# Lines and Curves

Computer Graphics – lecture 4

Dr. Zahraa Yasseen

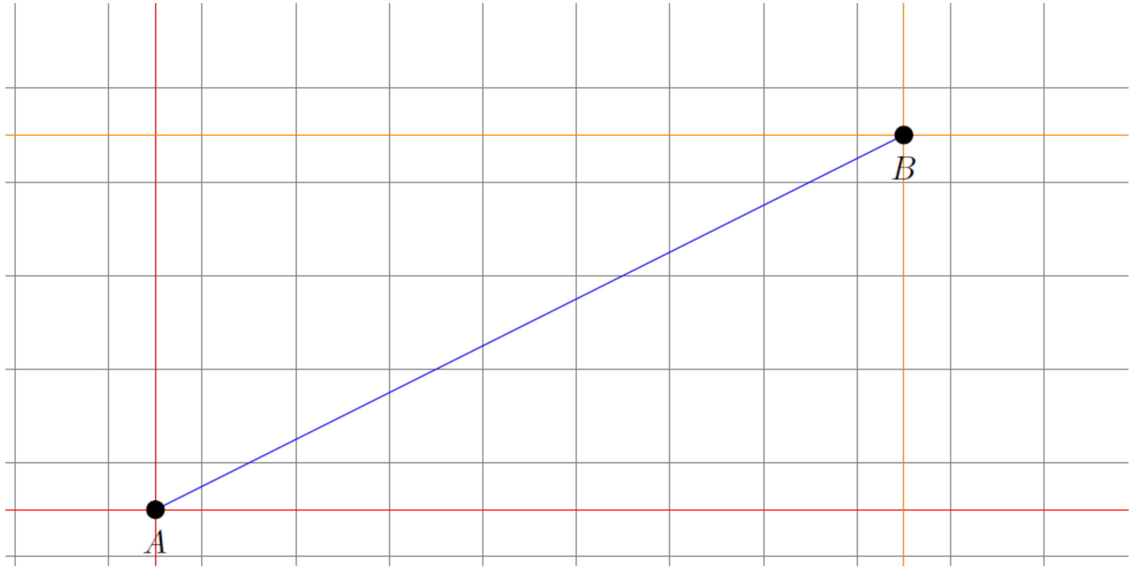
# Line Drawing

- A straight line segment connecting two given points
- An algorithm for drawing the line **has to decide exactly which pixels to color**.
- **Bresenham's algorithm** for line drawing, implements a very efficient procedure
- Complications involved:
  - Antialiasing
  - Line width
  - Line ends: round vs square caps

# Bresenham’s line algorithm

- We have a line segment specified by end points A and B with coordinates (m; n) and (m0; n0) respectively where m, n, m0 and n0 are integers.

y = mx + b



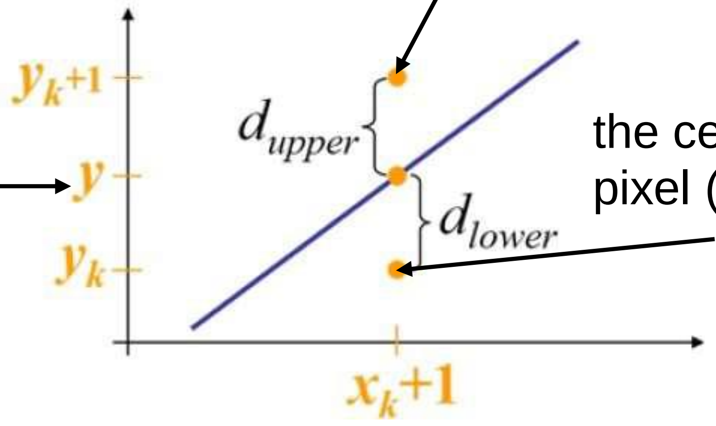
# Bresenham’s line algorithm (cont.)

- Suppose we have drawn a pixel in column $x_k$ and row $y_k$.
- We need to find out which of the two pixels to color in the next column $x_k$ +1.
- Since the pixels are adjacent vertically, it is natural to use distance as a way of determination.

the center of the pixel ($x_k$ +1, $\mathbf{y_k}$ **+1**),

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

y = mx + b being the equation of the line between the end points

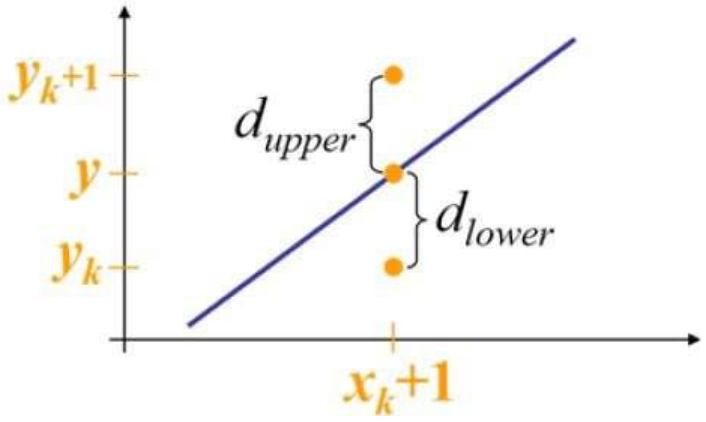



the center of the pixel ($x_k$ +1, $\mathbf{y_k}$),

# Derivation

- we only need the value d, which is the difference of them, to determine which pixel to color:

$$
\begin{aligned}
d_l &= y - y_k \\
&= m(x_k + 1) + b - y_k \\
d_u &= y_k + 1 - y \\
&= y_k + 1 - m(x_k + 1) - b
\end{aligned}
$$

