Computer Graphics

Selected Lecture Notes

Partha Bhowmick

IIT Kharagpur

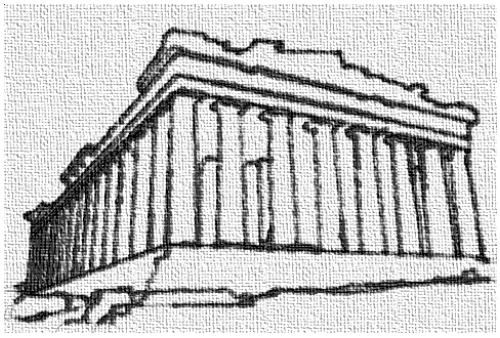
http://cse.iitkgp.ac.in/~pb

2018

Contents

Chapter 2

2D Drawing



When I see a white piece of paper, I feel I've got to draw. And drawing, for me, is the beginning of everything.

—Ellsworth Kelly

2.1 Line Drawing

A *digital line segment* (DLS) is obtained by digitization of a real straight line segment. Given two pixels $p_1(x_1, y_1)$ and $p_2(x_2, y_2)$, with $0 \leqslant x_1 < x_2, 0 \leqslant y_1 < y_2$, and $x_2 - x_1 \geqslant y_2 - y_1$, the DLS connecting p_1 and p_2 is a 0-connected digital curve satisfying the following properties:

1. Each pixel has $x_1 \leqslant x \leqslant x_2$.
2. The smaller between x-distance and y-distance of each pixel from the underlying real line is at most $\frac{1}{2}$.

Fig. 2.1 shows an example.

2.1.1 Bresenham's Algorithm

For designing the algorithm, we need the following observations.

Observation 1 For each integer $x \in [x_1, x_2]$, there is exactly one integer point (x, y) that belongs to the DSL. Hence, we can increment x from x_1 to x_2 at unit step and compute the corresponding integer value of y.

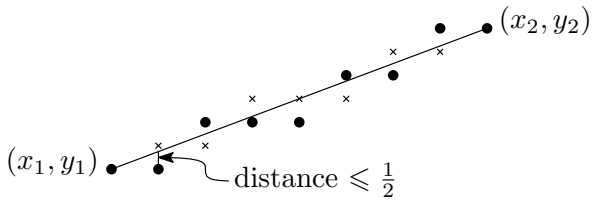


Figure 2.1: An example of DLS with $x_2 - x_1 = 8, y_2 - y_1 = 3$.

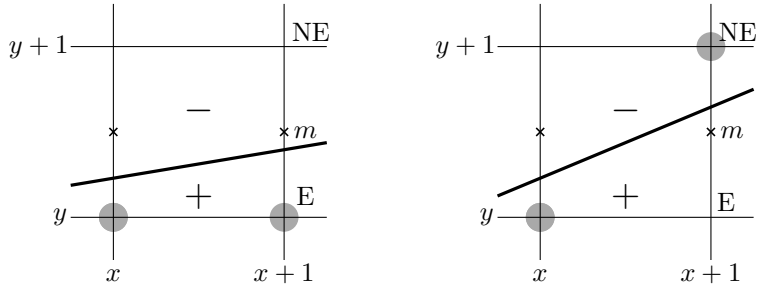


Figure 2.2: Two possible cases in Bresenham's line drawing algorithm. Left: E is selected, as m lies above l. Right: NE is selected, as m lies below l.

Observation 2 As the slope lies in the interval $[0, 1]$, from each pixel $p(x, y)$, the next pixel would be either East: $\mathrm{E}(x + 1, y)$ or Northeast: $\mathrm{NE}(x + 1, y + 1)$. See Figure 2.2.

Observation 3 Let $l : ax + by + c = 0$ be the real line passing through p_1 and p_2. Consider the *decision function* $f(x, y) = ax + by + c$. Let $p(u, v)$ lie on l and $p'(u, v + k)$ lie above l. Then $f(p) = 0$, and so $f(p') = f(p) + bk = bk$. As $k > 0$, the sign of the half-plane above l is same as that of b.

Selection of pixels Let $d_x = x_2 - x_1$ and $d_y = y_2 - y_1$. Then, with $s = d_y/d_x$, we get the equation of l as $y = sx + t$, or, $d_y x - d_x y + nd_x = 0$, or, $ax + by + c = 0$, where,

$$a = d_y,\ b = -d_x,\ c = td_x. \tag{2.1}$$

As $b < 0$, using Observation 3, we can conclude the following about any real point p (see Figure 2.2).

$$f(p) \begin{cases} > 0 & \iff p \text{ lies below } l \\ < 0 & \iff p \text{ lies above } l \\ = 0 & \iff p \text{ lies on } l \end{cases}$$

Let $p(x, y)$ be the current pixel selected by the algorithm (Figure 2.2). By Observation 2, the next pixel will be either E or NE depending on whether $m(x + 1, y + \frac{1}{2})$ lies above l or not. For this, compute $f(m) := a(x + 1) + b(y + \frac{1}{2}) + c$, and then select NE if $f(m) > 0$, and E otherwise.

Computation of f See Figure 2.3. To simplify the computation, we increment the value of $f(m)$ at a midpoint $m(x + 1, y + \frac{1}{2})$ to compute that of $f(m')$ at the next midpoint m'. Two possible increments:

i) if E is selected based on $f(m)$, then $m' = (x + 2, y + \frac{1}{2})$, or, $f(m') = f(m) + d_E$, where $d_E = a$.
ii) if NE is selected based on $f(m)$, then $m' = (x + 2, y + \frac{3}{2})$, or, $f(m') = f(m) + d_{NE}$, where $d_{NE} = a + b$.

The 1st midpoint is $m_1(x_1+1, y_1+\frac{1}{2})$. We get $f(m_1) = a(x_1+1)+b(y_1+\frac{1}{2})+c = f(p_1)+a+\frac{1}{2}b = a+\frac{1}{2}b$, since $f(p_1) = 0$. It's not an integer but a half-integer, if b is odd. To make all computations in the integer domain, we just double it. Doubling f does not affect its sign, but it's an advantage, as we need to always check the sign of an integer expression ($2a + b$ to start with). We redefine

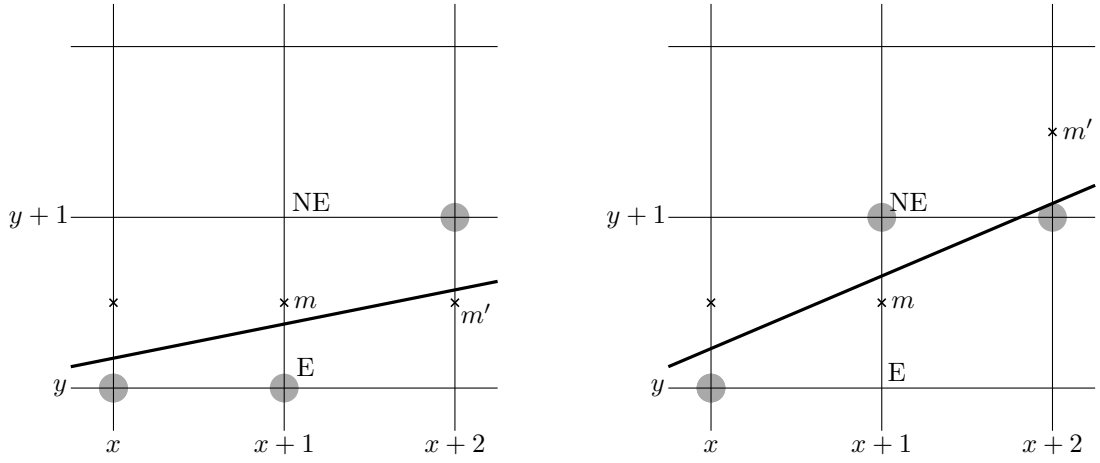


Figure 2.3: Computation of decision function f in Bresenham's algorithm. Left: E is selected, and so $f(m') = f(m) + d_E$. Right: NE is selected, and so $f(m') = f(m) + d_{NE}$.

