CSc 133 Lecture Notes

16 - Lines and Curves

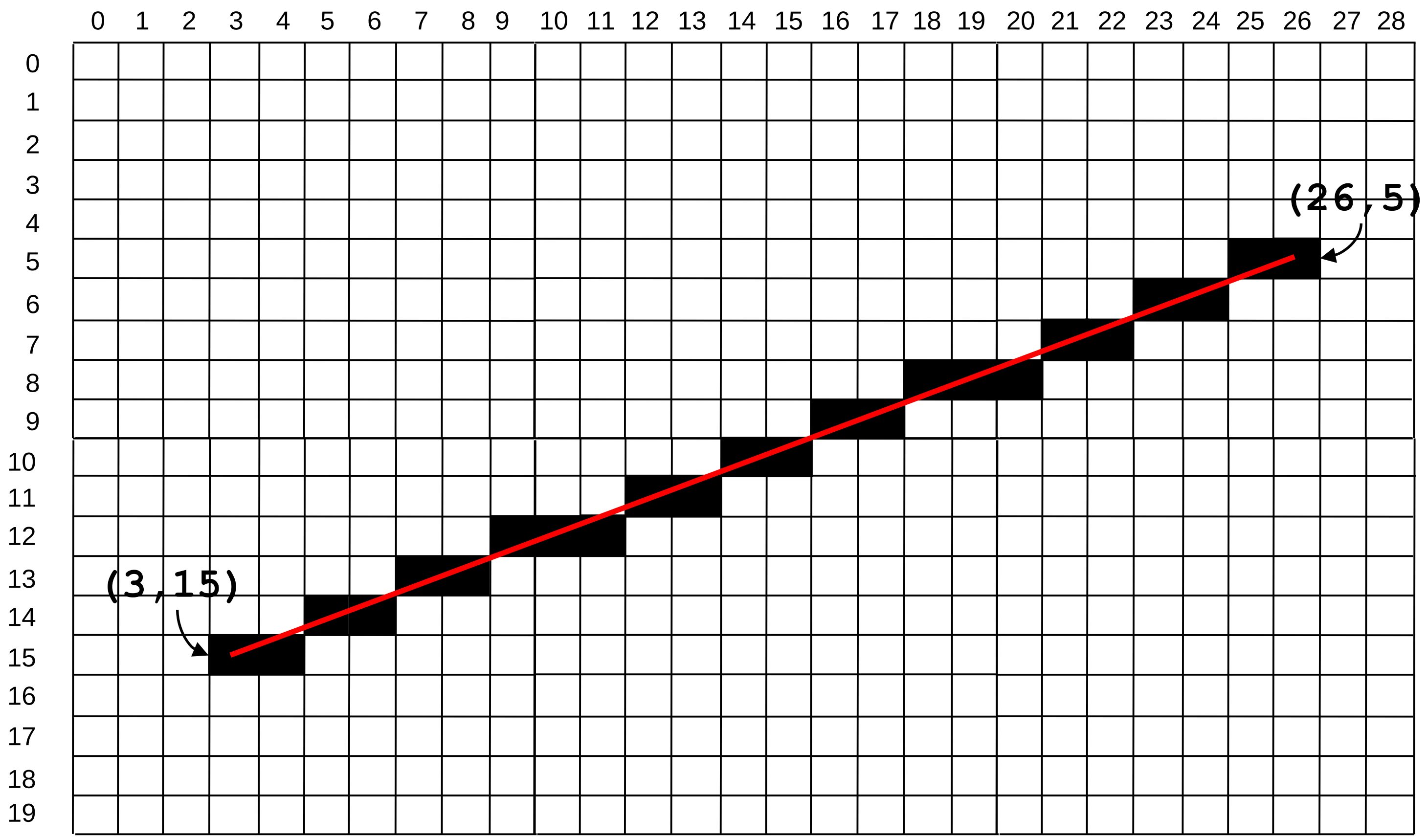
Computer Science Department
California State University, Sacramento

Overview

- **Rasterization**
- **Line-based Graphical Primitives**
- **Parametric Line Representation**
- **Quadratic & Cubic Bezier Curves**
    - **Geometric and analytical definitions**
- **Rendering Via Recursive Subdivision**
- **Applications of Curves**

Rasterization

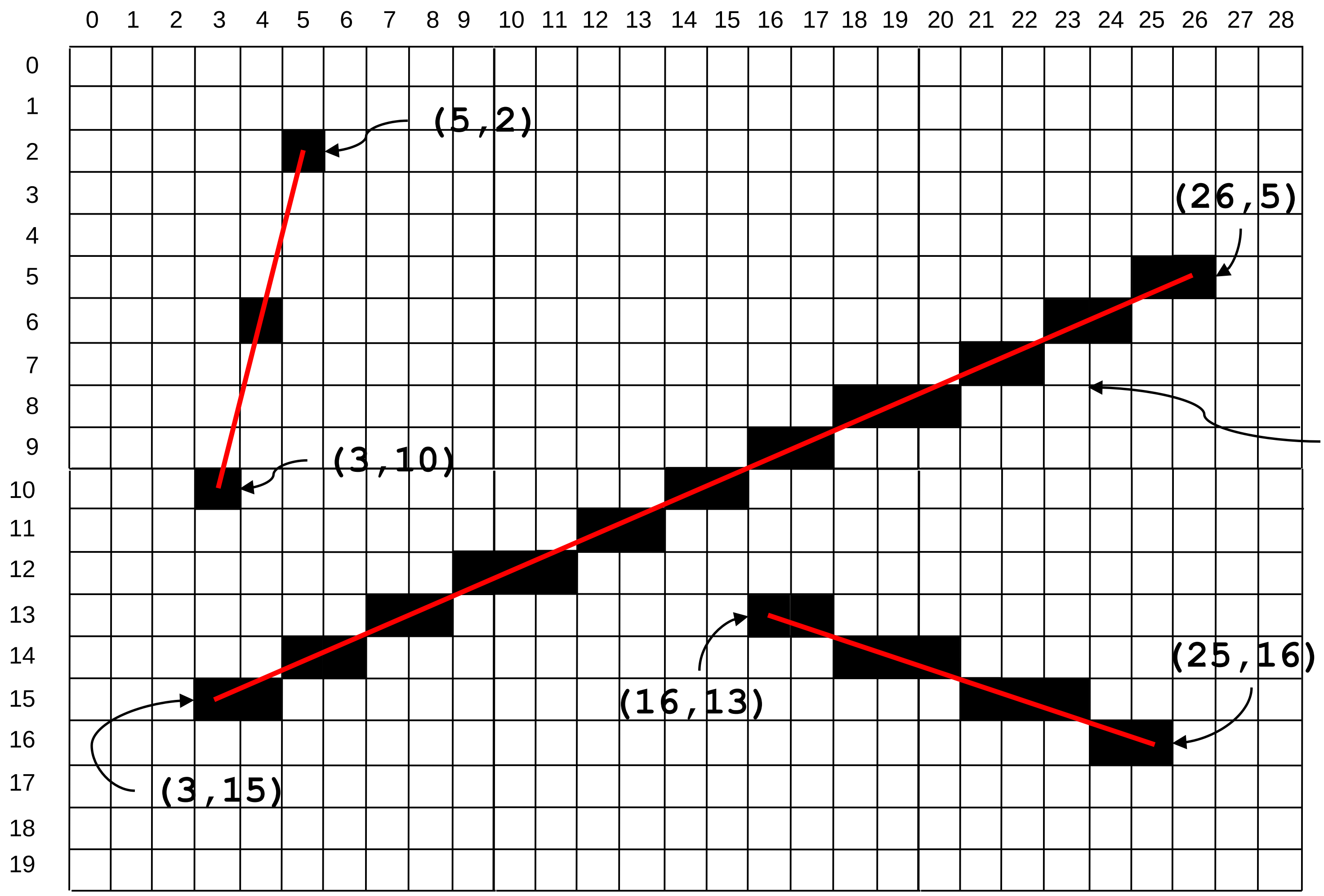
Rasterization is the task of taking an image described in a vector graphics format (shapes) and converting it into a raster image (pixels or dots) for output on a video display or printer, or for storage in a bitmap file format.



