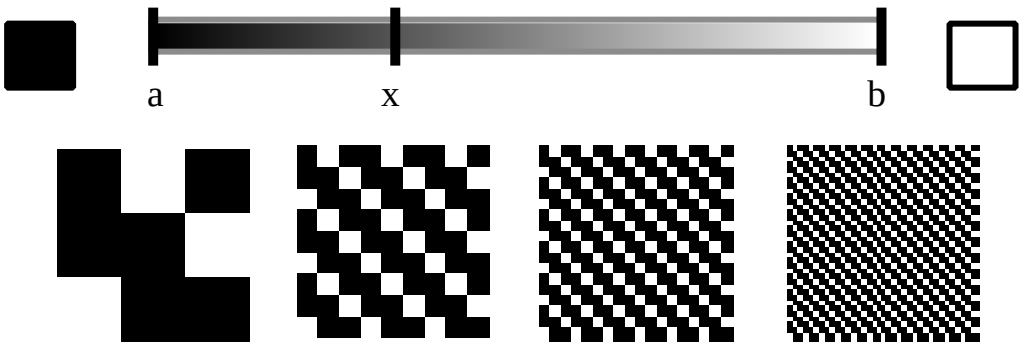
Dithering and Raster Graphics

Graphics devices like monitors or printers are able to handle only a limited number of colors. In order to display raster images, which contain colors that are not available on the given device, two methods can be used. One of them is **quantization**, where from the available color tables that one is selected which fits best to the color spectrum of the given image. The other one is **dithering** which simulates the missing colors by mixing the existing colors.

1. Dithering

The principle of dithering is demonstrated in Figure 1. In order to simulate tone x on a raster display the available colors a and b are mixed in an appropriate ratio.



In order to simulate tone x

$100 \cdot \frac{x-a}{b-a}\%$ b-Pixel and

$100 \cdot \frac{b-x}{b-a}\%$ a-Pixel

have to be used.

Figure 1. Simulation of missing colors by mixing the existing colors.

In printing industry we are not restricted to constant size pixels located at regular grid points. Therefore, in black-and-white printing the point density is equal everywhere and different gray levels are represented by points with different sizes. In color printing every primary color is rastered separately and different printing angles ensure unbiased results.

1.1.Classification of dithering methods

Dithering methods can be classified into two main categories. One of them is **threshold dithering**, where every pixel is compared to a threshold value in order to decide which existing color should substitute the given pixel color. For example, *ordered dithering*, *stochastic dithering*, and *dot diffusion dithering* belong to this category. The other alternative is **error diffusion dithering**, where the rounding error of every pixel is propagated to neighboring pixels and compensated there. A typical example of this category is *Floyd-Steinberg dithering.*

1.2. Threshold dithering

Using threshold dithering, each pixel value p is compared to a threshold t. If $p < t$ then $p = a$, otherwise $p = b$, where a and b are the available colors. The threshold t can be equal everywhere (for instance: $(a + b) / 2$, arbitrary value, mean value, median value, etc.) or it can be location dependent defined locally or globally. Figure 2 demonstrates **constant threshold dithering**, where threshold t is the same for each pixel.

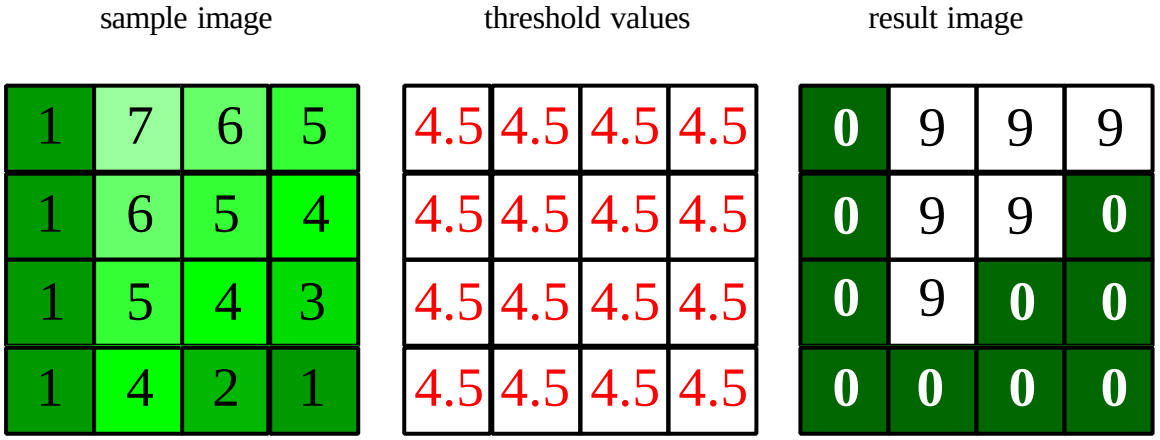
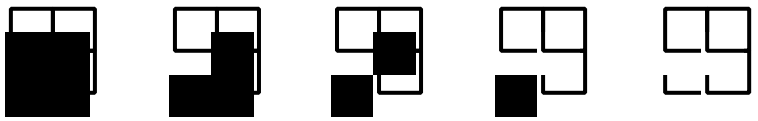
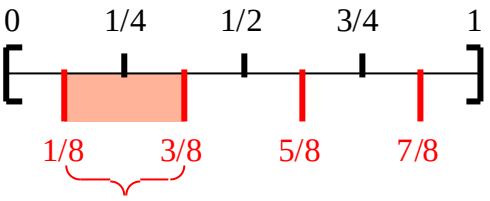