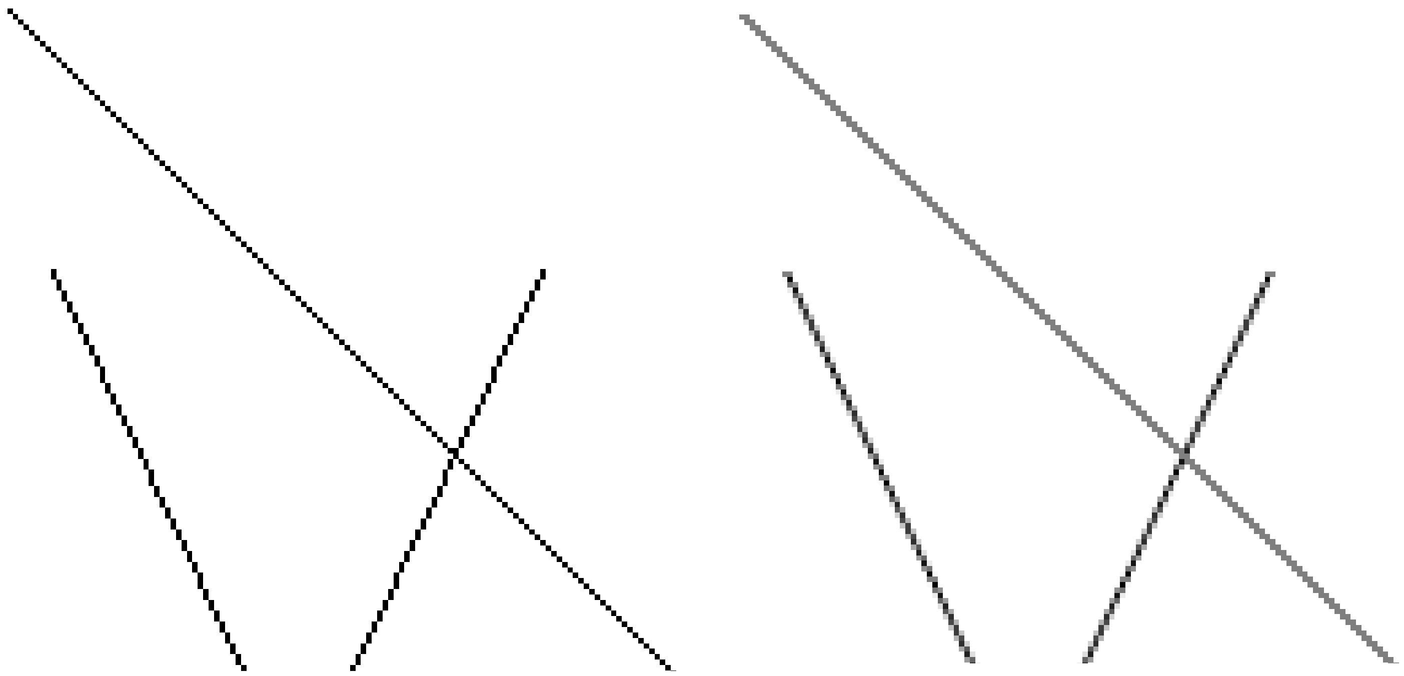



$$d = d_l - d_u = 2m(x_k + 1) + 2b - 2y_k - 1$$

# Anti-aliasing

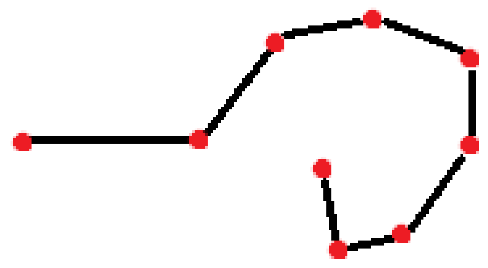
- The fact that some information is lost in the process of mapping something continuous in nature to a discrete space is called **aliasing**.

One way of reducing the raggedness of the line segment we draw is to color pixels in different grey scales.



# Polylines

- Sequence of vertices connected by straight line segments
- Useful, but not for smooth curves
- This is the representation that usually gets drawn in the end (a curve is converted into a polyline)

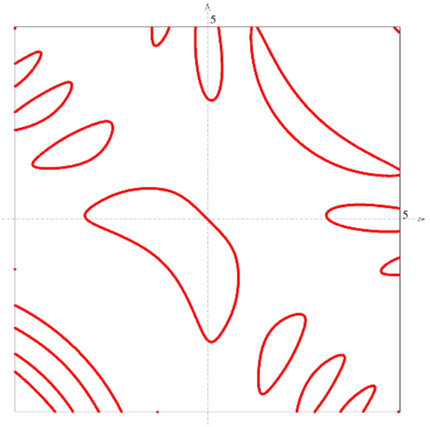


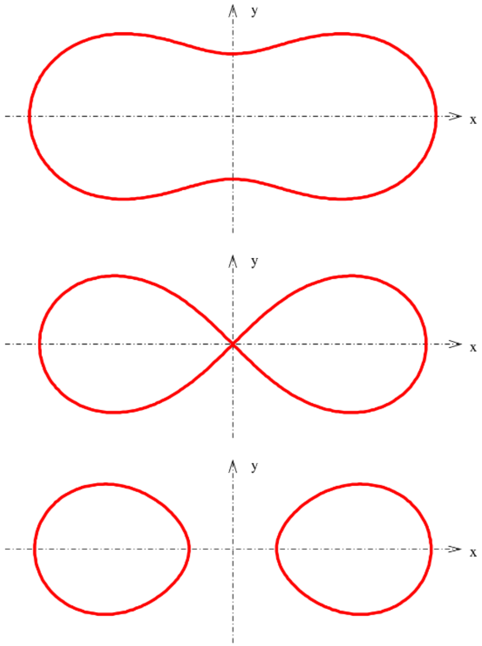
# Curves

- Three types:
  - Implicit
  - Explicit
  - Parametric curves

# Implicit Curves

- Defined by an implicit function of the form f(x, y) = 0
- Examples of implicit curves include:
  1. Line: ax + by + c = 0
  2. Circle: $x^2 + y^2 = R^2$
  3. Cassini oval: $(x^2+y^2)^2 - 2c^2(x^2-y^2) - (a^4-c^4) = 0$
  4. sin(x + y) – cos (xy) + 1 = 0