The Simple DDA Algorithm

```
/** Sets pixels on the line between points (xa,ya) and (xb,yb)
 *  to a specified color. This simple version assumes the absolute value of the
 *  slope of the line is < 1.
 */
void simpleLineDDA (int xa,ya, xb,yb; Color rgb) {
  int dx = xb - xa ;            // X-extent of the line
  int dy = yb - ya ;            // Y-extent of the line
  int xIncr = 1 ;               // increase in X per step = 1
  double yIncr = dy/dx ;        // increase in Y per step = slope
  double x = xa ;               // start at first input point
  double y = ya ;
  setPixel ((int)x, (int)y, rgb) ;
  for (int k=1; k<=dx; k++) {
    x = x + xIncr ;
    y = y + yIncr ;
    setPixel (round(x), round(y), rgb) ;
    }
}
```

Applying The DDA Algorithm



slope =

(5-15)/(26-3) =

-10/23 =

-0.43

Full DDA Algorithm

```
/** Sets pixels on the line between points (xa,ya) and (xb,yb) to a specified color.
 *  Works for lines of arbitrary slope with positive or negative direction.
 */
void LineDDA (int xa,ya, xb,yb; Color rgb) {
  int dx, dy ;               // distance in X and Y for the line
  int factor ;               // denominator used in xIncr and yIncr formulas
  double x, y ;              // 'current' loc on the line
  double xIncr, yIncr ;      // increment per step in X and Y
  dx = xb – xa ;             // X-extent of the line
  dy = yb – ya ;             // Y-extent of the line
  if abs(dy/dx) < 1 then
    factor = abs (dx)        // if abs(slope) < 1, to take unit steps in X, factor = abs(dx)= dx
  else
    factor = abs (dy) ;      // if abs(slope) >= 1, to take unit steps in Y, factor = abs(dy)
  xIncr = dx / factor ;      // increase in X per step. If abs(slope)<1, xIncr = 1. If
                             // abs(slope)>=1, xIncr = 1/abs(slope)= abs(dx)/abs(dy) = dx/abs(dy)
  yIncr = dy / factor ;      // increase in Y per step. If abs(slope)>=1, yIncr = 1 (if slope is
                             // positive) OR yIncr = -1 (if slope is negative). If abs(slope)<1,
                             // yIncr = slope = dy/dx = dy/abs(dx)
  x = xa ;                   // start at first input point
  y = ya ;
  setPixel ((int)x, (int)y, rgb) ;
  for (int k=1; k<=steps; k++) {
    x = x + xIncr ;
    y = y + yIncr ;
    setPixel (round(x), round(y), rgb) ;
  }
}
```

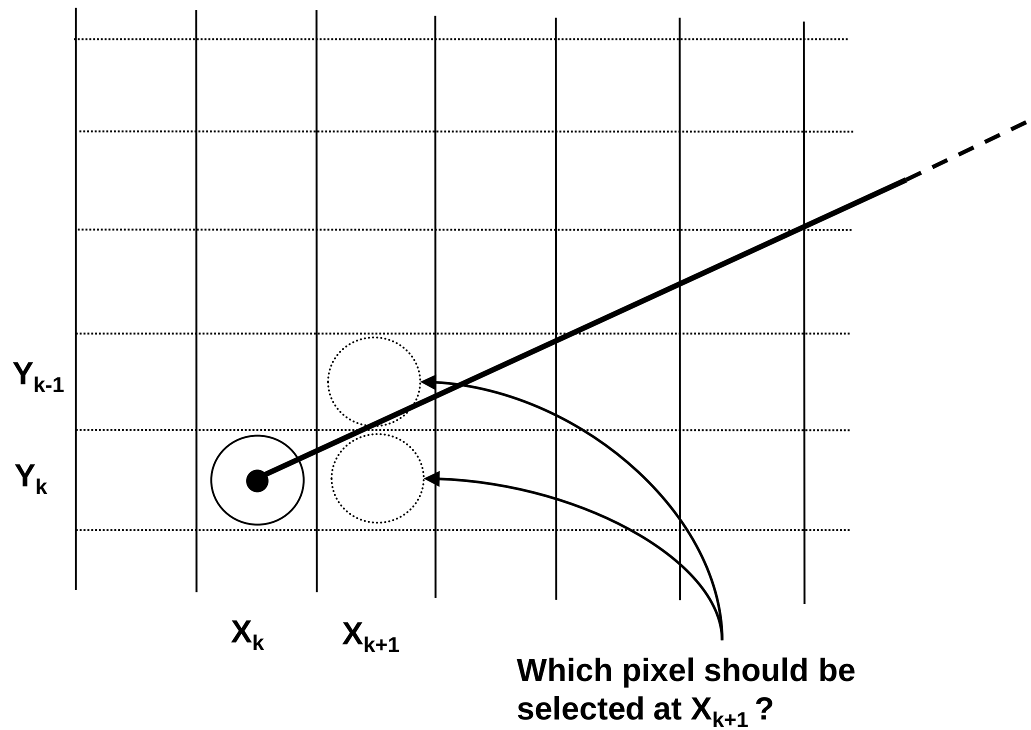
Problem with DDA Algorithm

- In the for-loop located at the end of algorithm it does a floating point arithmetic:
  - It is expensive when repeated many times.
  - It can cause a floating point error.