Figure 2. Constant threshold dithering.

Constant threshold dithering produces very bad results, since practically the intermediate tones are substituted by the nearest available colors. Better results can be achieved using location dependent dithering, where for a uniform area the same pattern is repeated. For example, using a 2×2 pattern the following five different gray tones can be simulated:



Using this patterns, the domain of the possible gray levels is divided into five intervals, where the interval borders are defined as follows:



According to these threshold values it is decided which pattern to use in a certain area. For instance, in a region where the gray levels are between *1/8* and *3/8* the second pattern is repeated which represents the gray value of *1/4*. This technique can be implemented using a **threshold matrix** which contains a regular pattern of the upper interval borders defining a threshold for each pixel location (Figure 3).

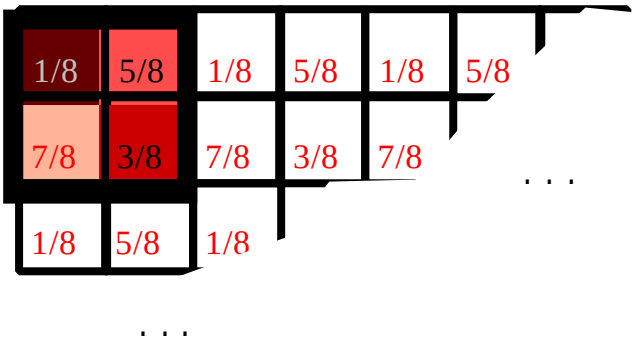
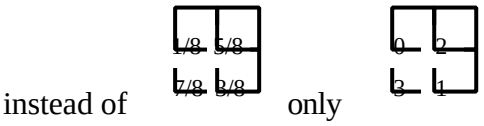


Figure 3. The threshold matrix.

In general case the distances between interval borders are normally equal, therefore it suffices to define the sequence of the pixel values in a **dither matrix**:



For instance, having an $n \times n$ matrix the values are in $[0, n^2 - 1]$, and value k corresponds to the threshold value $(2k + 1)/(2n^2)$. Figure 4 shows an example of a dither matrix and a corresponding threshold matrix. Figure 5 demonstrates the location dependent dithering using a 2×2 dither matrix.

7	2	3
0	4	8
5	6	1

a

$\frac{15}{18}$	$\frac{5}{18}$	$\frac{7}{18}$
$\frac{1}{18}$	$\frac{9}{18}$	$\frac{17}{18}$
$\frac{11}{18}$	$\frac{13}{18}$	$\frac{3}{18}$

b

Figure 4. An example of a dither matrix and a corresponding threshold matrix

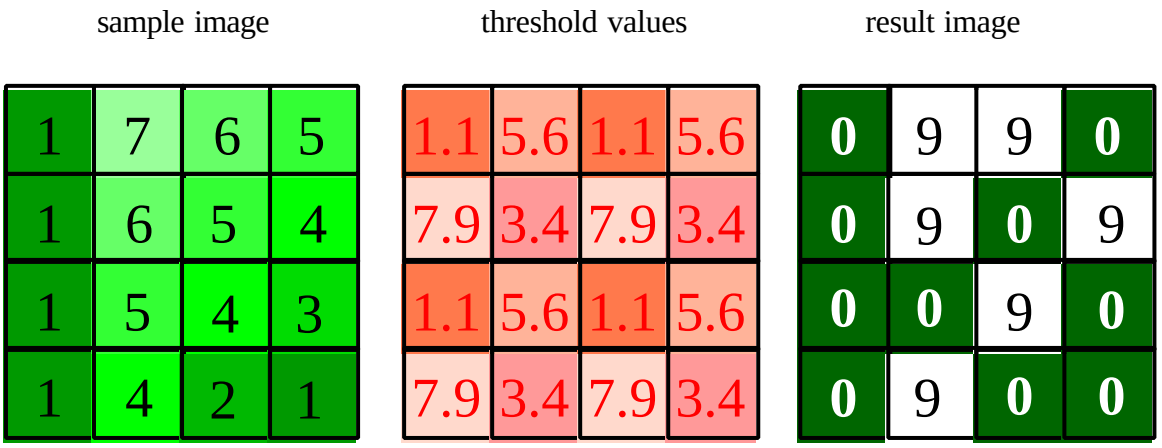


Figure 5. Threshold dithering.

