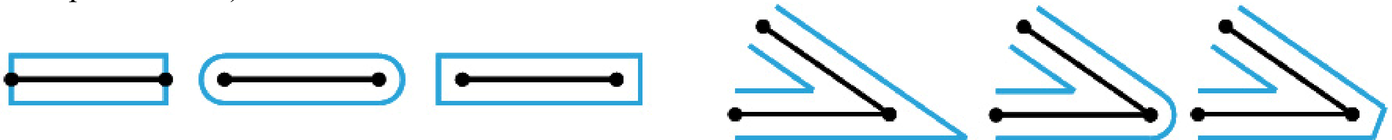
Attributes of Graphics Primitives

Graphics primitives can be created with a number of properties, also called attributes. This chapter also includes some filling algorithms, which are considered to be attributes of polygons, too.

Attributes of (Points and) Lines

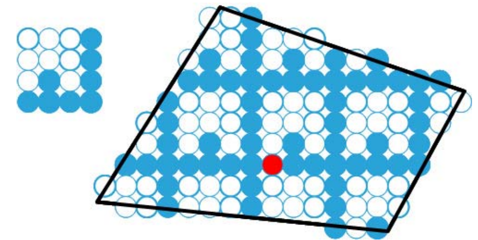
Besides well known line attributes like stroke width, line pattern, color or stroke type, there exist some other attributes which are less obvious. One of them is the type of line ending and the apex shape (when two lines end up in a corner) at thicker lines:



Furthermore, anti-aliasing is an important topic for lines. More details follow in later sections.

Attributes of (2D-)Polygons and Surfaces

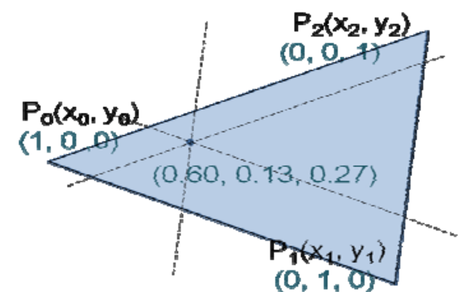
Obviously the attributes of a surface's edges are the same as those of simple lines. But now we additionally have to take into account the surface itself, which can be provided with a filling. Filling patterns are normally created by repeated application of a basic pattern originating from some reference point (also called seed point).



Many applications also require creating a combination of the newly drawn pattern and the background. There are several different ways which base upon logic operations like AND, OR, XOR. Generally the blending of color is performed by a linear combination of the existing background color B and the given foreground color F: $P = t*F + (1-t)*B$

Triangle Rasterization

For filling triangles, the use of *barycentric coordinates* is common. Each point in the plane is represented as weighted average of the 3 triangle vertices: $P=\alpha P_0 + \beta P_1 + \gamma P_2$.
(α, β, γ) are then called the barycentric coordinates of the point P, with $\alpha+\beta+\gamma=1$.
All points with $0< \alpha, \beta, \gamma<1$ are inside the triangle.
When filling a triangle, the barycentric coordinates of each pixel of a (preferably tight) bounding area are calculated, and all pixels with $0< \alpha, \beta, \gamma<1$ are rendered. The rendering color can easily be linearly interpolated from corner values using the available weights (α, β, γ). If we denote the line through P_1 and P_2 with $l_{12}(x,y) = a_{12}x+b_{12}y+c_{12}=0$ then α of the point P(x,y) is $\alpha = l_{12}(x,y) / l_{12}(x_0,y_0)$, β und γ analogous.
To avoid rendering the common border of 2 adjacent triangles twice, only pixels are rendered that lie inside the triangle borders. Pixels that are exactly on a border need a special treatment.



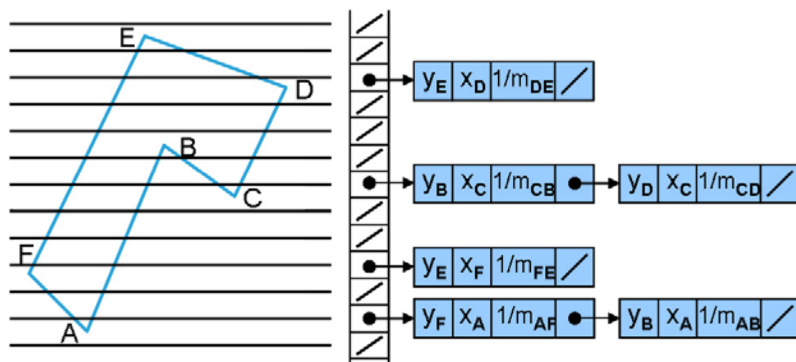
When filling a more complex polygon it has to be defined which parts are "inside" and which areas are "outside" (refer to "01 Graphics Primitives"). There are two categories of filling methods. The scan-line method intersects each scan-line with the polygon (or the surface border), and parts which lie inside (spans) are filled. The flood-fill method fills the surface originating from a seed point in all directions until the whole polygon is recognized to be filled.