- These problems can result is highly inaccurate rasterization results.

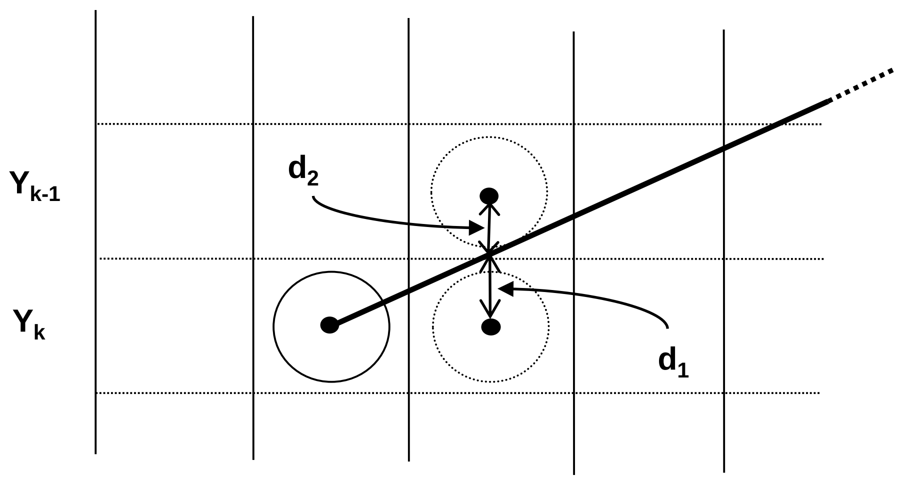
The “Pixel Selection” Decision

- Basic question: which is the best “next pixel”?



The "Pixel Decision" Parameter

- Choose the pixel *closest to the true line*



Same as "sign(d1-d2) is +"

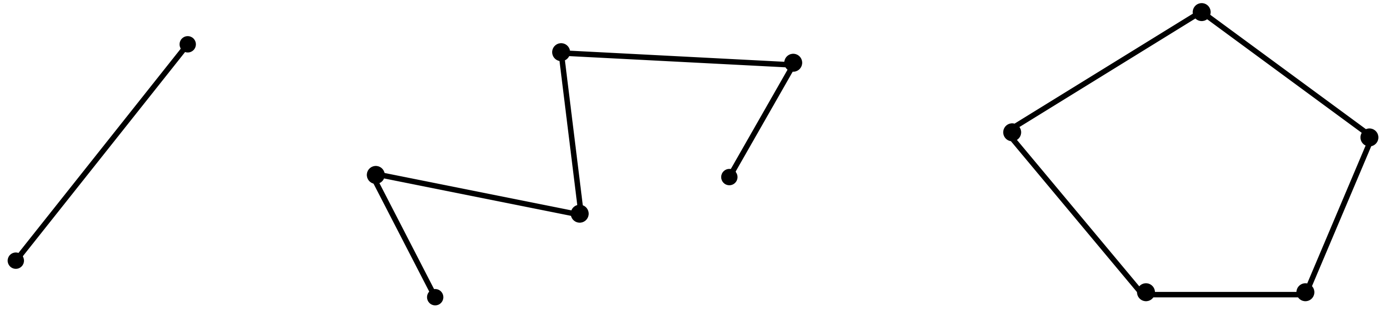
```
if ((d1-d2) > 0)
   choose pixel Yk-1
else
   choose pixel Yk
```

Bresenham's Algorithm

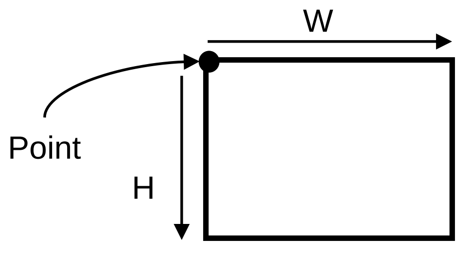
- Bresenham [IBM, 1962] figured out how to make the "sign(d2-d1) is positive" test using only integer arithmetic.
- No floating point involved!
- This results in rasterization that is at the same time faster and also more accurate (because it always chooses the "best next pixel").