In order to generate larger threshold matrices a recursive method can be used shown in Figure 6. Another opportunity is to use "magic squares" which produces less diagonal stripes (Figure 7).

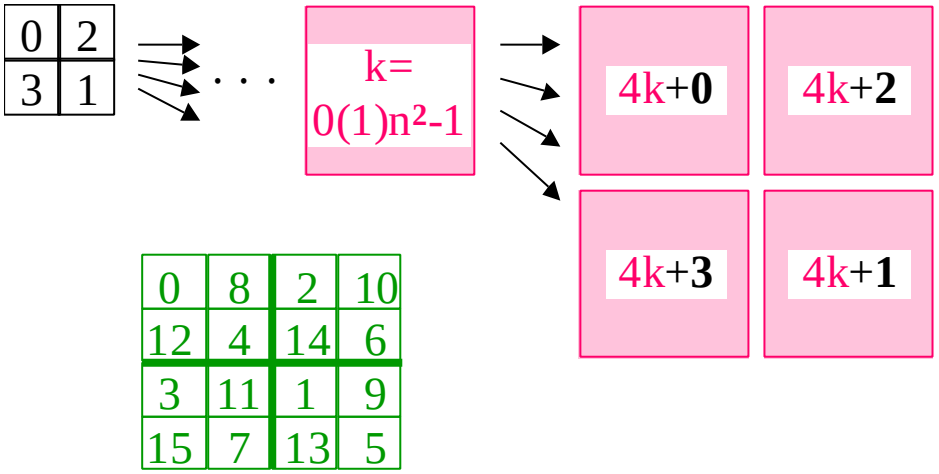


Figure 6. Recursive generation of threshold matrices.

0	14	3	13
11	5	8	6
12	2	15	1
7	9	4	10

Figure 7. "Magic squares".

1.3. Gray level dithering

Assume that, there are several gray levels available. Between neighboring levels *a* and *b* a threshold corresponding to a dither matrix value *k* is calculated as $a + (b - a)(2k + 1)/(2n^2)$. This calculation is done separately for each pixel and not once for a dither matrix, since it depends on the interval which contains the given pixel value. Figure 8 shows an example for gray level dithering using 4 available gray values (0, 3, 6, 9).

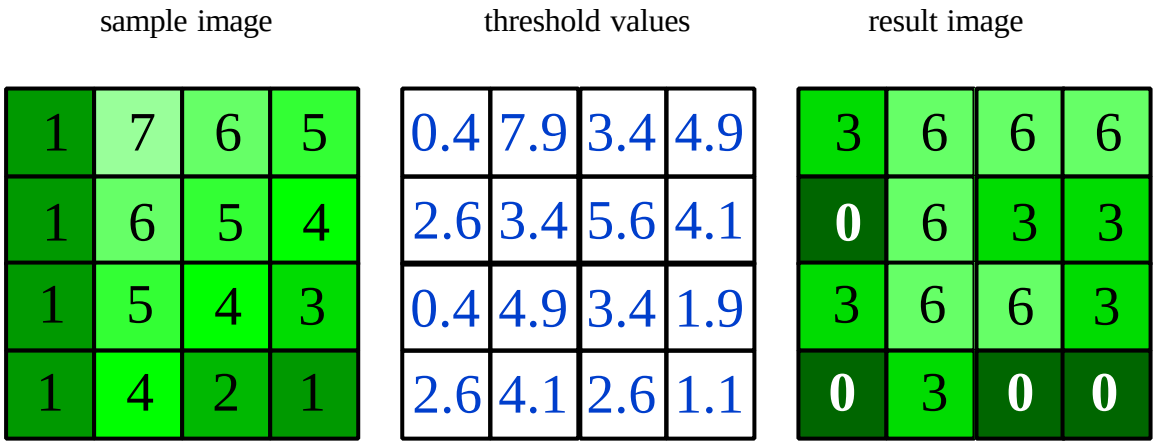


Figure 8. Gray level dithering example.

