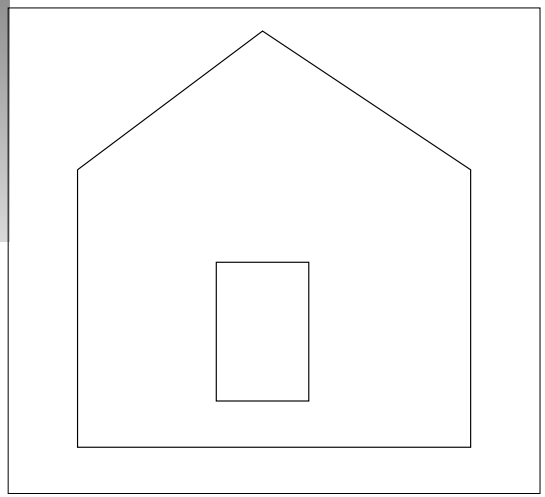
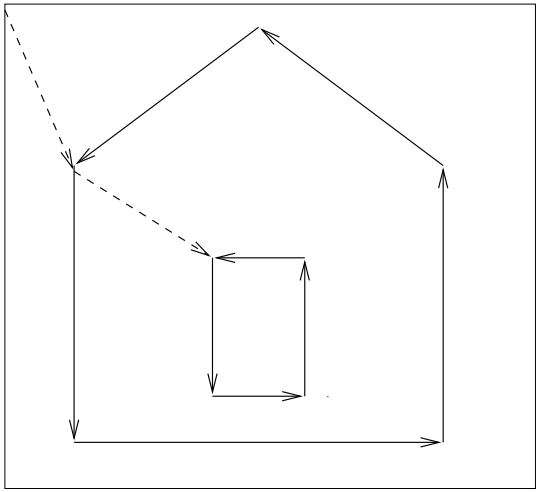
Representing an image

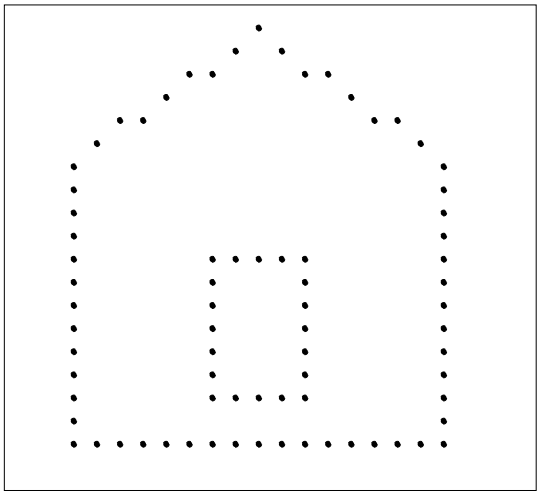


Original image



Vector graphics:

Representation by basic geometric objects(lines, circles, ellipses, cubic curves,. . .).



Raster graphics:

Representation in the form of a pixel matrix.

Raster graphics

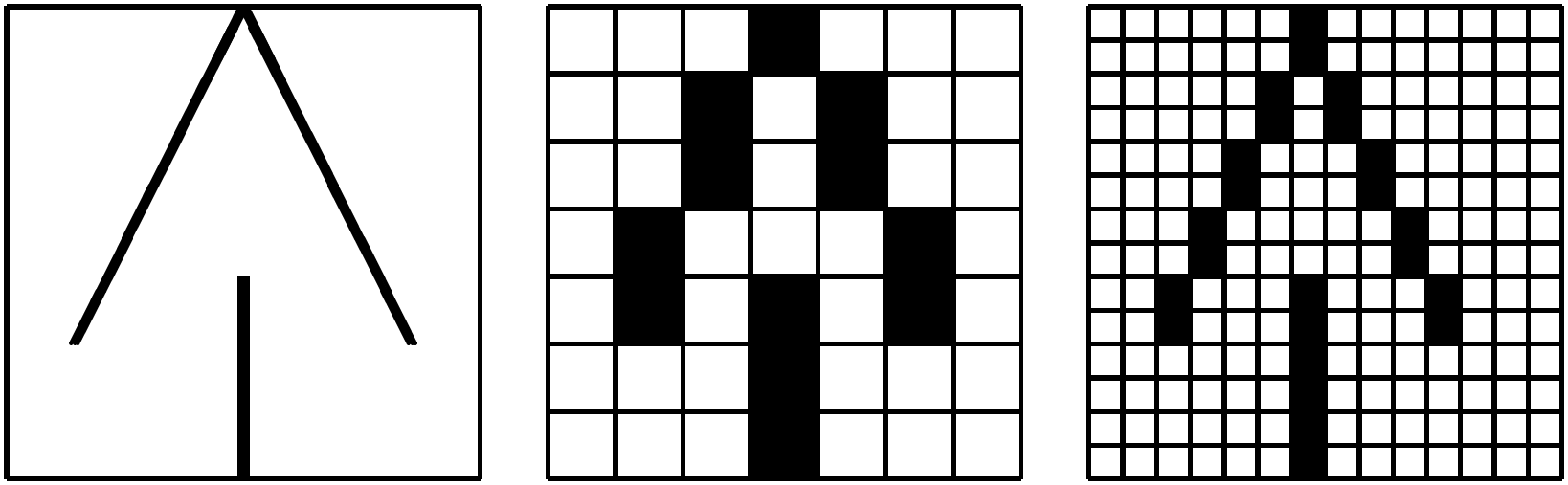
Raster graphics for cathode ray tube:

- The video controller reads the image buffer (row-wise from left to right and from right to left).
- At each pixel, the intensity of the ray is chosen according to the entry in the image buffer.
- $>$60Hz refresh rate $\rightarrow$ no flickering effects

Vector graphics

- scalable
- requires computations for display on a pixel-oriented medium (scan conversion)
- scan conversion can lead to **aliasing effects** (f.e. jagged edges) which occur in general when a a discrete sampling rate is used to measure a continuous signal.

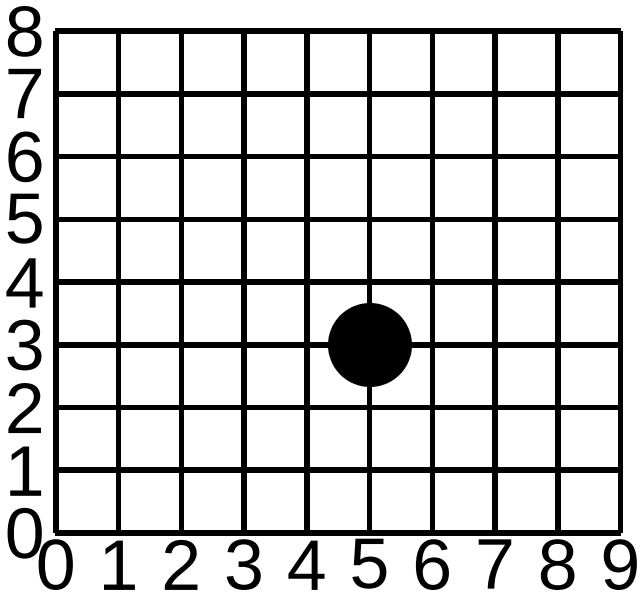
Raster graphics and scaling



The tip of an arrow displayed with two different resolutions.

Raster as a grid

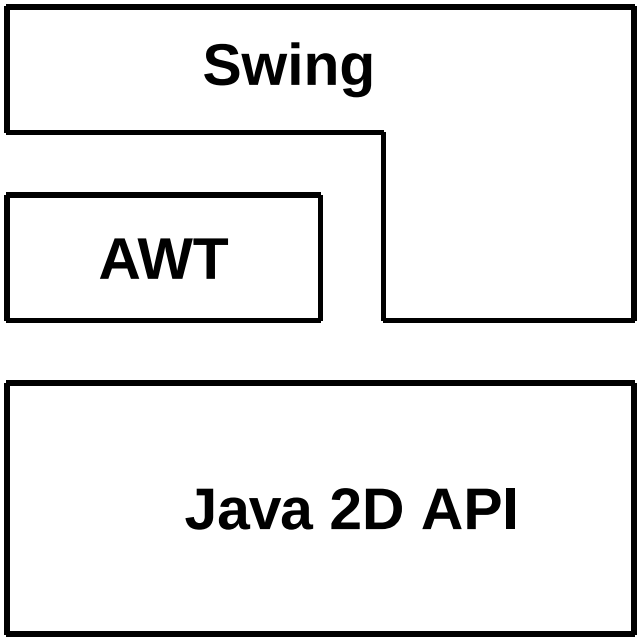
(Pixels lie on the grid points.)



Pixel with coordinates (5,3).

Sometimes the y-coordinates are counted from top to bottom (f.e. in Java).

Getting started with Java 2D



AWT components displayed on the screen have a `paint()` method with a `Graphics` object as argument.

Getting started with Java 2D

The class `Graphics2D` within Java 2D extends the class `Graphics`.

In order to exploit the options of Java 2D, the `Graphics` object must be casted into a `Graphics2D` object within the `paint()` method.

The window

```
import java.awt.*;

public class SimpleJava2DExample extends Frame
{
  SimpleJava2DExample()
  {
    addWindowListener(new MyFinishWindow());
  }

  public void paint(Graphics g)
  {
    Graphics2D g2d = (Graphics2D) g;
    g2d.drawString("Hello world!",30,50);
  }
```

The window

```
  public static void main(String[] argv)
  {
    SimpleJava2DExample f = new SimpleJava2DExample();
    f.setTitle("The first Java 2D program");
    f.setSize(250,80);
    f.setVisible(true);
  }
}
```

Window coordinates

- Coordinates in the upper left corner (0,0).
- Extension of the window to the right (here 250 pixels) and downwards (here 80 pixels).
- The y-axis points downwards.
- It is not possible to draw on the margins of the window.

Basic geometric objects

Points for the definition of other objects (f.e. a line connecting two points).

Lines, polylines or curves can be defined by two or more points.

Areas are usually bounded by **closed polylines** or **polygons**. Areas can be filled with a colour or a texture.

Basic geometric objects

Line (segment): Connecting line between two points.

Polyline: A sequence of line where the following line starts where the previous one ends.

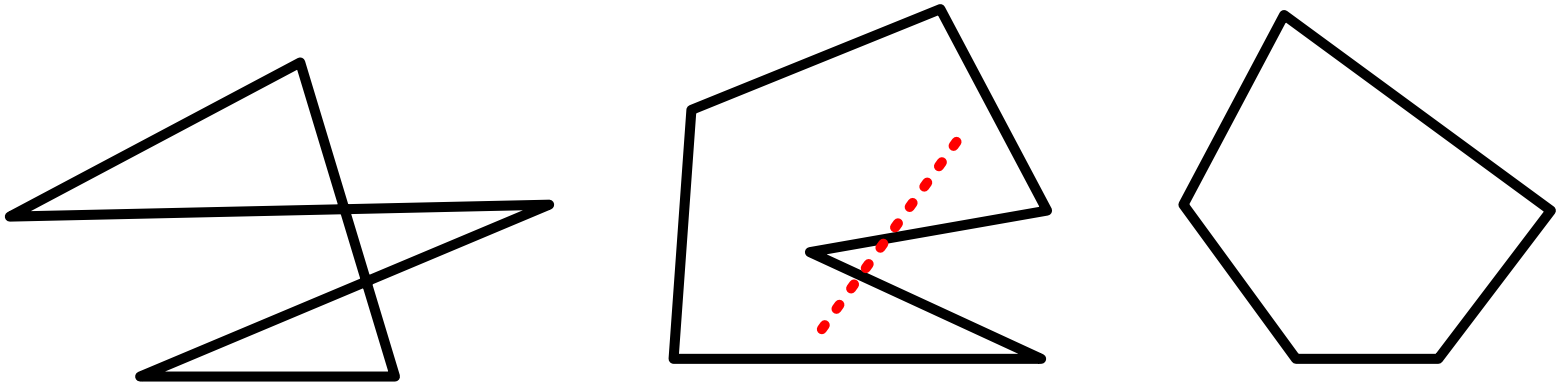
Polygon, closed polyline: The last line segment of a polyline ends where the first line segment started.

Polygons

Important additional properties of polygons:

Non-self-overlapping

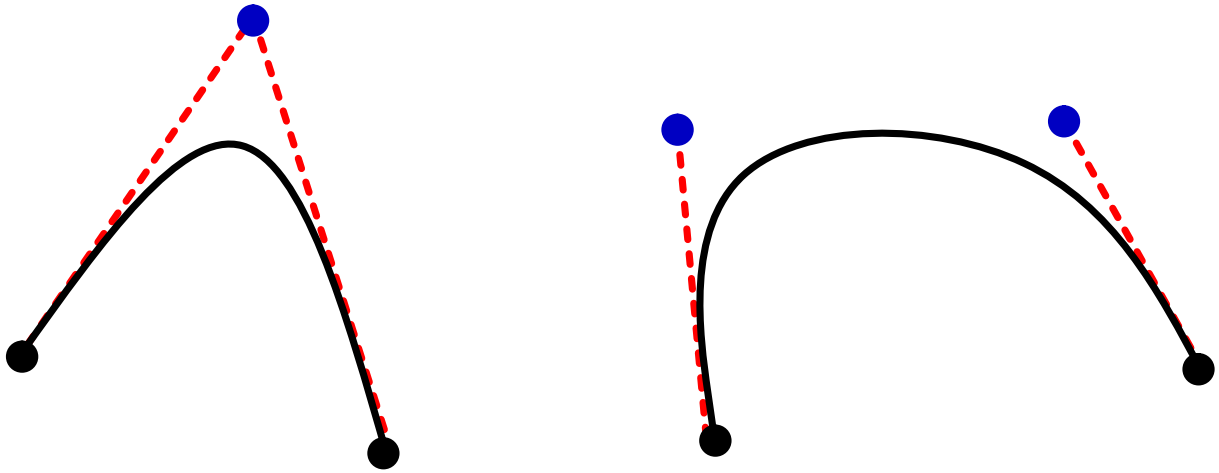
Convexity: Whenever two points are within the polygon the connecting line between these two points lies completely inside the polygon as well.
This definition of convexity applies also to arbitrary areas and 3D objects.



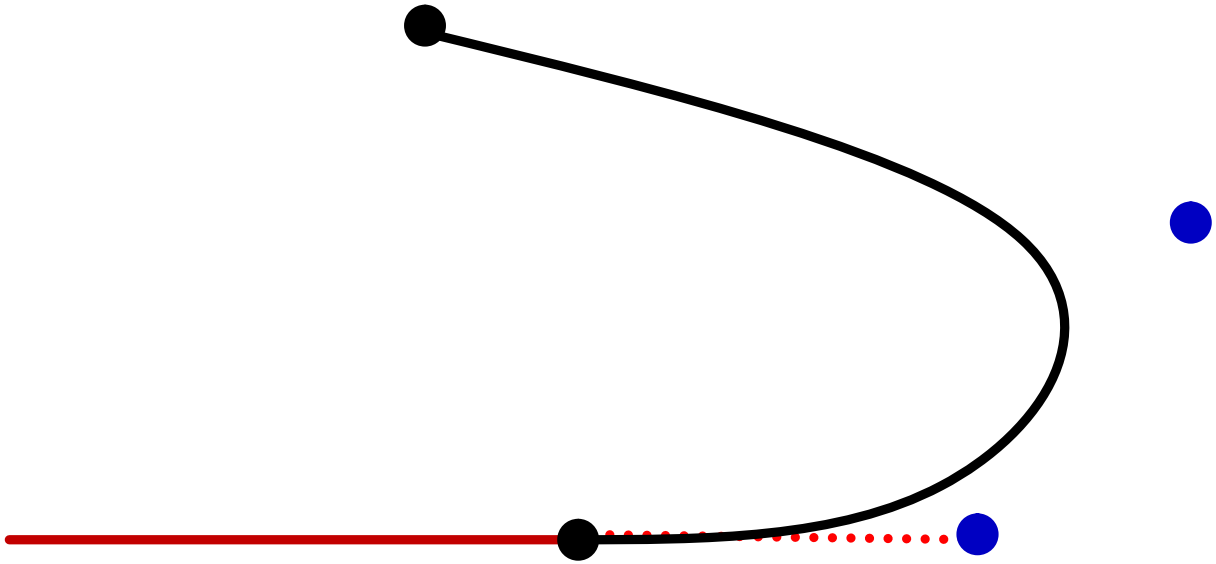
Parametric curves

quadratic curves: Two endpoints and one control point.

kubische Kurven: Two endpoints and two control points.



Parametric curves

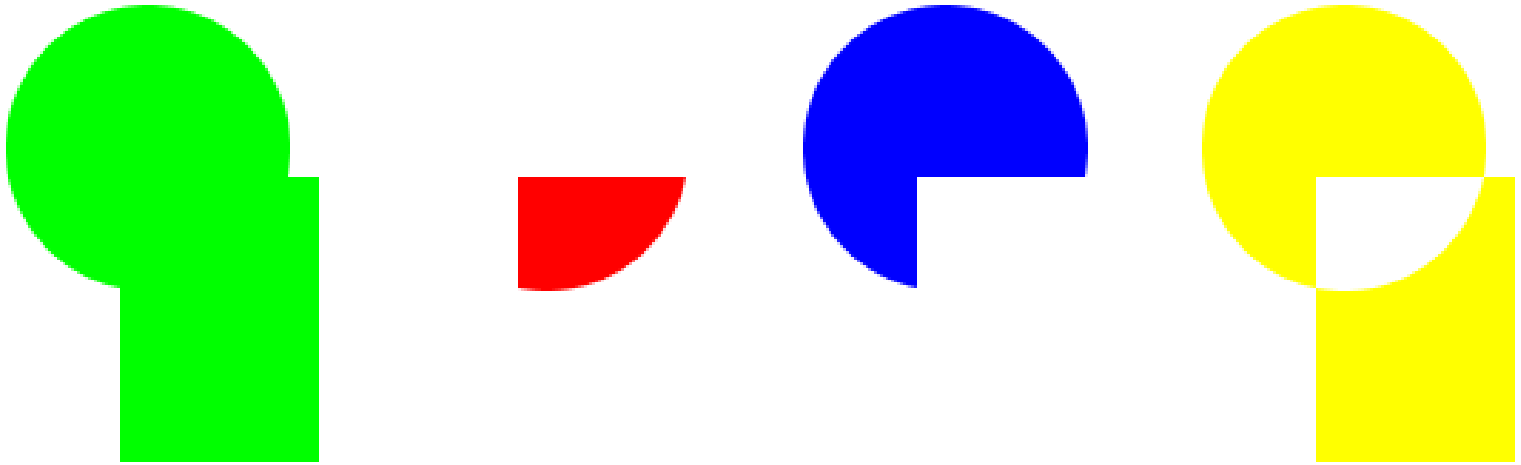


Avoiding sharp bends when attaching a cubic curve to a line.

Definition of areas

- Polygons or closed sequences of curves.
- Elementary geometric objects like circles, ellipses, (axes-parallel) rectangles.
- Modifying the shape of elementary geometric objects by geometric transformations.
- Application of set-theoretic operations like union, intersection, difference and symmetric difference to previously defined areas.

Set-theoretic operations



Union, intersection, difference and symmetric difference of a circle and a rectangle.

Geometric objects in Java 2D

- The abstract class `Shape` with its various subclasses allows the construction of various two-dimensional geometric objects.
- Vector graphics is used to define `Shape` objects, whose real-valued coordinates can either be given as `float`- or `double`-values.
- Shapes will not be drawn until the `draw` or the `fill` method is called with the corresponding `Shape` as argument in the form `graphics2d.draw(shape)` or `graphics2d.fill(shape)`, respectively.

Geometric objects in Java 2D

The abstract class `Point2D` is NOT a subclass of `Shape`.

Points cannot be drawn in Java.

They are only supposed to be used for the description of other objects.

Subclasses of `Point2D`: `Point2D.Float` and `Point2D.Double`.

Subclasses of Shape

Line (segment): Two endpoints are needed to define a line.

```
Line2D.Double line =
        new Line2D.Double(x1,y1,x2,y2);
```

Quadratic curve: Two endpoints and a control point are needed to define a quadratic curve.

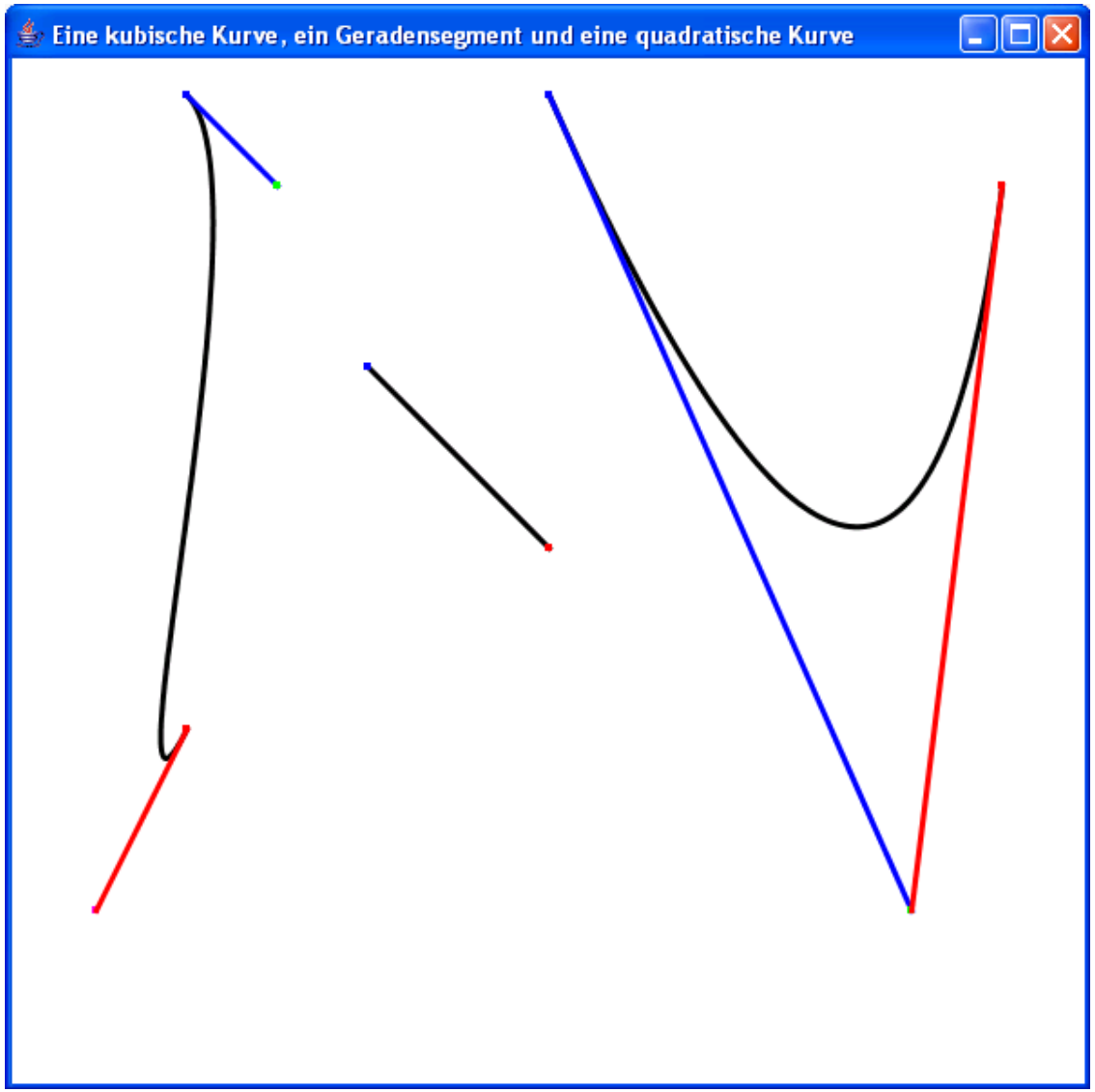
```
QuadCurve2D.Double qc =
  new QuadCurve2D.Double(x1,y1,
                         ctrlx,ctrly,
                         x2,y2);
```

Subclasses of Shape

cubic curve: Two endpoints and two control points are needed to define a cubic curve.

```
CubicCurve2D.Double cc =
new CubicCurve2D.Double(x1,y1,
                        ctrlx1,ctrly1,
                        ctrlx2,ctrly2,
                        x2,y2);
```

CurveDemo.java



Subclasses of Shape

General path: A `GeneralPath` is sequences of lines, quadratic and cubic curves.

```
GeneralPath gp = new GeneralPath();

gp.moveTo(50,50);
gp.lineTo(50,200);
gp.quadTo(150,500,250,200);
gp.curveTo(350,-100,150,150,100,100);
gp.lineTo(50,50);
```

Subclasses of Shape

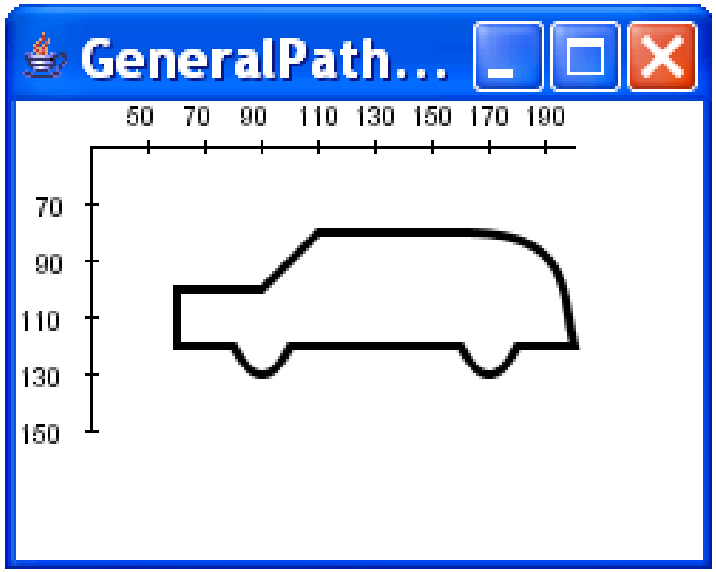
- A `GeneralPath` must always start with the method `moveTo`, defining the starting point of the general path.
- `lineTo` appends a line, starting from the (previous) last point of the `GeneralPath` to the specified endpoint.
- `quadTo` and `curveTo` append a quadratic and cubic curve, respectively, starting from the (previous) last point of the `GeneralPath` connecting it to the specified endpoint using the given control points.

GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

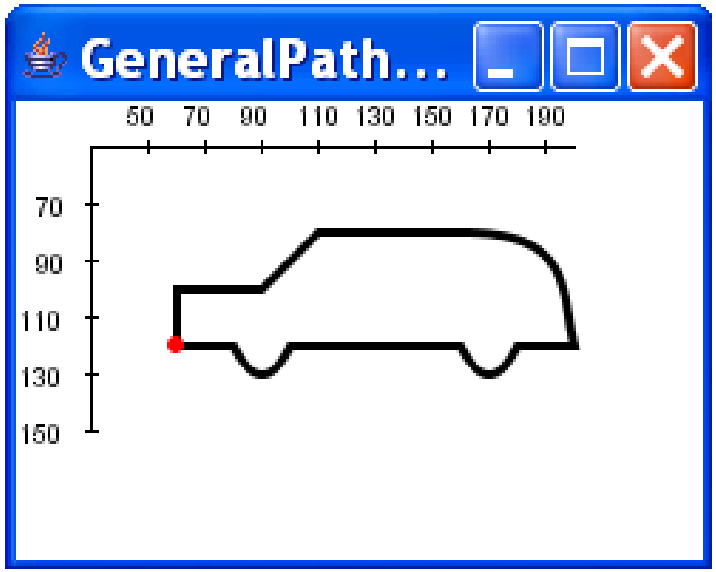


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

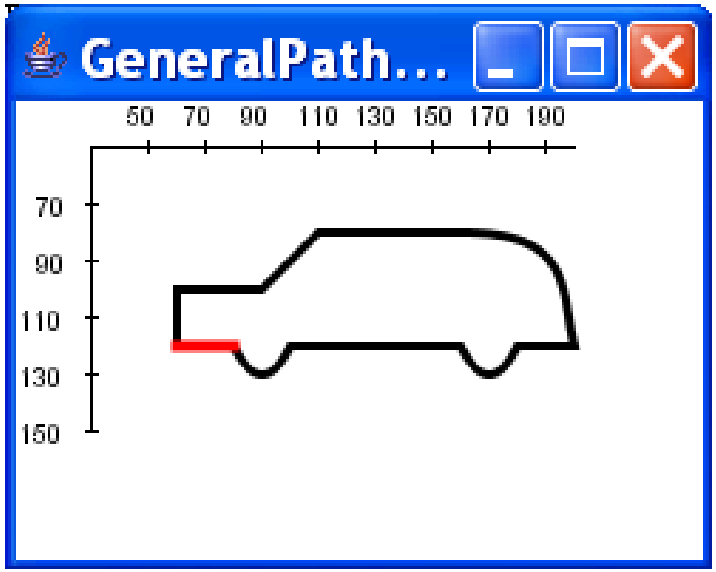


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

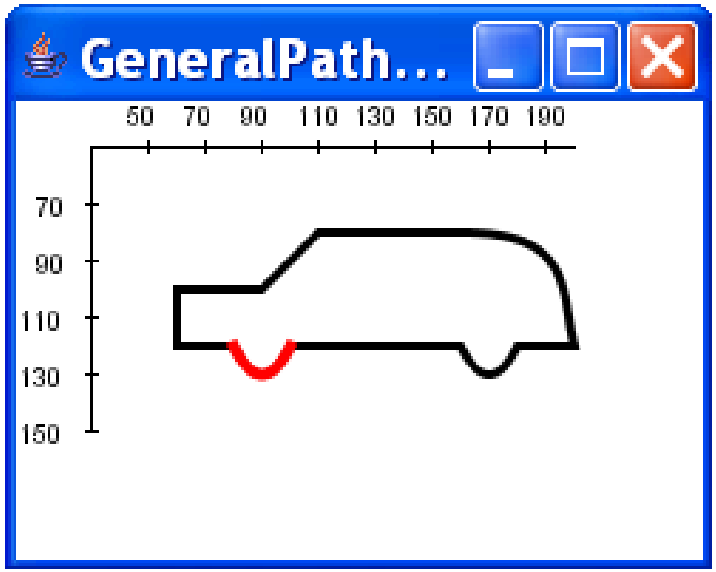


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

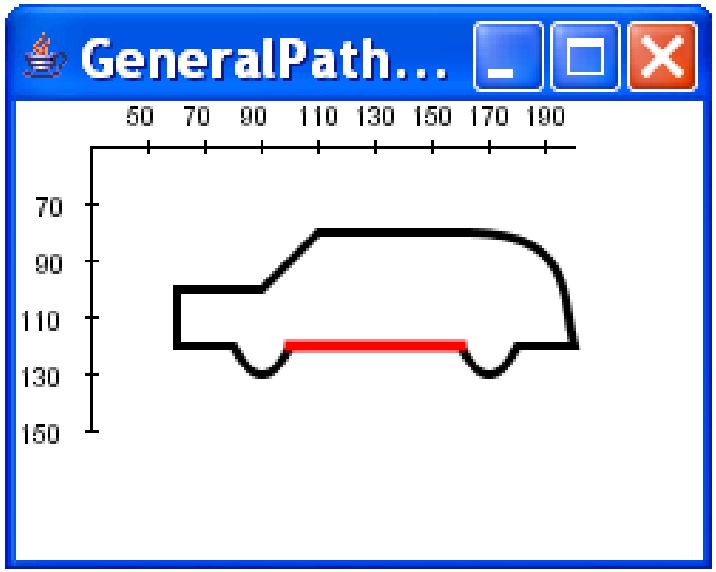


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

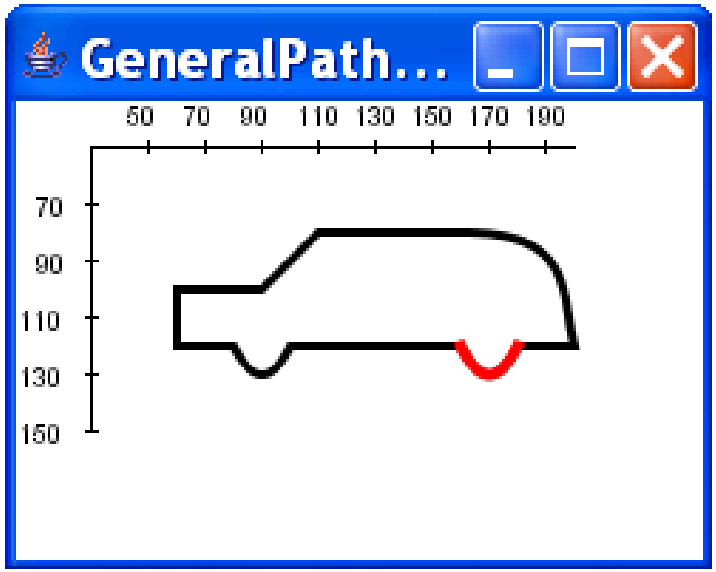


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

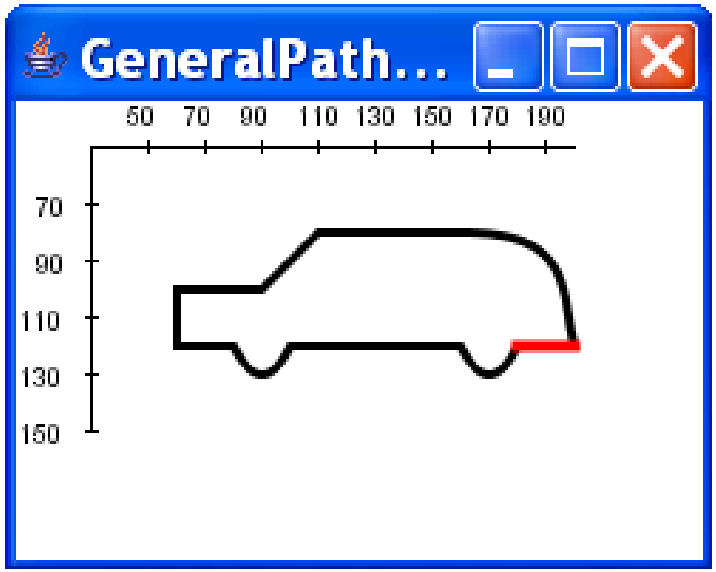


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

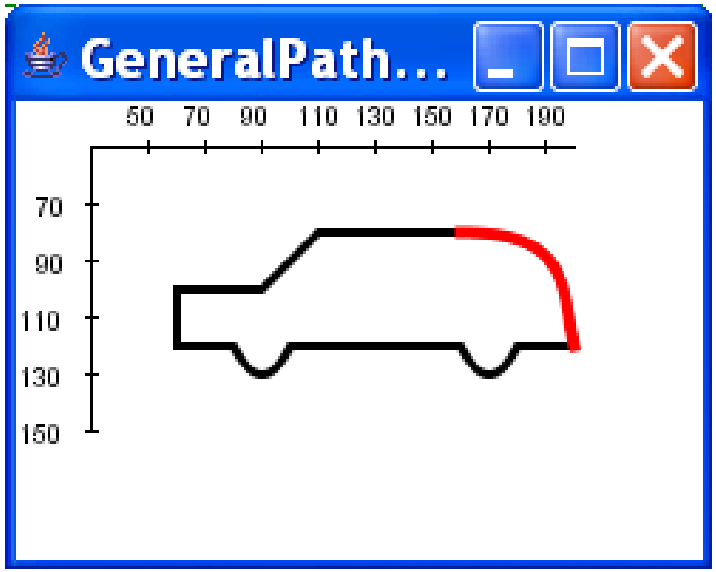


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

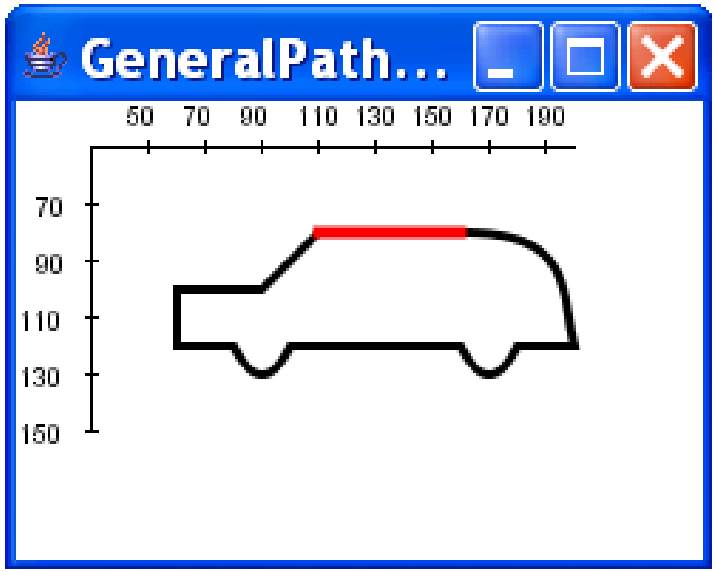


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

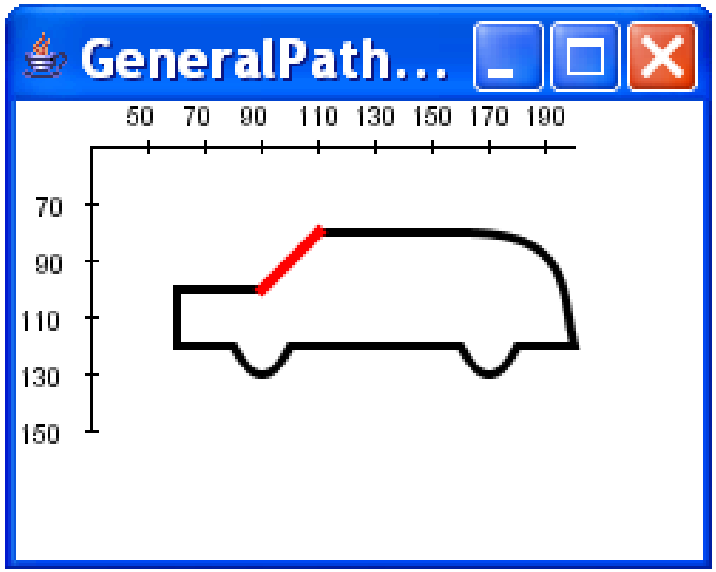


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

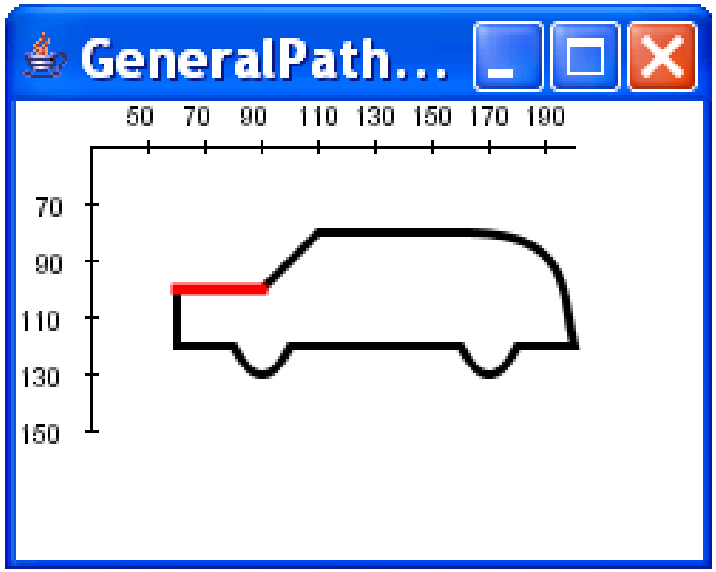


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```

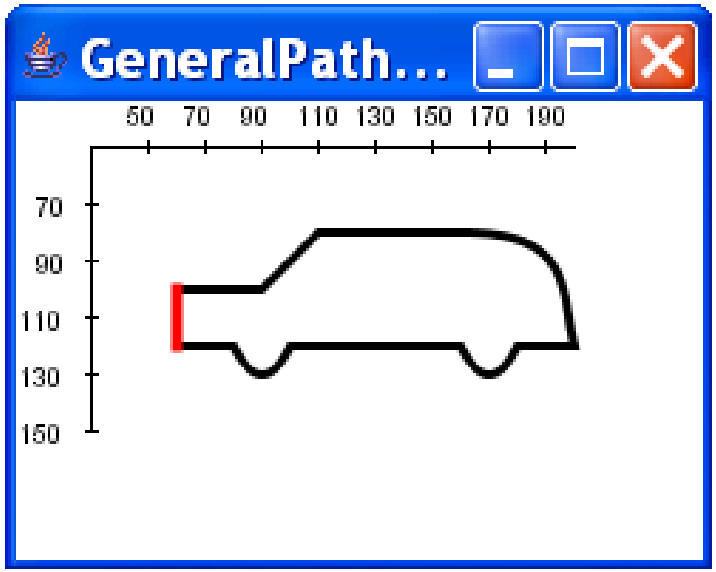


GeneralPathCar.java

```
GeneralPath gp = new GeneralPath();

gp.moveTo(60,120);
gp.lineTo(80,120);
gp.quadTo(90,140,100,120);
gp.lineTo(160,120);
gp.quadTo(170,140,180,120);
gp.lineTo(200,120);
gp.curveTo(195,100,
      200,80,160,80);
gp.lineTo(110,80);
gp.lineTo(90,100);
gp.lineTo(60,100);
gp.lineTo(60,120);

g2d.draw(gp);
```



Axes-parallel rectangles

Rectangle: By

```
Rectangle2D.Double r2d =
  new Rectangle2D.Double(
            x,y,width,height);
```

a rectangle is defined

- whose upper left corner has the coordinates `(x,y)`
- with width `width` and
- and height `height`.

Circles and ellipses

Circles and ellipses: by

```
Ellipse2D.Double elli =
new Ellipse2D.Double(x,y,width,height);
```

an axes-parallel ellipse is defined whose bounding rectangle is the `Rectangle2D` with the parameters `x,y,width,height`.

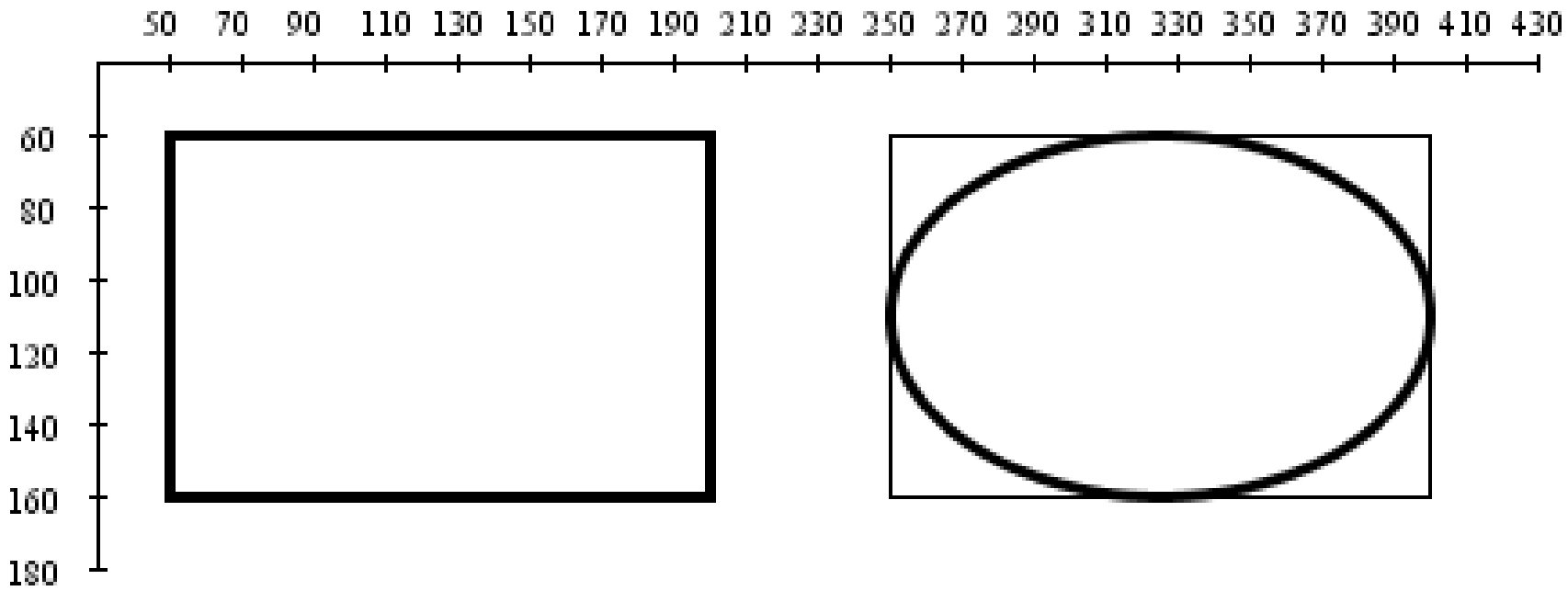
```
Ellipse2D.Double elli = new
Ellipse2D.Double(x-r,y-r,2*r,2*r);
```

defines a circle with centre `(x,y)` and radius `r`.

Example: Rectangle and ellipse

```
Rectangle2D.Double r2d =
  new Rectangle2D.Double(50,60,150,100);

Ellipse2D.Double elli =
  new Ellipse2D.Double(250,60,150,100);
```



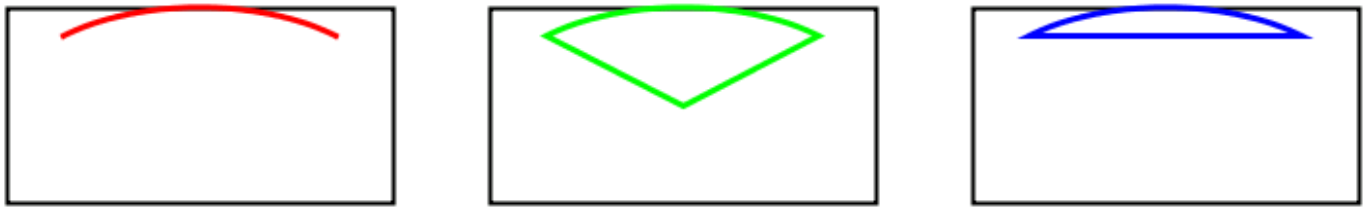
Ellipse arcs

Arcs (of ellipses) are defined by

```
Arc2D.Double arc = new Arc2D.Double(
                   rect,start,angle,type);
```

- `rect` is the bounding `Rectangle2D` of the corresponding ellipse.
- `start` is the angle (in degrees) where the arc is supposed to start relative to the bounding rectangle viewed as a square.
- `angle` is the opening angle of the arc, i.e., the arc extends from the start angle `start` to the angle `start + extend`. Again, relative to the bounding rectangle viewed as a square.
- `type` defines the type (OPEN, PIE, CHORD) of the arc.

ArcExampleColour.java



```
Rectangle2D.Double rect1 =
 new Rectangle2D.Double(50,50,200,100);
Arc2D.Double arcOpen =
 new Arc2D.Double(rect1,45,90,Arc2D.OPEN);
Rectangle2D.Double rect2 =
 new Rectangle2D.Double(300,50,200,100);
Arc2D.Double arcPie =
 new Arc2D.Double(rect2,45,90,Arc2D.PIE);
Rectangle2D.Double rect3 =
 new Rectangle2D.Double(550,50,200,100);
Arc2D.Double arcChord =
 new Arc2D.Double(rect3,45,90,Arc2D.CHORD);
```

`Area` *objects*

Area: The class `Area` allows to compute set-theoretic operations (f.e. for objects that were defined as `Shapes`.

- `Area a = new Area(s)` generates an `Area` object in form of the corresponding `Shape s`.

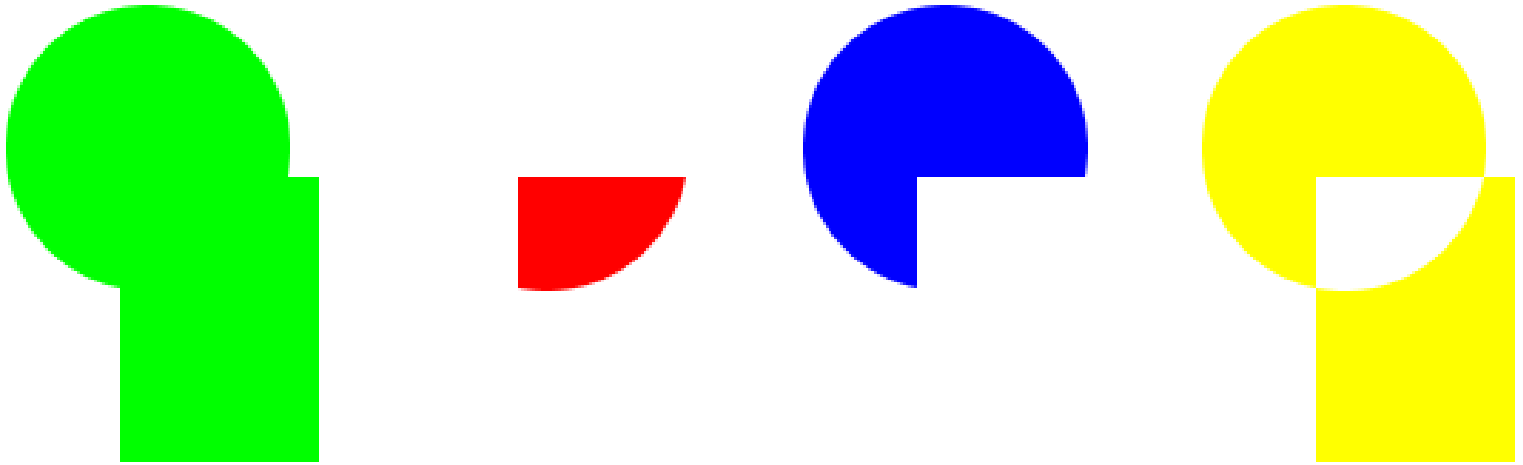
`a.add(b)`: Union of `Area a` and `Area b`.

`a.intersect(b)`: Intersection of `Area a` and `Area b`.

`a.subtract(b)`: `Area a` without `Area b`.

`a.exclusiveOr(b)`: Union of `Area a` and `Area b` without their intersection.

AreaExamplesColour.java



Geometric transformations (2D)

Scaling: Stretching/shrinking in the direction of the x- and the y-axis.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} s_x \cdot x \\ s_y \cdot y \end{pmatrix} = \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

Stretching in the direction of the x-axis $\Leftrightarrow |s_x| < 1$

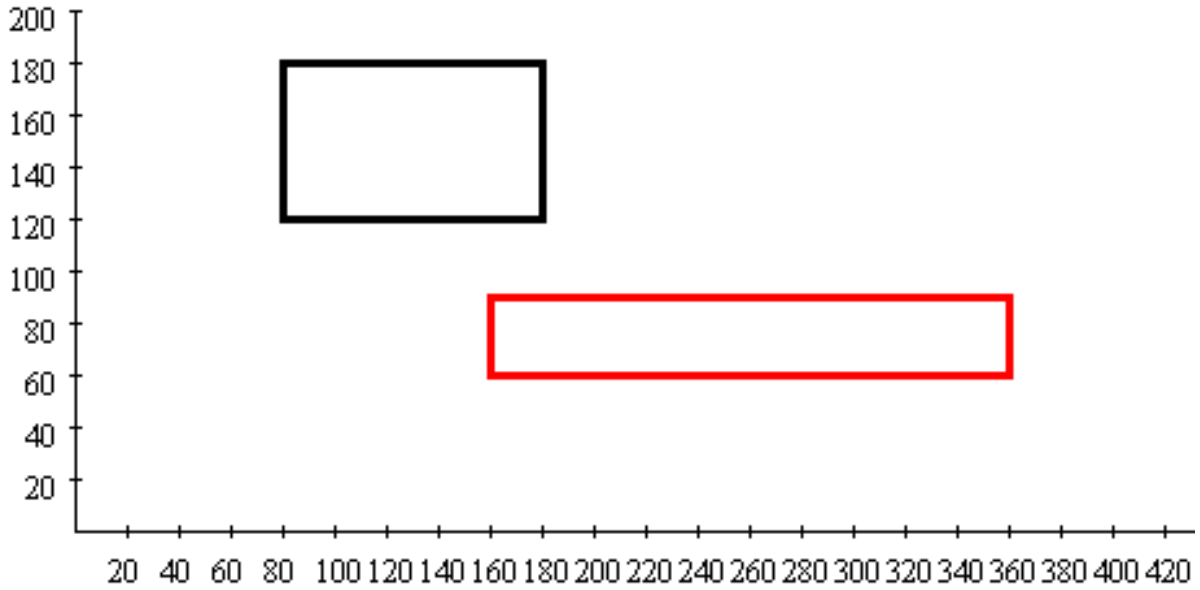
Shrinking in the direction of the x-axis $\Leftrightarrow |s_x| > 1$

Stretching in the direction of the y-axis $\Leftrightarrow |s_y| > 1$

Shrinking in the direction of the y-axis $\Leftrightarrow |s_y| < 1$

For negative values of s_x or s_y lead to an additional reflection w.r.t. the y- or the x-axis, respectively.

ScalingExampleColour.java



Scaling with $s_x = 2$, $s_y = 0.5$

Scaling is always carried out pointwise, i.e. w.r.t. the origin. Applying a scaling to an object that is not centred around the origin of the coordinate system will lead to a translation of the (centre of the) object in addition to the scaling.

Geometric transformations (2D)

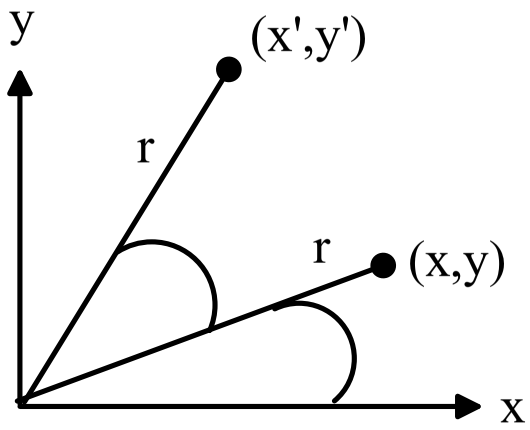
Rotation (anticlockwise) around the origin of the coordinate system through the angle θ.

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x \cdot \cos(\theta) - y \cdot \sin(\theta) \\ x \cdot \sin(\theta) + y \cdot \cos(\theta) \end{pmatrix}$$

$$= \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

In Java 2D the angle must be specified in radiant!

Derivation of the rotation matrix



Polar coordinates: $x = r \cdot \cos(\varphi), \quad y = r \cdot \sin(\varphi)$

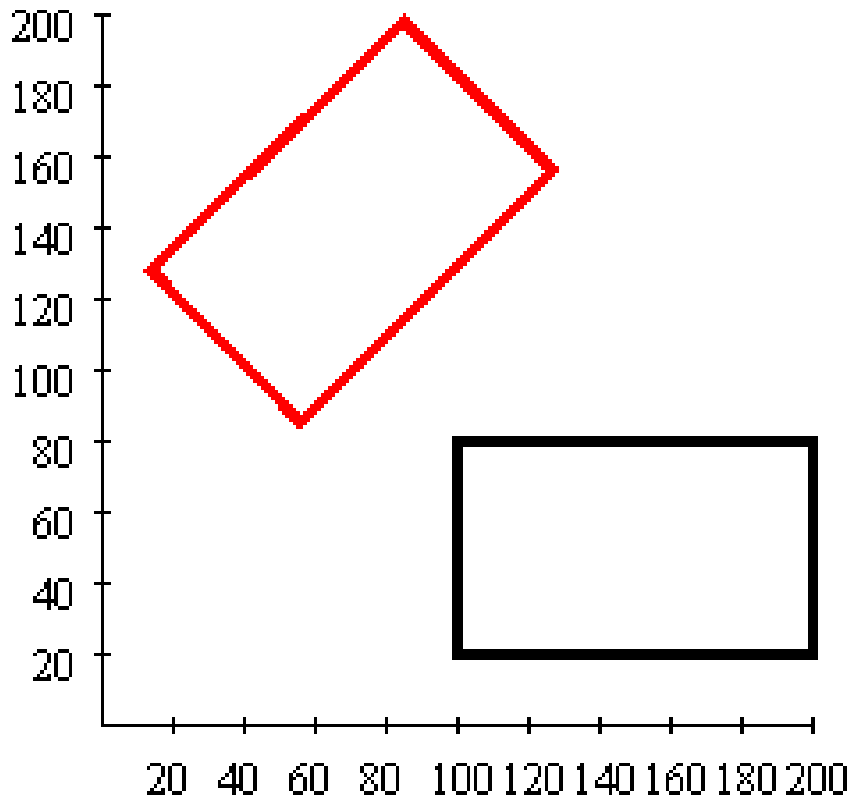
$$x' = r \cdot \cos(\varphi + \theta) = r \cdot \cos(\varphi) \cdot \cos(\theta) - r \cdot \sin(\varphi) \cdot \sin(\theta)$$

$$= x \cdot \cos(\theta) - y \cdot \sin(\theta)$$

$$y' = r \cdot \sin(\varphi + \theta) = r \cdot \cos(\varphi) \cdot \sin(\theta) + r \cdot \sin(\varphi) \cdot \cos(\theta)$$

$$= x \cdot \sin(\theta) + y \cdot \cos(\theta)$$

RotationExampleColour.java



Rotation through $\theta = \pi/4$

Rotation

- A rotation is always carried out around the origin of the coordinate system.
  Therefore, a similar shifting effect as in the case of scalings happens, when an object is not centred around the origin.
- In Java 2D a rotation through a postive angle corresponds to a clockwise rotation since the y-axis points downwards.

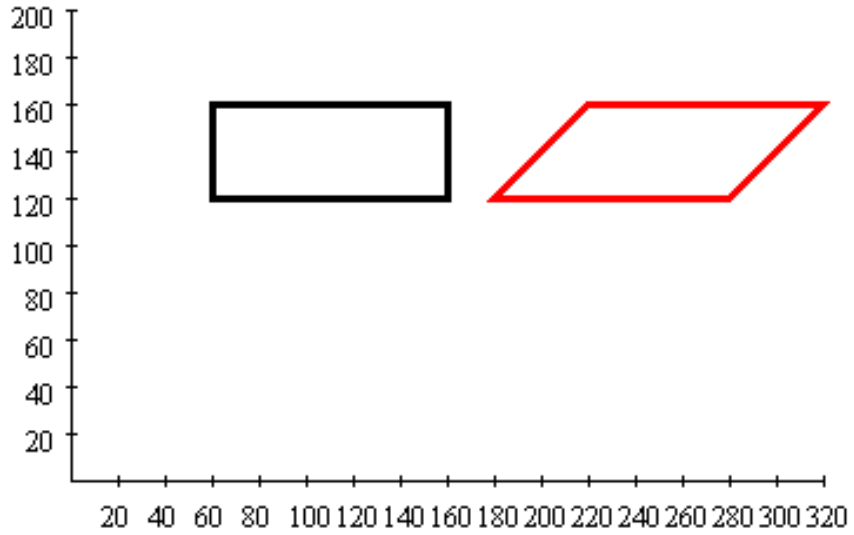
Geometric transformations (2D)

Shear transformation: Distortion of an elastic body:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x + s_x \cdot y \\ y + s_y \cdot x \end{pmatrix} = \begin{pmatrix} 1 & s_x \\ s_y & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix}$$

- $s_y = 0$: Shear transformation in the direction of the x-axis.
- $s_x = 0$: Shear transformation in the direction of the y-axis.

ShearingExampleColour.java



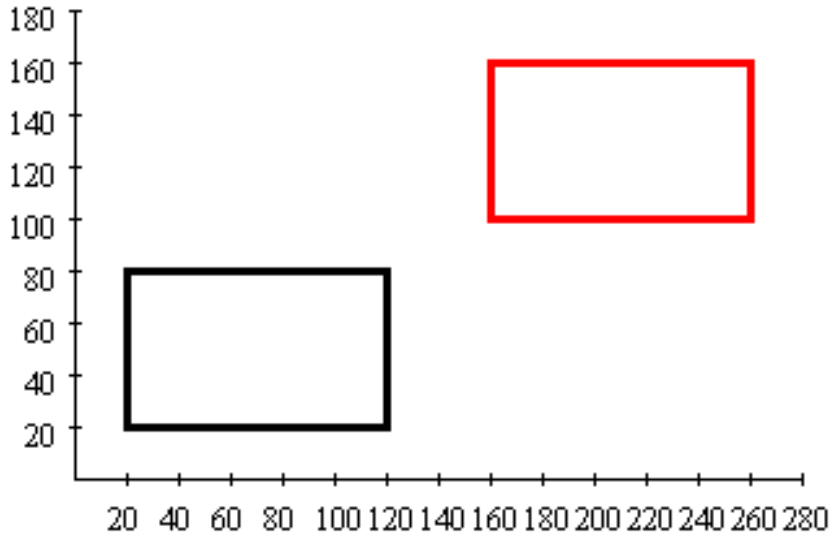
Shear transformation with $s_x = 1$, $s_y = 0$

The shear transformation is always carried out w.r.t. the origin of the coordinate system. Therefore, a shift might be involved if the object is not centred around the origin.

Geometric transformations (2D)

Translation: Shift by the vector d:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \end{pmatrix} = \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} d_x \\ d_y \end{pmatrix}$$



$d_x = 140,\ d_y = 80$

Geometric transformations (2D)

- A translation cannot be written in the form $\mathbf{v}' = A\mathbf{v}$ since $\mathbf{0} + \mathbf{d} \neq A \cdot \mathbf{0}$ holds for every translation $\mathbf{d} \neq \mathbf{0}$.
- All other above introduced geometric transformations can be written in matrix form $v' = A \cdot v$.
- Transformations of the form $\mathbf{v}' = A\mathbf{v}$ are called **lineare transformations**.
- Transformations of the form $\mathbf{v}' = A\mathbf{v} + \mathbf{d}$ are called **affine transformations**.

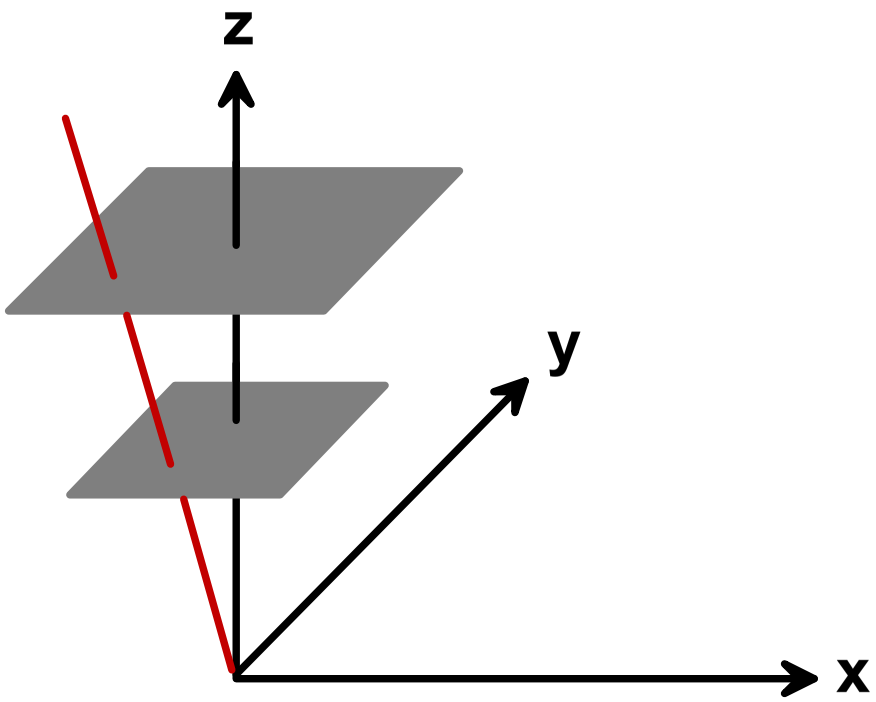
Homogeneous coordinates

In order to compute affine transformations based on matrix multiplications, **homogene Koordinaten** are introduced.

(Multiple) embedding of the plane into $\mathbb{R}^3 \setminus (\mathbb{R}^2 \times \{0\})$:

The point (x, y, z) (where $z \neq 0$) in homogeneous coordinates corresponds to the point $\left(\frac{x}{z}, \frac{y}{z}\right)$ in the plane in Cartesian coordinates.

Homogeneous coordinates



Among others, the point $(x, y, 1)$ corresponds to the point (x, y).

(Technically speaking: Two vectors/points $\mathbf{v}, \mathbf{v}' \in \mathbb{R}^3 \setminus (\mathbb{R}^2 \times \{0\})$ are equivalent (represent the same point in Cartesian coordinates) if there exists $\lambda \in \mathbb{R}$ such that $\mathbf{v}' = \lambda \mathbf{v}$ holds.)

Translation in hom. coordinates

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} x + d_x \\ y + d_y \\ 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

Translation matrix: $T(d_x, d_y) = \begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix}$

Transformation matrices

Transformation	Abbreviation	Matrix
Translation	$T(d_x, d_y)$	$\begin{pmatrix} 1 & 0 & d_x \\ 0 & 1 & d_y \\ 0 & 0 & 1 \end{pmatrix}$
Scaling	$S(s_x, s_y)$	$\begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Transformation matrices

Rotation $R(\theta)$

$$\begin{pmatrix} \cos(\theta) & -\sin(\theta) & 0 \\ \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Shear transformation $S(s_x, s_y)$

$$\begin{pmatrix} 1 & s_x & 0 \\ s_y & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Composition of transformations

The composition of geometric transformations can be computed by matrix multiplication in homogeneous coordinates.

So far, all transformation matrices had the form

$$\begin{pmatrix} a & c & e \\ b & d & f \\ 0 & 0 & 1 \end{pmatrix}.$$

The product of two such matrices yields a matrix of the same type.

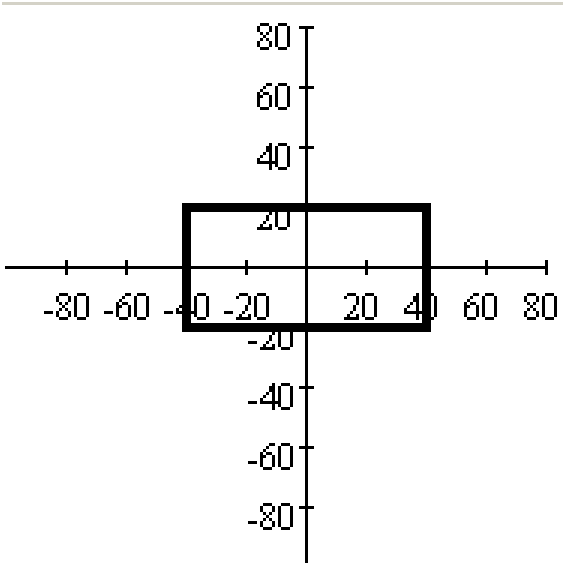
Composition of transformations

Rotations and translations preserve lengths and angles.
Scalings and shear transformations do not preserve lengths and angles in general, but at least parallel lines will be mapped to parallel lines again.

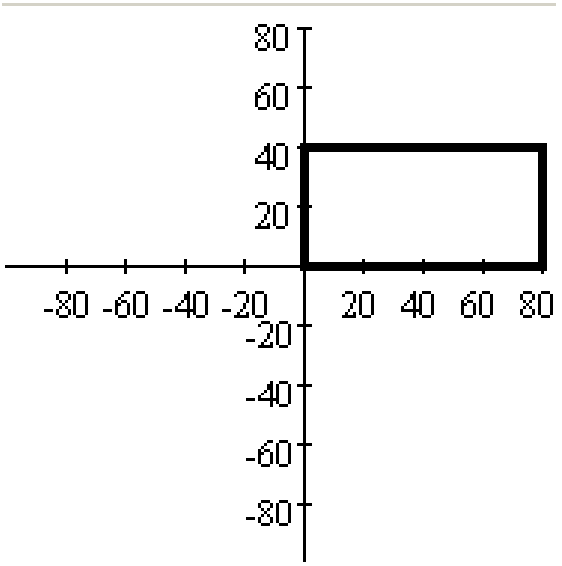
Matrix multiplication is noncommutative. The order in which geometric transformations are applied matters. Changing the order might lead to a different result.
Exceptions are:
Composition of the transformations of the same type (rotation with rotation, translation with Translation, scaling with scaling) and rotation with scaling with the same scaling factor for the x- and the y-axis.

Changing the order

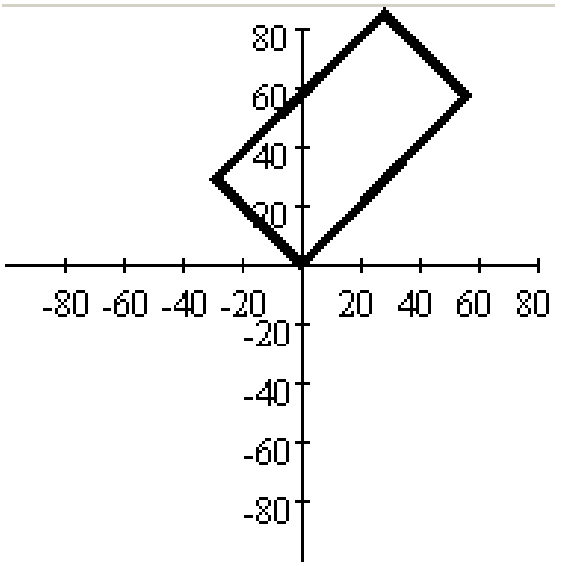
$$R(45^\circ) \cdot T(40, 20)$$



Original



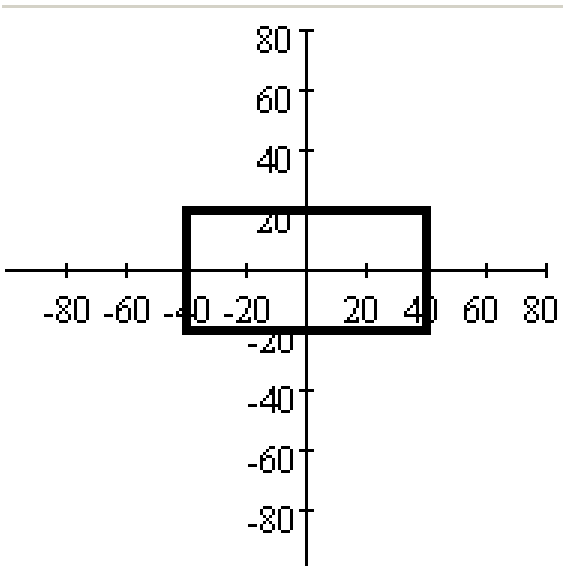
Translation



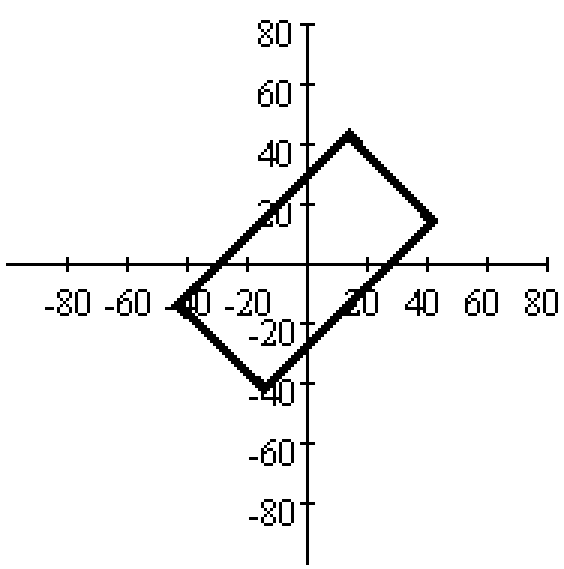
1. Translation
2. Rotation

Changing the order

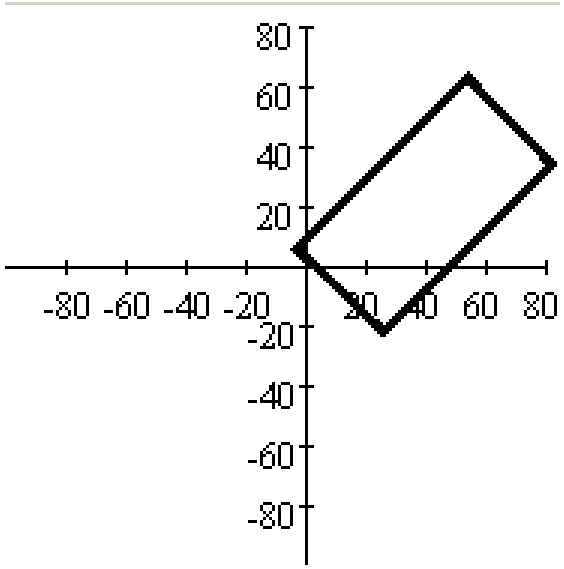
$$T(40, 20) \cdot R(45^\circ)$$



Original

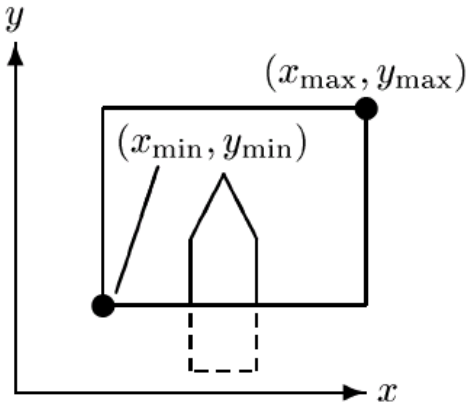


Rotation

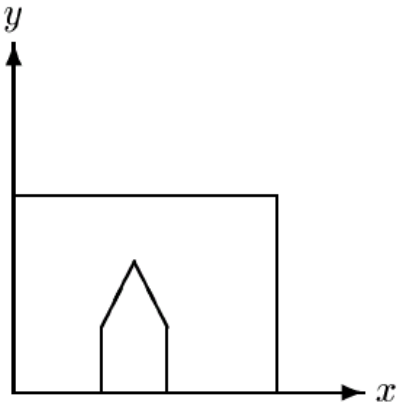


1. Rotation
2. Translation

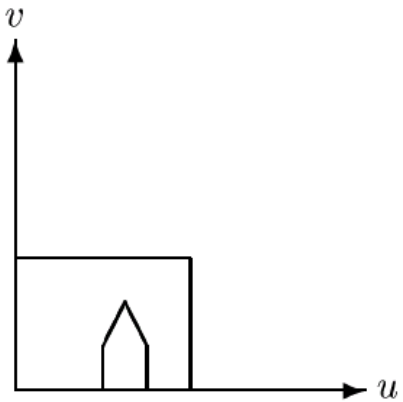
World $\rightarrow$ window coordinates



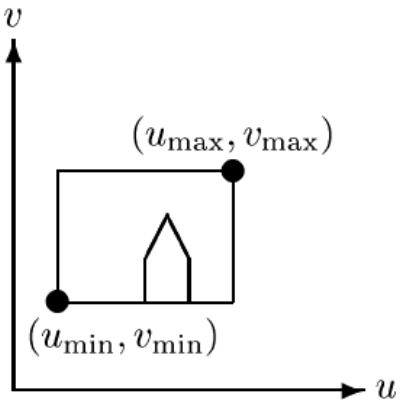
window in the world coordinate space



window in the origin of the world coordinate space



scaled window in the origin of the monitor coordinate space



final position of the window

World $\rightarrow$ window coordinates

The required transformation is

$$
\begin{aligned}
M_{WV} &= T(u_{\text{min}}, v_{\text{min}}) \cdot \\
&\quad S\left(\frac{u_{\text{max}} - u_{\text{min}}}{x_{\text{max}} - x_{\text{min}}}, \frac{v_{\text{max}} - v_{\text{min}}}{y_{\text{max}} - y_{\text{min}}}\right) \cdot \\
&\quad T(-x_{\text{min}}, -y_{\text{min}}).
\end{aligned}
$$

Rotation and scaling

Rotation w.r.t. the point (x_0, y_0):

$$R(\theta, x_0, y_0) \;=\; T(x_0, y_0) \cdot R(\theta) \cdot T(-x_0, -y_0)$$

Scaling w.r.t. the point (x_0, y_0):

$$S(s_x, s_y, x_0, y_0) \;=\; T(x_0, y_0) \cdot S(s_x, s_y) \cdot T(-x_0, -y_0)$$

Reflection w.r.t. the y-axis

- Standard window coordinates:
  - The origin is in the upper left corner.
  - The y-axis points downwards.
- (Sometimes) standard Cartesian coordinates are preferred.
  - The origin is in the lower left corner.
  - The y-axis points upwards.

Transformation to be applied for a window of height h:

$$T(0, h) \cdot S(1, -1)$$

Transformations in Java 2D

Class `AffineTransform` for affine transformations in homogeneous coordinates in Java 2D.

Constructors:

- `AffineTransform id = new AffineTransform()` generates the identical transformation encoded by the unity matrix.
- Explicit definition of the matrix:

  `new AffineTransform(a,b,c,d,e,f)`

  $a, \ldots, f$ are the six parameters of the transformation matrix given as `Double`-values.

Transformations in Java 2D

Rotation:

- `at.setToRotation(angle)` and `at.setToRotation(angle,x,y)` define the transformation `at` as a rotation through the angle `angle` around the origin or the point `(x,y)`, respectively.
- `at.rotation(angle)` and `at.rotation(angle,x,y)` concatenate the transformation `at` with a corresponding rotation (in terms of matrix multiplication from the right, so that the rotation is carried out before the original transformation `at`).

Transformations in Java 2D

Scaling:

- `at.setToScale(sx,sy)` defines the transformation `at` as a scaling with the scaling factors `sx` for the x- and `sy` for the y-axis w.r.t. to the origin of the coordinate system.
- `at.scale(sx,sy)` concatenates the transformation `at` with the corresponding scaling.

Transformations in Java 2D

Shear transformation:

- `at.setToShear(sx,sy)` defines the transformation `at` as a shear transformation with the parameters `sx` for the x- and `sy` for the y-axis w.r.t. to the origin of the coordinate system.
- `at.shear(sx,sy)` concatenates the transformation `at` with the corresponding shear transformation.

Transformations in Java 2D

Translation:

- The method `at.setToTranslation(dx,dy)` defines the transformation `at` as a translation by the vector $(dx, dy)^\top$.
- The method `at.translate(dx,dy)` concatenates the transformation `at` with the corresponding translation.

Transformations in Java 2D

Composition of transformations:

- `at1.concatenate(at2)` appends the affine transformation `at2` to the affine transformation `at1` (in terms of matrix multiplication from the right, so that `at2` is first carried out and the original transformation `at1` afterwards).
- `at1.preConcatenate(at2)` does the same as `at1.concatenate(at2)`, but in the reverse order. The matrix multiplication is carried out from the left, so that the original transformation `at1` is first carried out and the transformation `at2` afterwards.

Transformation of the whole image

When a transformation `at` is applied to the Graphics2D object `g2d` by `g2d.transform(at)`, the transformation `at` will be applied to every object before it is drawn.

In this way, the problem of the y-axis pointing downwards in the window, can be solved.

```
AffineTransform yUp =
                    new AffineTransform();
yUp.setToScale(1,-1);
AffineTransform translate =
                    new AffineTransform();
translate.setToTranslation(0,windowHeight);
yUp.preConcatenate(translate);
```

Transformations applied to objects

Many objects can be generated by applying transformations to elementary objects.

Example: A rectangle having its centre at the point (x, y) should be drawn. However, the rectangle should not be axes-parallel, but have an angle of $45°$ with the coordinate axes. One can generate an axes-parallel rectangle in the origin, rotate it by $45°$ and then translate it by the vector $(x, y)^\top$.

Transformations applied to objects

Applying a transformation `affTrans` to a `Shape s`:

```
Shape transformedShape =
   affTrans.createTransformedShape(s);
```

Correspondingly for an `Area a`:

```
Area transformedArea =
   affTrans.createTransformedArea(a);
```

Moving objects

Moving objects (animated graphics) can also be implemented based on transformations.

The transformations must describe the movement of the object(s).

- The object is drawn.
- The object's position in the next frame is computed based on transformations.
- The old object (or the whole window) is deleted.
- The updated object and the background are drawn again.

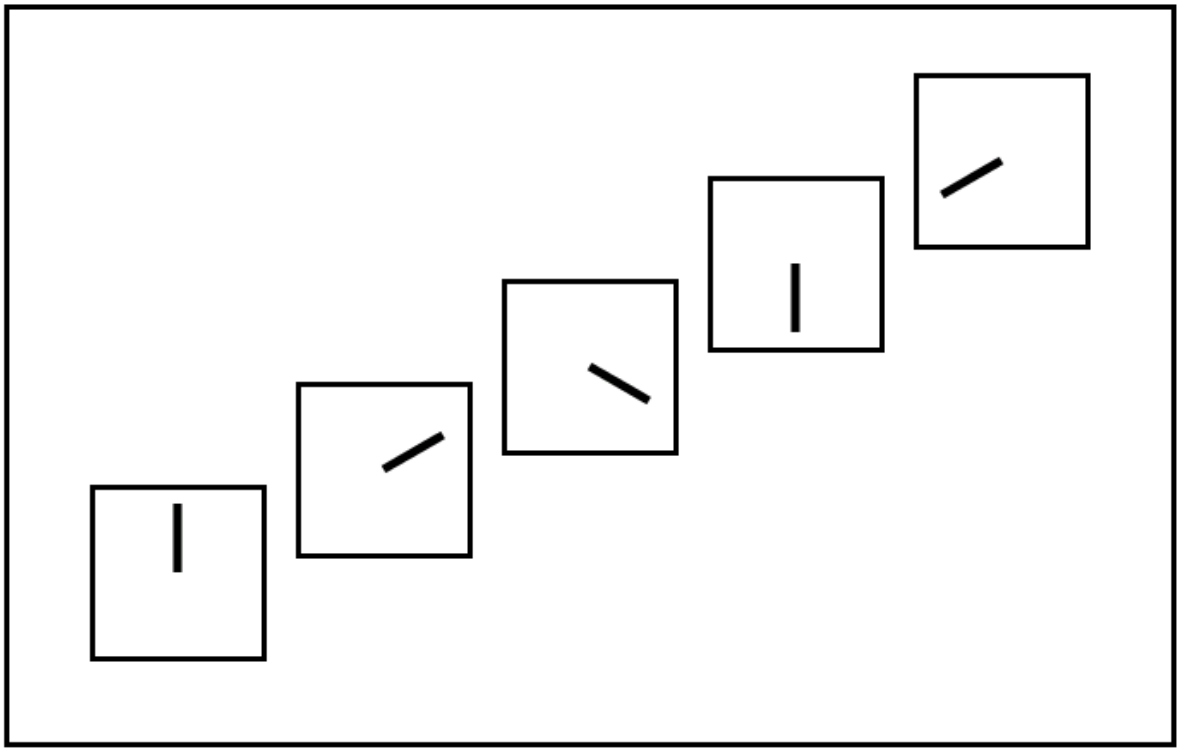
Moving objects

Method 1: Generate an object in the origin and keep track of the accumulated transformations defining the movement. Always apply the accumulated transformations to the object in the origin of the coordinate system in a suitable order.

Method 2: Keep track of the position, orientation and scaling of the object and apply the transformations step by step to the already transformed object.

Example for method 1

Moving clock:



Example for method 1

Shift (movement) of the clock in each step (frame):

$$T_{\text{clock,step}} = T(2, 1).$$

Rotation of the second hand in each step (frame):

$$T_{\text{hand,step}} = R(-\pi/180)$$

Example for method 1

$$T^{(\text{new})}_{\text{clock,accTrans}} = T_{\text{clock,step}} \circ T^{(\text{old})}_{\text{clock,accTrans}}$$

$$T^{(\text{new})}_{\text{hand,accRotation}} = T_{\text{hand,step}} \circ T^{(\text{old})}_{\text{hand,accRotation}}$$

$$T_{\text{hand,acc}} = T_{\text{clock,accTrans}} \circ T_{\text{hand,accRotation}}$$

Interpolators

Aim: Continuous transition from an initial state to final state.

Simplest case: Moving from position (point) $\mathbf{p}_0$ to $\mathbf{p}_1$.

The points $\mathbf{p}_\alpha$ on the line between $\mathbf{p}_0$ and $\mathbf{p}_1$ are the convex combinations

$$\mathbf{p}_\alpha = (1-\alpha)\cdot\mathbf{p}_0 + \alpha\cdot\mathbf{p}_1, \qquad \alpha \in [0,1].$$

Interpolation of transformations

Convex combination of two affine transformations with associated matrices M_0 and M_1:

$$M_\alpha = (1-\alpha)\cdot M_0 + \alpha \cdot M_1, \qquad \alpha \in [0,1].$$

Application: Transition of an object S_0 into another object S_1 where both objects were generated by transformations M_0 and M_1, respectively, applied to an object S.

Interpolation of transformations

The intermediate objects result from applying the convex combinations of the transformations $M_0 =$`initialTransform` and $M_1 =$`finalTransform` to the original object S.

```
double[] initialMatrix = new double[6];
initialTransform.getMatrix(initialMatrix);

double[] finalMatrix = new double[6];
finalTransform.getMatrix(finalMatrix);
```

(see `ConvexCombTransforms.java`)

Simple object interpolation

Two objects S and S' are each defined by n points $(x_1, y_1), \ldots, (x_n, y_n)$ and $(x'_1, y'_1), \ldots, (x'_n, y'_n)$, respectively, and lines and (quadratic and cubic) curves based on these points.

The lines and curves in the two object must correspond to each other. For instance, if the quadratic curve defined by the three points (x_1, y_1), (x_3, y_3) and (x_8, y_8) is part of the object S, then the corresponding quadratic curve defined by the points (x'_1, y'_1), (x'_3, y'_3) and (x'_8, y'_8) must be part of the object S' and vice versa.

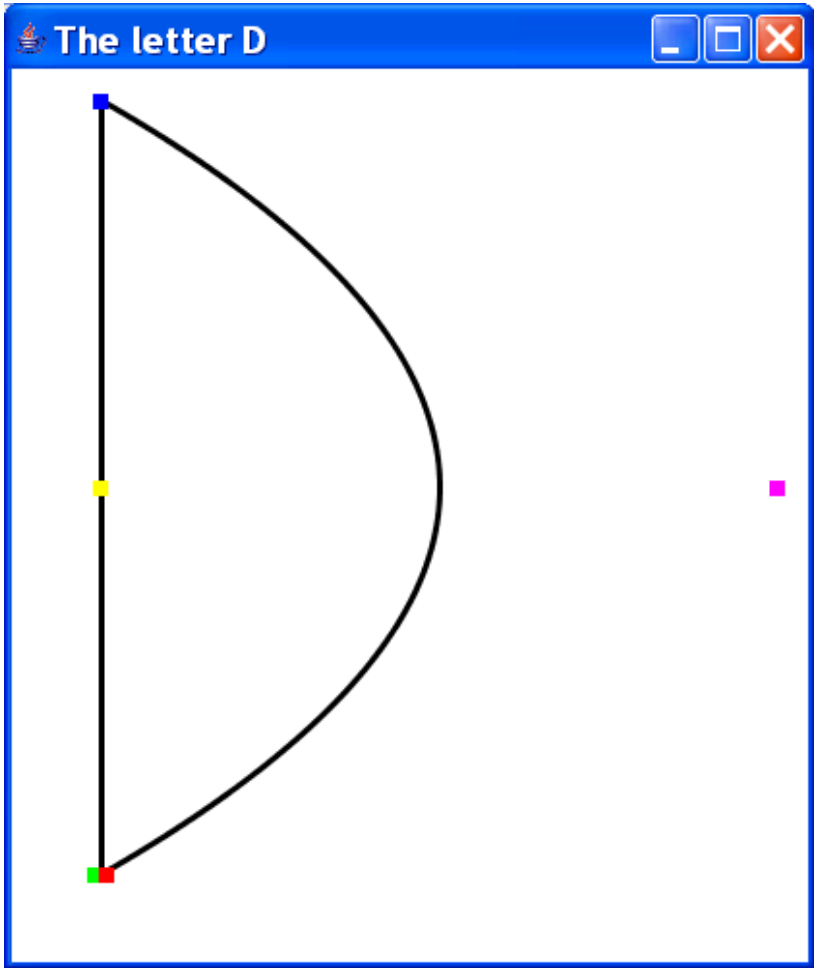
Simple object interpolation

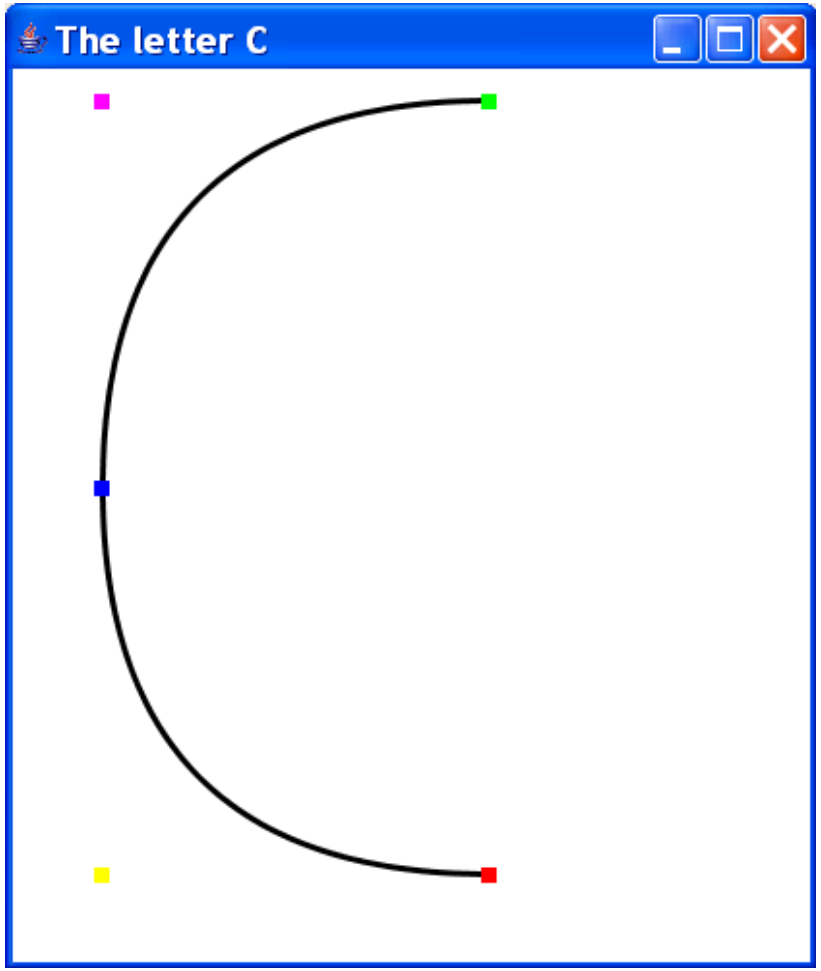
Example: The objects S and S' are the two letters D and C, respectively, defined by the points

$$
\begin{array}{lcllcl}
(x_1, y_1) & = & (50, 50) & (x'_1, y'_1) & = & (50, 250) \\
(x_2, y_2) & = & (50, 450) & (x'_2, y'_2) & = & (250, 50) \\
(x_3, y_3) & = & (400, 250) & (x'_3, y'_3) & = & (50, 50) \\
(x_4, y_4) & = & (50, 450) & (x'_4, y'_4) & = & (250, 450) \\
(x_5, y_5) & = & (50, 250) & (x'_5, y'_5) & = & (50, 450)
\end{array}
$$

and the quadratic curve given by the points no. 1.,3. and 2., and the quadratic curve given by the points no. 1.,5. and 4.

Simple object interpolation





Simple object interpolation

The intermediate frames for $\alpha \in [0, 1]$ require the computation of the convex combinations

$$(\tilde{x}_i, \tilde{y}_i) \;=\; (1 - \alpha) \cdot (x_i, y_i) + \alpha \cdot (x'_i, y'_i)$$

of the points (x_i, y_i) and (x'_i, y'_i) ($i = 1, \ldots, n$) and the drawing of the lines and curves based on the points $(\tilde{x}_i, \tilde{y}_i)$ corresponding to the lines and curves in S and S'.

(see `DToCMorphing.java`)

Roundoff errors

A second hand with a length of 100 pixels rotates around the origin of the coordinate system in steps of $6°$ (corresponding to one second).

Applying the rotations iteratively to the resulting pixels (integer arithmetics): Result after one minute: $(95, -2)$

Roundoff error example

Time	x	y
	double	
1 minute	99.99999999999973	−4.8572257327350600E-14
2 minutes	99.99999999999939	−9.2981178312356860E-14
3 minutes	99.99999999999906	−1.3739009929736312E-13
4 minutes	99.99999999999876	−1.4571677198205180E-13
5 minutes	99.99999999999857	−2.2204460492503130E-13
6 minutes	99.99999999999829	−2.9143354396410360E-13
7 minutes	99.99999999999803	−3.1641356201816960E-13
8 minutes	99.99999999999771	−3.7331249203020890E-13
9 minutes	99.99999999999747	−4.2604808569990380E-13
10 minutes	99.99999999999715	−4.5657921887709560E-13
8 hours	99.99999999986587	−2.9524993561125257E-11

Roundoff error example

Time	x	y
	`float`	
1 minute	100.00008	−1.1175871E-5
2 minutes	100.00020	−1.4901161E-5
3 minutes	100.00032	−1.8626451E-5
4 minutes	100.00044	−1.1920929E-5
5 minutes	100.00056	−8.9406970E-6
6 minutes	100.00068	−3.1292439E-5
7 minutes	100.00085	−5.3644180E-5
8 minutes	100.00100	−7.2270630E-5
9 minutes	100.00108	−8.0466270E-5
10 minutes	100.00113	−8.4191560E-5
8 hours	100.00328	−1.9669533E-4