Graphical Primitives

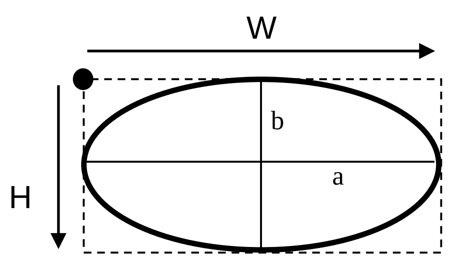
- Point- and Line-based



Line **"Polyline"** **"Polygon"**



Rectangle

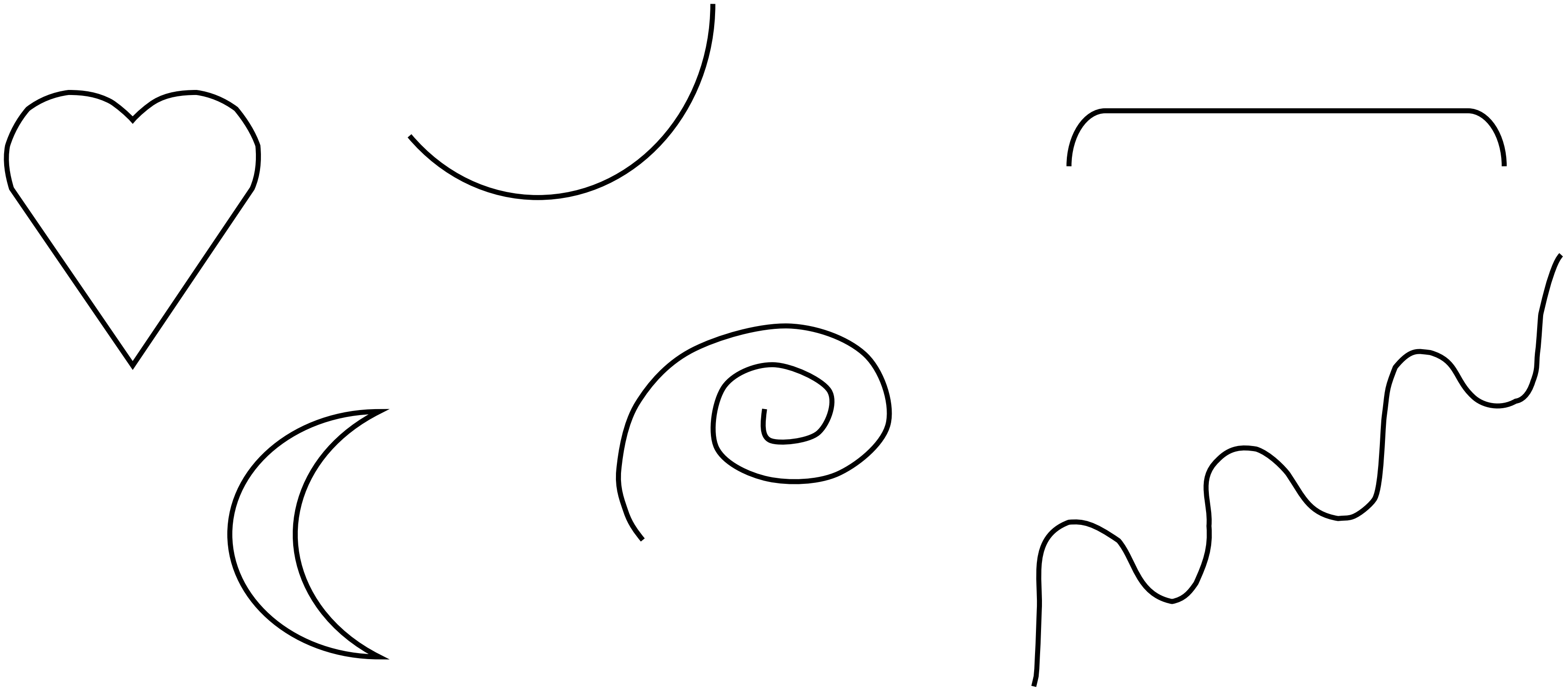


Oval

$$\frac{(x - xCenter)^2}{a^2} + \frac{(y - yCenter)^2}{b^2} = 1$$

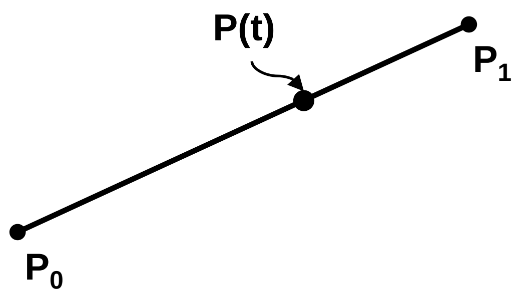
Curves Of Higher Complexity

- What if we want to draw shapes like these?



Parametric Line Representation

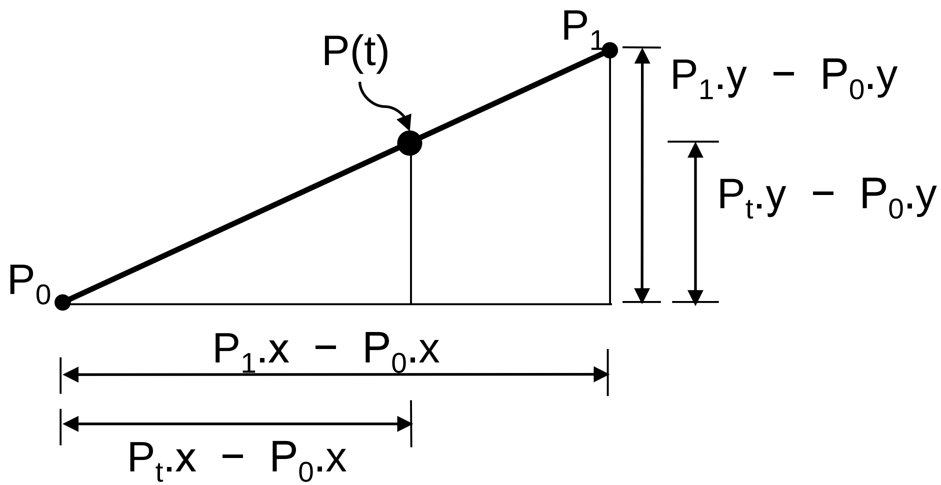
- Lines can be represented in terms of known quantities in several ways :
  - `Y = mX + b`// line with slope "m" and Y-intercept "b"
  - `(P0, P1)` // line containing $\mathbf{P_0}$ and $\mathbf{P_1}$
- Any point on **(P_0, P_1)** can be represented with a single *parameter value* ***'t'***



- **'t' is the ratio of [distance from P_0 to P(t)] to [distance from P_0 to P_1]**
- **Every point on the line has a unique 't' value**

Parametric Line Representation (cont.)

- Parametric equation for points ***P(t)*** on a line:



$$t = \frac{P_t - P_0}{P_1 - P_0}$$

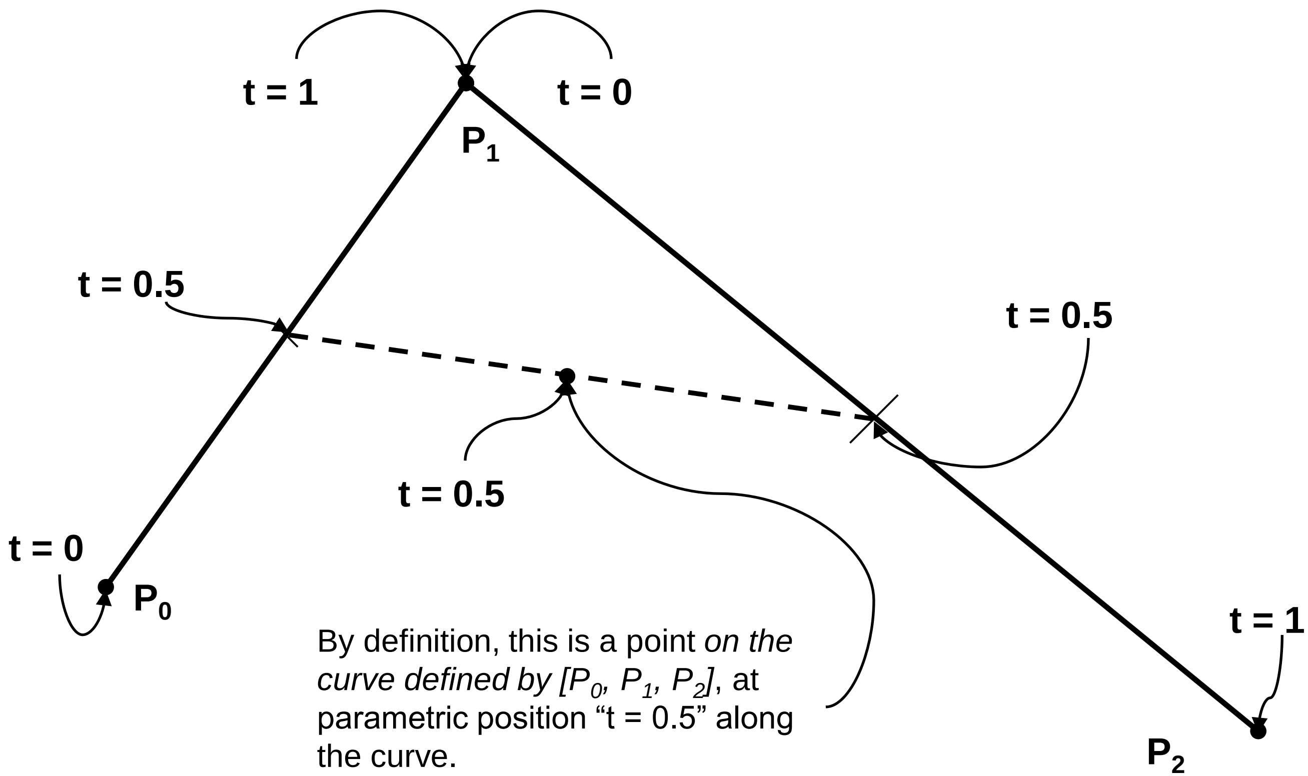
$$t\left(P_1 - P_0\right) = P_t - P_0$$

$$P_t = P_0 + t\left(P_1 - P_0\right)$$

$$P_t = \left(1 - t\right) P_0 + t\, P_1$$

Quadratic Bezier Curves

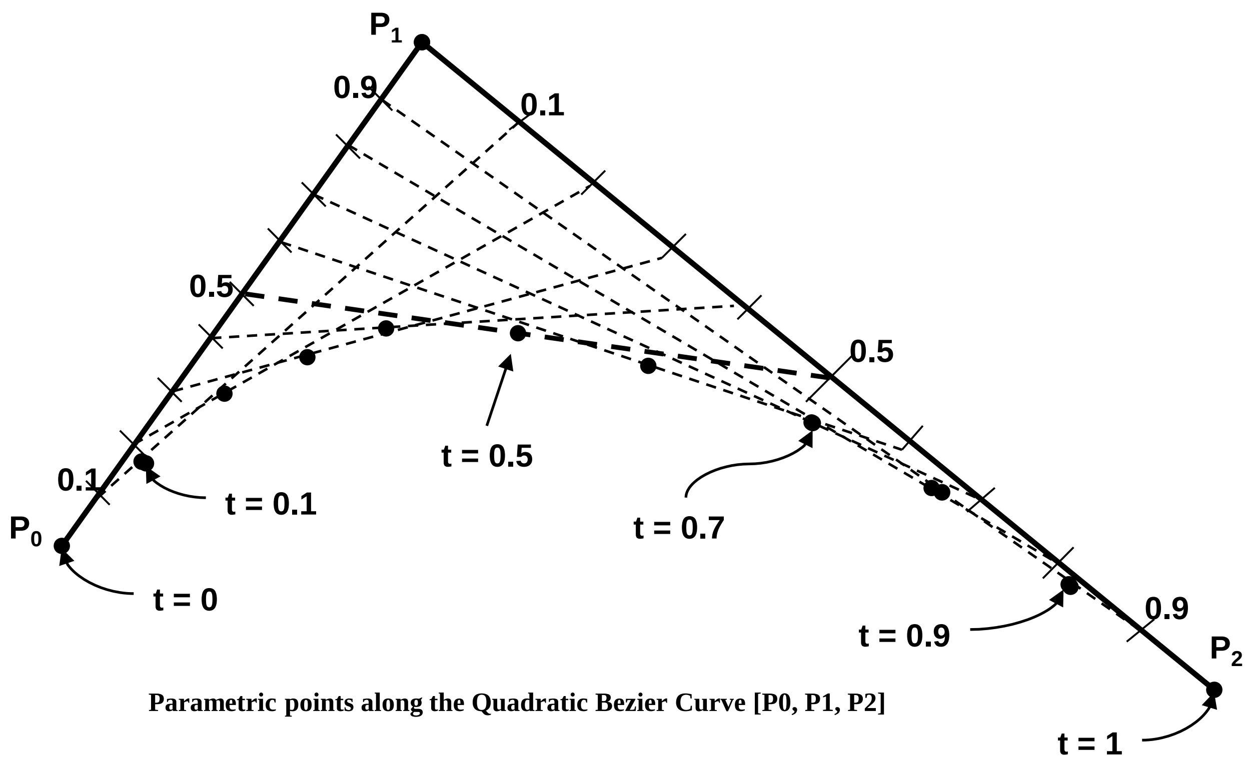
- Geometric description



By definition, this is a point *on the curve defined by [P_0, P_1, P_2]*, at parametric position “t = 0.5” along the curve.

Quadratic Bezier Curves (cont.)

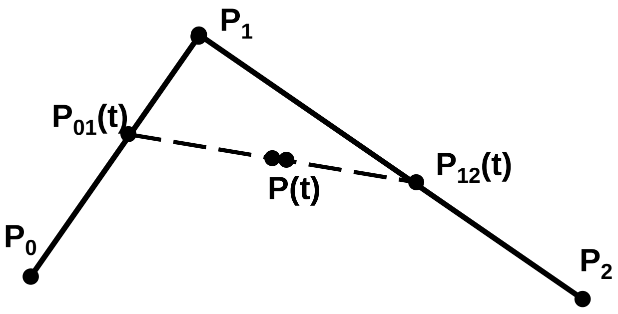
- Connecting points of equal parametric value generates a curve:



Parametric points along the Quadratic Bezier Curve [P0, P1, P2]

Quadratic Bezier Curves (cont.)