Scan-line Filling

All edges of the polygon are sorted by the lower y-value of their endpoints (bucket sort). For each edge, the following information is stored:

[max. y-value, start x-value, slope]

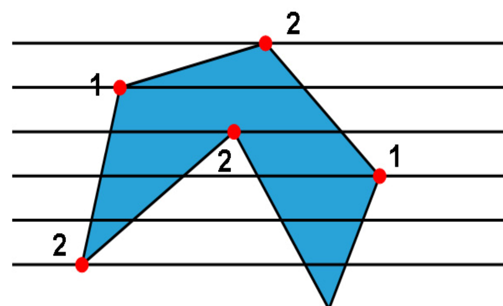
Using this data structure, for each scan-line in the list (bottom up) we create a list of “active” edges, which are those that intersect that scan-line. Then the scan-line is filled from the first intersection point to the second one, from the third to the fourth, from 5th to 6th and so on.



The list of active edges is generated incrementally. Starting with zero active edges, for each y-value we can look up in the sorted-edge-list, whether there is a new edge beginning there. If so, this edge is added to the current active-edge-list. At the same time, all edges whose maximum y-value has been exceeded are removed (not active anymore). The active edges in this list are always sorted by their intersection points (from the left to the right), so that drawing can always be performed instantaneously.

The intersection points can also be generated incrementally. Beginning from the (exact) intersection point (x_k,y_k) of a scan-line with an edge, we get the next higher intersection point (x_{k+1},y_{k+1}) by
$x_{k+1}= x_k+1/m$ and $y_{k+1}= y_k+1$. Now it also becomes clear why we’ve stored the slope 1/m in each node of the edge list.

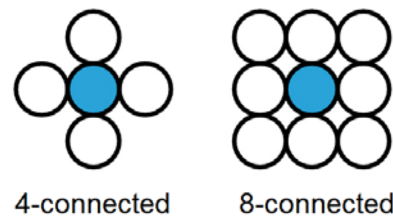
If a corner of the polygon lies exactly on a scanline we have to ensure that the number of intersection points for this point is correct. Some points have to be counted once, others twice (see figure). To avoid this problem, the point coordinates are often shifted up or down by a small value ε. This value has to be so small that it is not recognized, but large enough to move the point above or below the scan-line.



Flood-fill Algorithm

Originating from some start point (reference point, seed point), this algorithm fills in all directions until hitting the border. This border can either be defined explicitly, e.g. by a border of some certain color, or implicitly, e.g. in the case that an already colored surface is filled with a new color. There are also variants mixing implicit and explicit definitions. However, this definition is just used to formulate the stop criterion for the algorithm. Without loss of generality, for the following code we want to assume that the surface to be filled is already drawn in a defined color, and shall be filled with a new one.

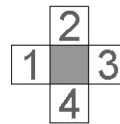
Proceeding in all directions requires a definition of the allowed directions. 4-connected means that a connection is defined only along the four main directions, 8-connected also accepts diagonal pixels as connected. It can easily be understood that for a 4-connected surface an 8-connected border is sufficient, but an 8-connected surface requires a 4-connected border line.