Cassini ovals:
(1) a=1.1, c=1 (above),
(2) a=c=1 (middle),
(3) a=1, c=1.05 (below)

# Implicit curve visualization

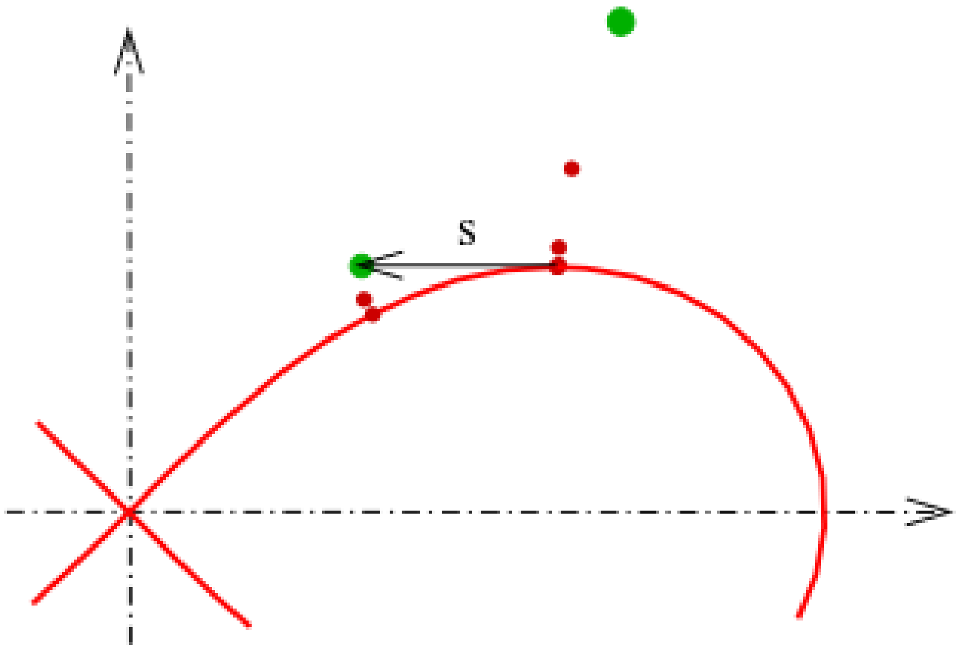
- Goal: determine a set of closely spaced points on the curve and draw as a polyline.
- A trivial approach is to solve for y in terms of x (using positive and negative square roots) and then to plot points (x,y) on the curve.

  This approach does not generalize easily and proves utterly impracticable for a more complicated relation such as ysinx + x cosy

- Alternative methods:
  - Tracing algorithm
  - Rasterization

# Implicit Curve Visualization (cont.)

- Tracing algorithm
  - Method:
    1. Choose a suitable starting point $P_n$ *near* the curve
    2. Using $P_n$, Determine a point $P_0$ *on* the curve
    3. Using the tangent to the curve on $P_0$, find another point near the curve and repeat the process to determine $P_1$ on the curve
  - Set back: If the implicit curve consists of several parts, it has to be started several times with suitable starting points.



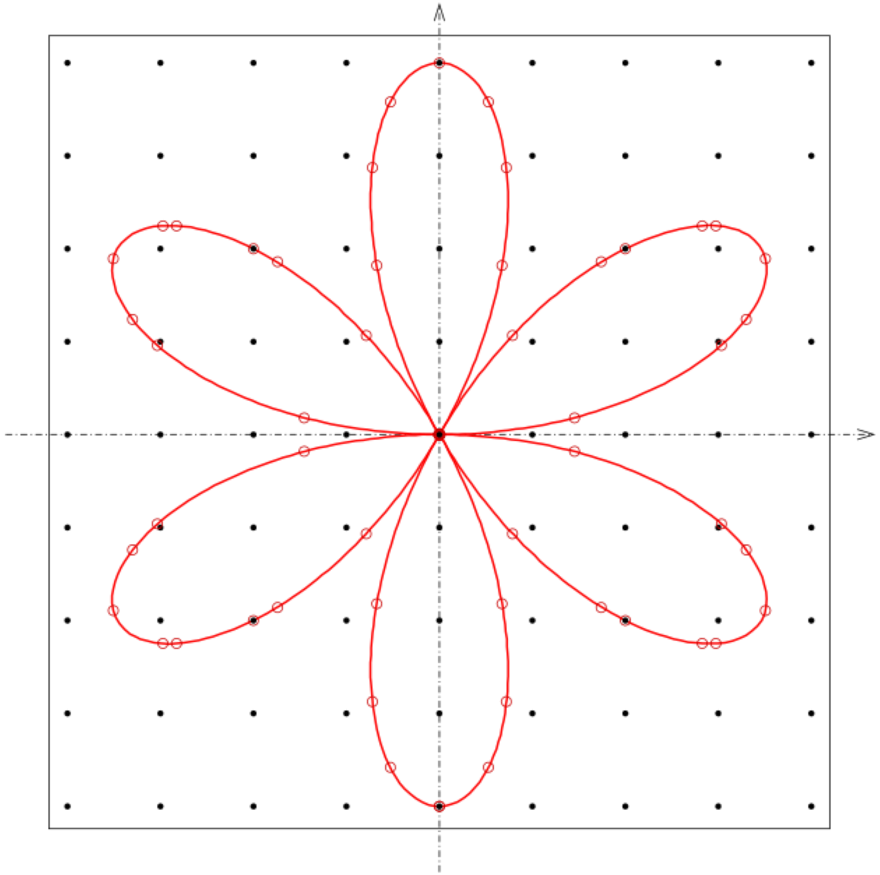

https://www.wikiwand.com/en/Implicit_curve

# Implicit Curve Visualization (cont.)

- Rasterization

  Produce a dense set of points without regard to their connectivity.

  On/off pixels rather than a set of points connected as a polyline.



