}()$. The other operators of Tables 6.2 and 6.3 should be clear to the reader, who is otherwise referred to the standard literature of image analysis.

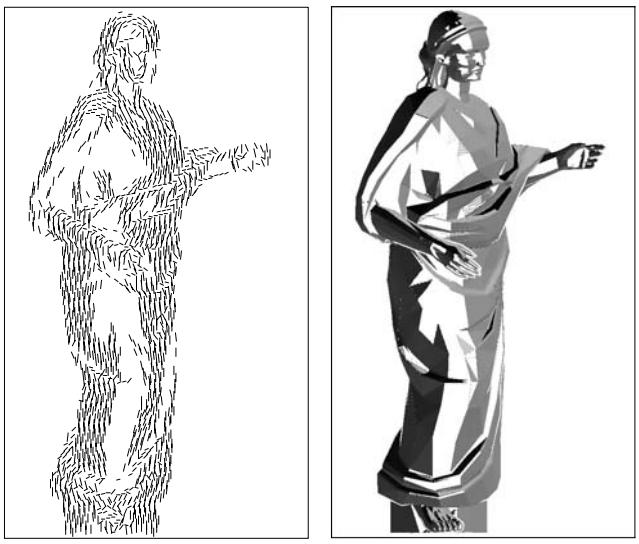


**Figure 6.4:** Visualization of *Iso* depth vectors and the associated G-buffer

Until now it was shown how G-buffers can be created by using image operators applied to the basic G-buffers. In the next subsection two half-toning schemes working on hatching lines are described. First, short hatching lines are distributed on the image, later the outline of lines is changed according to a given gray-scale value.

### 6.2.3 Half-Toning Using Short Hatching Lines

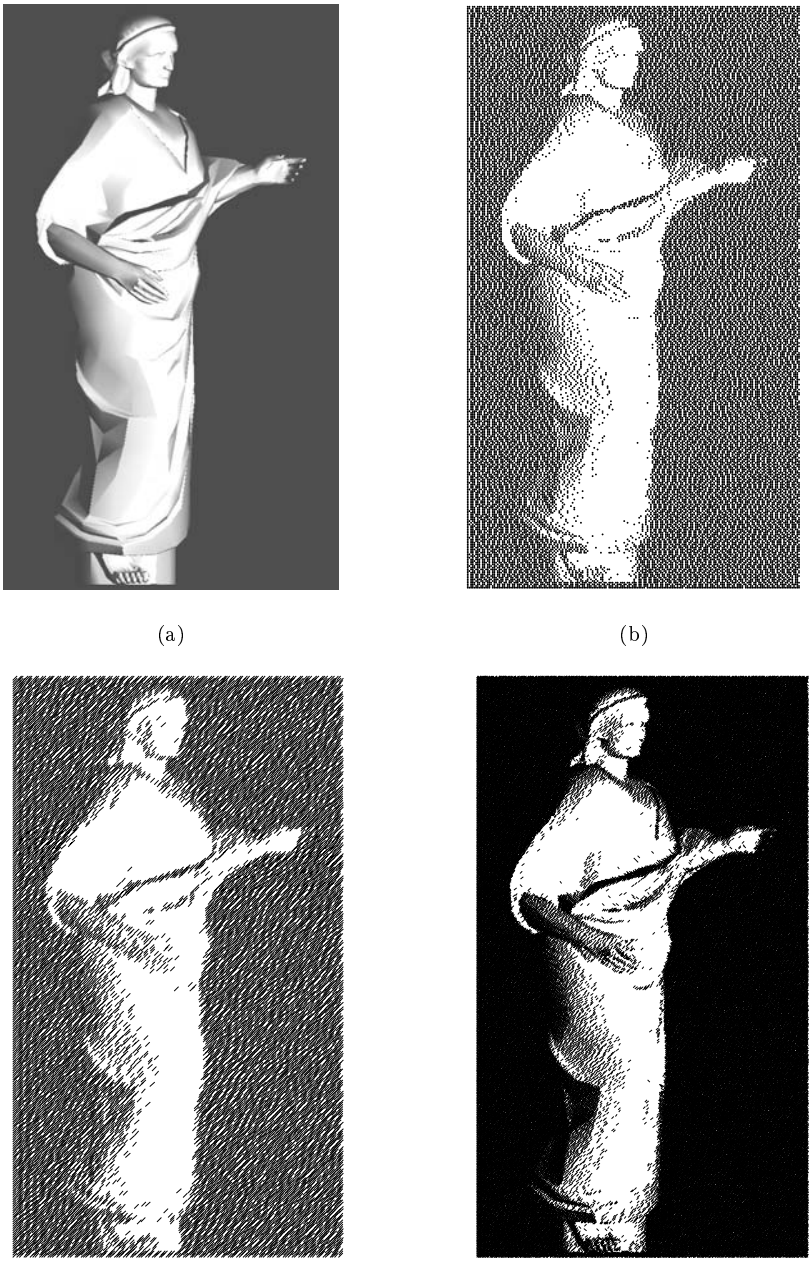
The half-toning process is a function which maps an image of gray-scale values to another image composed by the colors black and white. The half-toning process must preserve the integral gray-scale value over each (sufficiently large) part of the picture.