1.4. Dot diffusion dithering

Dot diffusion dithering simulates the traditional printing techniques for high resolution devices. This is demonstrated in Figure 9, where ordering of the threshold values generates larger dot areas.

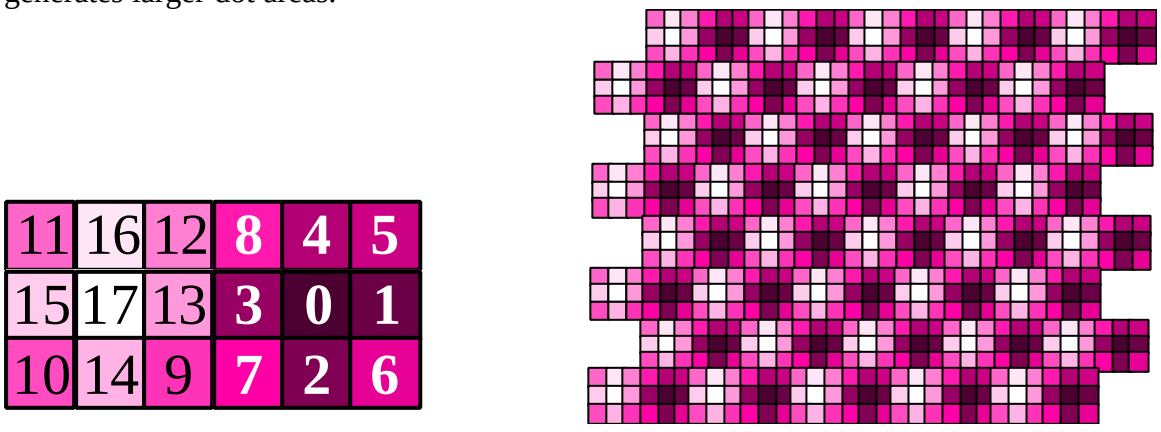


Figure 9. Dot diffusion dithering.

1.5. Stochastic dithering

Stochastic dithering methods use random numbers as threshold values. This technique has two advantages. The expectation value of the total error is zero and no regular artificial patterns appear. On the other hand, due to the bad distribution of the random numbers the results can be rather weak. One opportunity of improvement is to use a high resolution threshold matrix, where the threshold values are inserted one-by-one and always that position is used which is farthest away from all the other points. In order to determine the new location a force field function is used, therefore this method is called **forced random matrix dithering**.

1.6. Error diffusion dithering

Error diffusion dithering is represented by Floyd-Steinberg dithering, where the rounding error of every pixel is propagated to neighbor pixels and compensated there. There are variations of this technique according to that, which neighbor pixels are effected. Let's denote the correct values of a pixel line with $k_1, k_2, k_3 \ldots$, and the rounded values with r_1, r_2, $r_3 \ldots$. The rounding error e_1 of the first pixel is $e_1 = r_1 - k_1$ and in error e_i of each further pixel the error of the previous pixel is compensated, therefore $e_i = r_i - (k_i - e_{i-1})$.

k: 1 1 1 2 3 4 7 1 5 ...
r: 0 3 0 3 3 3 6 3 3 ...
f: -1 1 0 1 1 0 -1 1 -1 ...

$r_i := \text{round}\,(k_i - f_{i-1})$ $\qquad$ $f_i := r_i - k_i + f_{i-1}$

Figure 10. Error diffusion dithering.

Figure 10 demonstrates error diffusion dithering, where the available values are 0, 3, 6, and 9. In order to get better results the error can be distributed to several neighbors using normalized weighting. Figure 11 shows different weighted error distributions, and Figure 12 illustrates the dithering process using the first error weighting function.

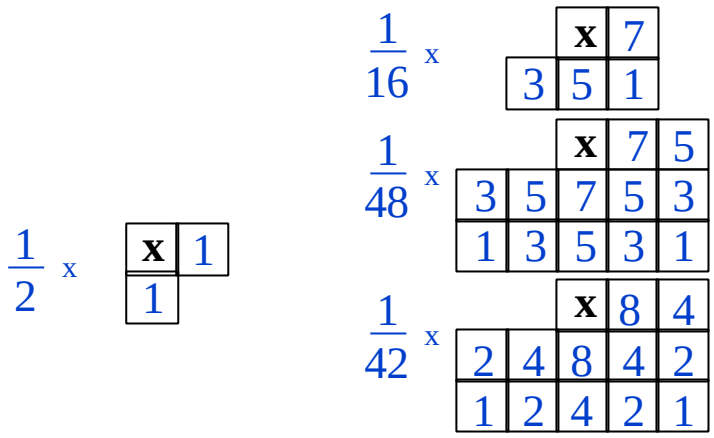


Figure 11. Different error distributions.

sample image | threshold values | result image

1	7	6	5
1	6	5	4
1	5	4	3
1	4	2	1

-1	1.5	.75	1.37
1.5	1.5	-.87	-.75
-.25	-1.37	.87	.06
-1.12	.75	-1.12	1.47

0	9	6	6
3	6	3	3
0	3	6	3
0	6	0	3

Figure 12. Error diffusion dithering.