# Explicit Curves

- Describe one coordinate of a curve or surface as an explicit function of the others: $y = f(x)$
- More straight forward to visualize.

```
For x in a:b
      add (x, f(x)) to polyline
Draw polyline
```

# Explicit curves visualization

- Select an interval to visualize. E.g.: $x \in [-10, 10]$
- Discretize the interval. The finer the discretization, the smoother the curve will be. E.g.: $dx = 0.1$
- Substitute values of x in f(x) to find the points of the polyline to draw. E.g.: $f(x) = 3x^2$: {(-10, 300), (-9.9, 294.03), …., (9.9, 294.03), (10, 300)}
- Depending on the interval and the differential step (dx), the number of generated points (size of the polyline) is determined:

$$\frac{interval\ length}{step\ size} = \frac{(10\ - (-10))}{0.1}$$

# Parametric Curves

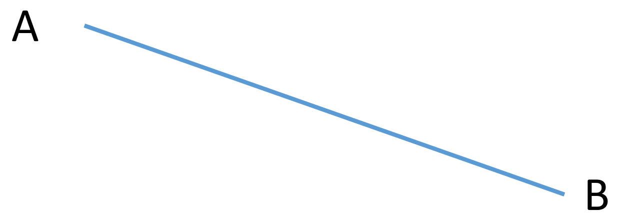
- Definition: A Parametric Curve in $\mathbb{R}^2$ is a set of equations x=x(t) and y=y(t) that trace a curve C as the Parameter t varies.
- Examples (https://tutorial.math.lamar.edu/classes/calcii/parametriceqn.aspx):
  - $x = r \times \cos(t)$ , $y = r \times \sin(t)$
  - $x = t^2 + t$, $y = 2t - 1$
  - $x = 5 \cos(t)$, $y = 2 \sin(t)$   $0 \leq t \leq 2\pi$

# Parametric curves visualization

- x = f(t) y = g(t)
- Similar to explicit curve
- Discretize the interval of t
- Generate the $(f(t_0), g(t_0)), (f(t_1), g(t_1)), \ldots. (f(t_N), g(t_N))$ couples
- Connect them as a polyline!

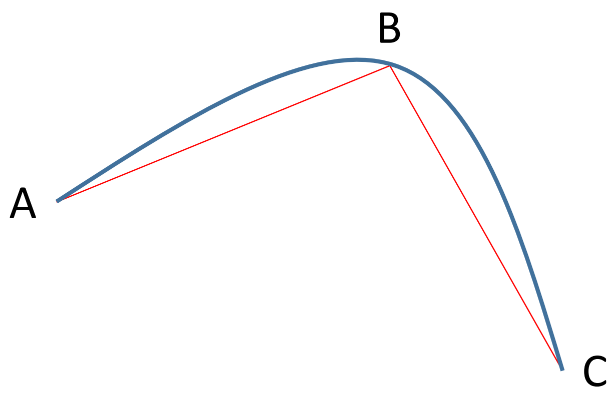
# Interpolation

- Problem: draw a smooth curve passing through given points




$$x(t) = (1 - t) \times x_A + t \times x_B$$

$$y(t) = (1 - t) \times y_A + t \times y_B$$




$$x(t) = f_1(t) \times x_A + f_2(t) \times x_B + f_3(t) \times x_C$$

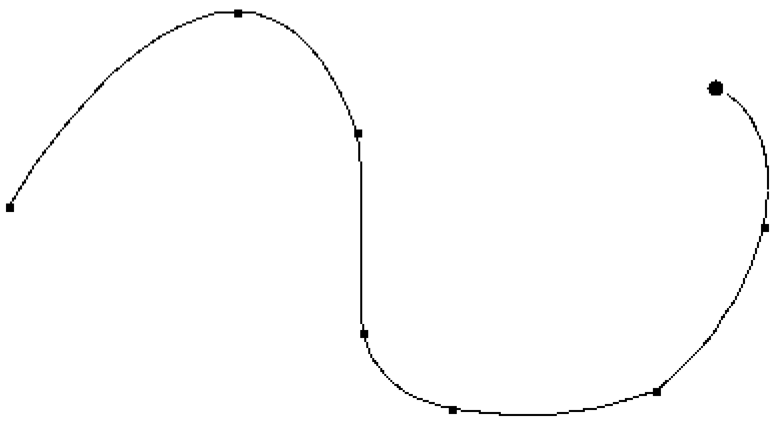
$$y(t) = f_1(t) \times y_A + f_2(t) \times y_B + f_3(t) \times y_C$$