Algorithm 1: Bresenham's line drawing algorithm.

```
1  int d_x ← x_2 − x_1, d_y ← y_2 − y_1
2  int f ← 2d_y − d_x, d_E ← 2d_y, d_NE ← 2(d_y − d_x)
3  int x ← x_1, y ← y_1
4  select(x, y)
5  while x < x_2 do
6      if f ⩽ 0 then
7          x ← x + 1, f ← f + d_E
8      else
9          x ← x + 1, y ← y + 1, f ← f + d_NE
10     select(x, y)
```

the increments of f accordingly, as follows.

$$\begin{aligned} d_E &= 2a = 2(y_2 - y_1), \\ d_{NE} &= 2(a + b) = 2(y_2 - y_1 - x_2 + x_1). \end{aligned} \tag{2.2}$$

The steps are shown in Algorithm 1.

2.1.2 Remarks

Bresenham's algorithm is widely used to quickly draw line segments in a bitmap (e.g. on a computer screen), as it uses only integer addition, subtraction and bit shifting, all of which are very cheap operations. It is one of the earliest algorithms developed in the field of computer graphics.

While algorithms such as Wu's algorithm are also frequently used in modern computer graphics because they support anti-aliasing, the speed and simplicity of Bresenham's line algorithm makes it attractive in hardware such as plotters and in modern graphics cards. It can also be found in many software graphics libraries.

2.2 Circle Drawing

A *digital circle* is obtained by digitization of a real circle with integer center and integer radius (Fig. 2.4). It is a closed 0-connected digital curve such that the smaller between x-distance and y-distance of each pixel from the corresponding real circle is at most $\frac{1}{2}$.

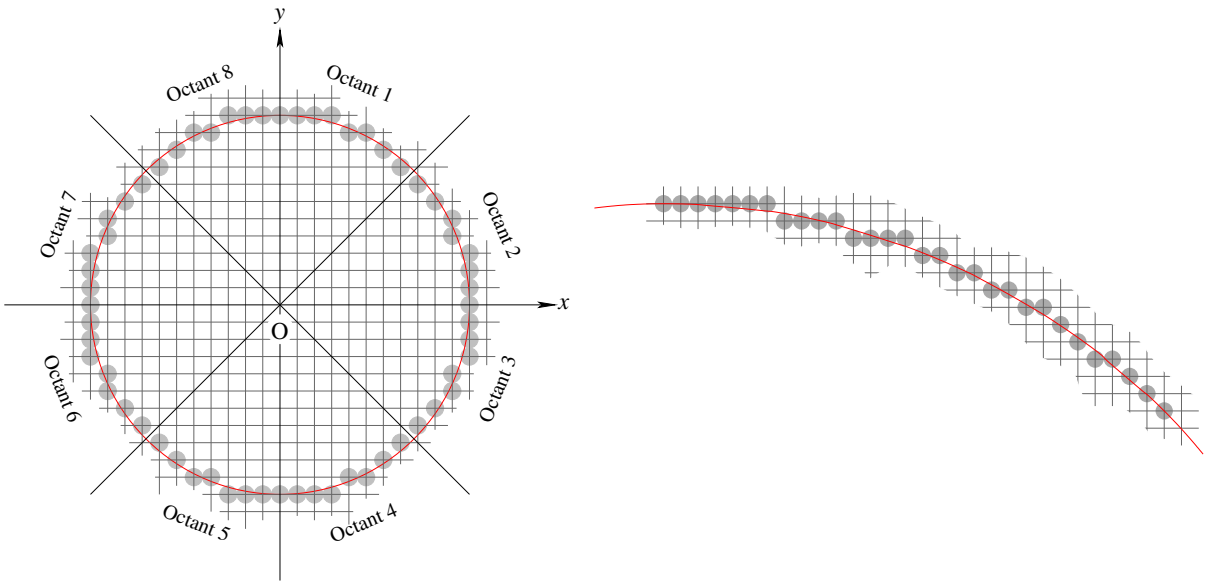


Figure 2.4: **Left:** A real circle (shown in red) of radius 11 and its corresponding digital circle. **Right:** A digital circular arc of radius 41.

2.2.1 Bresenham's Algorithm

Let $C(c, r)$ denote the digital circle having center at $c \in \mathbb{Z}^2$ and integer radius r. When the center is $o := (0, 0)$, we denote it by $C(o, r)$. The circle drawing algorithm is designed based on the following observations.

Observation 1 The digital circle $C(c, r)$ is simply given by all the pixels of $C(o, r)$, translated by the vector $\overrightarrow{oc}$. Hence, an algorithm for construction of $C(o, r)$ serves the purpose.

Observation 2 $C(o, r)$ comprises eight symmetric octants (Fig. 2.4). For $t = 1, 2, \ldots, 8$, let $C_t(o, r)$ denote the t-th octant of $C(o, r)$. In particular, for the 1st octant we have $0 \leqslant x \leqslant y$. If a pixel $p(x, y)$ belongs to $C_1(o, r)$, then the pixels in other seven octants are given by considering all possible signs of (x, y) along with their swapped values. That is, $(x, y) \in C_1(o, r) \implies \big\{(u, v) : \{|u|, |v|\} = \{x, y\}\big\} \subset C(o, r)$. For example, in Fig. 2.4, the pixel $(4, 6)$ belongs to $C_1(o, 7)$, and so the corresponding pixels in other seven octants are $(4, -6), (-4, 6), (-4, -6), (6, 4), (6, -4), (-6, 4), (-6, -4)$.

A digital circle is thus 8-symmetric in nature. The lines of symmetry are the four halves of two principal axes and the four halves of $\pm 45°$ lines, about o. The algorithm is designed, therefore, to generate the pixels of $C_1(o, r)$; then simply reflecting these pixels about the eight lines of symmetry produce the complete circle.

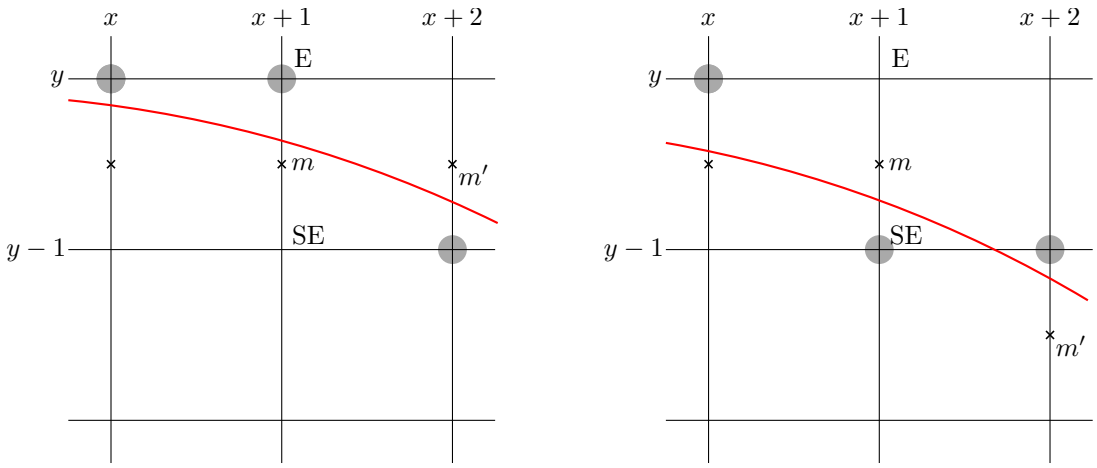


Figure 2.5: Two possible cases in Bresenham's circle drawing algorithm. Left: E is selected, as m lies inside the real circle. Right: SE is selected, as m lies outside.

