# Graphic Primitives

In order to produce graphical output on a device, the programming language in use has to provide accordant commands. The simplest graphical elements that can be created by these commands are called G*raphic Primitives*. Beside simple drawings, such fundamental elements can also be formatting commands and meta information. The most important commands are:

in 2D:
- points, lines
- polygons, circles, ellipses and other curves (can also be filled areas)
- bitmap-operations
- letters and symbols

in 3D:
- triangles and higher polygons
- free-form surfaces

Further commands are needed in order to define the properties of the primitives, such as color, fill pattern, texture, material property or transparency. Mostly these commands cause the subsequent primitives to be drawn with the last defined properties.

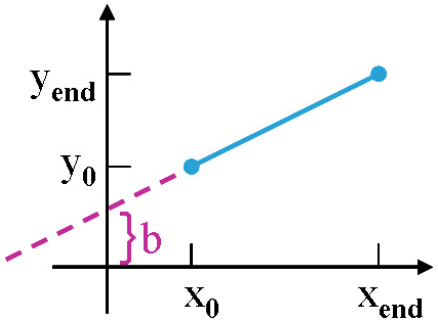
## Line Algorithms

Especially the drawing of straight lines on raster devices is a very important operation. The base method DDA (Digital Differential Analyzer) was cleverly refactored by Bresenham in a way that enables it to work only with integer operations, which made it *faster* and *easier to implement in hardware*.

Notation: A line is specified in the form $y = mx + b$,
where m is the slope of the line,
and (0,b) is the intersection with the y-axis.

m and b can be calculated from the line's endpoints $(x_0,y_0)$ and $(x_{end},y_{end})$ by

$m = (y_{end} - y_0) / (x_{end} - x_0)$ $\quad$ $b = y_0 - mx_0$



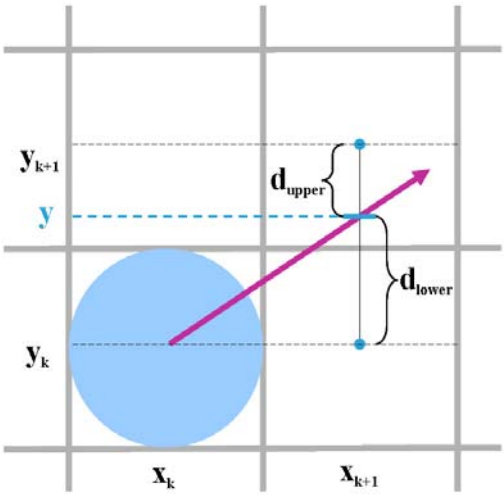

The simple ***DDA-Algorithm*** for $|m| < 1$ adds for each step to the right (x+=1) the value m to $y_0$, and rounds the result to an integer afterwards. This results in a line, which creates exactly one pixel on the line for each x-value.

```
dx = xEnd – x0; dy = yEnd – y0;
m = dy / dx;

x = x0;  y = y0;
setPixel (round(x), round(y));

for (k = 0; k < dx; k++)
  { x += 1; y += m;
  setPixel (round(x), round(y)}
```

For $|m| > 1$, x and y are swapped and the same procedure is executed in vertical direction. The following Bresenham-Algorithm will also only be explained for $0 < |m| < 1$, all other directions work by mirroring and rotation by 90°.

The ***Bresenham-Algorithm*** creates exactly the same result as the simple DDA, but suffices using only integer arithmetic. Thus it is faster, easier to implement in firm- or hardware, and furthermore it can easily be adopted to fit other curves like circles, ellipses, spline curves and so on.

For $0 < |m| < 1$, from the known location of the pixel in the column $x_k$ the y-value of the pixel in the next column $x_{k+1}$ is not being exactly calculated, but rather there's made a decision, whether $y_k$ or $y_{k+1}$ lies closer to the exact y-value.




