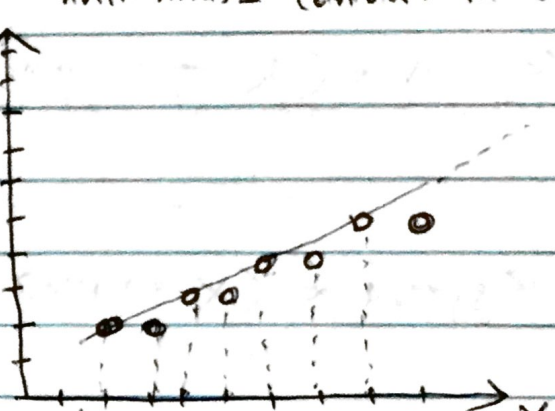
Chapter 4 : Graphics Algorithms for Drawing 2D Primitives

Topics :
① Line Drawing
② Clipping (lines + Polygons)
③ Anti-Aliasing

Alias → Staircase like pattern
Anti Alias – contour that

1. Scan-converting lines.

A scan-conversion algorithm for lines computes the co-ordinates of the pixels that lie on or near an ideal, infinitely thin straight line.

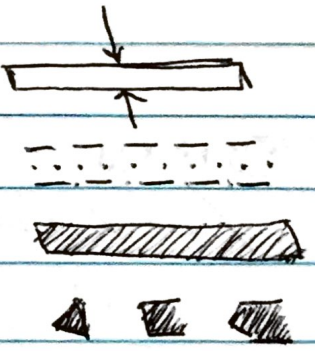


For lines $-1 \leq \text{Slope} \leq 1$, exactly 1 pixel should be drawn in each column.

For lines with other slopes, exactly 1 pixel should be drawn in each row.

Factors to consider:

① thickness
② line-style
③ pen style
④ Shape of end points



To draw a pixel in Java, we define a method :

```
void putpixel(Graphics g, int x, int y)
{
  g.drawline(x, y, x, y);
}
```

1.1. Basic Incremental Algorithm:

Simplest approach:

Slope $m = \Delta y / \Delta x$.

increment x by 1 from leftmost point $(-1 \leq m \leq 1)$

Use line equation:

$y_i = m x_i + c$ and then round off y_i to get y co-ordinate.

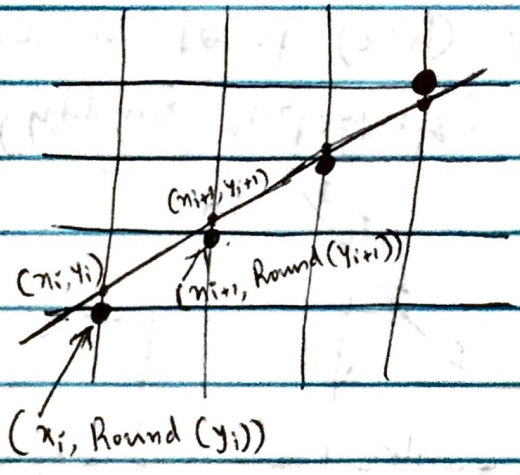
But inefficient due to Floating Point multiply, addition & rounding.

Optimizing it:

$$y_{i+1} = m x_{i+1} + B = m(x_i + 1) + B \quad [\because \text{every } x_i \text{ only increments by 1}]$$
$$= (m x_i) + m + (B)$$
$$= y_i + m$$

So, its called incremental algorithm:

At each step, we increment based on previous step.



Psendocode (for $|m| \leq 1$):

```
float y, m;
int x, dy, dx;
dy = yQ - yP;
dx = xQ - xP;
m = (float) dy/dx;
for(x = xP; x <= xQ; x++) {
    putpixel(g, x, Math-round(y));
    y = y + m;
}
```

Because of rounding, error of innacuracy

$$-0.5 \leq y_{exact} - y < 0.5$$

If $|m| > 1$, reverse the roles of x & y, i.e

$y_{i+1} = y_i + 1$, $x_{i+1} = x_i + 1/m$

Also, need to consider special cases of horizontal, vertical and diagonal lines.