2. Raster conversion

This chapter describes how to convert primitives like lines, circles etc. into pixels on a raster display. The most important requirement from these operations are efficiency and the support of hardware implementation.

2.1. Line conversion

In raster conversion of lines the following aspects have to be taken into account. The lines should appear straight even if they are short. They should also appear uniformly bright and the lightness should not depend on the direction. At last but not least, the endpoints should be exact. In order to fulfill these requirements the following **digital differential analyzer** (DDA) algorithm can be used:

```
#define ROUND(a) ((int)(a + 0.5))

void lineDDA (int xa, int ya, int xb, int yb)
 {
   int dx = xb – xa, dy = yb – ya, steps, k;
   float xIncrement, yIncrement, x = xa, y = ya;

   if(abs(dx) > abs(dy)) steps = abs(dx);
   else steps = abs(dy);

   xIncrement = dx / (float)steps;
   yIncrement = dy / (float)steps;

   setPixel(ROUND(x), ROUND(y));
   for(k = 0; k <steps; k++) {
     x += xIncrement;
     y += yIncrement;
     setPixel(ROUND(x), ROUND(y));
   }
 }
```

The main drawback of this simple DDA algorithm is the usage of real division, therefore it is not very efficient. The Bresenham's line drawing algorithm produces the same result using only integer operations like addition, subtraction and shift.

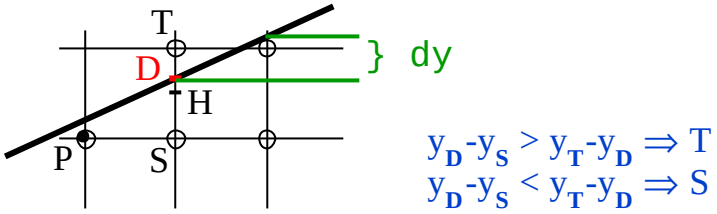


Figure 13. Raster conversion of a line.

This method is demonstrated in Figure 13. If $(y_d - y_s) > (y_t - y_d)$ then pixel T is the next point of the line, otherwise pixel S is drawn as the next point. If $(y_d - y_h) < 0$ (this is equivalent with $(y_d - y_s) < (y_t - y_d)$) then y_s and y_t do not change in the next step, otherwise $y_s = y_s + 1$ and $y_t = y_t + 1$. Here variable $d = y_d - y_h$ plays the role of a decision variable. In every step y_d is incremented with $dy = (y_2 - y_1) / (x_2 - x_1)$, thus $y_d = y_d + dy$. This algorithm works for a special case, where the steepness of the line is less than 45°. The general solution can be derived from this special case by mirroring and rotating with 90°, 180°, … angles. Using real division this algorithm could be implemented in the following way:

```
void BresenhamLine1(int x1, int y1, int x2, int y2)
 {
   int ys = y1;
   float d = - 0.5; // d = yD - yH
   float dy = (y2 – y1) / (x2 – x1);
   for(int xs = x1; xs <= x2; xs ++) {
     setPixel(xs, ys);
     d = d + dy;
     if(d > 0) {
       ys = ys + 1;
       d = d – 1; // because yH = yH + 1;
     }
   }
 }
```

Because of the floating point operations this implementation is not very efficient. Fortunately, introducing additional variables the real division can be avoided and the algorithm can be implemented using only integer operations, like addition, subtraction and shift:

```
void BresenhamLine2(int x1, int y1, int x2, int y2)
 {
   int ys = y1;
   int e = -((x2 – x1) >> 1);   // d * (x2 – x1)
   int de = (y2 – y1)         // dy * (x2 – x1)
   for(int xs = x1; xs <= x2; xs ++) {
     setPixel(xs, ys);
     e = e + de;
     if(e > 0) {
       ys = ys + 1;
       e = e – (x2 – x1);
     }
   }
 }
```

2.2. Circle conversion

The raster conversion of circles can also be done using Bresenham's circle drawing algorithm. In this case, the symmetry can be exploited, therefore only one eighth of the circle arc has to be calculated (Figure 14).