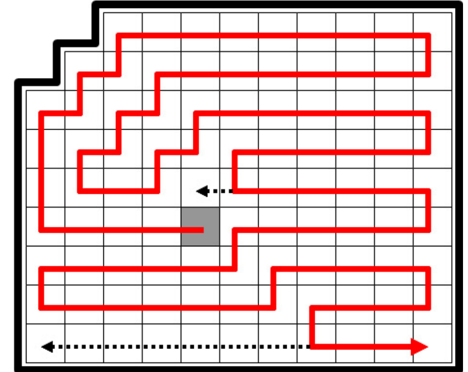


Floodfill for 4-connected surfaces can easily be implemented as follows:

```
void floodFill4 (int x, int y, int new, int old)
{ int color;
    /* set current color to new */
    getPixel (x, y, color);
    if (color = old) {
        setPixel (x, y);
        floodFill4 (x-1, y, new, old);  /* left */
        floodFill4 (x, y+1, new, old);  /* up */
```

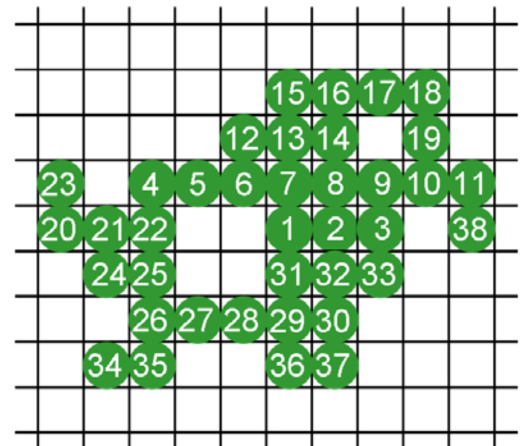
```
        floodFill4 (x+1, y, new, old);   /* right */
        floodFill4 (x, y-1, new, old);   /* down */
}}
```





However, this procedure creates a filling order which leads to a very high recursion depth, generally up to the number of pixels to fill. The upper left of the figure shows the recursion order, the long red arrow inside the polygon shows the depth of the recursion for this particular case.

To avoid this high recursion depth, filling can be performed iteratively in the horizontal direction and the recursion is only applied vertically. Of course we have to ensure that *all* spans above and below are called recursively (a span is a horizontal, continuous series of pixels which are processed collectively).



The example to the right demonstrates the filling order, where one of the pixels 1, 2 or 3 was chosen as start pixel. Recursion is first performed upwards, then downwards. At each call all spans in this direction have to be processed. After the span 4 to 11, the recursion goes upwards and draws from left to right, then goes downwards and draws from left to right as well. Already filled parts are recognized and the recursion terminates (e.g. going downwards from 4 to 11, 1 to 3 is already processed). Although the total depth of recursion theoretically can be very high with this method too, in practice it is proportional to the number of scanlines of the polygon.

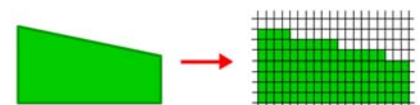
Attributes of Text

The attributes that can be assigned to text are common knowledge nowadays: Font (e.g. Courier, Arial, Times, **Broadway**, …), style (normal, **bold**, *italic*, underlined, …), size, text direction, color, alignment (left, right, centered, justified) and so on.

Attributes of 3D Polygons

Generally, 3D polygons are surface elements of objects in 3D space. Therefore their attributes mostly describe properties of those objects' surfaces: color, material parameters (roughness, absorption, …), transparency, texture, geometrical microstructure, reflection behavior etc., which create scene dependent effects under a given illumination. Additionally, normal vectors and texture coordinates for each vertex are often included to allow for fast and adequate calculation of the polygon's appearance. Later we will see how this is used efficiently.