Observation 3 In 1st octant, y-distance of a pixel in $C(o, r)$ from the real circle is smaller than x-distance. Hence, for a clockwise traversal in $C_1(o, r)$, the pixel next to (x, y) is either east $(x + 1, y)$ or southeast $(x + 1, y - 1)$.

Observation 4 By Observation 3, for each integer $x \in [0, r]$, there is exactly one integer point (x, y) with $x \leqslant y$ that belongs to $C_1(o, r)$. Hence, we can increment x from 0 to r at unit step and compute the corresponding integer value of y until $x > y$.

Decision function The idea of decision function is similar to the one used in Bresenham's line drawing algorithm. Let $f(x, y) = x^2 + y^2 - r^2$ be the decision function for $C(o, r)$. A point lies inside, on, or outside $C(o, r)$ accordingly as f is negative, zero, or positive at that point (Fig. 2.5). To decide the next pixel w.r.t. the current pixel $p(x, y) \in C_1(o, r)$, we check the location of the midpoint $m(x + 1, y - \frac{1}{2})$, using the following functional value.

$$f(m) = (x + 1)^2 + \left(y - \frac{1}{2}\right)^2 - r^2 \tag{2.3}$$

If $f(m)$ is negative, then $\mathrm{E}(x + 1, y)$ is selected, else $\mathrm{SE}(x + 1, y - 1)$.

Difference computation To optimize the computation, we use the value of $f(m)$ to obtain $f(m')$ at the next midpoint m' w.r.t. E or SE, as the case may be. It works as follows.

$$\text{E is selected:} \quad f(m') = (x + 2)^2 + \left(y - \frac{1}{2}\right)^2 - r^2 = f(m) + d_E, \tag{2.4}$$

$$\text{SE is selected:} \quad f(m') = (x + 2)^2 + \left(y - \frac{3}{2}\right)^2 - r^2 = f(m) + d_{SE}, \tag{2.5}$$

where,

$$d_E = 2x + 3 \quad \text{and} \quad d_{SE} = 2x - 2y + 5. \tag{2.6}$$

Algorithm 2: Bresenham's circle drawing algorithm.

```
1  int x ← 0, y ← r, f ← 1 − r, de ← 3, dse ← −2r + 5
2  select(u, v) : {|u|, |v|} = {x, y}
3  while x < y do
4      if f < 0 then //select E
5          f ← f + de
6          de ← de + 2,  dse ← dse + 2
7          x ← x + 1
8      else
9          f ← f + dse
10         de ← de + 2,  dse ← dse + 4
11         x ← x + 1,  y ← y + 1
12     select(u, v) : {|u|, |v|} = {x, y}
```

Double difference Eq. 2.6 provides the values of d_E and d_{SE} using the (x, y) coordinates of p. For the next pixel (E or SE), these values need to be recomputed. For further optimization, we increment their previous values as follows.

$$\text{E is selected} \begin{cases} d^2_{E|E} = & d_{E|E} - d_E = & 2(x+1) + 3 - (2x+3) = & 2. \\ d^2_{SE|E} = & d_{SE|E} - d_{SE} = & 2(x+1) - 2y + 5 - (2x - 2y + 5) = & 2. \end{cases} \tag{2.7}$$

$$\text{SE is selected} \begin{cases} d^2_{E|SE} = & d_{E|SE} - d_E = & 2(x+1) + 3 - (2x+3) = & 2. \\ d^2_{SE|SE} = d_{SE|SE} - d_{SE} = 2(x+1) - 2(y-1) + 5 - (2x - 2y + 5) = 4. \end{cases} \tag{2.8}$$

Initialization The first pixel selected for $C_1(o, r)$ is $(0, r)$. The corresponding midpoint is $m = (1, r - \frac{1}{2})$ at which $f(m) = 1^2 + (r - \frac{1}{2})^2 - r^2 = 1 - r + \frac{1}{4}$, which is not an integer. However, since r is an integer and so also the values of d_E and d_{SE}, the sign of of $f(m)$ does not get affected if we disregard its fractional part, $\frac{1}{4}$. As a result, all computations remain in integer domain, as shown in Algorithm 2. The respective initial values of d_E and d_{SE}, set at $(0, r)$, are $2 \cdot 0 + 3 = 3$ and $2 \cdot 0 - 2r + 5 = -2r + 5$.

2.3 Curve Drawing

In computer graphics, parametric cubic curves are predominantly used for drawing smooth and continuous 2D/3D curves and surfaces. The advantages are as follows.

1. *Explicit form* ($y = f(x), z = g(x)$): Not easy to draw because of the following reasons.
   - It is not possible to get multiple values of y or z for a single value of x, as needed in nonlinear curves like circles, ellipses, spirals, etc.
   - Rotational invariance is not possible.
   - Vertical tangents are difficult to be handled.
2. *Implicit form* ($f(x, y, z) = 0$): Problems are as follows.
   - May have multiple solutions but all may not be required, e.g., when drawing an arbitrarily oriented semicircle.
   - Lack of assurance to tangent continuity at the junction point of two implicit curves.
3. *Parametric form* ($x = x(t), y = y(t), z = z(t)$): Resolves the issues with the implicit and the explicit forms, and offers readiness and flexibility for curve drawing. Some of the important ones are as follows.
   - Parametric curves are designed with *parametric tangent vectors* (always having finite values) instead of geometric slopes (which may be infinite).
   - A long curve is flexibly represented by a sequence of *piecewise polynomial curves*, which can be joined smoothly with each other at the junction points.
4. *Cubic parametric curves* strike a balance between polynomial degree and computational cost. It requires four coefficients to define a cubic curve, which are obtained from four given knowns. These might be two endpoints and their tangents (Bezier curve) or might be four control points (B-spline). The reasons for choosing cubic polynomial are as follows.

   - With four knowns such as four control points in B-spline, we can get a non-planar curve; but with three, we always get it planar!
   - For higher degree, the curve may be wiggly or locally non-smooth. For cubic, the wiggling effect is minimal.

2.3.1 Parametric Equations

A curve segment $Q(t) := [x(t), y(t)]$ in 2D can be represented as follows.

$$x(t) = a_x t^3 + b_x t^2 + c_x t + d_x, \tag{2.9}$$

$$y(t) = a_y t^3 + b_y t^2 + c_y t + d_y, \tag{2.10}$$

$$\text{where } 0 \leqslant t \leqslant 1.$$

We set $0 \leqslant t \leqslant 1$ to get a segment of the cubic curve, as shown in Fig. 2.6. For 3D curves, we just have another cubic function $z(t)$ defined by four other coefficients. For simplicity, in this section we discuss about 2D parametric cubic curves, and all the equations can be extended for 3D parametric cubic curves as well.

