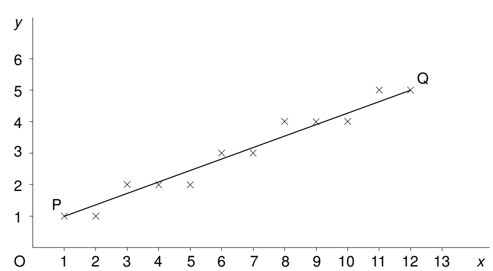
Chapter 4 Classic Algorithms

- Bresenham's Line Drawing
- Doubling Line-Drawing Speed
- Circles
- Cohen-Sutherland Line Clipping
- Sutherland–Hodgman Polygon Clipping
- Bézier Curves
- B-Spline Curve Fitting

Bresenham's Line Drawing

- A line-drawing (also called scan-conversion) algorithm computes the coordinates of the pixels that lie on or near an ideal, infinitely thin straight line



Bresenham's Line Drawing (cont'd)

- For lines $-1 \leq slope \leq 1$, exactly 1 pixel in each column.
- For lines with other slopes, exactly 1 pixel in each row.
- To draw a pixel in Java, we define a method

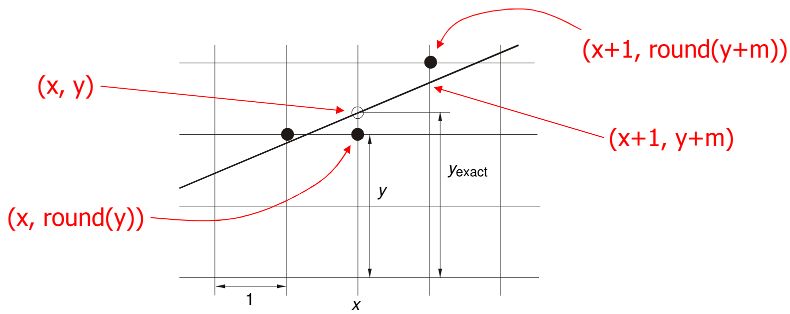
```
void putPixel(Graphics g, int x, int y)
{  g.drawLine(x, y, x, y);
}
```


Basic Incremental Algorithm

- Simplest approach:
  - Slope $m = \Delta y/\Delta x$
  - Increment x by 1 from leftmost point (if $-1 \leq m \leq 1$)
  - Use line equation $y_i = x_i m + B$ and round off y_i.
- But inefficient due to FP multiply, addition, and rounding

Basic Incremental Algorithm (cont'd)

- Let's optimize it:
  - $y_{i+1} = mx_{i+1} + B = m(xi + \Delta x) + B = y_i + m\Delta x$
  - So it's called incremental algorithm:
    - At each step, increment based on previous step



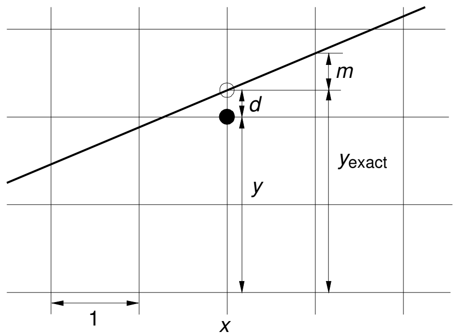
Basic Incremental Algorithm (cont'd)

- For $-1 \le m \le 1$:
  - int x;
  - float y, m = (float)(yQ - yP)/(float)(xQ - xP);
  - for (x= xP; x<=xQ; x++) {
  - putPixel(g, x, Math.round(y));
  - y = y + m; }
- Because of rounding, error of inaccuracy is
  - $-0.5 < y_{exact} - y \le 0.5$
- If $|m| > 1$, reverse the roles of x and y:
  - $y_{i+1} = y_i + 1$, $x_{i+1} = x_i + 1/m$
- Need to consider special cases of horizontal, vertical, and diagonal lines
- Major drawback: one of x and y is float, so is m, plus rounding.