Major drawback: One of x & y has to be a float, so is m, Rounding operation takes time.

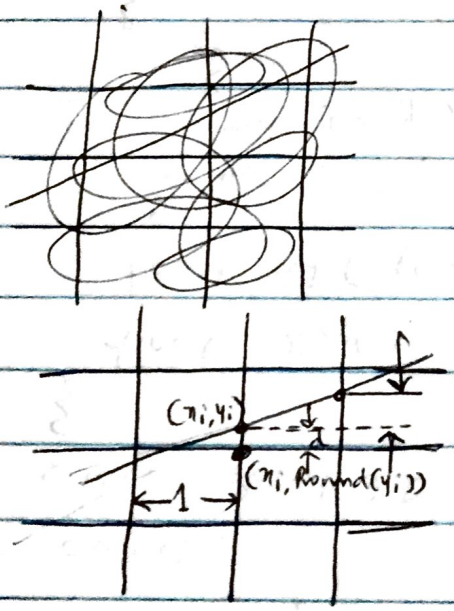
1.2 Bresenham Line Algorithm

Improve above incremental algorithm.

First, get rid of rounding operation, make y an integer, we separate its integer portion from fraction portion (i.e m + rounding part)

$y =$ [int] . [Frac]

↑



d = y − Round(y),

so $-0.5 \leq d < 0.5$.

Equivalent code:

```
float m, d = 0;
int x, y, dx, dy;
dy = yQ - yP; dx = xQ - xP;
m = (float) dy / dx;
for(x = xP; x ≤ xQ; x++){
    putpixel(G, x, y);
    d = d + m;
    if (d ≥ 0.5)
    {y++;
     d--;
    }
}
```

Second, get rid of float types m & d. To get rid of 0.5 for d, we double d to make it an integer. To get rid of division in m, we multiply m by $x_Q - x_P$. So, we introduce a scaling factor:

$C = 2 * (x_Q - x_P)$

$M = (m = 2(y_Q - y_P)$

$D = Cd.$

$H = C * 0.5 = x_Q - x_P$

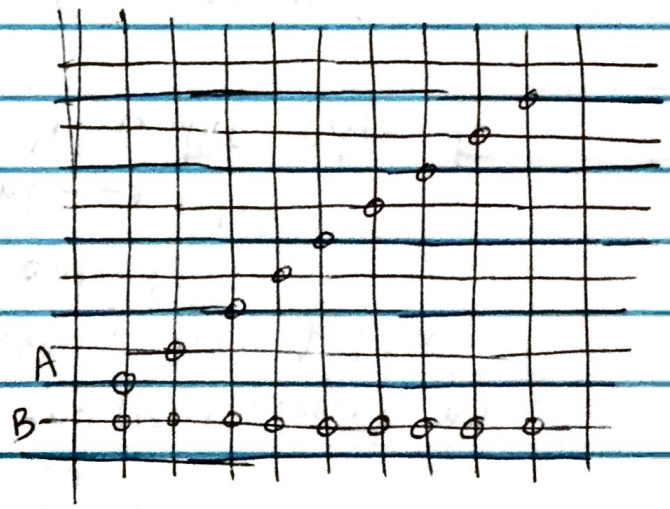
Then we get an integer version of the algorithm

```
int x, y, D=0, H= xQ-xP, C= 2*H;
  M= 2*(yQ-yP);
for (x=xP; x<=xQ; x++) {
        putPixel (g, x, y);
        D = D+M;
        if(D≥M) { y++; D=D-C; }.
}.
```

Now, we generalize the algorithm to handle all angles of slope & different orders of endpoints.