Using ***parameter vector*** (T) and ***coefficient matrix*** (C), we express Eqn. 2.9 succinctly as follows.

$$T = [t^3 \quad t^2 \quad t \quad 1], \quad C = \begin{bmatrix} a_x & a_y \\ b_x & b_y \\ c_x & c_y \\ d_x & d_y \end{bmatrix} \implies Q(t) = TC. \tag{2.11}$$

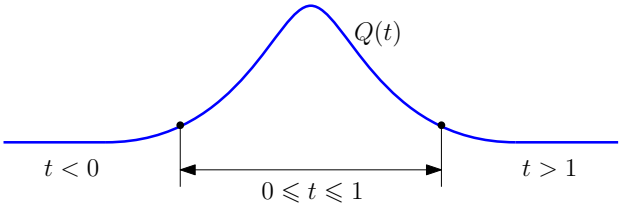


Figure 2.6: A parametric cubic curve and its segment for $0 \leqslant t \leqslant 1$.

Continuity and Smoothness The parametric *tangent vector* of the curve $Q(t)$ is given by

$$\frac{d}{dt}Q(t) = Q'(t) = \begin{bmatrix}3t^2 & 2t & 1 & 0\end{bmatrix}C. \tag{2.12}$$

While joining two curve segments, say $Q_1(t)$ and $Q_2(t)$, for drawing a larger curve segment, the end of the first segment should coincide with the start of the second (Fig. 2.7). That is, at the junction point, we should have $Q_1(t = 1) = Q_2(t = 0)$. This is called G^0 *continuity* (in both parametric and geometric sense). If the tangents drawn to $Q_1(t)$ and $Q_2(t)$ at the junction point (also called 'knot point') have same directions, then the resultant curve is continuous at the junction point and said to satisfy the *geometric continuity* G^1. If the tangent vectors are identical in both magnitude and direction, i.e., $Q_1'(t = 1) = Q_2'(t = 0)$, then the resultant curve satisfies the *parametric continuity* C^1. Clearly, satisfying C^1 means satisfying G^1, but not necessarily the converse.

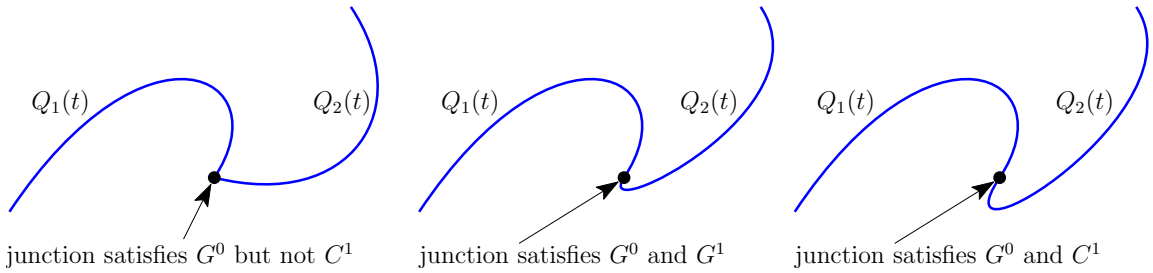


Figure 2.7: Joining of two parametric cubic curves with different levels of continuity. Left: only G^0. Middle: G^0 and G^1. Right: G^0 and C^1 (this is smoothest).

Blending Functions Using a ***basis matrix*** $M = [m_{ij}]_{4\times4}$ and a ***geometry vector*** $G = [G_1\ G_2\ G_3\ G_4]^T = \left[\begin{bmatrix}g_{ix} & g_{iy}\end{bmatrix}_{i=1}^{4}\right]^T$, the coefficient matrix can be rewritten as

$$[C]_{4\times2} = [M]_{4\times4}[G]_{4\times2}. \tag{2.13}$$

Hence, from Eqn. 2.11,

$$Q(t) = TMG, \tag{2.14}$$

from whose expansion we get

$$Q(t) = \begin{bmatrix}x(t) & y(t)\end{bmatrix} = \begin{bmatrix}t^3 & t^2 & t & 1\end{bmatrix}\begin{bmatrix}m_{11} & m_{12} & m_{13} & m_{14}\\ m_{21} & m_{22} & m_{23} & m_{24}\\ m_{31} & m_{32} & m_{33} & m_{34}\\ m_{41} & m_{42} & m_{43} & m_{44}\end{bmatrix}\begin{bmatrix}G_1\\ G_2\\ G_3\\ G_4\end{bmatrix}. \tag{2.15}$$

From Eqn. 2.15, therefore, we get

$$x(t) = \begin{array}{l}(t^3m_{11} + t^2m_{21} + tm_{31} + m_{41})g_{1x} + \\ (t^3m_{12} + t^2m_{22} + tm_{32} + m_{42})g_{2x} + \\ (t^3m_{13} + t^2m_{23} + tm_{33} + m_{43})g_{3x} + \\ (t^3m_{14} + t^2m_{24} + tm_{34} + m_{44})g_{4x}\end{array} = \text{weighted sum of elements } g_{ix} \text{ of } G.$$

$$y(t) = \begin{array}{l}(t^3m_{11} + t^2m_{21} + tm_{31} + m_{41})g_{1y} + \\ (t^3m_{12} + t^2m_{22} + tm_{32} + m_{42})g_{2y} + \\ (t^3m_{13} + t^2m_{23} + tm_{33} + m_{43})g_{3y} + \\ (t^3m_{14} + t^2m_{24} + tm_{34} + m_{44})g_{4y}\end{array} = \text{weighted sum of elements } g_{iy} \text{ of } G.$$

Each weight is a cubic polynomial of t and is called the ***blending function***. Thus, a ***blending matrix*** is defined by $B = TM$, from which we get

$$Q(t) = BG. \tag{2.16}$$

Significance of M: The column vector G is has four elements of geometric constraints. In case of 2D curves, each of these four elements is a 2-tuple (e.g., $G_1 = [g_{1x}\ g_{1y}]$), whereas for 3D curves, it is a 3-tuple (e.g., $G_1 = [g_{1x}\ g_{1y}\ g_{1z}]$). The geometric constraints are just the input, such as four control points (for B-splines) or two control points and their tangent vectors (for Hermite and Bezier curves). M is the basis matrix that decides the proportion in which the elements of G are mixed—by multiplying with the elements of M serving as 'weights'—as per the required nature of the cubic curve.

2.3.2 Hermite Curves

A cubic Hermite curve segment is defined by four constraints—two endpoints and their tangent vectors (Fig. 2.8). To set up Eq. 2.14 ($Q(t) = TMG$), we derive M using four equations from these four constraints. The geometry vector is taken as $G = [P_1\ P_4\ R_1\ R_4]^T$, where P_1 and P_4 are the endpoints, and R_1 and R_4 are respective tangent vectors. Using Eqns. 2.9, 2.10, and 2.11, the x-component is written as

$$x(t) = a_xt^3 + b_xt^2 + c_xt + d_x = TC_x = [t^3\ \ t^2\ \ t\ \ 1]MG_x \implies x'(t) = [3t^2\ \ 2t\ \ 1\ \ 0]MG_x,$$

where, $G_x = [g_{1x}\ \ g_{2x}\ \ g_{3x}\ \ g_{4x}]^T$. Hence,

$$\begin{aligned} P_1 = x(0) &= [0\ \ 0\ \ 0\ \ 1]MG_x, \\ P_4 = x(1) &= [1\ \ 1\ \ 1\ \ 1]MG_x, \\ R_1 = x'(0) &= [0\ \ 0\ \ 1\ \ 0]MG_x, \\ R_4 = x'(1) &= [3\ \ 2\ \ 1\ \ 0]MG_x. \end{aligned}$$

So,

$$G_x = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}_x = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} MG_x \implies I = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} M,$$

$$\text{or, } M = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.$$

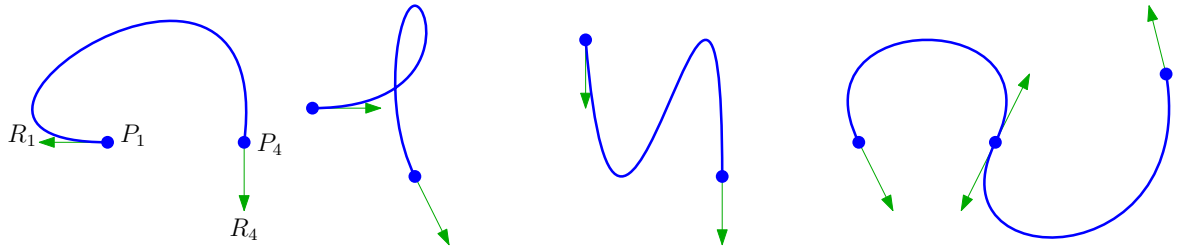


Figure 2.8: Examples of Hermite curves. First three are single curve segments each, the last one is where two curve segments share an endpoint and have two opposite tangents at that endpoint.