A classic method is the algorithm of FLOYD and STEINBERG [FS76]. The image is processed row by row. For each pixel the error that arises with drawing a black or white pixel is accumulated and added to the next pixel, thus spreading the error over a larger area.

In Fig. 6.5(b) the half-toned image of the given reference (Fig. 6.5(a)) is shown. Points can easily be replaced by short hatching lines, if the error is treated appropriately (cf. Fig. 6.5(c)). If two line directions and smaller lines are used, the result gives a good approximation of the reference image.

Usually, during half-toning no knowledge is assumed about the content of the image to be processed. This is not the case here, as we want to do a special kind of half-toning which leads us to hatched images.

Figure 6.6 shows how information about the model can be used by a half-toning process. Figure 6.6(a) shows the result of half-toning the robe of the

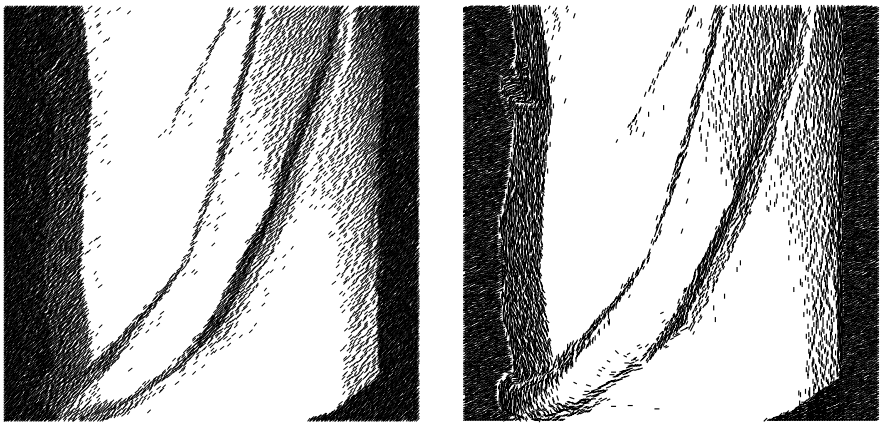


(a) (b)

(c) (d)

**Figure 6.5:** A half-toning process with short hatching lines: (a) reference image; (b) application of an error diffusion algorithm using points; (c) the same done by using short lines; (d) two line directions combined, smaller lines used

statue using short hatching lines with a slope of ±45 degrees. In Fig. 6.6(b) the same is done using iso depth values for redirecting the slope of the lines.



(a) (b)

**Figure 6.6:** Enhanced half-toning using short lines: (a) half-toning with two static line directions; (b) redirecting the slope of the lines to the direction of *iso* depth values

In the next section the focus is on generating long hatching lines. A half-toning method based on these lines is described below.

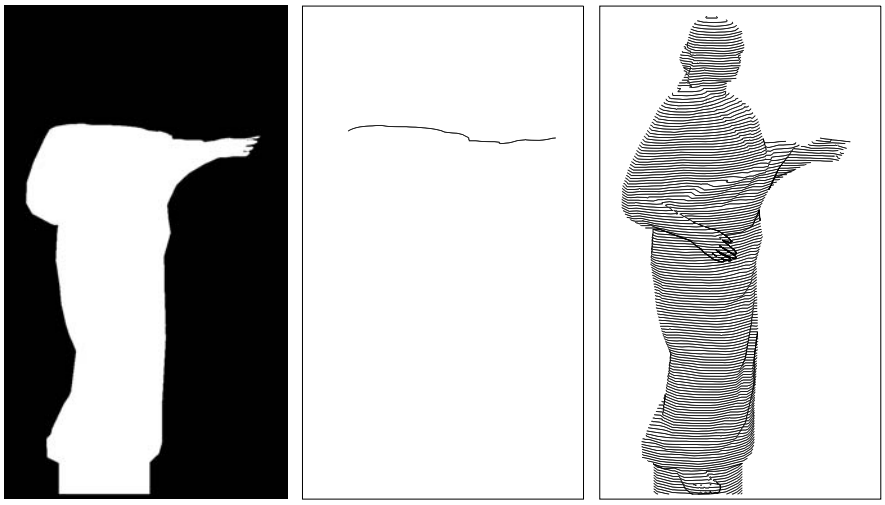
### 6.2.4 Generating Long Hatching Lines by Intersections

In scientific illustrations long hatching lines are widely used. The hatching lines in Fig. 1.1 might be interpreted as intersections between the object and a set of planes.