Line Intensity Problem:-

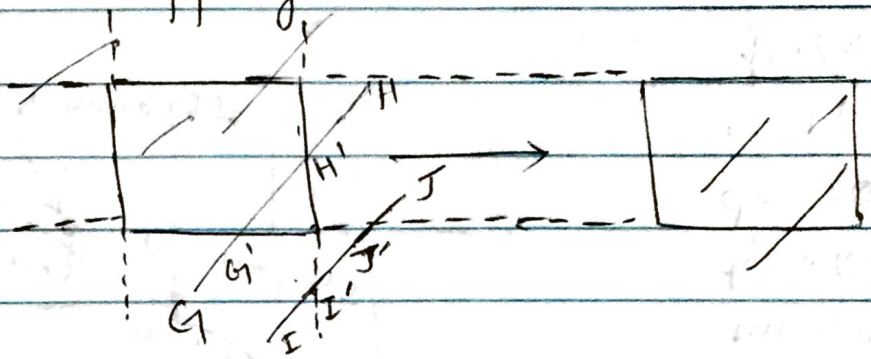
Since # of Pixels on both lines are the same, but line A is $\sqrt{2}$ as long as line B. So, line A looks thinner



Solutions :-

① Set intensity as function of slope
② Treating line as a rectangle, using anti aliasing approach.

2. Line Clipping :-



2.1. Clipping endpoints

For a point (x, y) to be inside the clip rectangle of boundaries

$x_{min} \longrightarrow x_{max}$, $y_{min} \longrightarrow y_{max}$.

$x_{min} \leq x \leq x_{max}$ AND $y_{min} \leq y \leq y_{max}$

2.2 Brute Force Approach.

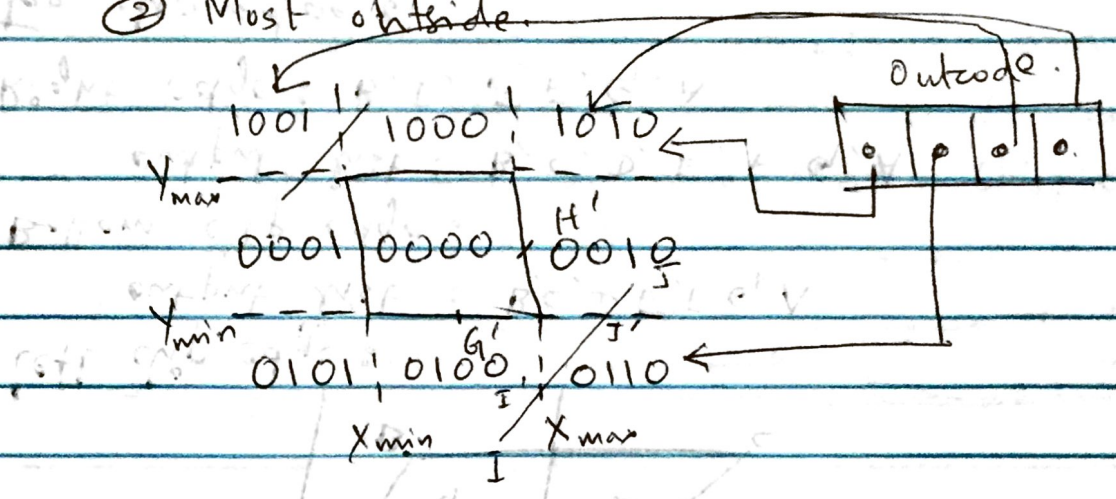
① If both endpoints inside clip rectangle, trivially accept.
② If one inside, one outside, compute intersection point (by solving 2 simultaneous equations)
③ If both outside, compute intersection points & check whether they are interior.

Very inefficient (since multiplication & division needed in solving equations).

Father of CG
Turing Award winner.

2.3 Cohen Sutherland Algorithm

* Based on "regions", most line segments could be trivially rejected.
* Very efficient for cases:
  ① Most primitives are inside chip rectang
  ② Most outside.