Recall that the *inverse* of a square matrix A (with determinant $|A| \neq 0$) is given by $A^{-1} = \frac{1}{|A|}C^T$, where C is the *cofactor matrix* of A given from its *minors* M as follows.

$$C_{ij} = (-1)^{i+j} M_{ij} = \begin{cases} M_{ij} & \text{if } i+j \text{ is even} \\ -M_{ij} & \text{if } i+j \text{ is odd} \end{cases}$$

2.3.3 Bézier Curves

Bézier curve replaces the two tangent vectors of Hermite curve by two control points that do basically fix the tangents when taken with the other two control points. By this, it interpolates the two end control points and approximates the other two. For four control points P_1, P_2, P_3, P_4 in sequence, the respective tangent vectors R_1 and R_4 at P_1 and P_4 are determined by the vectors $\overrightarrow{P_1P_2}$ and $\overrightarrow{P_3P_4}$. In particular, the respective tangents P_1 and P_4 are defined as

$$R_1 = Q'(0) = 3(P_2 - P_1), \quad R_4 = Q'(1) = 3(P_4 - P_3). \tag{2.17}$$

Accordingly, the ***Bézier geometry vector*** becomes

$$G_B = \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix}. \tag{2.18}$$

Then the matrix M_{HB} defines the relation $G_H = M_{HB} \cdot G_B$ between the Hermite geometry vector G_H and the Bézier geometry vector G_B. It just uses Eq. 2.17 and gives the following equation.

$$G_H = \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ -3 & 3 & 0 & 0 \\ 0 & 0 & -3 & 3 \end{bmatrix} \begin{bmatrix} P_1 \\ P_2 \\ P_3 \\ P_4 \end{bmatrix} = M_{HB} \cdot G_B \tag{2.19}$$

To find the ***Bézier basis matrix*** M_B, we start with the Hermite form, use Eq. 2.19, and proceed as follows.

$$Q(t) = T \cdot M_H \cdot G_H = T \cdot M_H \cdot \big(M_{HB} \cdot G_B\big) = T \cdot \big(M_H \cdot M_{HB}\big) \cdot G_B = T \cdot M_B \cdot G_B, \tag{2.20}$$

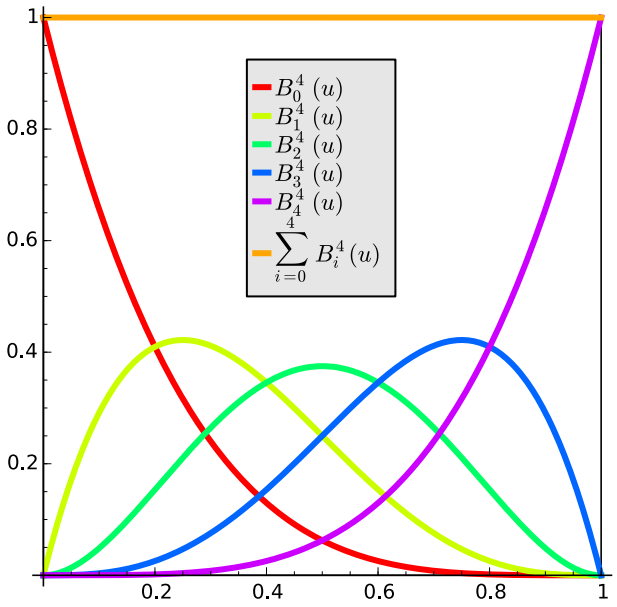


Figure 2.9: Plot of Bernstein polynomial functions up to degree 4 with summation of all four functions to show characteristic of partition of one. (source: https://en.wikipedia.org/wiki/Bernstein_polynomial)

where,

$$M_B = M_H \cdot M_{HB} = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}. \tag{2.21}$$

Thus, from Eq. 2.20, we get

$$Q(t) = T \cdot M_B \cdot G_B = (1-t)^3 P_1 + 3t(1-t)^2 P_2 + 3t^2(1-t)P_3 + t^3 P_4. \tag{2.22}$$

The four polynomials $B_B = T \cdot M_B$, which serve as the weights in Eq. 2.22, are called ***Bernstein polynomials*** (Fig. 2.9).[1]

Figure 2.10 shows two Bézier curve segments with a common endpoint P_4, serving as the knot point. If $P_4 - P_3 = k(P_5 - P_4)$ with $k > 0$ (i.e., the three control points are distinct and collinear), then G^1 continuity is ensured. For $k = 1$, we get C^1 continuity that makes the curve portion even smoother at the knot point.

At the knot point, we consider the 0-th order and 1st order continuities of x-values of left segment and right segment. We get

$$x_l(t=1) = x_r(t=0), \qquad \frac{d}{dt}x_l(t=1) = \frac{d}{dt}x_r(t=0),$$
$$\text{or,} \quad x_l(t=1) = x_r(t=0) = x_4, \quad 3(x_4 - x_3) = 3(x_5 - x_4) \implies \boxed{C_1 \text{ is ensured}}.$$

[1] See https://en.wikipedia.org/wiki/Bernstein_polynomial for further details.

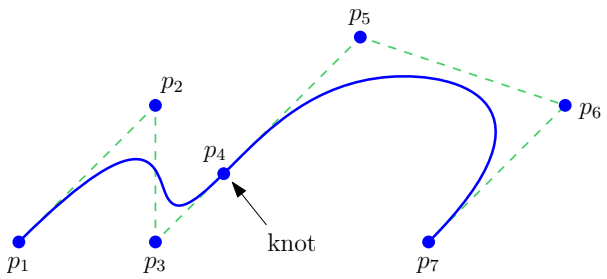


Figure 2.10: Two Bézier curves joined at P_4. Point P_3, P_4, and P_5 are collinear here, and hence G^1 is maintained. However, C^1 is not present here, as $P_4 - P_3 \neq P_5 - P_4$.

2.3.4 B-spline (Uniform Nonrational)

The basis matrix is given by

$$M = \frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}, \tag{2.23}$$

which is obtained from the parametric continuity and the geometric continuity of B-splines.

