UNIVERSITÀ DEGLI STUDI DI BRESCIA · UNIBS.it

AA 2026/27

PAJC · PROGRAMMAZIONE AVANZATA IN JAVA E C

L05b

Trasformazioni di coordinate

Dal mondo ai pixel: il contesto che si trasforma, pan e zoom di una forma, `AffineTransform` applicata a un oggetto grafico



prof. Massimiliano Redolfi

Università degli Studi di Brescia · Ingegneria Informatica

Oggi

1. **Due sistemi di riferimento**: il mondo e i pixel, la conversione a mano
2. **Trasformare il contesto**: `translate`, `scale`, `rotate`, l'ordine conta, isolare
3. **Pan e zoom**: una forma, una vista, il mouse che parla in pixel
4. **La trasformazione sull'oggetto**: `AffineTransform` come valore, `getShape()`, disegnare e colpire con la stessa forma

01

Due sistemi di riferimento



Il mondo e i pixel

Un grafico, una lavagna, un sistema solare: il problema ragiona nelle sue unità, con l'origine dove gli serve e la y in alto. Il componente ragiona in **pixel**, origine in alto a sinistra, y in basso.

- disegnare vuol dire portare ogni punto del **mondo** nel suo pixel: una proporzione per asse

```
private double worldToScreenX(double x) {
    return SINISTRA + (x - xMin) / (xMax - xMin) * larghezza();
}
private double worldToScreenY(double y) {
    return SOPRA + (yMax - y) / (yMax - yMin) * altezza();     // y verso l'alto
}
private double screenToWorldX(int pixel) {                     // per il mouse
    return xMin + (double) (pixel - SINISTRA) / larghezza() * (xMax - xMin);
}
```

Convertire punto per punto

```
g2.draw(new Line2D.Double(worldToScreenX(x1), worldToScreenY(y1),
                          worldToScreenX(x2), worldToScreenY(y2)));
```

- sicuro e trasparente: per un grafico basta, e il tratto e le etichette restano in pixel
- diventa scomodo quando le forme sono **molte o complicate**: ogni vertice di ogni `Path2D` convertito a mano, le dimensioni scalate a parte, una rotazione con la trigonometria su ogni punto

> La seconda strada: non convertire le figure, ma chiedere al contesto di **trasformare il proprio sistema di riferimento**. Le figure restano scritte nel mondo.

02

Trasformare il contesto



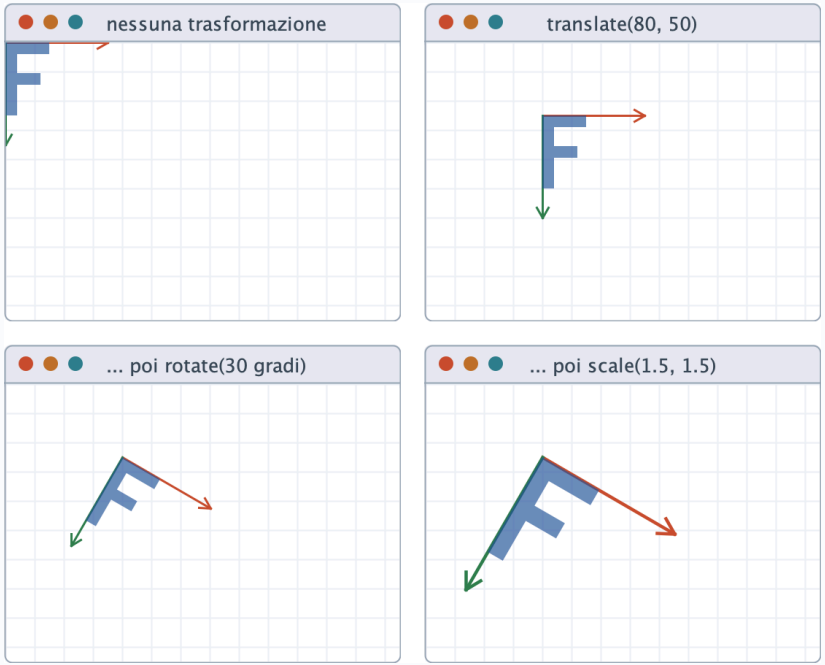
`Graphics2D` ha una trasformazione

Nel suo stato, accanto a colore e tratto, c'è la regola con cui le coordinate passate ai metodi di disegno diventano pixel. All'inizio è l'identità: $(10, 20)$ è il pixel $(10, 20)$.