Outcode bits can be considered sign bits of $y_{max} - y$, $y - y_{min}$, $x_{max} - x$, $x - x_{min}$ respectively.
(1 – negative, 0 – positive)

Checking for a line AB.

① If outcodes of both A & B are zero, trivially accept.
② If $outcode_A$ AND $outcode_B \neq 0$ trivially reject.
③ Otherwise, start from outside end point & find the intersection point, chip away the outside segment and replace outside endpoint with the intersection point; go to ①

Given the chosen outside endpoint, work on boundary edges by checking the 4-bits from left to right on its outcode.

top → bottom → right → left

The first encountered '1' is the edge to work on.

Consider line AD, $outcode_A = 0000$

$outcode_D = 1001$, Neither ① nor ②

Choose D, use top edge to clip to obtain AB, find $outcode_B = 0000$

So, according to ①, accept AB.

Consider EI, $outcode_E = 0100$,

$outcode_I = 1010$

Start from E,

Clip against bottom edge to obtain FI, which still meets neither ① nor ②

Since $outcode_F = 0000$, choose I.

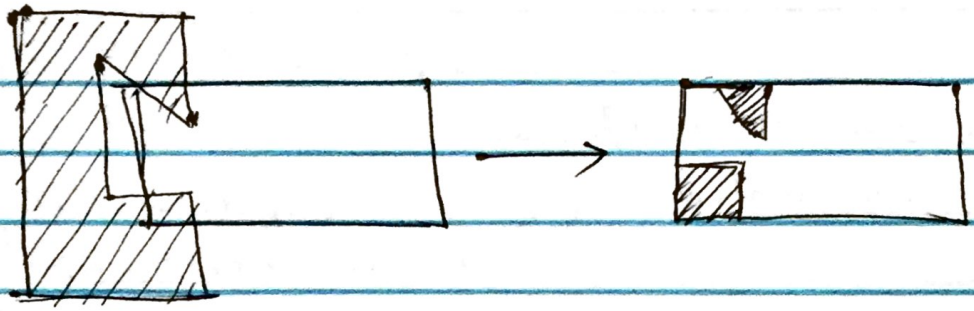
Use top edge to clip to get FH.

$Outcode_H = 0010$, use right edge to clip to get FG, which is then trivially accepted.

Starting from I, will obtain the same result.

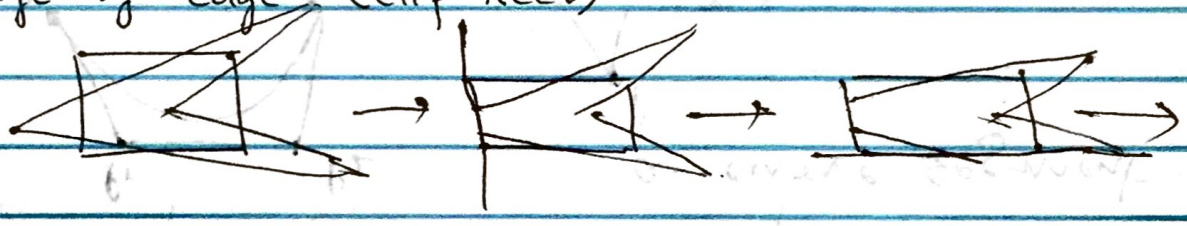
ClipLine.java.