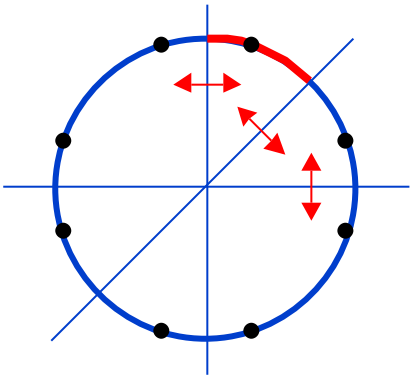


Figure 14. Exploiting symmetry in raster conversion of a circle.

The Brasenham's circle drawing algorithm uses a similar criterion as the line drawing method (Figure 15). If $(y_d - y_s) > (y_t - y_d)$ then pixel T is the next point of the circle, otherwise pixel S is drawn as the next point. If $(y_d - y_s) > (y_t - y_d)$ then point H is inside the circle. It means that $x_H^2 + y_H^2 < r^2$ or using the implicit equation of the circle: $f(x_H,y_H) = x_H^2 + y_H^2 - r^2 < 0$.

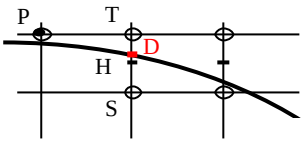


Figure 15. Raster conversion of a circle.

The new value d_{new} of the decision variable is calculated from the old value $d_{old} = d = f(x_p + 1, y_p - 1/2)$ as follows. If $d_{old} < 0$ then $H_{new} = H_{old} + (1,0)$, and $d_{new} = f(x_p+2, y_p-1/2) = (x_p+2)^2 + (y_p-1/2)^2 - r^2 \Rightarrow d_{new} = d_{old} + (2x_p+3)$. If $d_{old} > 0$ then $H_{new} = H_{old} + (1,-1)$, and $d_{new} = f(x_p+2, y_p-3/2) = (x_p+2)^2 + (y_p-3/2)^2 - r^2 \Rightarrow d_{new} = d_{old} + (2x_p - 2y_p + 5)$. This algorithm can be implemented the following way:

```
void BresenhamCircle(int r)
  {
    int x = 0, y = r; float d = 1.25 - r;
    do {
       Draw8(x, y);
       if(d < 0) d = d + 2 * x + 3;
       else {
         d = d + 2 * (x – y) + 5;
         y = y – 1;
       }
       x = x + 1;
    } while( y < x);
  }
```

The initialization of the decision variable is $d = 1.25 - r$ since $H = (1, r - 0.5) \Rightarrow d = f(H) = 1 + (r^2 - r + 0.25) - r^2 = 1.25 - r$.

2.3. Raster transformations

This section describes how to apply geometrical transformations to raster images. There are four major raster transformations: **translation**, **scaling**, **shearing**, and **rotation**. The translation is trivial, therefore no further discussion is necessary. Raster scaling requires resampling of the original image. Figure 16 and Figure 17 demonstrate upscaling and downscaling respectively, where the purple grid represents the old resolution, while the black grid represents the new resolution. In both cases the center point of a pixel in a new resolution defines its color.

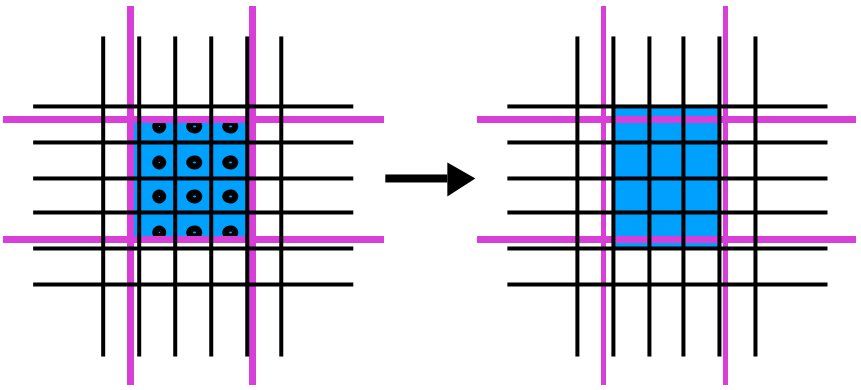


Figure 16. Scaling up a raster image.