- Analytic definition



$$P_{01}(t) = t \cdot P_1 + (1-t) \cdot P_0 \qquad [1]$$

and

$$P_{12}(t) = t \cdot P_2 + (1-t) \cdot P_1 \qquad [2]$$

and a point on the curve $[P_0 \quad P_1 \quad P_2]$ is defined as

$$P(t) = t \cdot (P_{12}(t)) + (1-t) \cdot (P_{01}(t)) \qquad [3]$$

Substituting [1] and [2] into [3] gives

$$P(t) = t \cdot (t \cdot P_2 + (1-t) \cdot P_1) + (1-t) \cdot (t \cdot P_1 + (1-t) \cdot P_0)$$

Factoring and combining the constant terms P_0, P_1, and P_2 gives

$$P(t) = (1-t)^2 \cdot P_0 + \left(-2t^2 + 2t\right) \cdot P_1 + \left(t^2\right) \cdot P_2$$

Curves as Weighted Sums

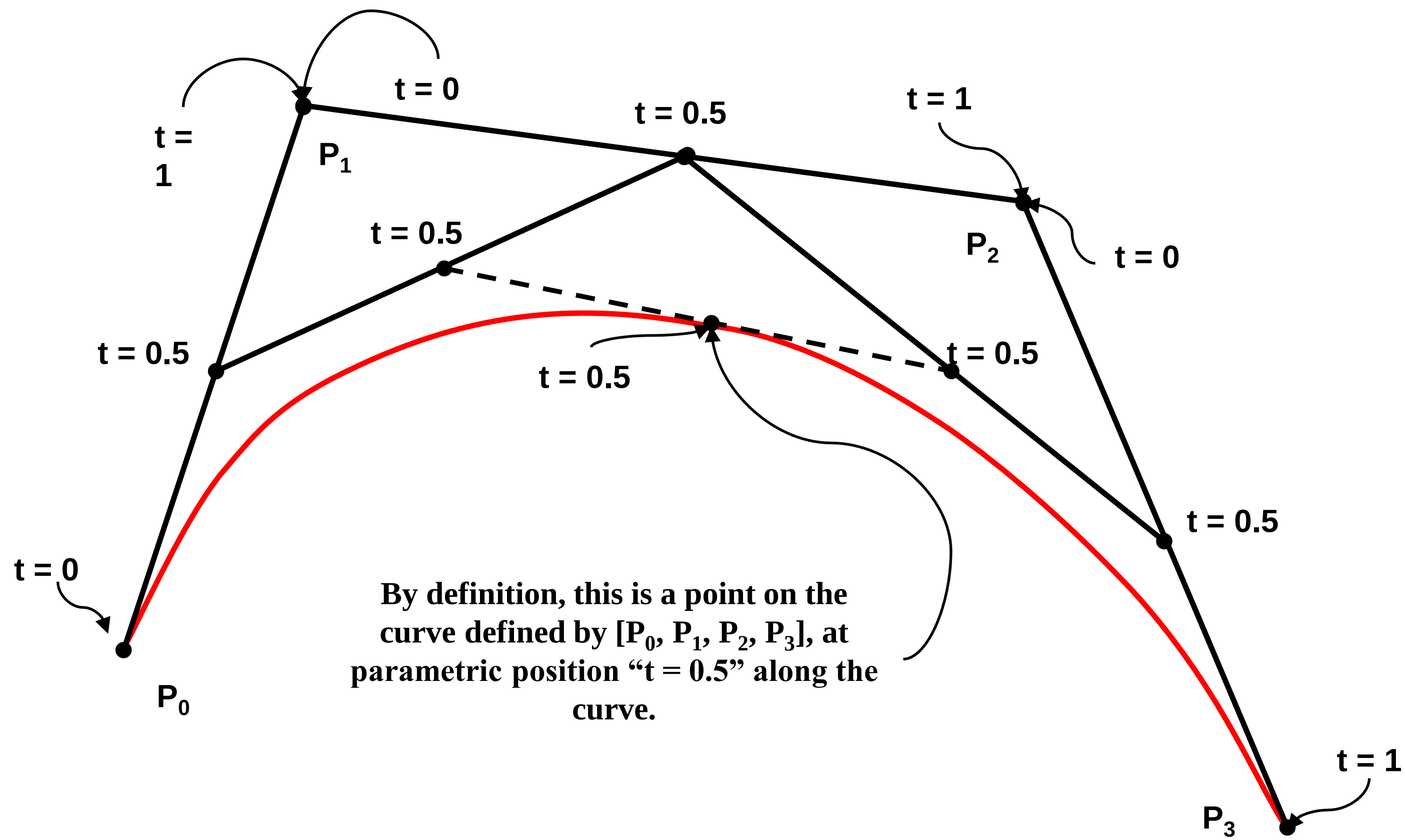
$$P(t) = (1-t)^2 \cdot P_0 + \left(-2t^2 + 2t\right) \cdot P_1 + \left(t^2\right) \cdot P_2$$

$$P(t) = \sum_{i=0}^{2} P_i \cdot B_i(t), where \begin{cases} B_0(t) = (1-t)^2 \\ B_1(t) = \left(-2t^2 + 2t\right) \\ B_2(t) = t^2 \end{cases}$$

- *A point on the curve is a* ***weighted sum*** *of the three "control points"*
    - The "weightings" are the quadratic polynomials, evaluated at "*t* "

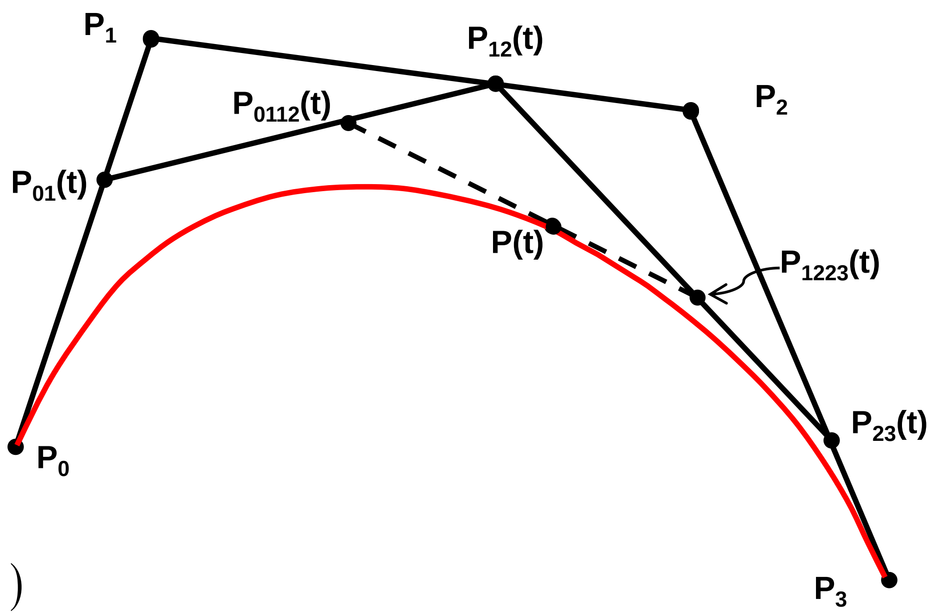
Cubic Bezier Curves

- Geometric description



Cubic Bezier Curves (cont.)