Breshenham Line Algorithm

Let's improve the incremental algorithm

- To get rid of rounding operation, make *y* an integer



Breshenham Line Algorithm (cont'd)

- $d = y - round(y)$, so $-0.5 < d \leq 0.5$
- We separate *y*'s integer portion from its fraction portion
  - int x, y;
  - float d = 0, m = (float)(yQ - yP)/(float)(xQ - xP);
  - for (x= xP; x<=xQ; x++) {
  - putPixel(g, x, y); d = d +m;
  - if (d > 0.5) {y++; d--; }

Breshenham Line Algorithm (cont'd)

- To get rid of floating types *m* and *d,* we
  - double *d* to make it an integer, and
  - multiply *m* by *xQ – xP*
- We thus introduce a scaling factor
  - $C = 2 * (xQ - xP)$
  - (why can we do this?)
- So:
  - $M = cm = 2(yQ - yP)$
  - $D = cd$

Breshenham Line Algorithm (cont'd)

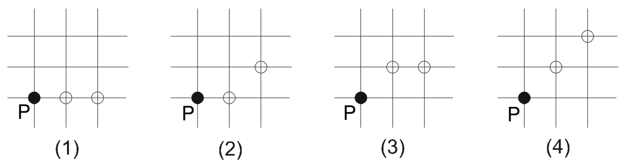
- We finally obtain a complete integer version of the algorithm (variables starting with lower case letters):

```
int x , y = yP, d = 0, dx = xQ - xP, c = 2 * dx, m = 2 * (yQ - yP);
for (x=xP; x<=xQ; x++) {
   putPixel(g, x, y);
   d += m;
   if (d >= dx) {y++; d -= c;}
}
```

- Now we can generalize the algorithm to handle all slopes and different orders of endpoints

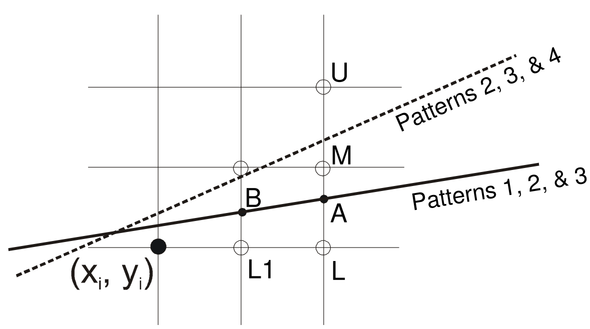
Doubling Line-Drawing Speed

- Bresenham algorithm:
  - Determines slope
  - Chooses 1 pixel between 2 based on *d*
- Double-step algorithm:
  - Halves the number of decisions by checking for next <u>TWO</u> pixels rather than 1



Double-Step Algorithm

- Patterns 1 and 4 cannot happen on the same line



Double-Step Algorithm (cont'd)

- For slope within [0, ½):
  - Pattern 1: 4dy < dx
  - Pattern 2: 4dy ≥ dx AND 2dy < dx
  - Pattern 3: 2dy ≥ dx
- Algorithm:
  - Set *d* initially at *4dy-dx*, check in each step
    - d < 0: Pattern 1 d = d+4dy
    - d ≥ 0, if d < 2dy Pattern 2 d = d + 4dy - 2dx
      d ≥ 2dy Pattern 3 d = d + 4dy - 2dx
  - x = x + 2

Circles

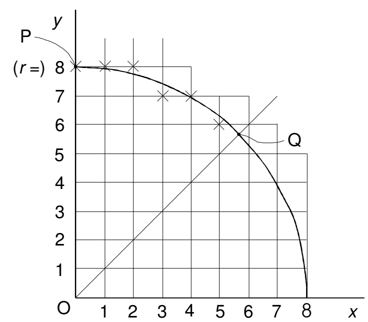
- How do we implement a circle-drawing method in Java
  - **drawCircle(Graphics g, int xC, int yC, int r)**
- A simplest way is
  - $x = xC + r \cos \varphi$
  - $y = yC + r \sin \varphi$

  where
  - $\varphi = i \times$ (i = 0, 1, 2, ..., $n - 1$)

  for some large value of *n*.
- But this method is time-consuming …

Circles (cont'd)

- According to circle formula
  - $x^2 + y^2 = r^2$
- Starting from *P*, to choose between *y* and *y-1*, we compare which of the following closer to *r*:
  - $x^2 + y^2$ and
  - $x^2 + (y - 1)^2$



Circles (cont'd)