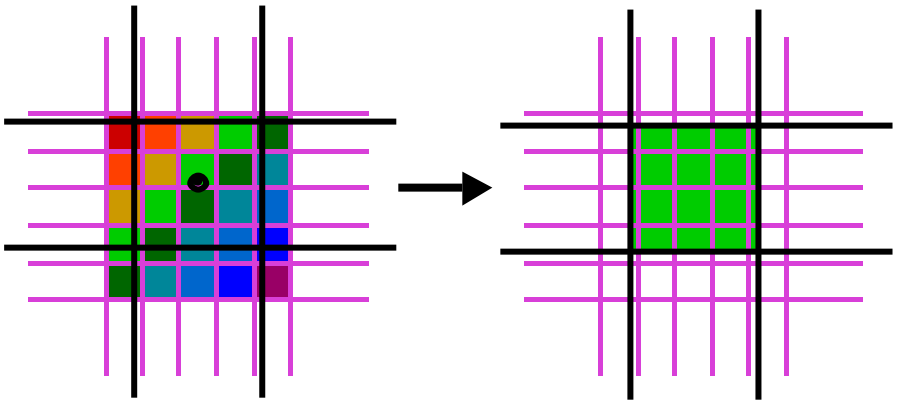
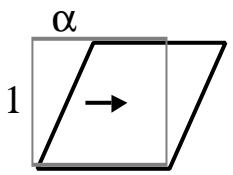


Figure 17. Scaling down a raster image.

A shearing transformation can be defined by a 2×2 transformation matrix. Figure 18 shows x-shearing and y-shearing with the corresponding matrix operations.

x-shearing

$$(x \;\; y) \times \begin{pmatrix} 1 & 0 \\ \alpha & 1 \end{pmatrix} = (x+\alpha y \;\; y)$$



y-shearing

$$(x \;\; y) \times \begin{pmatrix} 1 & \beta \\ 0 & 1 \end{pmatrix} = (x \;\; y+\beta x)$$

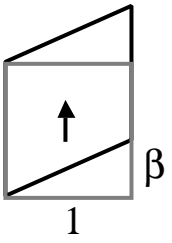


Figure 18. Shearing with matrix transformations.

Raster shearing can be performed as multiple application of line raster conversion (e.g. Bresenham) as it is demonstrated in Figure 19. Note that, there is no information loss during this operation.