- Analytic definition



$$P_{01}(t) = t \cdot P_1 + (1-t) \cdot P_0$$

$$P_{12}(t) = t \cdot P_2 + (1-t) \cdot P_1$$

$$P_{23}(t) = t \cdot P_3 + (1-t) \cdot P_2$$

$$P_{0112}(t) = t \cdot P_{12}(t) + (1-t) \cdot P_{01}(t)$$

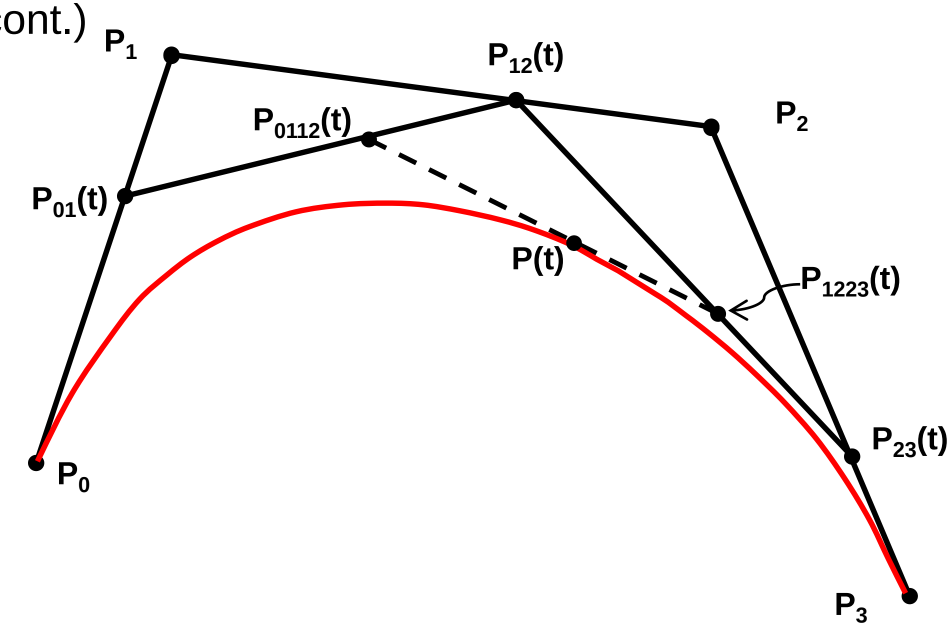
$$P_{1223}(t) = t \cdot P_{23}(t) + (1-t) \cdot P_{12}(t)$$

and a point on the curve $[P_0 \quad P_1 \quad P_2 \quad P_3]$ is defined as

$$P(t) = t \cdot (P_{1223}(t)) + (1-t) \cdot (P_{0112}(t))$$

Cubic Bezier Curves (cont.)

- Analytic definition (cont.)

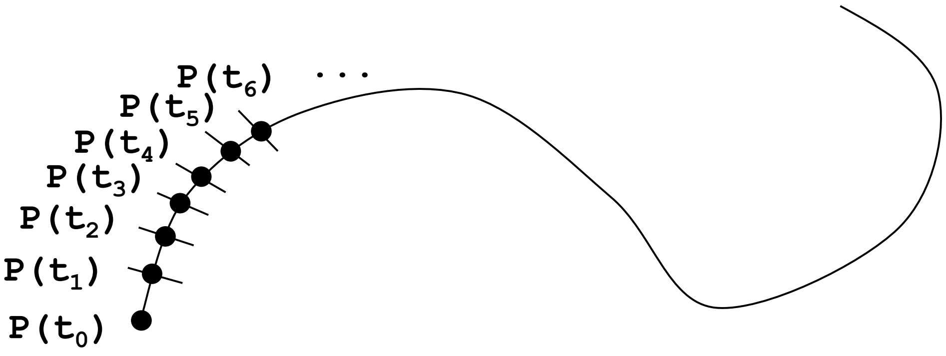


$$
\begin{aligned}
P(t) &= t \cdot \left(P_{1223}(t)\right) + (1-t) \cdot \left(P_{0112}(t)\right) \\
&= \underline{(1-t)^3} \cdot P_0 + \underline{\left(3t^3 - 6t^2 + 3t\right)} \cdot P_1 + \underline{\left(-3t^3 + 3t^2\right)} \cdot P_2 + \underline{\left(t^3\right)} \cdot P_3 \\
&= \sum_{i=0}^{3} P_i \cdot B_{i,3}(t)
\end{aligned}
$$

Drawing Bezier Curves

- Iterative approach

```
moveTo (P(t0));
drawTo (P(t1));
drawTo (P(t2));
drawTo (P(t3));
...
```



Drawing Bezier Curves (cont.)