- To avoid computing squares, use 3 new variables:
  - $u = (x + 1)2 – x2 = 2x + 1$
  - $v = y2 – (y – 1)2 = 2y – 1$
  - $E = x2 + y2 – r2$
- Starting at *P*
  - $x = 0$ and $y = r$, thus $u = 1$, $v = 2r – 1$ and $E = 0$
  - If $|E – v| < |E|$, then y-- which is the same as
  - $(E – v)2 < E2 \Rightarrow v(v – 2E) < 0$
- *v* is positive, thus we simply test
  - $v < 2E$

Circles (cont'd)

- Java code for the arc PQ:
  - void arc8(Graphics g, int r)
  - { int x = 0, y = r, u = 1, v = 2 * r - 1, e = 0;
  -    while (x <= y)
  -    { putPixel(g, x, y);
  -       x++; e += u; u += 2;
  -       if (v < 2 * e){y--; e -= v; v -= 2;}
  -    }
  - }

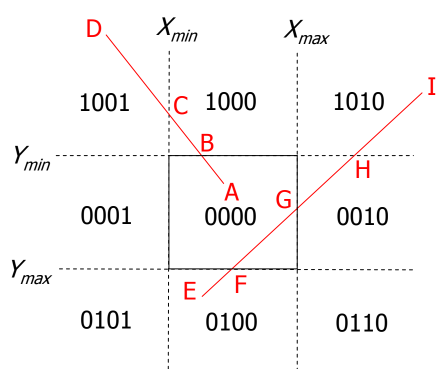
Line Clipping

- **Clipping endpoints**
  - **For a point (*x, y*) to be inside clip rectangle defined by x_{min}/x_{max} and y_{min}/y_{man}:**
    - $x_{min} \le x \le x_{max}$ ***AND*** $y_{min} \le y \le y_{max}$
- **Brute-Force Approach**
  - **If both endpoints inside clip rectangle, trivially accept**
  - **If one inside, one outside, compute intersection point**
  - **If both outside, compute intersection points and check whether they are interior**
- **Inefficient due to multiplication and division in computing intersections**

Cohen-Sutherland Algorithm

- **Based on "regions", more line segments could be trivially rejected**
- **Efficient for cases**
  - **Most line segments are inside clip rectangle**
  - **Most line segments are outside of clip rectangle**

Cohen-Sutherland Algorithm (cont'd)

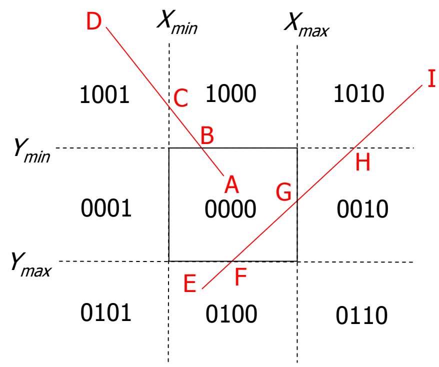


Outcode:

$Y_{max}-y$	$y-Y_{min}$	$X_{max}-x$	$x-X_{min}$

- **Check for a line**
  1. If $Outcode_A = Outcode_B = 0000$, trivially accept
  2. If $Outcode_A$ AND $Outcode_B \neq 0$, trivially reject
  3. Otherwise, start from outside endpoint and find intersection point, clip away outside segment, and replace outside endpoint with intersection point, go to (1)
- **Order of boundary from outside:**
  - **Top ⇒ bottom ⇒ right ⇒ left**

Cohen-Sutherland Algorithm (cont'd)

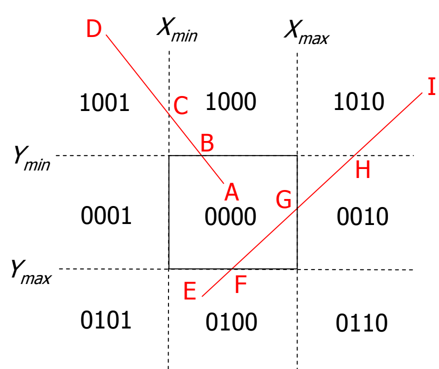


Outcode:

Y_{max}-y	y-Y_{min}	X_{max}-x	x-X_{min}

- **Consider line AD:**
  - Outcode_A = 0000, Outcode_D = 1001, neither accept nor accept
  - Choose D, use top edge to clip to AB
  - Find Outcode_B = 0000, according to (1), accept AB