Such a set of intersecting lines should help in directing subsequent half-toning operators along the surface of the objects. The lines can be created either analytically by intersecting the model with a set of planes, or by a combination of pixel-based operations and image analysis. In both cases the result should be a set of curves in 2D.

Currently the intersections are generated using the latter method (cf. Fig. 6.7). The process has several steps: First, the full model is shaded using flat shading and a dark background. By applying the operator *Diff1a* and performing a pixel-vector conversion, the outline of the model is generated.

Now the same is done by using additional clipping planes (Fig. 6.7(a)). The operator *Diff1a* is applied again and the outline of the model is subtracted. What remains is the pixel representation of the intersecting line (Fig. 6.7(b)). A pixel-vector conversion is carried out to achieve the desired pixel vector. The whole set of vectors can be seen in Fig. 6.7(c).



(a) (b) (c)

**Figure 6.7:** Pixel based generation of intersecting lines: (a) flat shaded image of the model by using an additional clipping plane, (b) resulting vector after subtracting the outline of the full model, (c) the whole set of intersecting lines

### 6.2.5 Half-Toning Using Long Hatching Lines

The generated lines have to be drawn in an appropriate way. To match the half-toning requirements, each line $l_i$ is responsible for a tube $t_i$ (cf. Fig. 6.8). The outline of the tube lies between the line and its neighbors. Using such tubes results in partitionizing the surface of the object to be hatched.

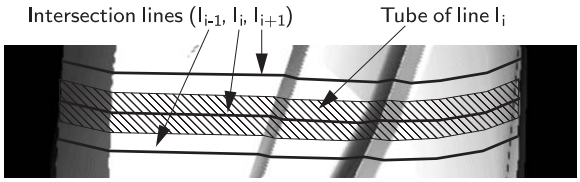



**Figure 6.8:** For half-toning the appearance of each intersection line $l_i$ is responsible for approximating the gray-scale values belonging to a tube around the line

If the overall intensity of the line $l_i$ equals the intensity of the pixels in $t_i$, a correct half-toning is achieved. A simple method is to modulate the width of $l_i$ according to the intensities along the line. Doing so supposes that we have uniform intensities in $t_i$ perpendicular to $l_i$.

In Fig. 6.9, a set of intersection lines is used for half-toning the bones of a foot. The intensities of the image at the top were used for controlling