- `translate(dx, dy)` sposta l'**origine** nel punto (d_x, d_y) del sistema corrente
- `scale(sx, sy)` cambia l'**unità**: una unità lungo x vale s_x unità di prima; un fattore negativo capovolge l'asse
- `rotate(theta)` ruota gli **assi** attorno all'origine corrente, in radianti; con la y in basso, un angolo positivo gira in senso orario

```
Non spostano le figure: spostano il sistema di riferimento.
Tutto cio' che si disegna dopo usa il sistema nuovo.
```

La stessa F, quattro sistemi

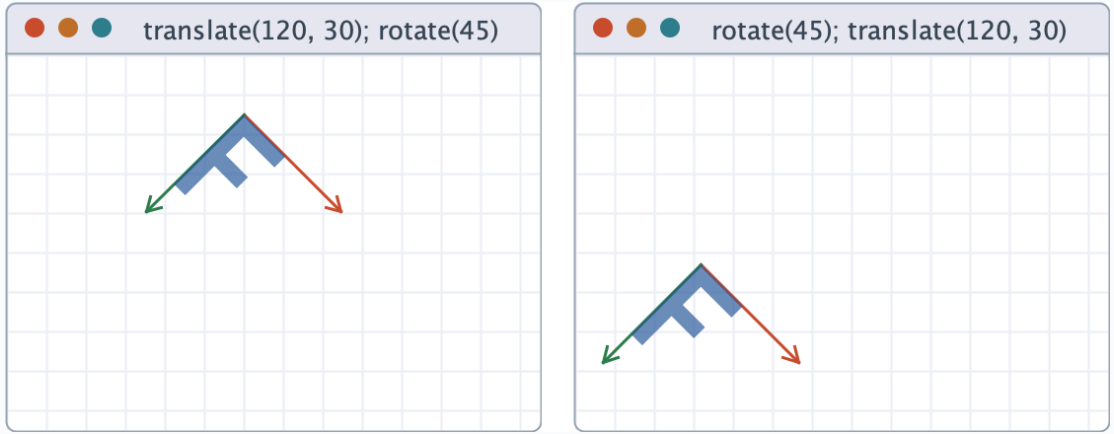


Stesse coordinate (asta da (0, 0) a (0, 50), bracci a destra); frecce: gli assi correnti, rosso x e verde y

Si compongono

- `translate(80, 50)` porta l'origine in $(80, 50)$, e la F con lei
- `rotate` aggiunto dopo ruota gli assi di 30 gradi attorno alla **nuova** origine: la F si inclina
- `scale(1.5, 1.5)` aggiunto dopo ancora allunga le unità degli assi **già ruotati**: più grande, e ancora inclinata
- nessuna coordinata della F è cambiata: ogni chiamata agisce sul sistema **lasciato dalle precedenti**

L’ordine conta



- a sinistra: l’origine va in $(120, 30)$ e la F **ruota attorno a quel punto**
- a destra: la rotazione avviene prima, attorno all’angolo del pannello, e i 120 e 30 si contano **lungo gli assi già ruotati**

Ricetta: prevedere un disegno

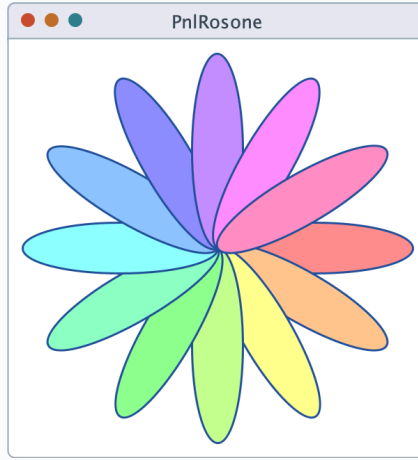
1. disegnare gli assi iniziali: origine in alto a sinistra, x a destra, y in basso
2. per ogni `translate(dx, dy)`: spostare l'origine di d_x e d_y lungo gli assi **correnti**
3. per ogni `rotate(a)`: ruotare gli assi attorno all'origine corrente, in senso orario se $a > 0$
4. per ogni `scale(sx, sy)`: annotare la nuova unità su ciascun asse; capovolgerlo se il fattore è negativo
5. disegnare la figura nel sistema finale, con le sue coordinate

> Lo stesso metodo serve a leggere un codice altrui e a rispondere a un quesito d'esame: una riga alla volta, con i due assi davanti.