# Interpolation

- Many methods exist
- One solution is using LaGrange interpolated curves

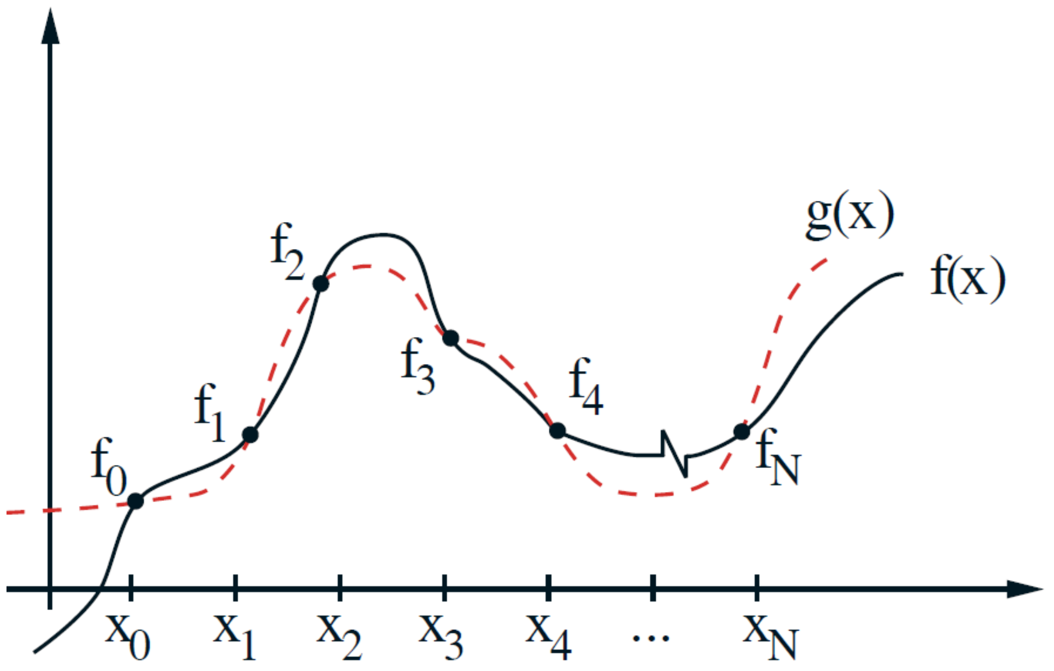
# LaGrange interpolated curves

- Create a smooth curve that passes through an ordered group of points (called the **control points**).
- The general idea is to use the control points to generate parametric equations.
- To draw the curve, all that is then needed is to step through u at some small amount, drawing straight lines between the calculated points. The smaller the step size the more smooth the curve will appear.



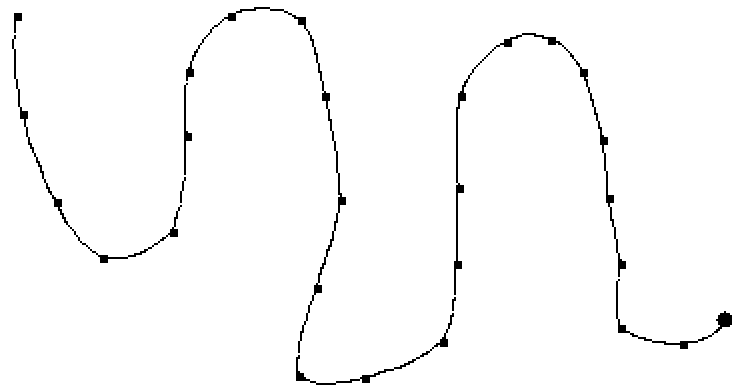
# LaGrange interpolated curves

- Fit n+1 points with an $n^{th}$ degree polynomial



# Drawing LaGrange curves

- Specify a curve that will pass through any number of control points.
- The curve's function can be constructed as a sum of terms, one for each control point:
  - fx(u) = sum from i = 1 to n of x[i] B[i](u)
  - fy(u) = sum from i = 1 to n of y[i] B[i](u)
- Each function B[i](u) specifies how much the ith control point effects the position of the curve.
- The name for these functions are **Blending Functions.**



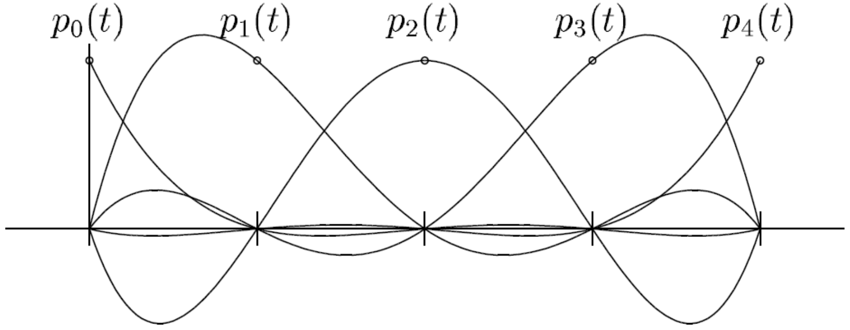
# LaGrange Blending Functions

- Theorem: the solution to the polynomial interpolation problem can be expressed as:

$$P(t) = p_0(t)P_0 + \cdots + p_n(t)P_n, \text{ where for each } i,$$

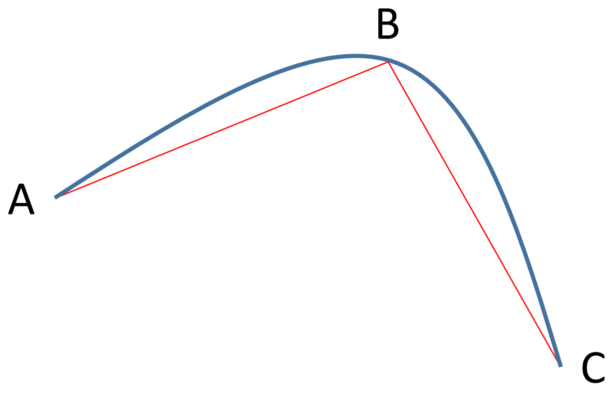
$$p_i(t) = \frac{\prod_{j \neq i}(t - t_j)}{\prod_{j \neq i}(t_i - t_j)} = \prod_{j \neq i} \frac{t - t_j}{t_i - t_j}.$$

- The polynomials $p_i(t)$ are independent of the control points



# LaGrange Blending Functions

$$p_i(t) = \frac{\prod_{j \neq i}(t - t_j)}{\prod_{j \neq i}(t_i - t_j)} = \prod_{j \neq i} \frac{t - t_j}{t_i - t_j}.$$




$$x(t) = p_1(t) \times x_A + p_2(t) \times x_B + p_3(t) \times x_C$$

$$y(t) = p_1(t) \times y_A + p_2(t) \times y_B + p_3(t) \times y_C$$

# Example: for three control points

To find a curve of degree at most 2 such that P(-1) = A, P(0) = B, P(1) = C

We have $n = 2$, $t_0 = -1, t_1 = 0, t_2 = 1$.