The geometry vector is constructed from the 4 control points, namely (x_0, y_0), (x_1, y_1), (x_2, y_2), and (x_3, y_3), defining a B-spline segment as follows:

$$G = \begin{bmatrix} x_0 & y_0 \\ x_1 & y_1 \\ x_2 & y_2 \\ x_3 & y_3 \end{bmatrix}. \tag{2.24}$$

Hence, from Eqn. 2.14, we get

$$a_x = \frac{1}{6}(-x_0 + 3x_1 - 3x_2 + x_3) \tag{2.25}$$

$$b_x = \frac{1}{2}(x_0 - 2x_1 + x_2) \tag{2.26}$$

$$c_x = \frac{1}{2}(-x_0 + x_2) \tag{2.27}$$

$$d_x = \frac{1}{6}(x_0 + 4x_1 + x_2) \tag{2.28}$$

Chapter 11

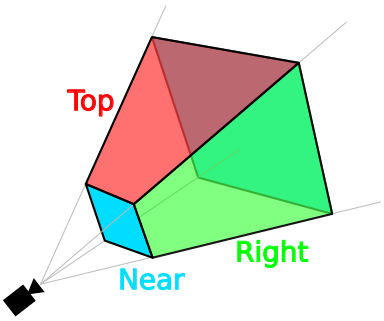
Clipping

You can't depend on your eyes when your imagination is out of focus.

—*Mark Twain*

Clipping is somewhat like focusing—it is a method to selectively enable or disable rendering operations within a defined region of interest (called *clip window* in 2D). A rendering algorithm only draws pixels in the intersection between the clip region and the scene model.

By clipping, lines and surfaces outside the view volume (aka. *frustum*) are removed[1]. The inset figure shows a view frustum, with near- and far-clip planes; only the shaded volume is rendered. A clip window or clip regions is commonly specified to improve render performance. It allows the renderer to save time and energy by skipping calculations related to pixels that the user cannot see. Pixels that will be drawn are said to be within the clip window. Pixels that will not be drawn are outside the clip window. More informally, pixels that will not be drawn are said to be "clipped".



11.1 Line Clipping using Region Codes

Cohen-Sutherland algorithm is used for line clipping in 2D when the clip window W is rectilinear, i.e., an axis-parallel rectangle. If it is a rectangle but not axis-parallel, then the coordinate system can be rotated accordingly so that W becomes axis-parallel. The algorithm divides the two-dimensional space into 9 regions, encodes them uniquely into 4-bit *region codes*, and then performs clipping on each input line. The algorithm was developed in 1967 during flight-simulator work by Danny Cohen and Ivan Sutherland.[2]

[1] Gary Bertoline and Wiebe, Eric (2002): Fundamentals of Graphics Communication (3rd ed.), McGraw-Hill, p. G-3.

[2] Principles of Interactive Computer Graphics, p. 124, 252, by Bob Sproull and William M. Newman, 1973, McGraw-Hill Education, International edition.

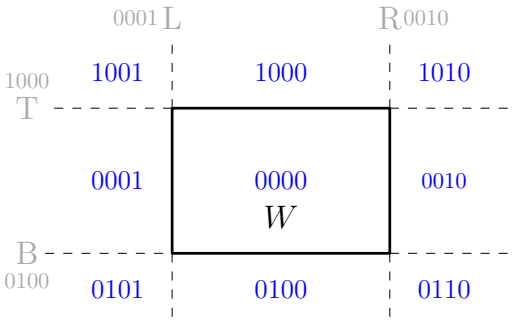


Figure 11.1: 4-bit region codes (in TBRL format).

Region codes The clip window W is specified by four coordinates: $x_{\min}, x_{\max}, y_{\min}, y_{\max}$. These coordinates provide the four boundary lines of W, namely L (left), R (right), B (bottom), and T (top)—in this order by convention, although any other convention would also do. Accordingly, for each of L, R, B, T, we get a pair of half-planes—one containing W and the other without. The bit of the half-plane containing W is set to `0`, and the other to `1`. For example, its rightmost bit is set to `0` if it lies in the left half-plane of L, else it is set to `1`. Thus, for encoding the location of a point, 4 bits are enough (6 bits in the three-dimensional case). Other three bits are set in a similar manner depending on its location w.r.t. R, B, T. The resultant codes of nine regions are referred to as rCodes and shown in Fig. 11.1.

An rCode is computed for each of the two endpoints of the line. It must be recomputed in each iteration after the clipping occurs. The algorithm includes, excludes, or partially includes the input line segment based on analysis of three cases (Fig. 11.2) as follows.

1. Both endpoints are in W (bitwise OR $= 0000$): trivial accept.
2. Both endpoints lie in the same half-plane that does not contain W (bitwise AND $\neq 0000$): trivial reject.
3. Neither of the above: nontrivial case (leads to partial or complete rejection at the end).
   3.1. Find the outpoint (one of the two endpoints that lies outside W).
   3.2. Compute the intersection of the line with L/R/B/T.
   3.3. Replace the outpoint by the intersection point.
   3.4. Repeat until a trivial accept or trivial reject occurs.

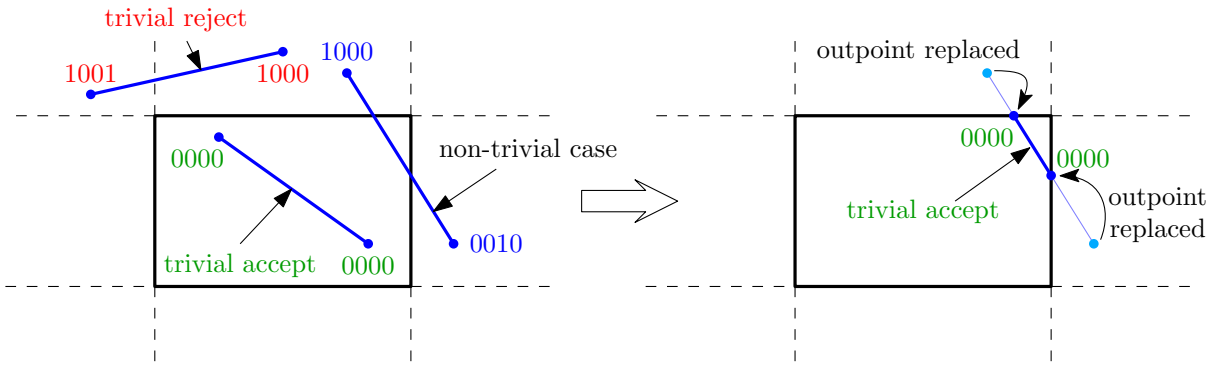


Figure 11.2: 3 cases in Cohen-Sutherland algorithm.

```
typedef int rCode;
const int INSIDE = 0; // 0000
const int LEFT = 1;   // 0001
const int RIGHT = 2;  // 0010
const int BOTTOM = 4; // 0100
const int TOP = 8;    // 1000

rCode Compute_rCode(double x, double y){
    rCode code = INSIDE; // initialized as being inside W
    if (x < xmin) code |= LEFT;
      else code |= RIGHT;
    if (y < ymin) code |= BOTTOM;
      else code |= TOP;
    return code;
}

void CohenSutherlandLineClipAndDraw(double x0, double y0, double x1, double y1){
    rCode rCode0 = Compute_rCode(x0, y0);
    rCode rCode1 = Compute_rCode(x1, y1);
    bool accept;
    while (true) {
        if (!(rCode0 | rCode1)) {
            accept = true; break; }
        else if (rCode0 & rCode1) {
            accept = false; break;}
        else {
            double x, y;

            // Find the outpoint -- it has non-zero rCode.
            rCode rCodeOut = rCode0 ? rCode0 : rCode1;

            // Find the intersection point.
            if (rCodeOut & TOP) {         // point is above W
                x = x0 + (x1 - x0) * (ymax - y0) / (y1 - y0);
                y = ymax; }
            else if (rCodeOut & BOTTOM) { // point is below W
                x = x0 + (x1 - x0) * (ymin - y0) / (y1 - y0);
                y = ymin; }
            else if (rCodeOut & RIGHT) {  // point is to the right of W
                y = y0 + (y1 - y0) * (xmax - x0) / (x1 - x0);
                x = xmax; }
            else if (rCodeOut & LEFT) {   // point is to the left of W
                y = y0 + (y1 - y0) * (xmin - x0) / (x1 - x0);
                x = xmin; }

            // Move outpoint to intersection point for next pass.
            if (rCodeOut == rCode0) {
                x0 = x, y0 = y;
                rCode0 = Compute_rCode(x0, y0); }
            else {
                x1 = x, y1 = y;
                rCode1 = Compute_rCode(x1, y1); }
        }
    }
    if (accept) {
        DrawLineSegment(x0, y0, x1, y1); }
}
```

Figure 11.3: Pseudo-code of Cohen-Sutherland line clipping algorithm.