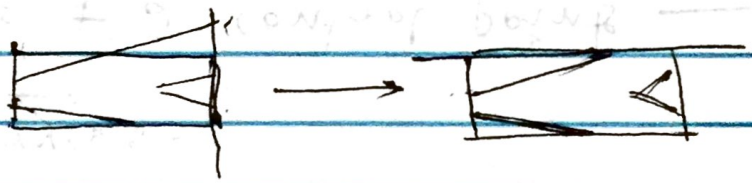
Polygon Clipping Algorithm :-



Sutherland- Hodgman Algorithm (Divide-and-Conquer)

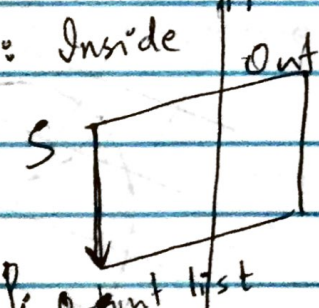
- General : a polygon (Concave or Convex) can be clipped against any convex clipping polygon Edge by Edge (Clip Rect)





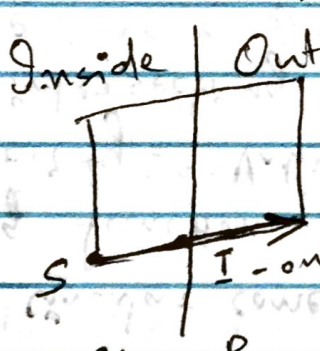
Foreach Polygon edge :

The algorithm moves around the polygon, with ends. polygon edge, 1 or 2 vertices are added to the output list of vertices that defines the clipped polygon. There are 4 possible cases : Inside | Outside.



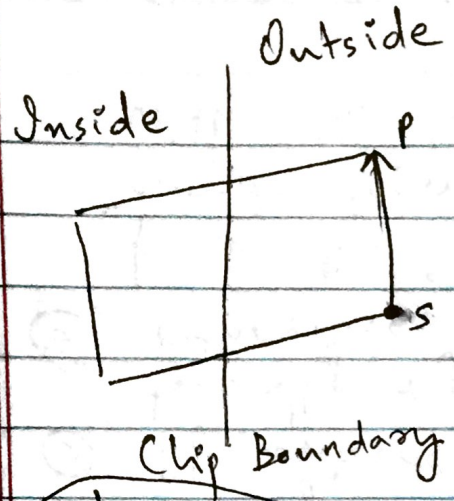
Clipping boundary

1 Output

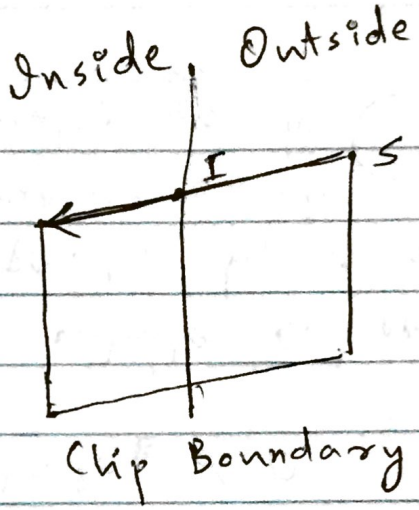


Clip Boundary

1 Output

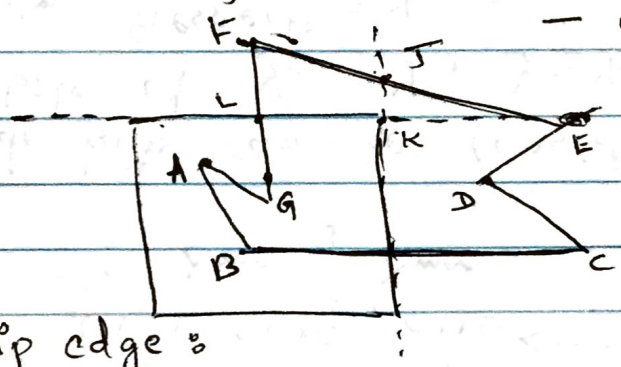


No Output



When travel from inside to outside
— output 1 point.

When travel from outside to inside,
— output 2 points.