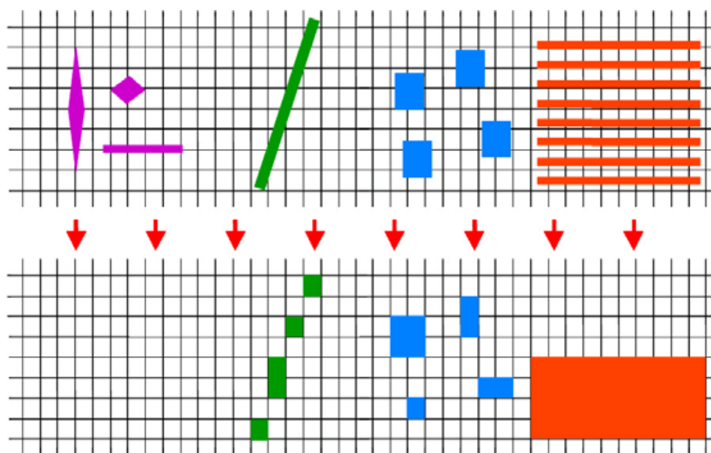
Aliasing and Anti-aliasing



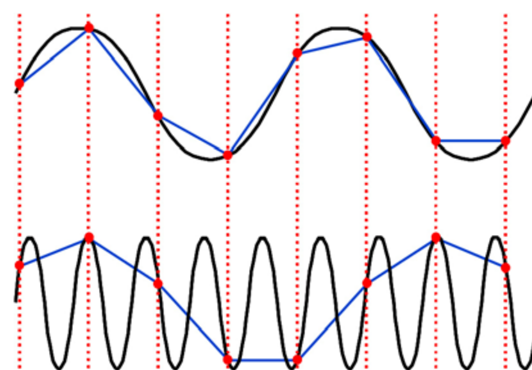
Aliasing effects ['eiliæsiŋ] are errors which result from the transformation (discretization) of analog data to digital information. The most prominent aliasing effect in computer graphics comes from the fact that in raster images pixels can store just one single value, while they actually represent a whole (small) area of the image containing a range of signal values (loss of information). Visible

aliasing effects can be caused by too low resolution, too few colors available, too few images per second, geometrical errors, and numerical errors.

Anti-aliasing is the term for methods that reduce the unwanted aliasing artifacts. Since an enhancement of hardware is unrealistic in most cases, software techniques are mainly used. In the following we only cover anti-aliasing that treats the resolution problem. Beside the jagged edges some other known effects are: disappearance of small objects, discontinuous narrow objects, equal-sized objects appearing in different size, complete destruction of fine detailed textures (see figure to the right).

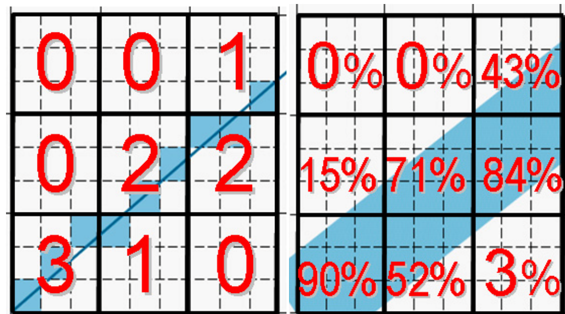


The cause of aliasing is an insufficient sampling rate of the original image. The theoretical basis is described by the Shannon *Sampling Theorem*. This theorem says that information can only be correctly reconstructed if the used sampling frequency (sampling rate) is at least double as high as the highest information frequency within the picture. This limit is called the *Nyquist limit*. The figure shows how a too coarse sampling rate of a signal (curve) can lead to a completely false reconstruction (polygon strip). Such errors can be reduced either by prefiltering of the input signal or by post-processing of the resulting image. In any case prefiltering leads to better results than post-processing. The central strategy of prefiltering is *super-sampling* (also called *oversampling*).



Anti-aliasing of Lines

Pixels which are more centrally intersected by some line should get more of the line color than pixels which are just touched slightly by the line. Therefore we subdivide each pixel into a number of sub-pixels and choose a result intensity for the pixel proportional to the number of sub pixels which lie on the line. For broader lines we choose the intensity of the line color according to the percentage of coverage of the pixel by the line. Because a pixel's center is more relevant than its border, sometimes sub-pixels in the center get more weight than those at the pixel's border („weighted oversampling").



Anti-Aliasing of Polygon Edges

For polygon edges we have the same alternatives as for lines: either we work with super-sampling or we calculate the pixel's degree of coverage by the polygon analytically. The calculation of the degree of coverage is done at raster conversion time, i.e. when creating the border lines and the filling of the polygon. When calculating the endpoints of the spans (scanline filling method), we have enough information available to extract the degree of coverage value almost directly without further calculation. We recall the decision variable p_k of the Bresenham line algorithm, whose sign indicates which pixel is to be drawn next. This variable can be transformed in a way such that its value equals the percentage of coverage of the last pixel: $p' = y - y_{mid}$, where $y_{mid} = (y_k + y_{k+1})/2$, has the same sign as p_k. If we use $p = p' + (1 - m)$, then we have to compare with $(1 - m)$ instead of 0 though, but $0 \le p \le 1$ is guaranteed, and p equals the degree of coverage at the point x_k. In this way anti-aliasing can be calculated incrementally very quickly. For other angles 90°-rotations and/or mirrored versions of this method are used.