The pseudo-code in C/C++ is given in Fig. 11.3. The coordinates of the two endpoints are denoted by (x_0, y_0) and (x_1, y_1). For finding the intersection point, we use the following:

$$\begin{aligned} \text{slope} = &\quad \frac{y_1 - y_0}{x_1 - x_0}, \\ x = &\quad x_0 + \frac{1}{\text{slope}}(y_m - y_0), \text{ where } y_m \text{ is } y_{\min} \text{ or } y_{\max}, \\ y = &\quad y_0 + \text{slope}(x_m - x_0), \text{ where } x_m \text{ is } x_{\min} \text{ or } x_{\max}. \end{aligned}$$

No need to worry about divide-by-zero because, in each case, the rCode-bit being tested guarantees the denominator is non-zero.

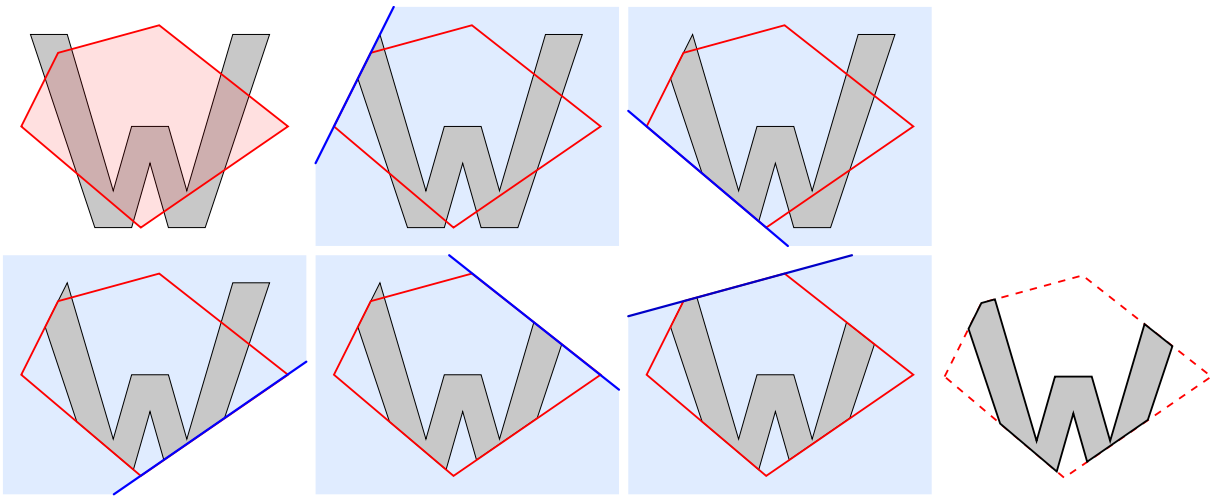


Figure 11.4: Steps for clipping a concave polygon 'W' with a 5-sided convex polygon using Sutherland-Hodgman algorithm (source: https://en.wikipedia.org/wiki/Sutherland%E2%80%93Hodgman_algorithm).

11.2 Polygon Clipping

In computer graphics and games development, polygons are clipped based on a window, which may be rectilinear or convex or, in general, of any arbitrary shape. As a polygon is defined by a sequence of vertices or edges, the subject or candidate polygon is usually supplied in a list of vertices in a specific order. The portion(s) of the subject polygon that lies inside the window is kept, and the rest is clipped. A polygon clipping algorithm is usually designed to handle different cases such as convex polygon (simplest case), polygon with holes, non-convex polygon, and self-intersecting polygon.

11.2.1 Sutherland-Hodgman Algorithm

This algorithm is widely used for clipping polygons[3] when the clip polygon (i.e., window) is convex. It works by extending each line of the convex clip polygon in turn and selecting only vertices from the subject polygon that are on the visible side (Fig. 11.4).

The algorithm begins with an input list of all vertices in the subject polygon. Next, one side of the clip polygon is extended infinitely in both directions, and the path of the subject polygon is traversed. Vertices from the input list are inserted into an output list if they lie on the visible side of the extended clip polygon line, and new vertices are added to the output list where the subject polygon path crosses the extended clip polygon line.

This process is repeated iteratively for each side of the clip polygon side, using the output list from one stage as the input list for the next. Once all sides of the clip polygon have been processed, the final generated list of vertices defines a new single polygon that is entirely visible. Note that if the subject polygon was concave at vertices outside the clipping polygon, then the

[3]Ivan Sutherland and Gary W. Hodgman: Reentrant Polygon Clipping. *Communications of the ACM*, **17**:32–42, 1974.

```
List outputList = subjectPolygon;
for (Edge clipEdge in clipPolygon) do
   List inputList = outputList;
   outputList.clear();
   Point S = inputList.last;
   for (Point E in inputList) do
      if (E inside clipEdge) then
         if (S not inside clipEdge) then
            outputList.add(ComputeIntersection(S,E,clipEdge));
         end if
         outputList.add(E);
      else if (S inside clipEdge) then
         outputList.add(ComputeIntersection(S,E,clipEdge));
      end if
      S = E;
   done
done
```

Figure 11.5: Pseudo-code of Sutherland-Hodgman polygon clipping algorithm.

new polygon may have coincident (i.e., overlapping) edges—this is acceptable for rendering, but not for other applications such as computing shadows.

Given a list of edges in a clip polygon, and a list of vertices in a subject polygon, the procedure for clipping the subject polygon against the clip polygon is shown in Fig. 11.5.

The vertices of the clipped polygon are to be found in `outputList` when the algorithm terminates. Note that a point is defined as being inside an edge if it lies on the same side of the edge as the remainder of the polygon. If the vertices of the clip polygon are consistently listed in a counter-clockwise direction, then this is equivalent to testing whether the point lies to the left of the line (left means inside, while right means outside), and can be implemented simply by using a cross product.

`ComputeIntersection` is a function, omitted here for clarity, which returns the intersection of a line segment and an infinite edge. Note that it is only called if such an intersection is known to exist, and hence can simply treat both lines as being infinitely long.

The Weiler-Atherton algorithm overcomes the limitations of Sutherland-Hodgman algorithm by returning a set of divided polygons, but is more complex and computationally more expensive; so, Sutherland-Hodgman is used for many rendering applications. Sutherland-Hodgman can also be extended into 3D space by clipping the polygon paths based on the boundaries of planes defined by the viewing space.

11.2.2 Other Algorithms

Weiler-Atherton Algorithm[4] It allows clipping of a subject polygon by an arbitrarily shaped clipping polygon/area/region. Although generally applied in 2D, it can be used in 3D through visible surface determination and with improved efficiency through Z-ordering.[5] The algorithm

[4]Weiler, Kevin and Atherton, Peter: Hidden Surface Removal using Polygon Area Sorting, *Computer Graphics*, **11**(2):214–222, 1977. URL: https://www.cs.drexel.edu/~david/Classes/CS430/HWs/p214-weiler.pdf.