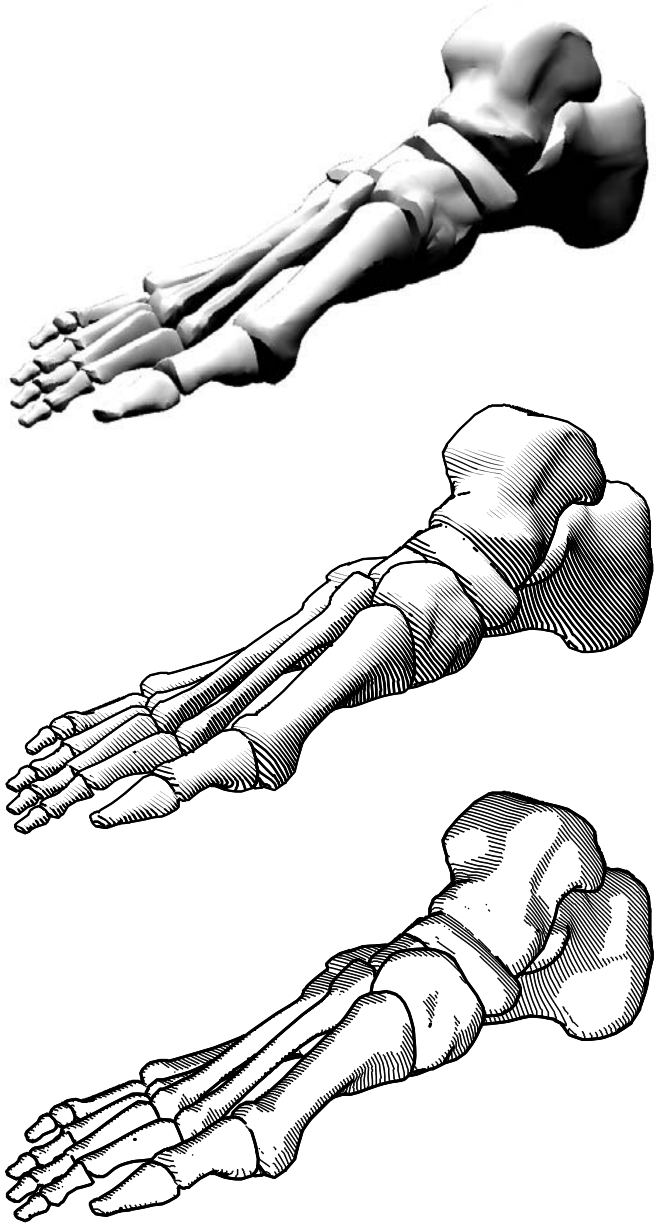


**Figure 6.9:** Using intersection lines for hatching: A set of intersection planes was used to generate the lines. The intensities of the image at teh top were used for controlling the line width of the picture in the middle. At the bottom a style similar to Fig. 1.1 was approximated

the line width of the picture in the middle by using the above method. At the bottom a style similar to Fig. 1.1 was approximated. Here, a uniform line width was chosen, and the line was drawn for those places where the intensities of the picture belong to a given interval.

### 6.2.6 Computer Generated Copper Plates

As mentioned in the introduction to this chapter, LEISTER used a modified ray tracing algorithm in combination with black and white volume textures to simulate the generation of copper plates. This approach is quite similar to explicitly generating intersections, as was done above. If the volume texture consists of parallel planes, the ray tracing algorithm tests for each pixel if one of these planes is present at the point where the ray hits the surface of the visible object.

The advantage of explicitly generating intersections is that these intersections can be postprocessed later by applying a half-toning method that generates the hatching lines individually. In Fig. 6.11 a copper plate is simulated. Several parts of the model were processed separately and later combined by using $z$-buffers. The size of the generated dots was chosen according to the intensity of the model's image $I$.

Artists use several tricks while developing copper plates. It is too simple to assume one can achieve a computer generated copper plate by just intersecting the model with a set of parallel planes. Real copper plates are made by using non-parallel planes or even other objects. Sometimes several planes are overlaid as can be seen in the face of the nun in Fig. 6.10 (left). In Fig. 6.10 (right) different styles are used for the light and the dark parts of the face.

**Figure 6.10:** Two copper plates of nuns demonstrate the usage of different styles within the same image [Gra90]

The same was done for the computer generated copper plate of Fig. 6.11. The face was drawn by using two sets of intersecting planes and the rest was

**Figure 6.11:** A computer generated copper plate showing a bust of Beethoven. The face was drawn using two sets of intersecting planes, the rest using one set

generated using one set. The main difficulty was to direct the orientation of the intersection planes in an appropriate way.

## 6.3 Concluding Remarks

In this chapter a pixel-based rendering pipeline was given. The pipeline consists of three steps. First, basic G-buffers have to be generated. By application of image operators additional buffers are created in a second step. One may also create intersection information, if appropriate. In the last step a half-toning process is applied to the data which maps the intensities of the image buffer $I$ to the size, direction, and style of the generated lines.

For high quality images sometimes a high resolution of intermediate G-buffers is needed. Therefore, it can be useful to partition the buffers (OpenGL works with images of at most $2\,000 \times 2\,000$ pixels).

Analytical and pixel-oriented methods can be combined in many different ways. On one hand the analytical generation of intersections can be used to

get 3D results, on the other hand their pixel-based generation can be mapped into 3D by using back projection (parts of the intersections may be missing if not visible). The results are now usable in an analytic approach.

G-buffers can be used in places where it is sufficient to store information at discrete points. Every analytic approach involves the creation of the image for a pre-defined view. At this point, pre-computed G-buffers may introduce additional information like vector fields (e.g., iso-values) or depth information.

Pixel-oriented methods may also be used to obtain some kind of image-based control on the number of and distance between hatching lines. The avoidance of Moiré patterns is an important problem during the generation of line drawings. It depends on the resolution of the output devices how closely curves can be placed together without obtaining a Moiré pattern. If a line drawing is to be rendered for a specific output device with given resolution, G-buffers may allow efficient calculation of the pixel distance from line to line.

The generation of intersecting lines may now be an iterative process: A new intersection is generated, the maximal and minimal pixel distances to their neighbors is measured and the intersection plane is moved if the distances do not match pre-defined criteria. Such methods were proposed by SAITO and TAKAHASHI in [ST90] and also by TURK and BANKS [TB96] for generating stream lines. Their application to line drawings is future work.

The fast generation and rendering of pixel-based hatching lines can be used for displaying and interacting with complex environments like botanical scenes. Based on our work on modeling and rendering realistic plants and plant ecosystems [DL97, DHL+98], the techniques presented in this chapter will be applied to those scenes in order to provide interaction with objects that otherwise require the display of tens of millions of polygons.

*Contributor of Chap. 6:* Oliver DEUSSEN