Cohen-Sutherland Algorithm (cont'd)

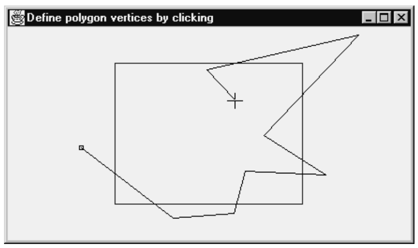


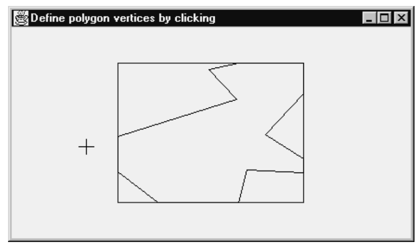
Outcode:

Y_{max}-y	y-Y_{min}	X_{max}-x	x-X_{min}

- **Consider line EI:**
  - Outcode_E = 0100, Outcode_I = 1010,
  - Start from E, clip to FI, neither (1) nor (2)
  - Since Outcode_F = 0000, choose I
  - Use top edge to clop to FH
  - Outcode_H = 0010, use right edge to clip to FG
  - According to (1), accept FG
- **Same result if start from I**

Polygon Clipping

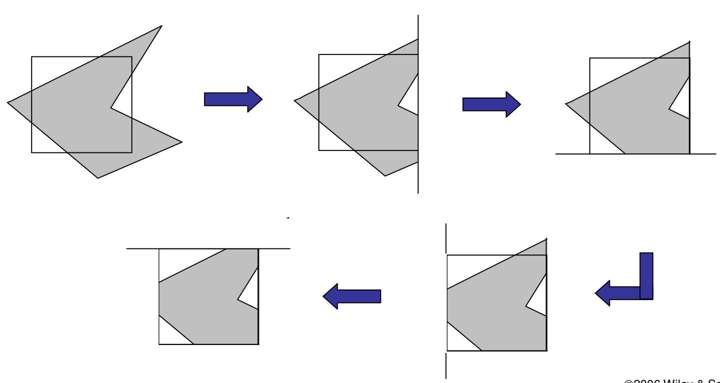




- **Sutherland-Hodgman Algorithm:** divide & conquer
  - General – a polygon (convex or concave) can be clipped against any convex clipping polygon

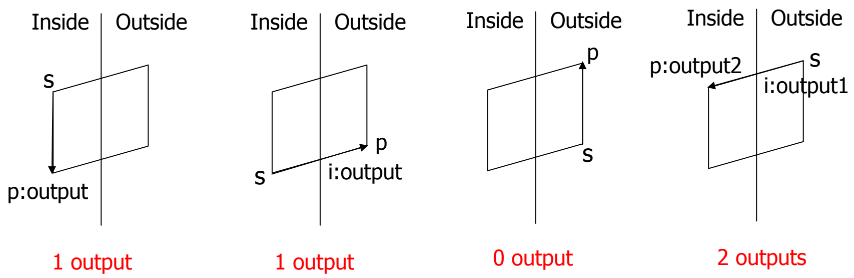
Sutherland-Hodgman Algorithm

- Clip the given polygon against one clip edge at a time

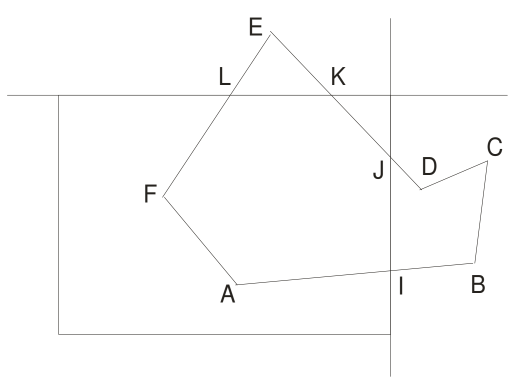


Sutherland-Hodgman Algorithm (cont'd)

- The algorithm clips every polygon edge against each clipping line
- Use an output list to store newly clipped polygon vertices
- With each polygon edge, 1 or 2 vertices are added to the output list