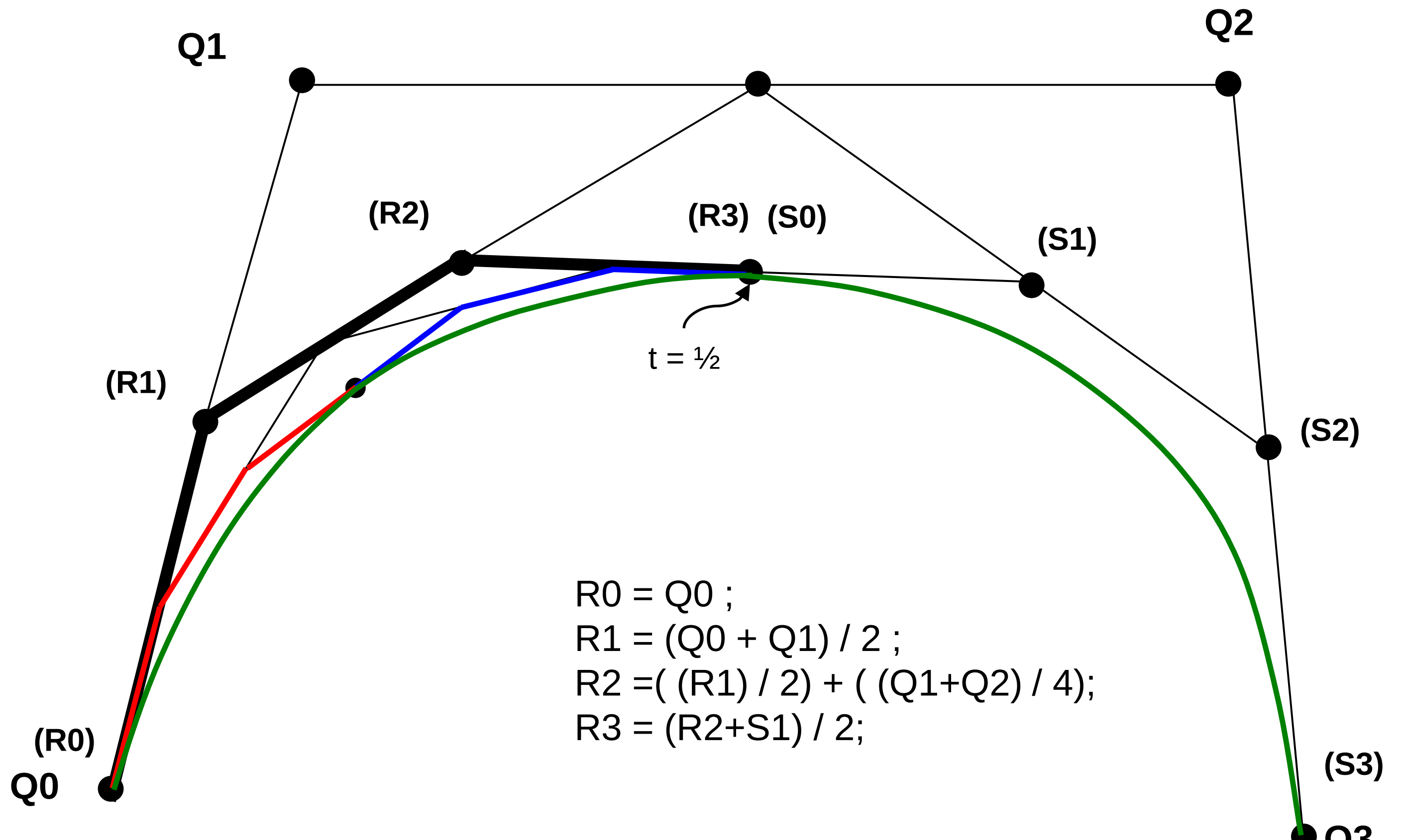
```
/** A routine to draw the (cubic) Bezier Curve represented by the (1x4) input
 *  Control Point Array using iterative plotting along the curve and an explicit
 *  computation which produces a weighted sum of control points for each new point.
 *  Note: This is (Java-like) pseudo code, not real Java code.
 */
void drawBezierCurve (controlPointArray) {
  currentPoint = controlPointArray [0] ; // start drawing at first control point
  t = 0 ;    // vary the parametric value "t" over the length of the curve
  while ( t<=1 ) {
    // compute next point on the curve as the sum of the Control Points, each
    // weighted by the appropriate polynomial evaluated at 't'.
    nextPoint = (0,0) ;
    for (int i=0; i<=3; i++) {
      nextPoint = nextPoint + ( blendingFunction(i,t) * controlPointArray[i] );
    }
    drawLine (currentPoint,nextPoint);
    currentPoint = nextPoint;
    t = t + smallFloatIncrement;
  }
}
```

Drawing Bezier Curves (cont.)

```
/** Returns the value of the "ith" cubic Bernstein polynomial blending
 *  function at parametric location 't'
 */
double blendingFunction (int i, double t) {
  switch (i)  {
     case 0: return ( (1-t) * (1-t) * (1-t) ) ;     // (1-t)^3
     case 1: return ( 3 * t * (1-t) * (1-t) ) ;     // 3t(1-t)^2
     case 2: return ( 3 * t * t * (1-t) ) ;         // 3t^2(1-t)
     case 3: return ( t * t * t ) ;                 // t^3
  }
}
```

Control Mesh Subdivision

- Split the control mesh [Q] at t=1/2
  - Produces two meshes [R] and [S]



R0 = Q0 ;
R1 = (Q0 + Q1) / 2 ;
R2 =((R1) / 2) + ((Q1+Q2) / 4);
R3 = (R2+S1) / 2;

Recursive Subdivision

```
/** Draws the (cubic) Bezier curve represented by the (1x4) input Control Point Vector
 *  by recursively subdividing the Control Point Vector until the control points are
 *  within some tolerance of being colinear, at which time the Control Points are deemed
 *  "close enough" to the curve for the 1st and last control points to be used as the
 *  ends of a line segment representing a short piece of the actual Bezier curve.
*   Note: This is (Java-like) pseudo code, not real Java code. */
void drawBezierCurve (ControlPointVector) {
  if ( straightEnough (ControlPointVector))
      Draw Line from 1st Control Point to last Control Point ;
  else  {
      subdivideCurve (ControlPointVector, LeftSubVector, RightSubVector) ;
      drawBezierCurve (LeftSubVector) ;
      drawBezierCurve (RightSubVector) ;
  }
}
 /** Splits the input control point vector Q into two control point
  *  vectors R and S such that R and S define two Bezier curve segments that
  *  together exactly match the Bezier curve defined by Q.
  */
void subdivideCurve (ControlPointVector Q,R,S) {
  R(0) = Q(0) ;
  R(1) = (Q(0)+Q(1)) / 2.0 ;
  R(2) = (R(1)/2.0) + (Q(1)+Q(2))/4.0 ;
  S(3) = Q(3) ;
  S(2) = (Q(2)+Q(3)) / 2.0 ;
  S(1) = (Q(1)+Q(2))/4.0 + S(2)/2.0 ;
  R(3) = (R(2)+S(1)) / 2.0 ;
  S(0) = R(3) ;
}
```

Recursive Subdivision (cont.)

```
/** determines whether the four points Q0,Q1,Q2,Q3 in the input array of Control
 * Points are within some tolerance "epsilon" of being colinear.
 */
boolean straightEnough (ControlPointVector) {
  // find length around control polygon
  d1 = lengthOf(Q0,Q1) + lengthOf(Q1,Q2) + lengthOf(Q2,Q3);
  // find distance directly between first and last control point
  d2 = lengthOf(Q0,Q3) ;
  if ( abs(d1-d2) < epsilon )      // epsilon (“tolerance”) = (e.g.) .001
      return true ;
  else
      return false ;
}
```

Applications Of Curves

- Two types of "fonts"
  - Bit-mapped
  - Outline