[5]Foley, James, Andries van Dam, Steven Feiner, and John Hughes: *Computer Graphics: Principle and Practice.* Addison-Wesley Publishing Company. Reading, Massachusetts: 1987. pages 689–693.

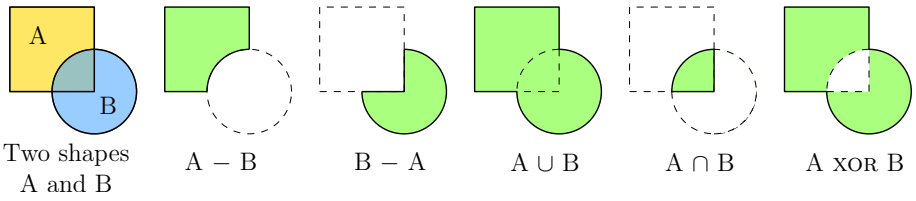


Figure 11.6: Object clipping as realized as boolean set-theoretic operations.

can support holes, but requires additional algorithms to decide which polygons are holes, after which merging of the polygons can be performed using a variant of the algorithm.

Vatti's clipping algorithm[6] This algorithm allows clipping of any number of arbitrarily shaped subject polygons by any number of arbitrarily shaped clip polygons (Fig. 11.7). Unlike the Sutherland-Hodgman and Weiler-Atherton polygon clipping algorithms, the Vatti algorithm does not restrict the types of polygons that can be used as subjects or clips. Even complex (self-intersecting) polygons, and polygons with holes can be processed. The algorithm is generally applicable only in 2D space.

While clipping usually involves finding the ***intersections*** (regions of overlap) of subject and clip polygons, clipping is conceptualized in Vatti's algorithm as the interaction of subject and clip polygons, and realized as boolean set-theoretic operations (Fig. 11.6):

- ***difference***—clipping polygons remove overlapping regions from the subject.
- ***union***—clipping returns the regions covered by either subject or clip polygons.
- ***xor***—clipping returns the regions covered by either subject or clip polygons except where they are covered by both subject and clip polygons.

Vatti's algorithm involves processing both subject and clipping polygon edges in an orderly fashion, starting with the lowermost edges and working towards the top; this is conceptually similar to the Bentley-Ottmann algorithm. This sweep line approach divides the problem space by scanlines, imaginary horizontal lines that pass through every vertex of the participating polygons. These scanlines outline scanbeams the spaces between adjacent scanlines. These scanbeams are processed in turn, starting with the lowest scanbeam, with the algorithm adding points of intersection within these scanbeams into the solution polygons.

11.3 3D Clipping

In 3D graphics, clipping is required in several tasks such as *frustum culling*, which is a process of discarding objects that are not visible on the screen (Fig. 11.8).

[6]Bala R. Vatti: A generic solution to polygon clipping, *Communications of the ACM*, **35**(7):56–63, 1992.

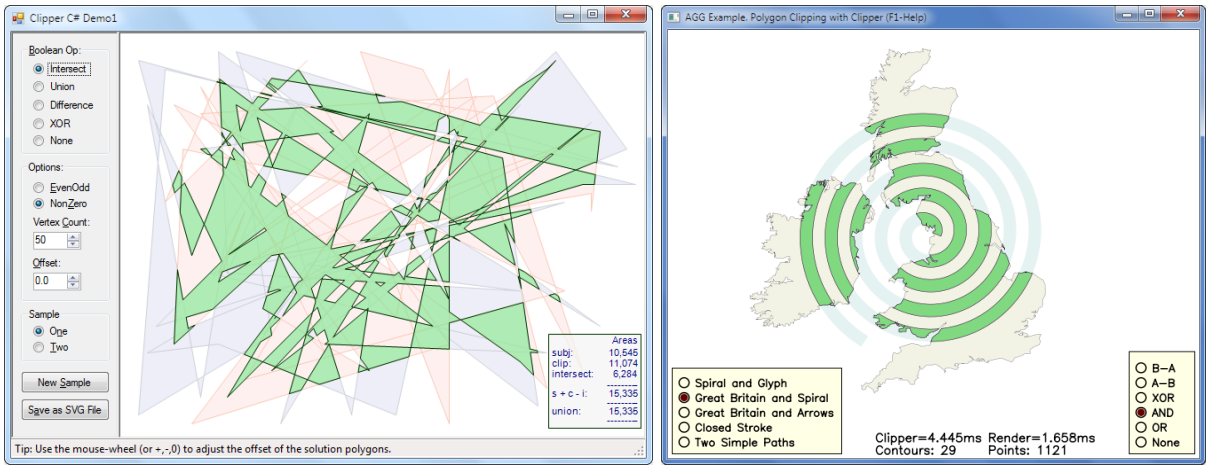


Figure 11.7: Polygon clipping (intersection, union, difference, and xor, based on Vatti's clipping algorithm. (source: Clipper—an open source freeware library for clipping and offsetting lines and polygons. URL: http://www.angusj.com/delphi/clipper.php).

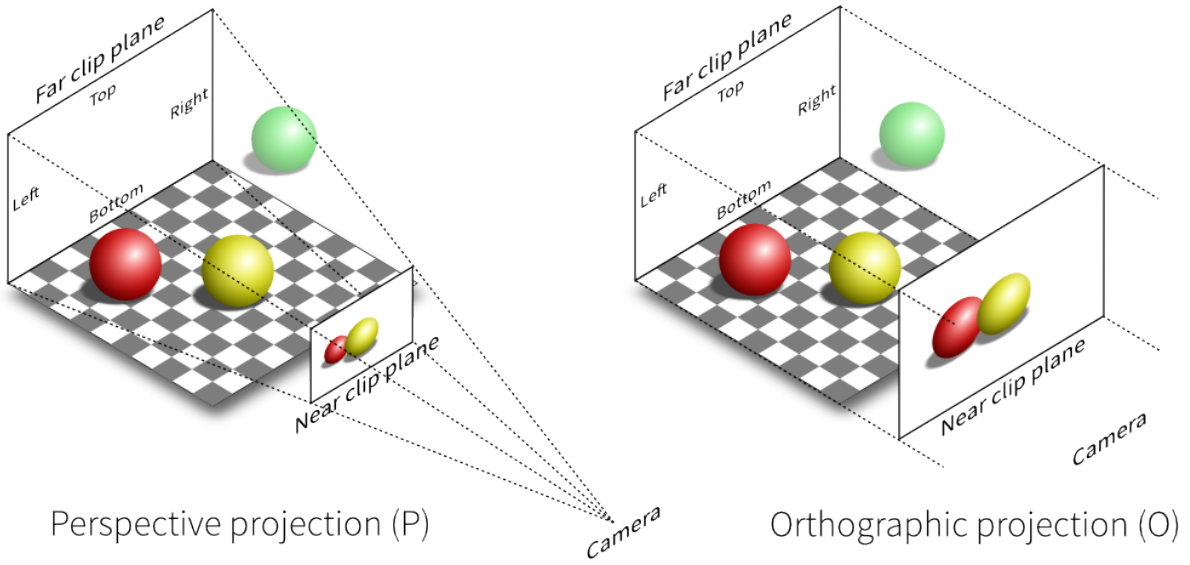


Figure 11.8: Frustum culling as 3D clipping.

Figure 11.9: 3D clipping on a triangulated object named `mother&child` containing 10 thousand triangles with the rectilinear clip window translating along a principal axis in discrete steps.