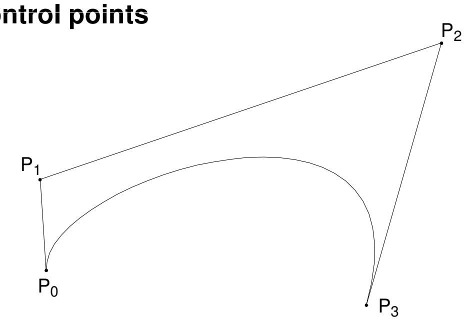
Sutherland-Hodgman Algorithm (cont'd)



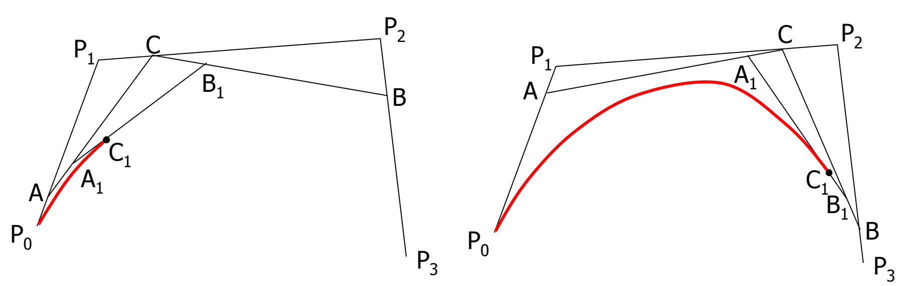
- Output vertices I, J, K, L, F, and A,

Bézier Curves

- 2 endpoints + 2 control points -> a curve segment
  - **P_0 and P_3 are endpoints**
  - **P_1 and P_2 are control points**



Bézier Curves (cont'd)



- C_1 is the point for drawing the curve

Bézier Curves (cont'd)

- Analytically
  - $A(t) = P_0 + t^*P_0P_1$ ($0 \le t \le 1$, t may be considered time)
  - $A(t) = P_0 + t(P_1 - P_0) = (1 - t)P_0 + t^*P_1$
- Similarly
  - $B(t) = (1 - t)P_2 + t^*P_3$
  - $C(t) = (1 - t)P_1 + t^*P_2$
  - $A_1(t) = (1 - t)A + t^*C$
  - $B_1(t) = (1 - t)C + t^*B$
  - $C_1(t) = (1 - t)A1 + t^*B_1$
- So
  - $C_1(t) = (1 - t)((1 - t)A + t^*C) + t^*(1 - t)C + t^*B)$
  - ,,,,,,
  - $C_1(t) = (1 - t)^3P_0 + 3(1 - t)^2t^*P_1 + 3t^2(1 - t)P_2 + t^{3*}P_3$

Bézier Curves (cont'd)

```
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));
   }
}
```

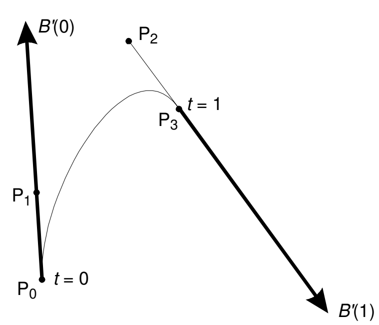
Bézier Curves (cont'd)

- Further manipulation:
  - $C_1(t) = (-P_0+3P_1-3P_2+P_3)t$
    $+ (P_0-2P_1+P_2)t$
    $- 3(P_1-P_0))t$
    $+ P_0$

```
void bezier2(Graphics g, Point2D[] p)
{  int n = 200;
   float dt = 1.0F/n,
   cx3 = -p[0].x + 3 * (p[1].x - p[2].x) + p[3].x,
   cy3 = -p[0].y + 3 * (p[1].y - p[2].y) + p[3].y,
   cx2 = 3 * (p[0].x - 2 * p[1].x + p[2].x),
   cy2 = 3 * (p[0].y - 2 * p[1].y + p[2].y),
   cx1 = 3 * (p[1].x - p[0].x),
   cy1 = 3 * (p[1].y - p[0].y),
   cx0 = p[0].x, cy0 = p[0].y,
   x = p[0].x, y = p[0].y, x0, y0;
   for (int i=1; i<=n; i++)
   {  float t = i * dt;
      x0 = x; y0 = y;
      x = ((cx3 * t + cx2) * t + cx1) * t + cx0;
      y = ((cy3 * t + cy2) * t + cy1) * t + cy0;
      g.drawLine(iX(x0), iY(y0), iX(x), iY(y));
   }
}
```