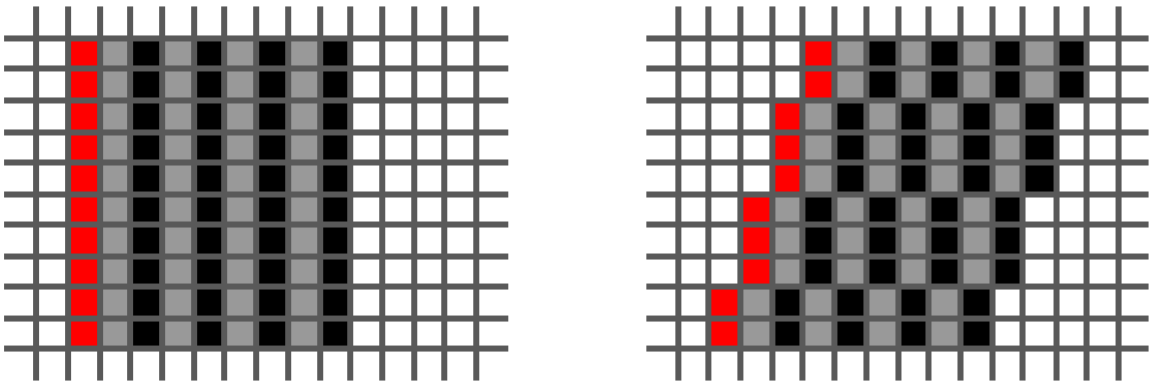
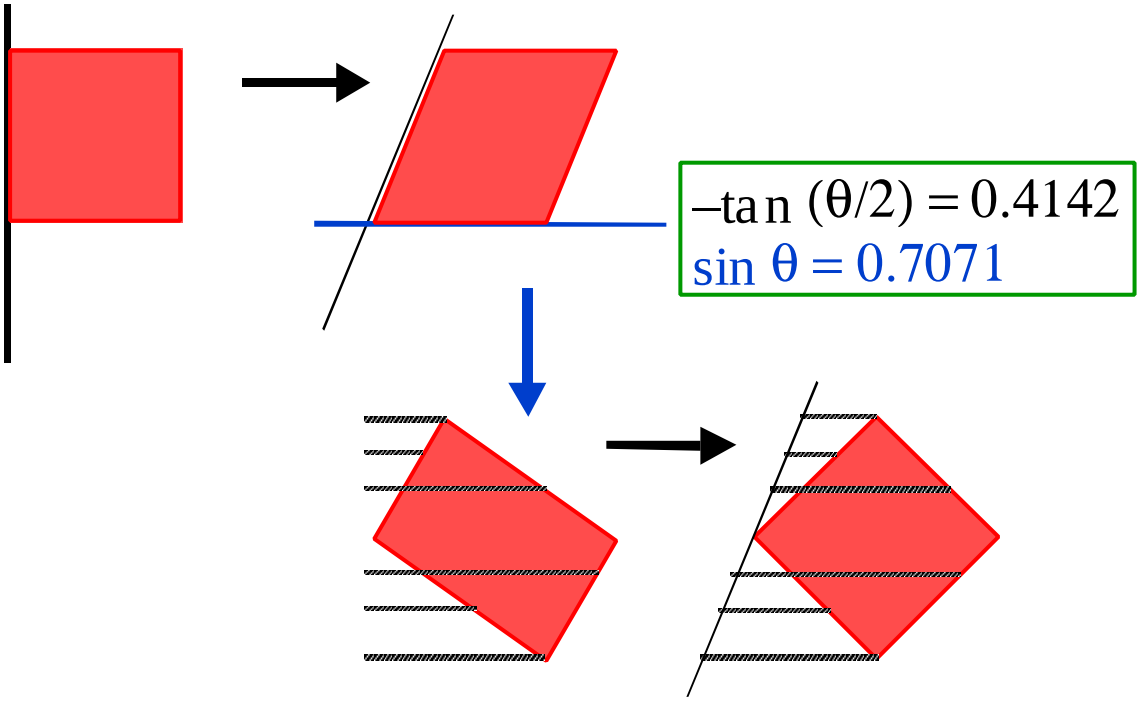
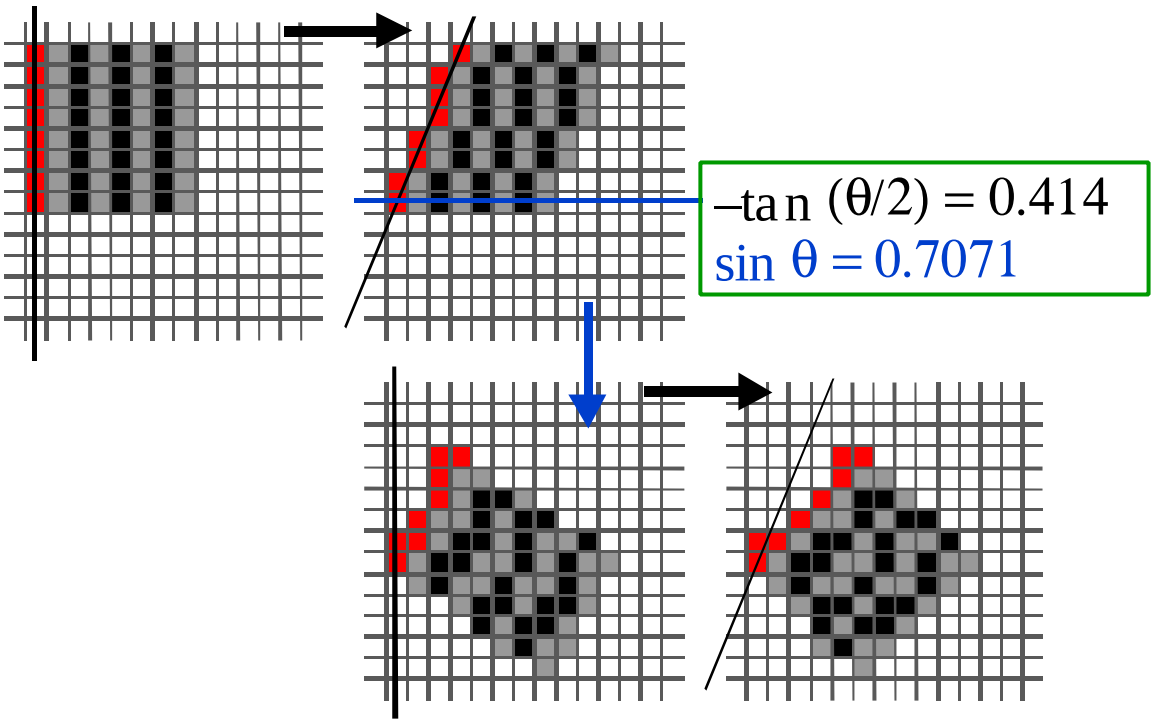


Figure 19. Raster shearing.

Raster rotation can be done subdividing the rotation transformation into three shear operations (Figure 20). Using this approach no resampling is necessary for rotation, therefore it is computationally efficient.





$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix} = \begin{pmatrix} 1 & 0 \\ \alpha & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & \beta \\ 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 \\ \alpha & 1 \end{pmatrix}$$

where $\alpha = -\tan(\theta/2)$

and $\beta = \sin\theta$

Figure 20. Raster rotation using three shears.