From $y = mx + b$ follows the exact y-value for the column right to $x_k$

$$y = m\cdot(x_k + 1) + b$$

The distance to $y_k$ is $\mathbf{d_{lower}} = y - y_k = m(x_k + 1) + b - y_k$
The distance to $y_{k+1}$ is $\mathbf{d_{upper}} = (y_k + 1) - y = y_k + 1 - m(x_k + 1) - b$

If the difference $d_{lower} - d_{upper} = 2m\cdot(x_k + 1) - 2y_k + 2b - 1$
is negative, then the lower point $(x_{k+1},y_k)$ is chosen. Otherwise, if the difference is positive, the upper point $(x_{k+1},y_{k+1})$ is set.

Substituting $m = \Delta y/\Delta x$ ( $\Delta x = x_{end} - x_0$, $\Delta y = y_{end} - y_0$ ), and multiplying this difference with $\Delta x$ results in a decision variable $p_k = \Delta x\cdot(d_{lower} - d_{upper}) = 2\Delta y\cdot x_k - 2\Delta x\cdot y_k + c$ , which has *the same sign* as $d_{lower} - d_{upper}$, but doesn't need any division.

Now, when the decision variable $p_k = 2\Delta y\cdot x_k - 2\Delta x\cdot y_k + c$ for $x_k$ is known, the decision variable for $x_{k+1}$ can easily be calculated:

$$\mathbf{p_{k+1}} = 2\Delta y\cdot x_{k+1} - 2\Delta x\cdot y_{k+1} + c + p_k - 2\Delta y\cdot x_k + 2\Delta x\cdot y_k - c = \mathbf{p_k + 2\Delta y - 2\Delta x\cdot(y_{k+1} - y_k)}$$

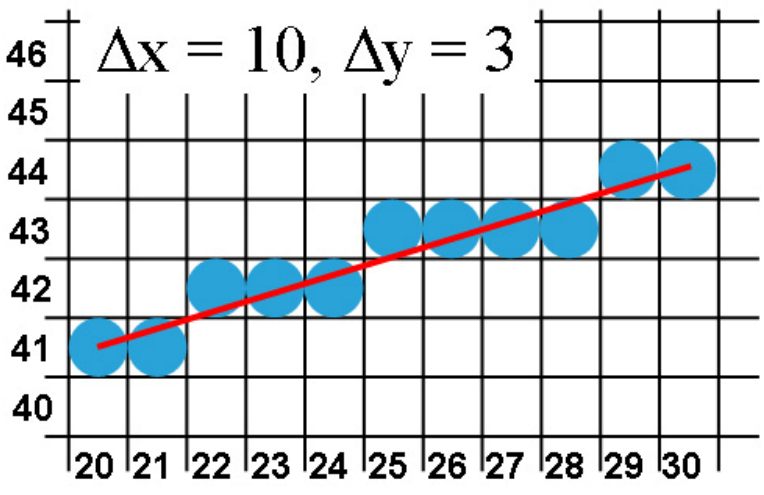
namely by just adding a number, which is constant for all points of the line. $p_0 = 2\Delta y - \Delta x$

With this the Bresenham-Algorithm roughly looks like this:

1. store left line endpoint in $(x_0,y_0)$
2. plot pixel $(x_0,y_0)$
3. calculate constants $\Delta x$, $\Delta y$, $2\Delta y$, $2\Delta y - 2\Delta x$, and obtain $p_0 = 2\Delta y - \Delta x$
4. At each $x_k$ along the line, perform test:
5. if $p_k<0$
6. then plot pixel $(x_k+1,y_k)$; $p_{k+1} = p_k + 2\Delta y$
7. else plot pixel $(x_k+1,y_k+1)$; $p_{k+1} = p_k + 2\Delta y - 2\Delta x$
8. perform step 4 $(\Delta x - 1)$ times.