Left clip edge :
output list : B, C, D, E, F, G, A

Bottom clip edge :
output list : B, C, D, E, F, G, A

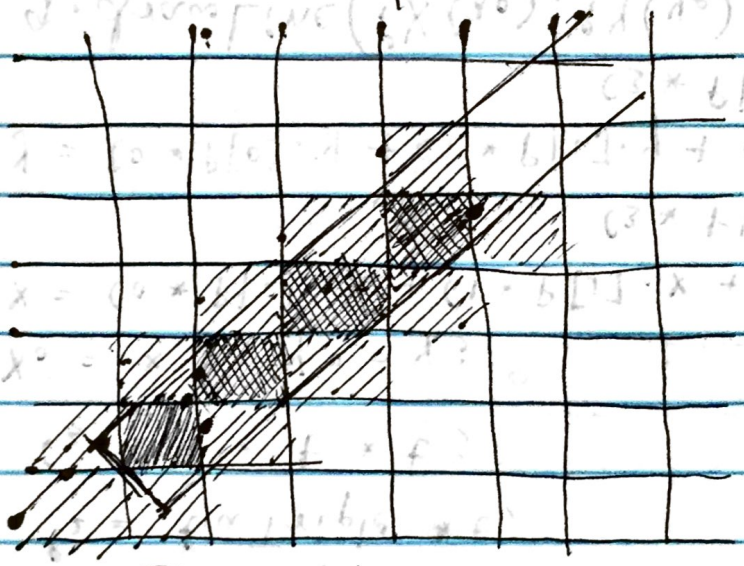
Right edge : B, I, J, F, G, A

Top clip edge :
output list : B, I, K, L, G, A

4. Antialiasing:

An effective way to overcome the staircasing (aliasing) problem on high-resolution displays.

① Unweighted area sampling -
setting intensity proportional to the amount of area covered by the line.



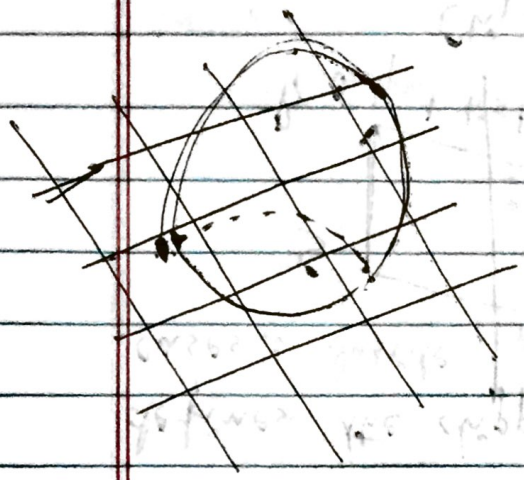
Features: (A) The farther away a line is, the less influence it has on pixel's intensity.

(B) If the line does not intersect a pixel, it cannot influence the pixel's intensity.

(C) Equal areas contribute equal intensity (Regardless of the distance b/w pixel's center & the area)

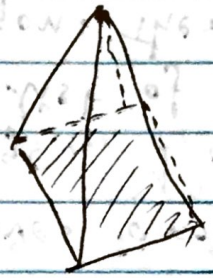
② Weighted area Sampling - keep (A) & (B) improving (C)

(C) A small area closet to the pixel center has greater influence than does one at a greater distance.



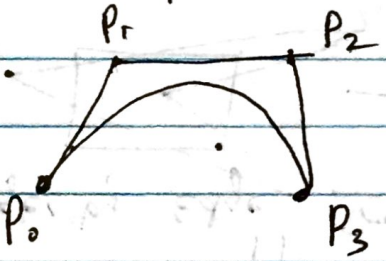
Use a circular cone as the weighting function:

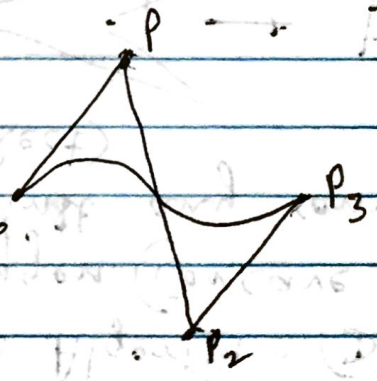
pixel intensity = Volume of the cone intersecting with the line.