Bézier Curves (cont'd)

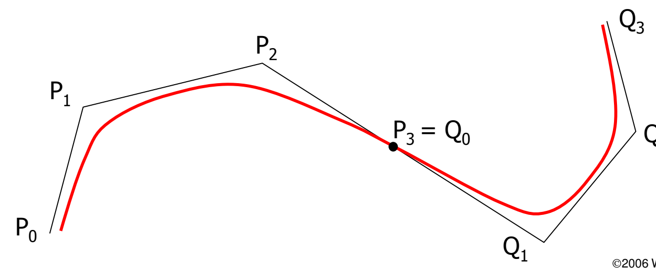
- $C_1(t)$ is the position of the curve at time t, its derivative C1'(t) is velocity:
  - $C_1'(t) = -3(t-1)^2P_0+3(3t-1)(t-1)P_1-3t(3t-2)P_2+3t^2P_3$



- So:
  - $C_1'(0) = 3(P_1 - P_0)$
  - $C_1'(1) = 3(P_3 - P_2)$

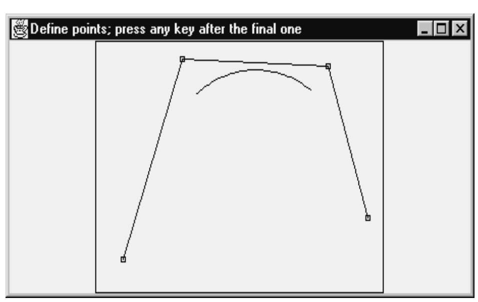
Bézier Curves (cont'd)

- When two Bézier curves *a* (P_0P_3) and *b* (Q_0Q_3) are combined, to make the connecting point smooth,
  - $C_{1a}'(1) = C_{1b}'(0)$
    i.e. the final velocity of curve *a* equals the initial velocity of curve *b*
  - The condition is guaranteed if P_3 ($=Q_0$) is the midpoint of line P_2Q_1



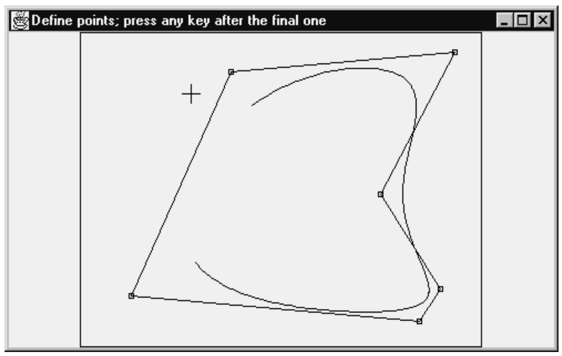
B-Spline Curve Fitting

- Number of control points = number of curve segments + 3



B-Spline Curve Fitting (cont'd)

- For example, following curve consists of 5 segments, 8 control points (left 2 repeated)
- Smooth connections between curve segments



B-Spline Curve Fitting (cont'd)

- The mathematics for B-splines (first 1st curve segment) can be expressed as ($0 \le t \le 1$):

$$B(t) = \frac{1}{6}\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}\begin{bmatrix} P_0 \\ P_1 \\ P_2 \\ P_3 \end{bmatrix}$$

$$B(t) = \frac{1}{6}\begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix}\begin{bmatrix} -P_0 + 3P_1 - 3P_2 + P_3 \\ 3P_0 - 6P_1 + 3P_2 \\ -3P_0 + 3P_2 \\ P_0 + 4P_1 + P_2 \end{bmatrix}$$

B-Spline Curve Fitting (cont'd)

$$B(t) = \frac{1}{6}(-P_0 + 3P_1 - 3P_2 + P_3)t^3 + \frac{1}{2}(P_0 - 2P_1 + P_2)t^2 + \frac{1}{2}(-P_0 + P_2)t + \frac{1}{6}(P_0 + 4P_1 + P_2)$$