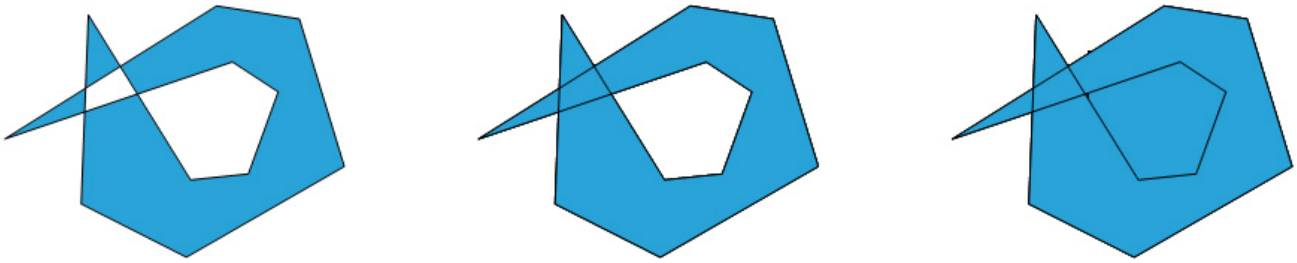
Example:

| k | $p_k$ | $(x_{k+1},y_{k+1})$ |
|---|---|---|
| | | (20,41) |
| 0 | -4 | (21,41) |
| 1 | 2 | (22,42) |
| 2 | -12 | (23,42) |
| 3 | -6 | (24,42) |
| 4 | 0 | (25,43) |
| 5 | -14 | (26,43) |
| 6 | -8 | (27,43) |
| 7 | -2 | (28,43) |
| 8 | 4 | (29,44) |
| 9 | -10 | (30,44) |



## Filled Polygons

When filling polygons you have to decide, what to fill at all. With simple closed curves, “inside” and “outside” can easily be defined. But what about more complicated curves?



***Odd-Even-Rule:*** When shooting an arbitrary half-ray originating in a given point, then the point lies inside, if the number of intersections of the ray with the curve is odd, otherwise the point lies outside (left picture).

***Nonzero-Winding-Number-Rule:*** Points lie outside, if an arbitrary half-ray intersects the same number of clockwise and counter-clockwise edges, otherwise they lie inside (center picture).

***All-In-Rule:*** Any point which is somehow surrounded by a closed subset of the curve’s edges lies inside. Rarely used, mostly when playing Poker☺ (right picture).

After deciding, which points are interpreted to be “inside”, filling can be performed by either filling line by line (*Scanline*-Algorithms), or originating from an inner point filling in all directions (*Flood-Fill*-Algorithms).

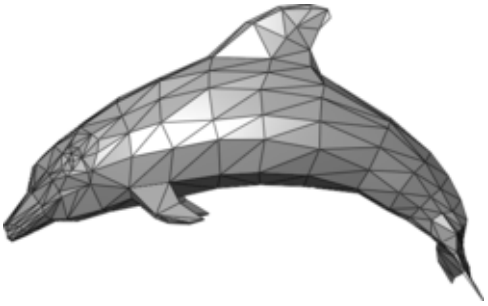
A polygon is called “convex”, if all inner angles are smaller than 180°, otherwise it’s called “concave”. Since convex polygons create much less special cases, most algorithms are only designed for convex polygons. Therefore, methods are needed to cut down concave polygons to a number of convex ones.

## Letters And Symbols

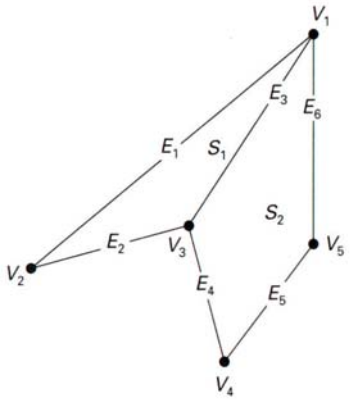
Text is defined by fonts and properties. Fonts with serifs (upper example) are better suited for flow text, fonts without serifs are better suited for catchy, striking text. Additionally, font properties like italic/non-italic, normal/bold, underlined etc. are defined. A font’s shape is normally defined by the silhouette curves of its letters, in some applications also by a pixel raster graphic.


## Polygon Lists

3-dimensional objects are mostly represented by polygon lists (in most cases triangles). A number of polygons, which together describe the surface of an object, is called *Boundary Representation* („B-Rep“). Beside its geometric information, B-Rep data structures also contain attributes. The geometry consists of point lists, edge lists, surface lists and has to be verified for consistency and completeness.



The following example shows for a simple object with two polygons, how point list (vertex table), edge list (edge table) and surface list (surface table) refer to each other. The geometry information itself is stored only in the point list, the other lists describe the topology of the object.




**VERTEX TABLE**

| | |
|---|---|
| $V_1$: | $x_1, y_1, z_1$ |
| $V_2$: | $x_2, y_2, z_2$ |
| $V_3$: | $x_3, y_3, z_3$ |
| $V_4$: | $x_4, y_4, z_4$ |
| $V_5$: | $x_5, y_5, z_5$ |

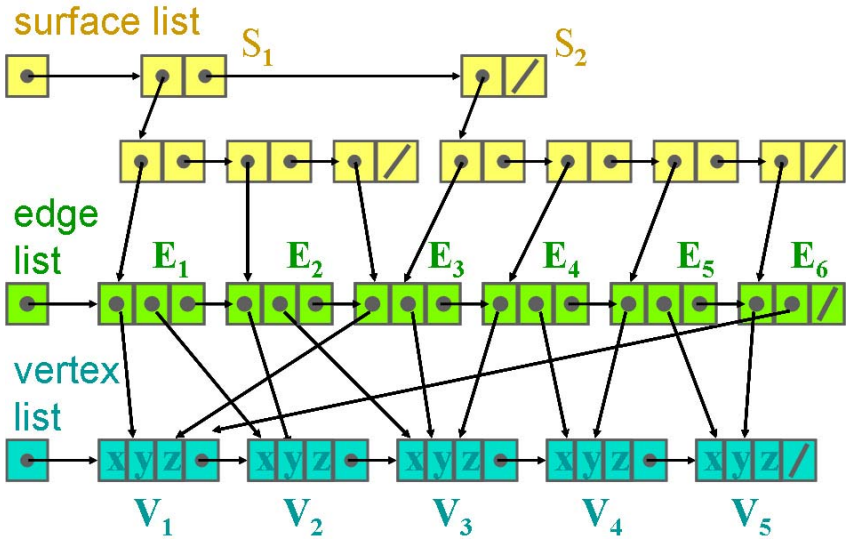
**EDGE TABLE**

| | |
|---|---|
| $E_1$: | $V_1, V_2$ |
| $E_2$: | $V_2, V_3$ |
| $E_3$: | $V_3, V_1$ |
| $E_4$: | $V_3, V_4$ |
| $E_5$: | $V_4, V_5$ |
| $E_6$: | $V_5, V_1$ |

**POLYGON-SURFACE TABLE**

| | |
|---|---|
| $S_1$: | $E_1, E_2, E_3$ |
| $S_2$: | $E_3, E_4, E_5, E_6$ |

The same structure can be described by pointer lists:




The representation of each individual polygon surface includes the plane Ax + By + Cz + D = 0, in which it is embedded, and its corner points $V_1$ to $V_n$. From the plane parameters A, B, C, D we retrieve the surface normal vector on the plane (A, B, C). The two sides of the polygon are defined as *Backface*, which is oriented towards the inside of the object, and *Frontface*, which is part of the outside form of the object. "Behind the polygon" lie all those points, which can be seen from its backface, "in front of the polygon" all those points from where one can directly see its front face.

When assuming a right-handed coordinate system and arranging the vertices of each polygon (seen from its front side) in a mathematically positive sense (that is, counter-clockwise), then we can say for a point (x, y, z):

if Ax + By + Cz + D = 0 then the point lies **on** the plane (intersects the plane)
if Ax + By + Cz + D < 0 then the point lies **behind** the plane
if Ax + By + Cz + D > 0 then the point lies **in front** of the plane

Similarly, from a set of three consecutive vertices $V_1$, $V_2$, $V_3$ we can calculate a normal vector N with the formula $N= (V_2 – V_1) \times (V_3 – V_1)$ which is oriented outwards. We will need these facts later on.