$$p_0(t) = \frac{(t-0)(t-1)}{(-1-0)(-1-1)} = \tfrac{1}{2}(t^2 - t)$$

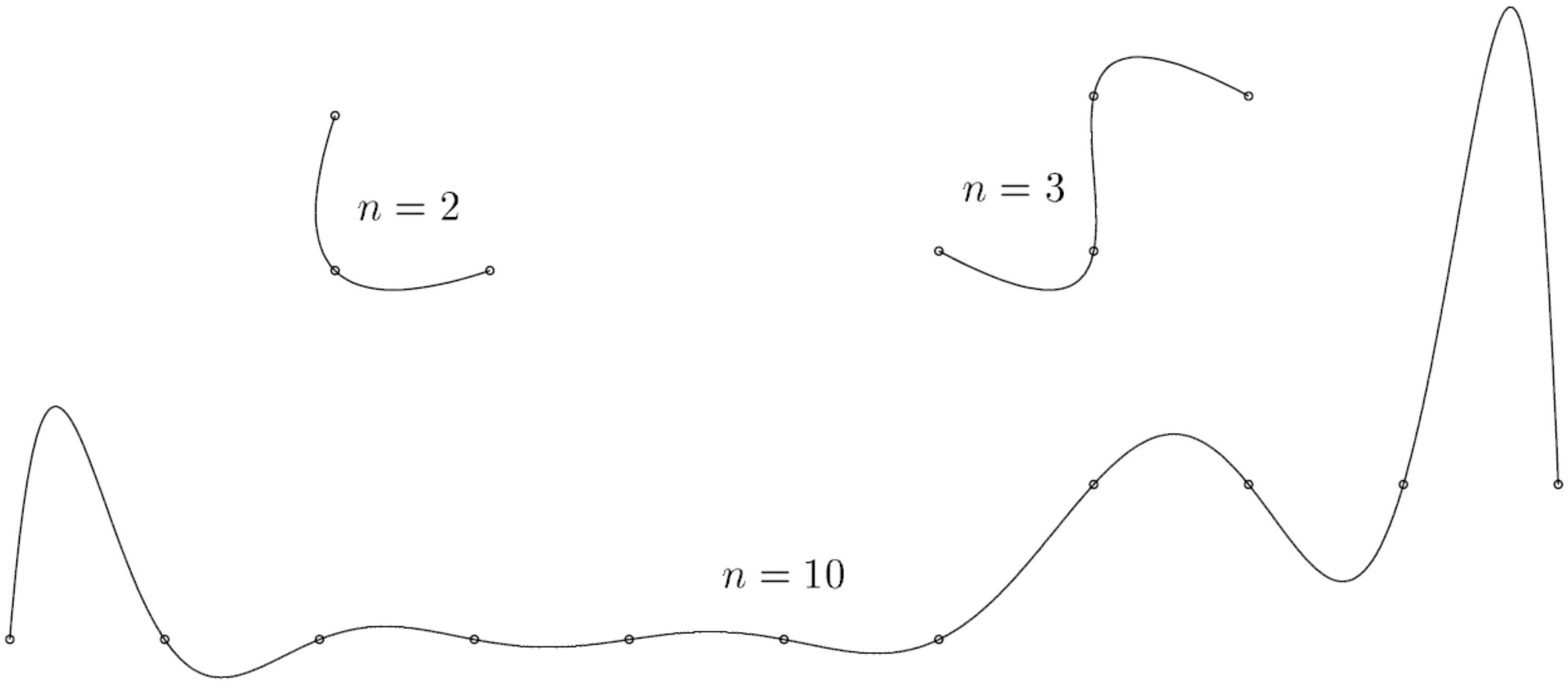
$$p_1(t) = \frac{(t-(-1))(t-1)}{(0-(-1))(0-1)} = -(t^2 - 1)$$

$$p_2(t) = \frac{(t-(-1))(t-0)}{(1-(-1))(1-0)} = \tfrac{1}{2}(t^2 + t)$$

$$P(t) = \tfrac{1}{2}(t^2 - t)(1,0) - (t^2 - 1)(0,0) + \tfrac{1}{2}(t^2 + t)(0,1) = (\tfrac{1}{2}(t^2 - t), \tfrac{1}{2}(t^2 + t))$$

# Examples

Note how the function for n = 10 is too wavy.
This is one of the disadvantages of working with high degree polynomials.



# Exercise

- Derive cubic Lagrange blending function and write an application that accepts 4 control points by mouse clicks and interpolates the points.
  1. Derive the n = 4 blending functions: $p_0$, $p_1$, $p_2$, $p_3$
  2. Given 4 control points A, B, C and D, construct a curve by generating the couples (x(t), y(t)) such that

$$x(t) = p_0(t)\, x_A + p_1(t)\, x_B + p_2(t)\, x_C + p_3(t)\, x_D$$

And

$$y(t) = p_0(t)\, y_A + p_1(t)\, y_B + p_2(t)\, y_C + p_3(t)\, y_D$$

# Exercise solution: derivation

$$p_0(t) = \frac{(t-0)(t-1)(t-2)}{(-1-0)(-1-1)(-1-2)} = \frac{-1}{6}(t^3 - 3t^2 + 2t)$$

$$p_1(t) = \frac{(t-(-1))(t-1)(t-2)}{(0-(-1))(0-1)(0-2)} = \frac{1}{2}(t^3 - 2t^2 - t + 2)$$

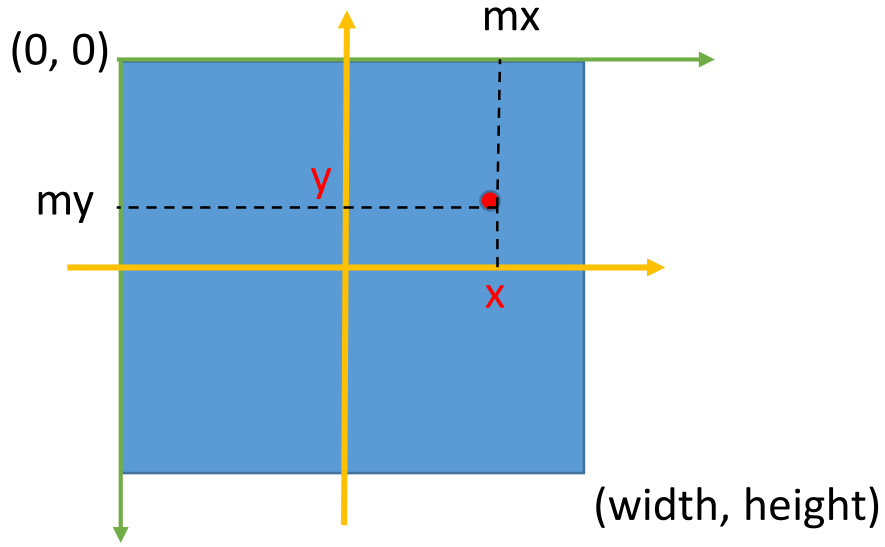
$$p_2(t) = \frac{(t-(-1))(t-0)(t-2)}{(1-(-1))(1-0)(1-2)} = \frac{-1}{2}(t^3 - t^2 - 2t)$$

$$p_3(t) = \frac{(t-(-1))(t-0)(t-1)}{(2-(-1))(2-0)(2-1)} = \frac{1}{6}(t^3 - t)$$

# Convert from screen space to world space

- From [0, width] to [-1, 1]
  - Divide by width (width – 0) => mx/width
  - Multiply by 2 (1 – (-1)) => (mx/width) * 2
- Translate from (-1, 1) to (0 , 0)
  - Subtract by 1

  ⇒(mx/width) * 2 – 1
- Multiply the y coordinate by -1 to invert the direction of the axis



# User input using freeglut

```
void mousebutton(int button, int state, int mx, int my)
{
    if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
    {
        float x = mx * 2.0 / g_width - 1.0;
        float y = -1 * (mx * 2.0 / g_height - 1.0);
        curve.addCtrlPoint(x, y);
    }
}
```

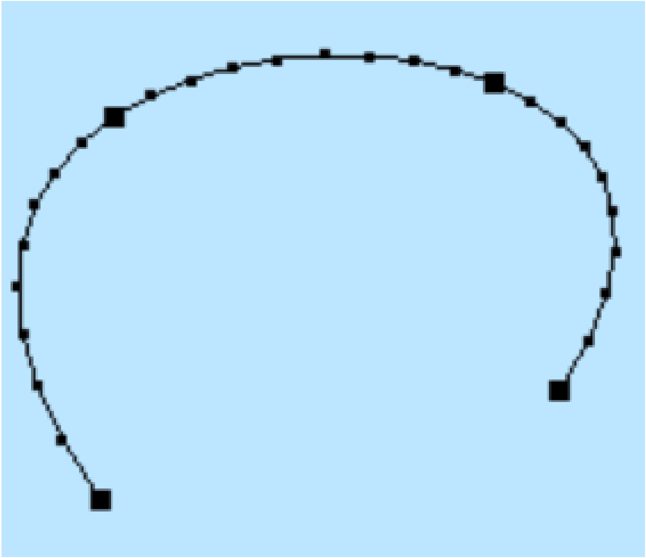
The mapping from the mouse position in the screen coordinates **(**[0, g_width] **,** [0, g_height]**)** to world coordinates **(**[-1, 1] **,** [-1,1]**)**

```
void initGlutState() {
    glutInitDisplayMode(GLUT_RGBA | GLUT_DOUBLE | GLUT_DEPTH);
    glutInitWindowSize(g_width, g_height);
    glutCreateWindow("Hello World");
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);

    glutMouseFunc(mousebutton);
}
```

# The structure of our data

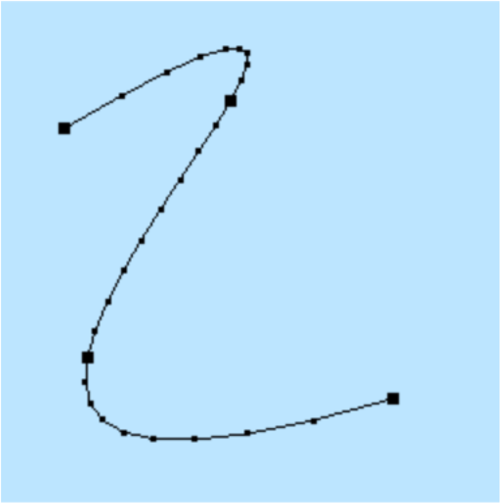
- An array for the coordinates of the 4 input points
- An array for the coordinates of the points
- Vertex buffer object handles for both arrays
- PLUS: three essential methods:
  - Initialize geometry
  - Initialize VBOs
  - Draw the object

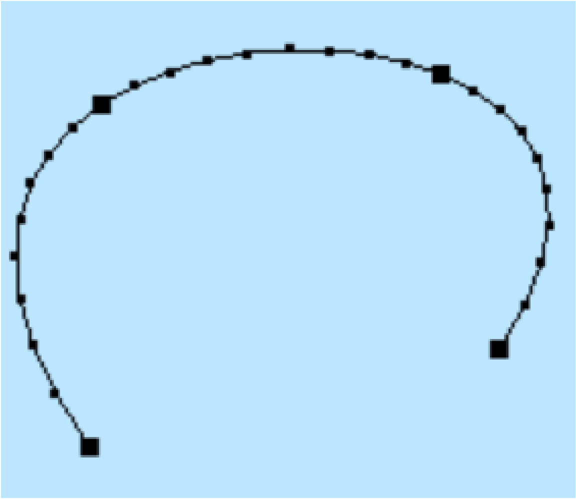


# Number of points on the curve

```
#define NUM_PTS_SEG 8
#define N (4 + NUM_PTS_SEG * 3)
```

**8:** number of point samples on the curve segment between the control points. (<u>this is our choice</u>)
**4:** number of control points
**3:** number of curve segments between control points





■ Control points entered by mouse clicks

▪ P(t) generated by the program

# The Class

```
class CubicLaGrange
{
    std::vector<vect2> ctrlPoints;
    GLfloat ctrlArray[4 * 2];
    GLfloat curvePoints[N * 2];
    GLuint curveVertBO, ctrlPtsBO;

    public:

    void addCtrlPoint(float x, float y);
    void initCurve();
    void initVBOs();
    void drawObj(GLuint h_aVertex);
};
```

```
class vect2 {
    public:
        float x, y;
        vect2(float x, float y) {
            this->x = x;
            this->y = y;
        }
};
```