Disegnare attorno all'origine

Invece di calcolare dove va ogni vertice, la forma si disegna nella posizione più comoda, centrata sull'origine, e si porta al suo posto trasformando il sistema.

```
g2.translate(getWidth() / 2.0, getHeight() / 2.0);
Ellipse2D petalo = new Ellipse2D.Double(0, -r * 0.13, r, r * 0.26);
for (int i = 0; i < N; i++) {
    g2.setColor(Color.getHSBColor(i / (float) N, 0.45f, 1f));
    g2.fill(petalo);
    g2.rotate(2 * Math.PI / N);        // il prossimo petalo
}
```



Il petalo non sa niente di angoli: è sempre lo stesso oggetto, la trigonometria la fa il contesto. Le rotazioni **si accumulano**, ed è ciò che si vuole qui.

Isolare: una copia per oggetto

Altre volte l'accumulo è un problema: più oggetti, ciascuno con posizione e rotazione sue, e fra uno e l'altro si deve **tornare al sistema di partenza**.

```
for (Pianeta p : pianeti) {
    Graphics2D gp = (Graphics2D) g2.create();   // parte dal sistema di g2
    gp.rotate(p.angolo());                      // l'orbita
    gp.translate(p.distanza(), 0);              // la posizione sull'orbita
    double r = p.raggio();
    gp.fill(new Ellipse2D.Double(-r, -r, 2 * r, 2 * r));   // centrato sull'origine
    gp.dispose();                               // g2 e' com'era
}
```

- l'alternativa: `getTransform()` prima e `setTransform(salvata)` dopo; ripristina solo la trasformazione

> `setTransform(new AffineTransform())` non riporta all'inizio del pannello ma all'identità della **superficie** di Swing, spesso la finestra intera: si ripristina solo una trasformazione salvata prima.

Dietro i tre metodi: `AffineTransform`

Un solo oggetto di `java.awt.geom`: una matrice 3×3 per qualunque combinazione di traslazioni, rotazioni, scale e inclinazioni.

Sul contesto	Come valore
`g2.translate / scale / rotate`	`t.translate / scale / rotate` sull'oggetto
`g2.getTransform()` la corrente	`new AffineTransform()` l'identità
`g2.transform(t)` la compone	`t.concatenate(u)`: prima `u`, poi `t`
`g2.setTransform(t)` la sostituisce	`t.transform(p, null)` porta un punto
	`t.inverseTransform(p, null)` lo riporta
	`t.createTransformedShape(s)` porta una forma

Due usi: **applicata al contesto** per guardare un mondo (pan e zoom), **applicata a una forma** per collocare un oggetto.

03

Pan e zoom



Una forma, una vista

La figura sta nelle **coordinate del mondo** e non cambia mai. La **vista** è la trasformazione che la porta sullo schermo: pan e zoom cambiano la vista, non la figura.

```
public class PnlVista extends JPanel {
    private final Shape figura = casa();           // nel mondo, attorno all'origine
    private double zoom = 1;                       // pixel per unita'
    private double panX, panY;                     // spostamento della vista, in pixel

    /** Dal mondo allo schermo: il centro del pannello, il pan, poi lo zoom. */
    public AffineTransform worldToScreen() {
        AffineTransform t = new AffineTransform();
        t.translate(getWidth() / 2.0 + panX, getHeight() / 2.0 + panY);
        t.scale(zoom, zoom);
        return t;
    }
```

`paintComponent`: una riga

```
@Override
protected void paintComponent(Graphics g) {
    super.paintComponent(g);
    Graphics2D g2 = (Graphics2D) g;                  // ... antialiasing ...
    g2.transform(worldToScreen());                   // da qui: coordinate del mondo
    disegnaGriglia(g2);
    g2.setColor(new Color(0xC94F2E));
    g2.fill(figura);                                 // la stessa figura, a ogni zoom
}
```

- `transform` **compone** la vista con la trasformazione corrente del contesto, che non è l'identità: Swing ha già portato l'origine nell'angolo del pannello
- tutto ciò che segue è scritto nel mondo: la griglia, la figura, domani cento figure

Il mouse parla in pixel

Il contesto va dal mondo ai pixel; il mouse arriva in pixel, e vuole la strada **inversa**.

```java
/** Dallo schermo al mondo: la trasformazione inversa. */
public Point2D screenToWorld(Point p) {
    try {
        return worldToScreen().inverseTransform(p, null);
    } catch (NoninvertibleTransformException e) {
        throw new IllegalStateException(e);         // con zoom > 0 non succede
    }
}
```

- il risultato è un `Point2D`, reale: a zoom 400% un pixel vale un quarto di unità
- ogni punto del mouse passa di qui **prima** di entrare nel mondo: dove è stato cliccato, che cosa c'è sotto il cursore

Pan: trascinare la vista

Lo spostamento è una proprietà della vista e si misura in pixel: non serve convertire niente.

```
MouseAdapter mouse = new MouseAdapter() {
    private Point ultimo;

    @Override public void mousePressed(MouseEvent e) { ultimo = e.getPoint(); }

    @Override public void mouseDragged(MouseEvent e) {
        panX += e.getX() - ultimo.x;                // quanto si e' mosso il mouse
        panY += e.getY() - ultimo.y;
        ultimo = e.getPoint();
        repaint();
    }
};
addMouseListener(mouse);
addMouseMotionListener(mouse);
```

Zoom attorno al mouse

Moltiplicare `zoom` e basta ingrandisce attorno al centro: la figura sotto il cursore scappa. Il punto del mondo sotto il mouse deve **restare sotto il mouse**.

```
addMouseWheelListener(e -> zoomSu(e.getPoint(), Math.pow(1.1, -e.getWheelRotation())));

private void zoomSu(Point pixel, double fattore) {
    Point2D fisso = screenToWorld(pixel);                    // 1. il punto sotto il mouse
    zoom *= fattore;                                         // 2. la nuova scala
    Point2D dopo = worldToScreen().transform(fisso, null);   // 3. dove finirebbe adesso
    panX += pixel.x - dopo.getX();                           // 4. si riporta sotto il mouse
    panY += pixel.y - dopo.getY();
    repaint();
}
```

Le due direzioni lavorano insieme: l'inversa per leggere il mouse, la diretta per prevedere dove finisce un punto.

Il tratto scala con il sistema

Con `scale(20, 20)` una linea di spessore 1 è larga **20 pixel**: la griglia diventa una scacchiera di bande.

```
private void disegnaGriglia(Graphics2D g2) {
    Point2D a = screenToWorld(new Point(0, 0));                    // gli angoli del pannello,
    Point2D b = screenToWorld(new Point(getWidth(), getHeight())); // nel mondo
    g2.setStroke(new BasicStroke((float) (1 / zoom)));             // un pixel, a ogni zoom
    double x0 = Math.floor(a.getX() / passo) * passo;              // il primo multiplo visibile
    for (double x = x0; x <= b.getX(); x += passo)
        g2.draw(new Line2D.Double(x, a.getY(), x, b.getY()));
    // ... e le righe orizzontali
}
```

- il tratto della griglia si **compensa** con $1/zoom$: appartiene allo schermo
- il tratto della figura cresce con lo zoom: appartiene al disegno, come la sua dimensione

Capovolgere la y

Per un mondo con la y verso l'alto, come in fisica, basta un fattore negativo nella vista: `t.scale(zoom, -zoom)`.

- gli assi escono come in un quaderno di matematica; un punto in $(100, 50)$ sta in alto a destra
- `rotate` cambia verso: nel sistema capovolto un angolo positivo gira in senso **antiorario**. Il verso di `rotate` è da x verso y, e dipende da dove punta la y
- il **testo** è un disegno come gli altri e viene capovolto: per un'etichetta si fa una copia del contesto, `translate` sul punto e `scale(1, -1)` di nuovo, e si scrive lì

> `getWidth() / 2` è una divisione intera: con una larghezza dispari l'origine cade mezzo pixel a sinistra del centro. Per le trasformazioni si scrive `getWidth() / 2.0`.

04

La trasformazione sull’oggetto



Non il contesto: la forma

`AffineTransform` è un valore, e si può applicare a una forma **senza toccare il contesto**: il risultato è una `Shape` nuova, già nel sistema di arrivo.

```
AffineTransform vista = worldToScreen();
Shape sulloSchermo = vista.createTransformedShape(figura);   // in pixel
g2.setStroke(new BasicStroke(2));                            // 2 pixel, a ogni zoom
g2.draw(sulloSchermo);
```

- il tratto resta in pixel senza compensare, il testo non si capovolge
- la forma trasformata è un **oggetto**: si può interrogare (`contains`, `getBounds`, `intersects`) e intersecare con un'`Area`
- il prezzo: una chiamata per forma, e una `Shape` nuova a ogni disegno

Un oggetto con posizione e rotta

Una navicella: una **sagoma** fissa, attorno all'origine con la punta verso $+x$, e uno stato che cambia nel tempo.

```
public class Nave {
    private static final Shape SAGOMA = sagoma();   // attorno all'origine, punta verso +x
    private double x, y, angolo;                    // posizione e rotta, nel mondo
    private double velocita;                        // unita' al secondo

    public void stepNext(double dt) {
        x += Math.cos(angolo) * velocita * dt;
        y += Math.sin(angolo) * velocita * dt;
    }
    public void ruota(double da) { angolo += da; }

    private static Shape sagoma() {                 // un triangolo con la coda
        Path2D p = new Path2D.Double();
        p.moveTo(20, 0); p.lineTo(-12, 10); p.lineTo(-6, 0); p.lineTo(-12, -10); p.closePath();
        return p;
    }
```

`getShape()`: la sagoma, dove sta la nave

```java
/** La sagoma portata nella posizione e nella rotta correnti, in coordinate del mondo.
public Shape getShape() {
    AffineTransform t = new AffineTransform();
    t.translate(x, y);                              // prima: dove sta
    t.rotate(angolo);                               // poi: come e' girata
    return t.createTransformedShape(SAGOMA);
}
```

- `translate` **poi** `rotate`: la rotazione avviene attorno al centro della nave, non all'origine del mondo
- la sagoma è centrata sull'origine proprio per questo: il centro di rotazione è il suo $(0, 0)$
- `getShape()` non sa niente di `Graphics2D`: è un metodo del **modello**

Disegnare e colpire con la stessa forma

```
// nella vista
for (Nave n : navi)
    g2.fill(n.getShape());

// nel modello: due oggetti si toccano?
public boolean collide(Nave altra) {
    Area a = new Area(getShape());
    a.intersect(new Area(altra.getShape()));
    return !a.isEmpty();
}

// nel controllore: l'utente ha cliccato sulla nave?
if (nave.getShape().contains(puntoNelMondo)) ...
```

Una sola geometria: ciò che si vede è ciò che si colpisce. Con il contesto trasformato non si può: il disegno finisce sui pixel e non torna indietro.

Le due trasformazioni si compongono




```
g2.transform(worldToScreen());          // la vista, sul contesto
g2.fill(nave.getShape());               // l'oggetto, sulla forma
```

- ogni livello trasforma nel sistema del livello sopra: la nave ruota attorno a sé, la vista ingrandisce attorno al mouse, e nessuno dei due sa dell'altro
- il mouse fa la strada inversa un livello alla volta: `screenToWorld` riporta il pixel nel mondo, e lì `getShape().contains` risponde

In pratica

	Conversione a mano	Contesto trasformato	Forma trasformata
quando	un grafico: pochi punti, griglia in pixel	un mondo da guardare: pan, zoom, molte figure	un oggetto da collocare e interrogare
tratto e testo	in pixel	scalano: si compensano	in pixel
collisioni, `contains`	no	no	sì: la forma è un oggetto
dove vive	nella vista	nella vista	nel modello (`getShape()`)

> Le tre strade convivono: la vista trasforma il contesto, ogni oggetto trasforma la propria sagoma, un'etichetta si converte a mano.

Errori frequenti

- `scale` **prima di** `translate`: anche la traslazione viene scalata
- `rotate` **senza isolare**: dal secondo oggetto in poi le rotazioni si sommano
- `setTransform(new AffineTransform())`: si perde la trasformazione di Swing
- **la griglia con il tratto predefinito** dopo uno `scale`: bande larghe quanto lo zoom
- **il mouse usato in pixel** dentro il mondo: la figura nasce fuori dallo schermo
- **la sagoma non centrata**: la nave ruota attorno a un suo angolo
- `rotate` **poi** `translate` **in** `getShape()`: la nave orbita attorno all'origine del mondo

Trasformazioni, in due righe

- il contesto ha una trasformazione: `translate`, `scale` e `rotate` spostano il **sistema**, si compongono nell'ordine, si isolano con una copia; una vista è un `AffineTransform` applicato al contesto e la sua **inversa** legge il mouse
- la stessa trasformazione, applicata a una forma con `createTransformedShape`, colloca un oggetto nel mondo: `getShape()` serve a disegnarlo e a colpirlo, con la stessa geometria