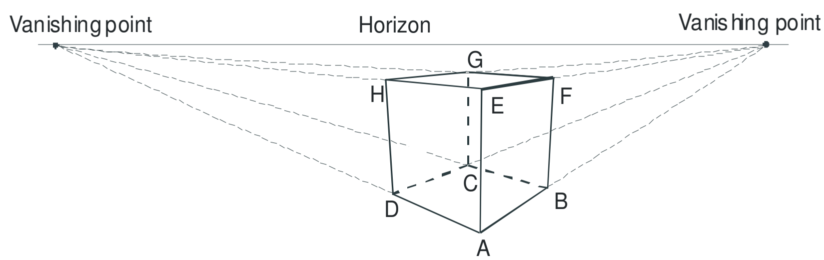
B-Spline Curve Fitting (cont'd)

```
void bspline(Graphics g, Point2D[] p)
  {  int m = 50, n = p.length;
     float xA, yA, xB, yB, xC, yC, xD, yD,
        a0, a1, a2, a3, b0, b1, b2, b3, x=0, y=0, x0, y0;
     boolean first = true;
     for (int i=1; i<n-2; i++)
     {  xA=p[i-1].x; xB=p[i].x; xC=p[i+1].x; xD=p[i+2].x;
        yA=p[i-1].y; yB=p[i].y; yC=p[i+1].y; yD=p[i+2].y;
        a3=(-xA+3*(xB-xC)+xD)/6; b3=(-yA+3*(yB-yC)+yD)/6;
        a2=(xA-2*xB+xC)/2;      b2=(yA-2*yB+yC)/2;
        a1=(xC-xA)/2;                  b1=(yC-yA)/2;
        a0=(xA+4*xB+xC)/6;      b0=(yA+4*yB+yC)/6;
        for (int j=0; j<=m; j++)
        {  x0 = x; y0 = y;
           float t = (float)j/(float)m;
           x = ((a3*t+a2)*t+a1)*t+a0; y = ((b3*t+b2)*t+b1)*t+b0;
           if (first) first = false;
           else  g.drawLine(iX(x0), iY(y0), iX(x), iY(y));
        }
     }
```

Chapter 5 Perspective

- Basic concepts
- Viewing Transformation
- Perspective Transformation
- A Cube Example
- Some Useful Classes
- Wire-Frame Drawings

Perspective Concepts



- Viewpoint
- Parallel (orthographic) projection
- Perspective projection

Perspective Concepts (cont'd)

World coordinates (*x*w, *y*w, *z*w) – 3D

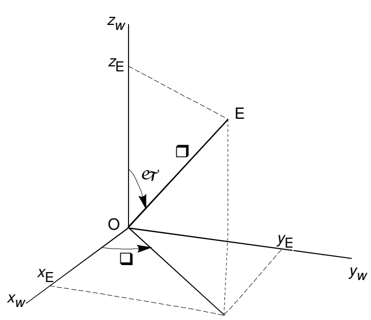
↓ *Viewing transformation*

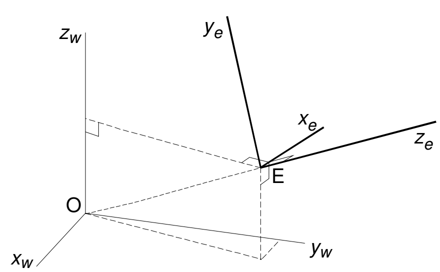
Eye coordinates (*x*e, *y*e, *z*e) – 3D

↓ *Perspective transformation*

Screen coordinates (*X*, *Y*) – 2D

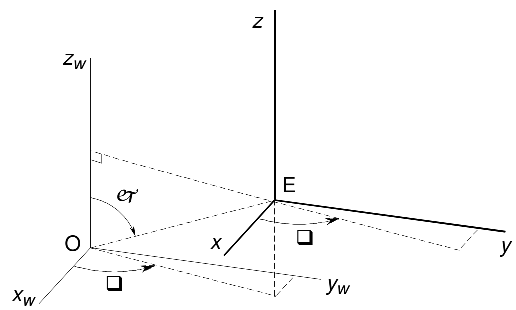
Viewing Transformation





Viewing Transformation (cont'd)