* 2 is for the x and y coordinates

# Initialize geometry: initCurve

```
void CubicLaGrange::initCurve() {
    float dt = (2.0f - (-1.0f)) / (N-1);
    float t = -1;
    for (size_t i = 0; i < N; i++){
        float t3 = t * t * t;
        float t2 = t * t;
        float p[4] = { (-1.0f / 6) * (t3 - 3 * t2 + 2 * t),
                        (0.5f) * (t3 - 2 * t2 - t + 2),
                        (-0.5f) * (t3 - t2 - 2 * t),
                        (1.0f / 6) * (t3 - t) };
        curvePoints[i * 2] = curvePoints[i * 2 + 1] = 0;
        for (size_t j = 0; j < 4; j++){
            curvePoints[i * 2] += p[j] * ctrlPoints[j].x;
            curvePoints[i * 2 + 1] += p[j] * ctrlPoints[j].y;
        }
        t += dt;
    }
}
```

The *t* parameter for the 1st point on the curve is -1, the 2nd 0, the 3rd is 1 and the 4th is 2. The t values for all points on the curve in between belong to the interval [-1, 2].

N – 1 is the number of intervals between the points.

# Initialize geometry: (cont.)

```
void CubicLaGrange::initCurve() {
    float dt = (2.0f - (-1.0f)) / (N-1);
    float t = -1;
    for (size_t i = 0; i < N; i++){
        float t3 = t * t * t;
        float t2 = t * t;
        float p[4] = { (-1.0f / 6) * (t3 - 3 * t2 + 2
                       (0.5f) * (t3 - 2 * t2 - t + 
                       (-0.5f) * (t3 - t2 - 2 * t),
                       (1.0f / 6) * (t3 - t) };
        curvePoints[i * 2] = curvePoints[i * 2 + 1] =
        for (size_t j = 0; j < 4; j++){
            curvePoints[i * 2] += p[j] * ctrlPoints[
            curvePoints[i * 2 + 1] += p[j] * ctrlPoin
        }
        t += dt;
    }
}
```

Calculate the coefficients of the control points A, B, C, and D where:
$P(t) = c[0]*P_1 + c[1]*P_2 + c[2]*P_3 + c[3]*P_4$
And c[i] are computed according to the formulae that

$$p_0(t) = \frac{(t-0)(t-1)(t-2)}{(-1-0)(-1-1)(-1-2)} = \frac{-1}{6}(t^3 - 3t^2 + 2t)$$

$$p_1(t) = \frac{(t-(-1))(t-1)(t-2)}{(0-(-1))(0-1)(0-2)} = \frac{1}{2}(t^3 - 2t^2 - t + 2)$$

$$p_2(t) = \frac{(t-(-1))(t-0)(t-2)}{(1-(-1))(1-0)(1-2)} = \frac{-1}{2}(t^3 - t^2 - 2t)$$

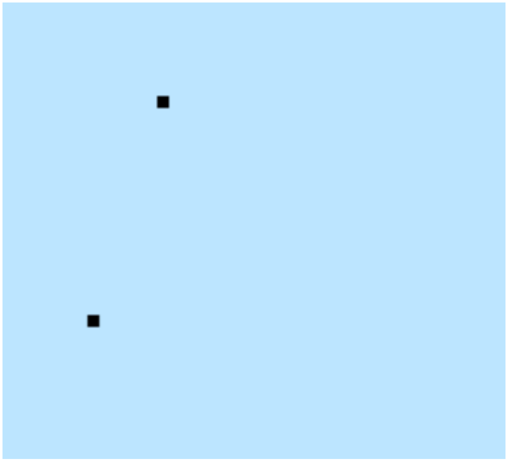
$$p_3(t) = \frac{(t-(-1))(t-0)(t-1)}{(2-(-1))(2-0)(2-1)} = \frac{1}{6}(t^3 - t)$$

curvePoints[2*i] is the x coordinate of the $i^{th}$ point and curvePoints [2*i +1] is its y coordinate.

# Initialize VBOs

```
void CubicLaGrange::initVBOs(void) {
    glGenBuffers(1, &curveVertBO);
    glBindBuffer(GL_ARRAY_BUFFER, curveVertBO);
    glBufferData(
    GL_ARRAY_BUFFER,
    N * 2 * sizeof(GLfloat),
    curvePoints,
    GL_STATIC_DRAW);
    //ready the array to draw the control points separately
    if (ctrlPoints.size() > 0) {
        glGenBuffers(1, &ctrlPtsBO);
        glBindBuffer(GL_ARRAY_BUFFER, ctrlPtsBO);
        glBufferData(
        GL_ARRAY_BUFFER,
        ctrlPoints.size() * 2 * sizeof(GLfloat),
        ctrlArray,
        GL_STATIC_DRAW);
    }
}
```

While the user is entering the control points, their number is not 4 yet, it is only `ctrlPoints.size()`



# Drawing the curve

```
void CubicLaGrange::drawObj(GLuint h_aVertex) {
	if (ctrlPoints.size() == 4)
	{
		glBindBuffer(GL_ARRAY_BUFFER, curveVertBO);
		safe_glVertexAttribPointer(h_aVertex, 2, GL_FLOAT, GL_FALS
		safe_glEnableVertexAttribArray(h_aVertex);
		glDrawArrays(GL_LINE_STRIP, 0, N);
		glPointSize(3);
		glDrawArrays(GL_POINTS, 0, N);
		safe_glDisableVertexAttribArray(h_aVertex);
	}
	if (ctrlPoints.size() > 0) {
		glBindBuffer(GL_ARRAY_BUFFER, ctrlPtsBO);
		safe_glVertexAttribPointer(h_aVertex, 2, GL_FLOAT, GL_FALSE, 0, 0);
		safe_glEnableVertexAttribArray(h_aVertex);
		glPointSize(6);
		glDrawArrays(GL_POINTS, 0, ctrlPoints.size());
		safe_glDisableVertexAttribArray(h_aVertex);
	}
}
```

If all 4 points have been entered (size ctrlPoinsts is 4):

draw curvePoints as a line strip

draw curvePoints as a points of size 3

If the control points array is not empty, draw them as points of size 6