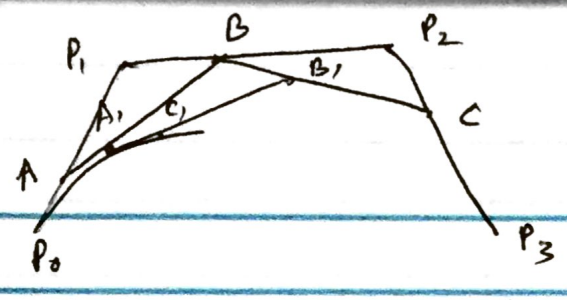
5. Bezier Curves :-

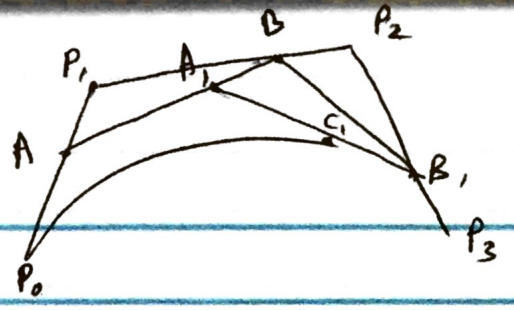
2 endpoints + 2 control points ⟶ a curve segment.





The basic idea :-





C_1 is the point for drawing the curve.

Analytically $A(t) = P_0 + t\,P_0P_1 \quad (0 \le t \le 1)$

t maybe considered as time

$$A(t) = P_0 + t(P_1 - P_0)$$
$$= (1-t)P_0 + tP_1$$

Similarly,

$$B(t) = (1-t)P_2 + tP_3$$
$$C(t) = (1-t)P_1 + tP_2$$
$$A_1(t) = (1-t)A + tC,$$
$$B_1(t) = (1-t)C + tB$$
$$C_1(t) = (1-t)A_1 + tB_1$$
$$C_1(t) = (1-t)((1-t)A + tC) + t((1-t)C + tB)$$
$$= (1-t)^2A + 2(1-t)tC + t^2B.$$
$$= (1-t^2)((1-t).P_0 + tP_1)$$
$$= 2(1-t)t\,((1-t)P_1 + tP_2) + t^2((1-t)P_2 + tP_3)$$
$$C_1(t) = (1-t)^3P_0 + 3(1-t)^2tP_1 + 3t^2(1-t)P_2 + t^3P_3 \quad \text{— (1)}$$

Alternatively, expand equation <1> :

$$C_1(t) = (-t^3 + 3t^2 - 3t + 1)P_0 + (3t^3 - 6t^2 + 3t)P_1 + (-3t^3 + 3t^2)P_2 + t^3P_3$$
$$= (-P_0 + 3P_1 + 3P_2 + P_3)t^3 + 3(P_0 - 2P_1 + P_2)t^2 + 3(P_1 - P_0)t + P_0$$

Since the points $P_0 \sim P_3$ are known, they can be combined to compute the co-efficients in advance

```
Void bezier1 (Graphics g, Point2D[] p)
{ int n = 200;
  float dt = 1.0f/n, x = p[0].x, y = p[0].y,
                                 x0, y0;
  for (int i=1 ; i<=n ; i++)
  {
    float t= i * dt , u= 1-t,
    tuTriple = 3 * t * u,
    c0 = u * u * u;
    c1 = tuTriple * u,
    c2 = tuTriple * t,
    c3 = t * t * t;
  x0 = x ; y0 = y;
  x = c0 * P[0].x + c1 * P[1].x + c2 * P[2].x +
                                  c3 * P[3].x ;
  y = c0 * P[0].y + c1 * P[1].y + c2 * P[2].y +
                                  c3 * P[3].y ;
  g.drawLine(iX(x0), iY(y0), iX(x), iY(y));
  }
}
```