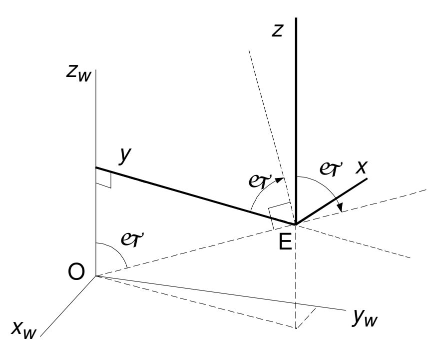
$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -x_E & -y_E & -z_E & 1 \end{bmatrix}$$



 43

Viewing Transformation (cont'd)

$$R_z = \begin{bmatrix} \cos(-\theta-90^\circ) & \sin(-\theta-90^\circ) & 0 & 0 \\ -\sin(-\theta-90^\circ) & \cos(-\theta-90^\circ) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} -\sin\theta & -\cos\theta & 0 & 0 \\ \cos\theta & -\sin\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



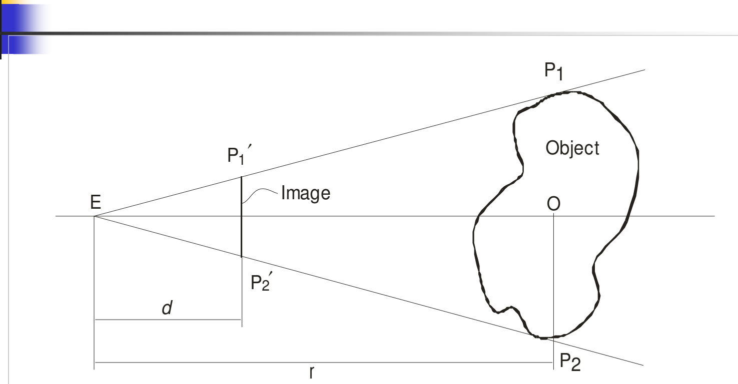
 44

Viewing Transformation (cont'd)

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(-\varphi) & \sin(-\varphi) & 0 \\ 0 & -\sin(-\varphi) & \cos(-\varphi) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\varphi & -\sin\varphi & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$V = TR_z R_x = \begin{bmatrix} -\sin\theta & -\cos\varphi\cos\theta & \sin\varphi\cos\theta & 0 \\ \cos\theta & -\cos\varphi\sin\theta & \sin\varphi\sin\theta & 0 \\ 0 & \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & -\rho & 1 \end{bmatrix}$$

Perspective Transformation



Changing r can change perspective.
It becomes parallel projection if r = ∞

Perspective Transformation (cont'd)

Due to similar triangles EQP′ and EOP:

$$\frac{P'Q}{EQ} = \frac{PR}{ER}$$

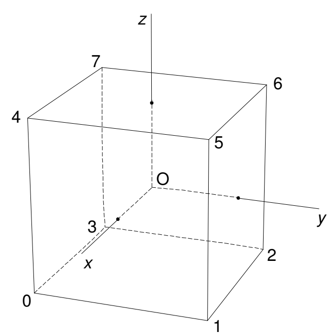
Applied to X-x_e and Y-y_e relationship:

$$\frac{X}{d} = \frac{x}{-z} \rightarrow \quad X = -d \cdot \frac{x}{z} \qquad Y = -d \cdot \frac{y}{z}$$

$$\frac{d}{\rho} = \frac{\text{image size}}{\text{object size}}$$

A Cube Example

- Draw a cube in perspective, given the viewing distance and object size.



A Cube Example (cont'd)

- Implementation
  - Class *Obj* contains 3D data and transformations
    - World coordinates for the cube – 3D
    - *ObjectSize* = SquareRoot(12)
    - Viewing distance *r* = 5 * *ObjectSize*
  - Prepare matrix elements
  - Transformations (viewing and perspective)
  - Draw cube (in paint)
    - Find center of world coordinate system
    - *d – r*ImageSize/ObjectSize*
    - Transformations
    - Draw cube edges according to screen coordinates

Some Useful Classes

- *Input*: for file input operations
- *Obj3D*: to store 3D objects
- *Tria*: to store triangles by their vertex numbers
- *Polygon3D*: to store 3D polygons
- *Canvas3D*: an abstract class to adapt the Java class *Canvas*
- *Fr3D*: a frame class for 3D programs

Wire-Frame Drawings

- Using all the previous classes, implement the following: