# Werkzeuge der Informatik

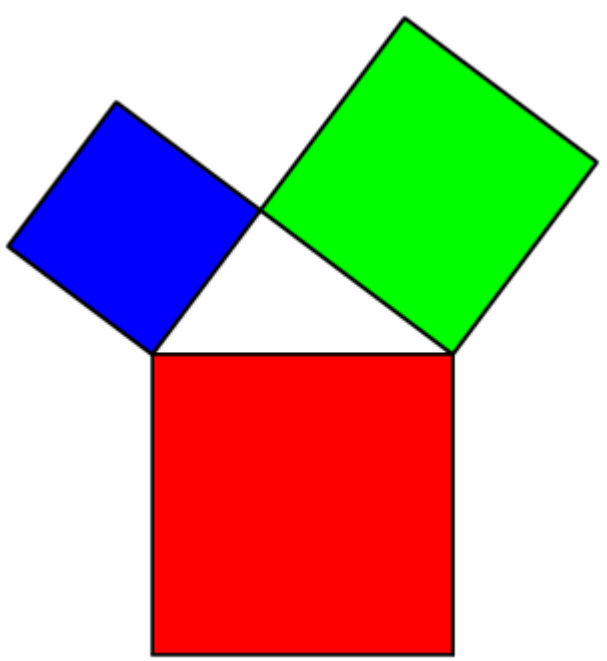
## Einführung in PostScript®

Prof. Dr. Kai Hormann

Institut für Informatik

TU Clausthal

01.02.2008



# Organisatorisches

- Vorlesung
  - Fr, 01.02.2008, 13–17 Uhr, Hörsaal A, Institut für Mathematik
- Webseite
  - Folien, Übungsblätter, zusätzliches Material
  - http://zach.in.tu-clausthal.de/teaching/werkzeuge_0708/
- Fragestunde
  - Di, 05.02.2008, 13–17 Uhr, Raum 302, Institut für Informatik
- Übungen
  - Mi, 06.02.2007, 17–19 Uhr, Raum 302, Institut für Informatik
  - Do, 07.02.2007, 17–19 Uhr, Raum 302, Institut für Informatik
- Klausur
  - nur für Studiengänge Informationstechnik und Physik
  - Sa, 29.03.2008, 11–12 Uhr, Raum wird noch bekannt gegeben

# Literatur

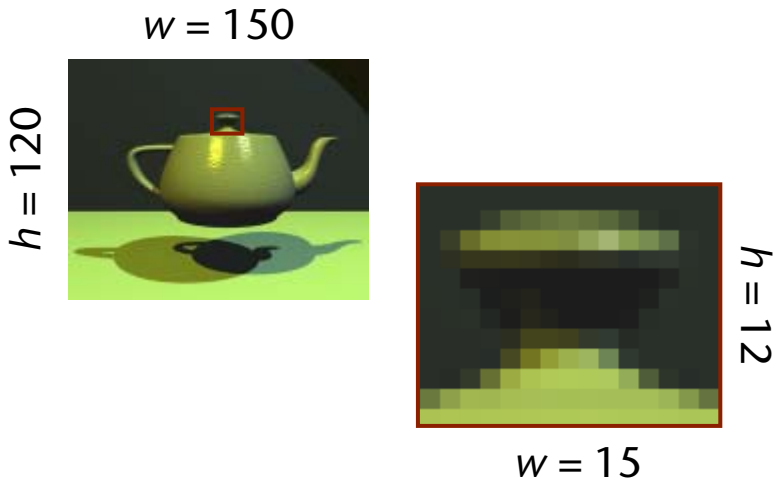
- Bücher
  - Adobe Systems Inc.: *Postscript® Handbuch*, 2. Auflage, Addison-Wesley, 1989
  - Adobe Systems Inc.: *Postscript® Language Reference*, 3. Auflage, Addison-Wesley, 1999
  - Bill Casselman: *Mathematical Illustrations*, Cambridge University Press, 2004
- Online Ressourcen
  - de.wikipedia.org/wiki/Postscript
  - www.adobe.com/products/postscript/pdfs/PLRM.pdf
  - www.math.ubc.ca/people/faculty/cass/graphics/text/www/

# Übersicht

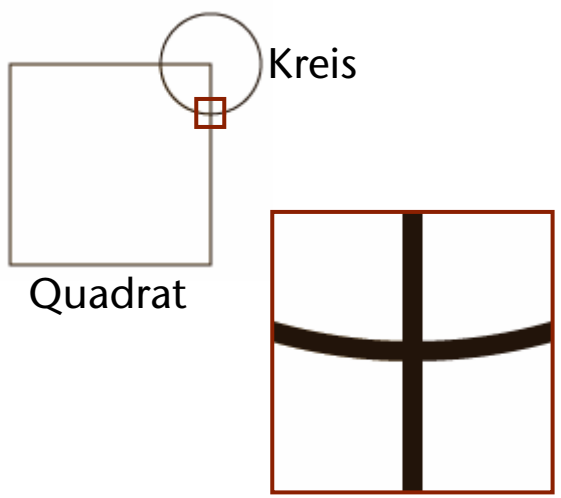
- Vektor- vs. Pixelgrafik
- Historischer Überblick
- PostScript-Syntax
- Grafik-Befehle
- lineare Transformationen
- Definition von Makros
- arithmetische Operationen
- einfache Schleifen
- Ausgabe von Text

# Vektor- vs. Pixelgrafik

- Pixelgrafik oder Bitmap
  - Matrix von $w \times h$ Bildpunkten
  - ein Farbwert pro Bildpunkt
  - abhängig von der Auflösung




- Vektorgrafik
  - Beschreibung durch grafische Primitive (z.B. Punkt, Linie, Kreis, Polygon, etc.)
  - unabhängig von der Auflösung
  - Umwandlung in Pixel bei Bedarf



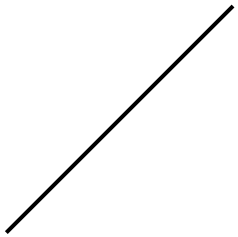
# PostScript®

- Programmiersprache zur Beschreibung von Vektorgrafik
  - geräte- und auflösungsunabhängig
  - Interpreter wandelt Vektorgrafik in Pixelgrafik um
- unterstützte Elemente
  - gerade Linien, Kreisbögen, kubische Kurven
  - ausgefüllte Formen, z.B. Kreise, geschlossene Polygone
  - lineare Transformationen, z.B. Translation, Rotation, Skalierung
  - Text
  - Pixelgrafiken
- Programm = Beschreibung einer Vektorgrafik

# PostScript®

- Beispiele
  - Beschreibung einer Linie

```
100 100 moveto
200 200 lineto
stroke
```



  - gestrichelter Halbkreis

```
[5] 0 setdash
200 200 100 0 180 arc
stroke
```

# Historischer Überblick

- 1976 – *John Warnock* von der *Evans & Sutherland Computer Corporation* entwickelt *Design Systems*, eine Sprache für CAD-Anwendungen
- 1978 – *John Warnock* und *Martin Newell* am *Xerox Palo Alto Research Center* entwickeln Design Systems weiter zu *JaM*, zum Entwurf von VLSI-Schaltungen und für Satz, Druck und Grafik
- 1982 – *John Warnock* und *Chuck Geschke* gründen *Adobe Systems Inc.* und entwickeln *PostScript*, eine Sprache zur Beschreibung von 2D-Vektorgrafik

# Historischer Überblick

- 1984 – *PostScript Level 1* kommt auf den Markt
- 1985 – *Apple LaserWriter* erscheint als erster Drucker mit einem *PostScript* Interface
- 1991 – *PostScript Level 2* unterstützt komprimierte Bildformate (JPEG) und ist schneller und zuverlässiger
- 1993 – Teile von *PostScript* werden für das *Portable Document Format 1.0* (PDF) verwendet
- 1997 – *PostScript 3* mit erweitertem Farbmanagement

# Getting started...

- PostScript-Programme erstellen
  - per Hand mit einem Text-Editor, z.B. `WinEdt` oder `emacs`
  - mit einem selbstgeschriebenen Programm, z.B. in `C`
  - Ausgabe eines Anwendungsprogramms
  - speichern in einer Datei, z.B. `grafik.ps`
- PostScript-Programme ausführen
  - erfordert Interpreter und Ausgabegerät
    z.B. `GSview` und Bildschirm oder PostScript-Drucker
  - wandelt Grafikbefehle auflösungsabhängig in Pixel um
    (Ausnahme: Plotter)

# Aufbau eines PostScript-Programms

- allgemeine Struktur
  - Header
    - allgemeine Angaben und Kommentare z.B. Format, Autor, Titel, Datum, etc.
  - Prolog
    - Definition eigener Befehle
  - Skript
    - Zeichenbefehle
  - Trailer
    - abschließende Angaben und Kommentare

```
%!PS-Adobe-1.0
%%Creator: Kai Hormann
%%Title: Beispiel
%%CreationDate: 02/01/2008
%%BoundingBox: 0 0 595 842
%%EndComments

% eigene Definitionen
%%EndProlog

100 100 moveto
200 200 lineto
stroke

%%Pages: 1
%%EOF
```

- Speichern als einfache Text-Datei im ASCII-Format

# Kommentare

- allgemeine Kommentare
  - alles zwischen % und Zeilenende
  - beliebiger Text
  - wird komplett ignoriert
- spezielle Kommentare
  - beginnen mit %% oder %!
  - danach ein Schlüsselwort
  - optional ein : und durch Leerzeichen getrennte Parameter
  - werden nicht von allen Interpretern berücksichtigt

```
%!PS-Adobe-1.0
%%Creator: Kai Hormann
%%Title: Beispiel
%%CreationDate: 02/01/2008
%%BoundingBox: 0 0 595 842
%%EndComments

% eigene Definitionen
%%EndProlog

100 100 moveto
200 200 lineto
stroke

%%Pages: 1
%%EOF
```

# Header

- erste Zeile
  - gibt PostScript-Version an

    hier: PostScript Level 1
  - beginnt immer mit `%!`
- Dokument-Informationen
  - Autor, Titel, Datum
- Zeichenbereich
  - Schlüsselwort `BoundingBox`
  - Parameter `x y w h` – linke untere Ecke, Breite, Höhe
  - Angaben in pt – 1 pt = 1/72 in – 1 in = 2.54 cm

```
%!PS-Adobe-1.0
%%Creator: Kai Hormann
%%Title: Beispiel
%%CreationDate: 02/01/2008
%%BoundingBox: 0 0 595 842
%%EndComments
% eigene Definitionen
%%EndProlog
100 100 moveto
200 200 lineto
stroke
%%Pages: 1
%%EOF
```

# Prolog und Trailer

- Definition eigener Befehle
  - Abkürzungen
  - Makros
  - Unterprogramme
- Angabe der Seitenzahlen
  - PostScript erlaubt die Definition mehrerer Seiten
  - wir beschränken uns auf 1 Seite
- Kennzeichnung des Programmendes

```
%!PS-Adobe-1.0
%%Creator: Kai Hormann
%%Title: Beispiel
%%CreationDate: 02/01/2008
%%BoundingBox: 0 0 595 842
%%EndComments
```

```
/mt /moveto load def
%%EndProlog
```

```
100 100 mt
200 200 lineto
stroke
```

```
%%Pages: 1
%%EOF
```

# Skript

- Sequenz von Objekten
  - Daten-Objekte
    z.B. Zahlen, Boolean-Werte,
    Strings, Arrays, etc.
  - Programm-Objekte
    z.B. Operatoren, Befehle,
    Namen, Prozeduren, etc.
  - getrennt durch Leerzeichen,
    Tab oder Zeilenende

```
%!PS-Adobe-1.0
%%Creator: Kai Hormann
%%Title: Beispiel
%%CreationDate: 02/01/2008
%%BoundingBox: 0 0 595 842
%%EndComments

% eigene Definitionen
%%EndProlog

100 100 moveto
200 200 lineto
stroke

%%Pages: 1
%%EOF
```

- Interpreter liest Objekte sequentiell und führt sie aus
  - Ausführung hängt vom Objekttyp ab

# Skript

- Beispiel `100 100 moveto`
- `100` und `100` sind Zahlen
  - Bei der Ausführung werden sie auf den Operanden-Stack geschoben
- `moveto` ist ein Operator
  - Bei der Ausführung werden die zwei obersten Elemente des Operanden-Stacks geholt, als Koordinaten interpretiert und der Zeichenstift dorthin gesetzt
- PostScript benutzt also die Postfixnotation

```
%!PS-Adobe-1.0
%%Creator: Kai Hormann
%%Title: Beispiel
%%CreationDate: 02/01/2008
%%BoundingBox: 0 0 595 842
%%EndComments

% eigene Definitionen
%%EndProlog

100 100 moveto
200 200 lineto
stroke

%%Pages: 1
%%EOF
```

# Syntax

- Zeichensatz
  - erlaubt sind alle ASCII-Zeichen
  - Leerzeichen, Tab und Return werden gleich behandelt
  - `(` , `)` , `<` , `>` , `[` , `]` , `{` , `}` , `/` , `%` sind Sonderzeichen
- Zahlen
  - ganze Zahlen `123` `-98` `43445` `0` `+17`
  - reelle Zahlen `-.002` `34.5` `12.6e10` `-1.` `0.0`
  - Radix-Zahen `8#1777` `16#FFFE` `2#1000`

    allgemeines Format: Basis#Zahl

    bei Basis > 10: Buchstaben für Ziffern > 9

# Syntax

- Strings
  - (fast) beliebige Zeichenfolge zwischen runden Klammern
  - `(` , `)` und `\` müssen „maskiert“ werden: `\(` , `\)` , `\\`
  - Beispiele
    ```
    (Dies ist ein String)
    (Ein String mit !\(*@] Sonderzeichen)
    ()
    ```
- Namen
  - jede Zeichenfolge ohne Sonderzeichen, die nicht eine Zahl ist
  - für Variablen, Befehle, Makros, etc.
  - Beispiele `abc` `Offset` `@@` `a.b` `&Zeichen` `$Hallo`

# Syntax

- Arrays
  - durch Leerzeichen getrennte Objekte zwischen eckigen Klammern
  - Beispiel `[123 abd (Hallo Welt)]`
- Prozeduren
  - durch Leerzeichen getrennte Objekte zwischen geschweiften Klammern
  - Beispiel `{add 2 div}`
  - bündelt die enthaltenen Objekte zu einem einzigen Objekt und führt sie aus, wenn dieses neue Objekt aufgerufen wird

# Grafik-Befehle für Linien

- Setzen des Zeichenstifts an die Position (x, y)

```
x y moveto
```

- Zeichnen einer Linie von der aktuellen Position zu einer neuen Position (x, y)

```
x y lineto
```

- Angabe der Position (x, y) *relativ* zur aktuellen Position

```
dx dy rmoveto
```

```
dx dy rlineto
```

- Tatsächliches Zeichnen der Linienzüge

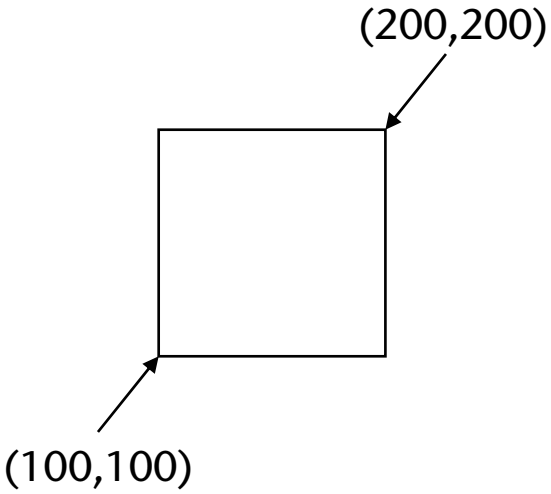
```
stroke
```

# Grafik-Befehle für Linien

- Beispiel 1

```
100 100 moveto
200 100 lineto
200 200 lineto
100 200 lineto
100 100 lineto
stroke
```

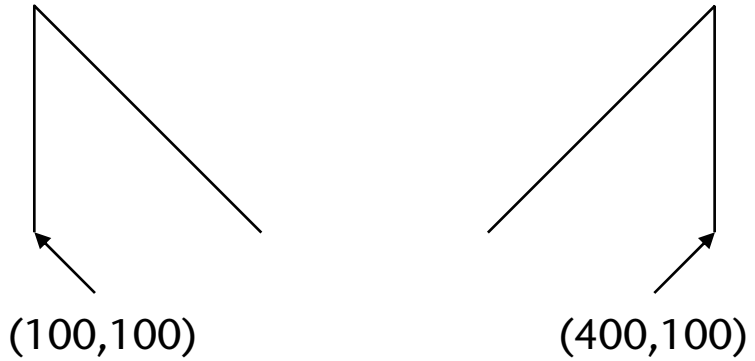
```
100 100 moveto
100 0   rlineto
0 100   rlineto
-100 0  rlineto
0 -100  rlineto
stroke
```




  - beide Programme erzeugen dieselbe Grafik

- Beispiel 2

```
100 4#1210 moveto
100 8#310  lineto
16#C8 100  lineto
5#400 0.0  rmoveto
100 20#50  rlineto
2#0 -100   rlineto
stroke
```



# Grafik-Befehle für Kurven

- Kreisbogen mit Mittelpunkt (x, y), Radius r, Startwinkel s und Endwinkel t, gegen den Uhrzeigersinn

  ```
  x y r s t arc
  ```

  Winkel in Grad (0–360), wobei 0 = positive x-Richtung

- analoger Kreisbogen im Uhrzeigersinn

  ```
  x y r s t arcn
  ```

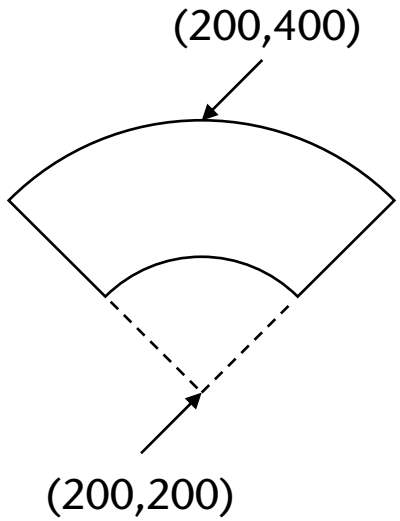
- Freiform-Kurve vom aktuellen Punkt ($x_0$, $y_0$) zum Punkt ($x_3$, $y_3$) und „vorbei“ an ($x_1$, $y_1$) und ($x_2$, $y_2$)

  ```
  x1 y1 x2 y2 x3 y3 curveto
  ```

# Grafik-Befehle für Kurven

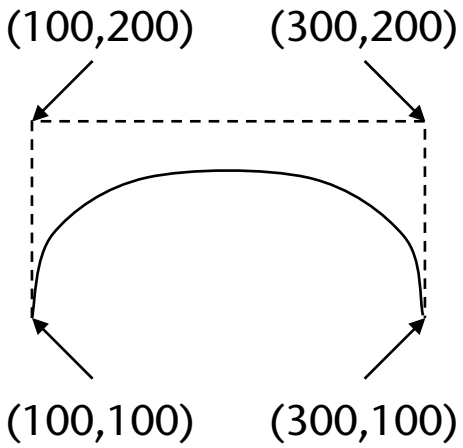
- Beispiel 1

```
200 200 100 45 135 arc
200 200 50 135 45 arcn
35.355 35.355 rlineto
stroke
```




- Beispiel 2

```
100 100 moveto
100 200 300 200 300 100 curveto
stroke
[5] 0 setdash
100 100 moveto
300 100 300 200 100 200 3 {lineto} repeat
stroke
```



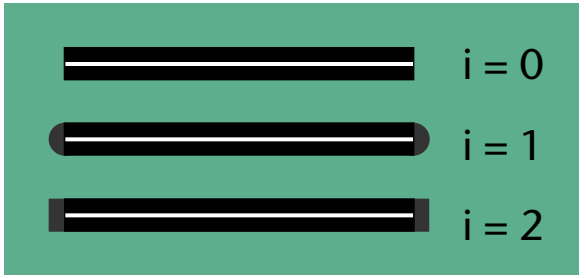
# Grafik-Befehle für den „Line-Style“

- Dicke der zu zeichnenden Linie oder Kurve

  ```
  d setlinewidth
  ```

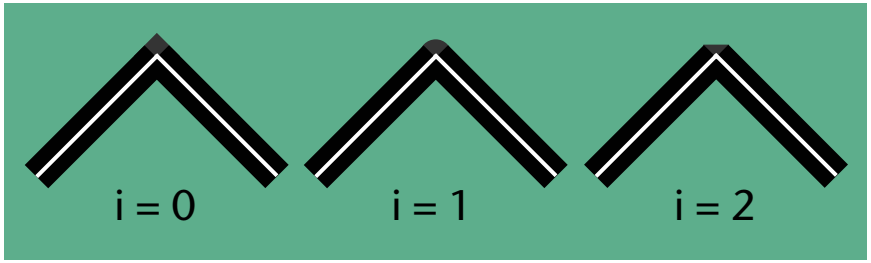
- Aussehen des Linien-Endes

  ```
  i setlinecap
  ```




- Aussehen der Linien-Winkel

  ```
  i setlinejoin
  ```




- Grauwert $k \in [0,1]$

  ```
  k setgray
  ```

- Farbwert $r,g,b \in [0,1]$

  ```
  r g b setrgbcolor
  ```

# Grafik-Befehle für den „Line-Style“

- gestrichelte Linien mit Linienmuster ($x_0$, $x_1$, ... )

```
[x0 x1 x2 ... xn] z setdash
```

die $x_i$ geben abwechselnd die Längen der durchgezogenen Striche und der Lücken an, dieses Muster wird um die Länge z verschoben

- Beispiele

| | |
|---|---|
| `[3] 0 setdash` | 3 an, 3 aus, 3 an, etc. |
| `[3] 2 setdash` | 1 an, 3 aus, 3 an, etc. |
| `[4,2] 0 setdash` | 4 an, 2 aus, 4 an, etc. |
| `[4,2] 5 setdash` | 1 aus, 4 an, 2 aus, 4 an, etc. |
| `[2,3,1] 9 setdash` | 2 an, 1 aus, 2 an, 3 aus, etc. |
| `[] 0 setdash` | durchgezogene Linie |

# Grafik-Befehle für den „Line-Style“

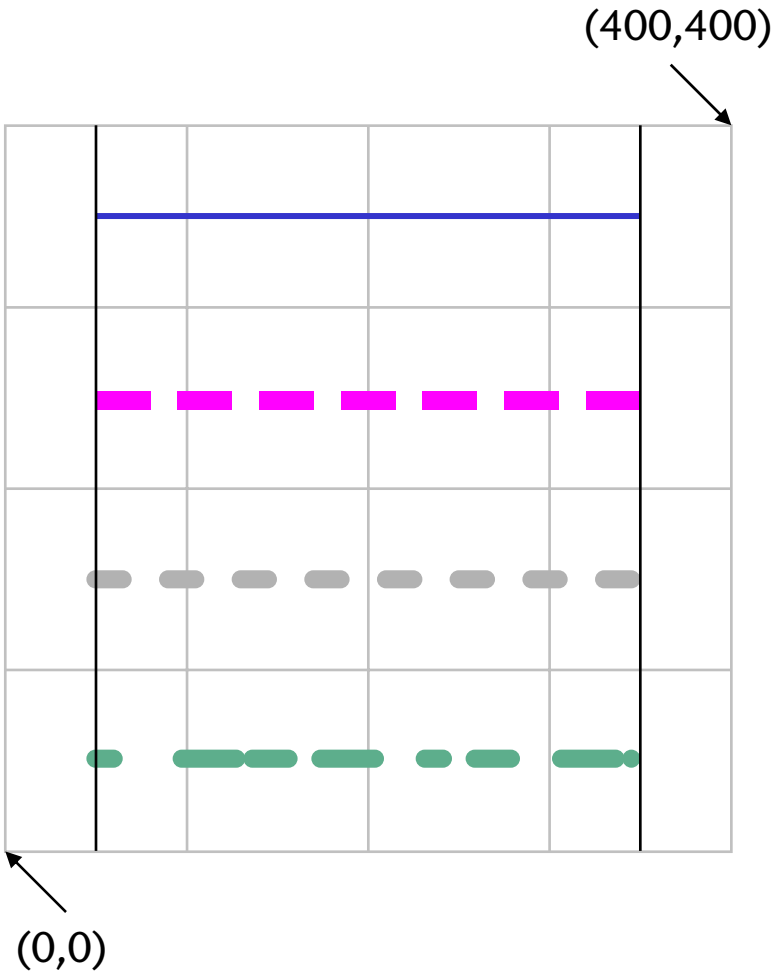
- Beispiel

```
0 0 1 setrgbcolor
50 350 moveto 300 0 rlineto
stroke
```

```
1 0 1 setrgbcolor
10 setlinewidth
[30 15] 0 setdash
50 250 moveto 300 0 rlineto
stroke
```

```
0.5 setgray
1 setlinecap
[15 25] 0 setdash
50 150 moveto 300 0 rlineto
stroke
```

```
0 0.5 0 setrgbcolor
[10 20 20 30 30] 40 setdash
50 50 moveto 300 0 rlineto
stroke
```



# Geschlossene Grafik-Objekte

- Erzeugen eines geschlossenen Pfades

  `closepath`

  verbindet aktuelle Position mit Ausgangsposition und schließt das Objekt ab

- Füllen einer geschlossenen Form mit aktueller Farbe

  `fill` statt `stroke`

- Erzeugen eines neuen Pfades

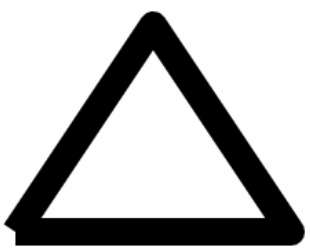
  `newpath`

  geschieht automatisch nach jedem `stroke` oder `fill`

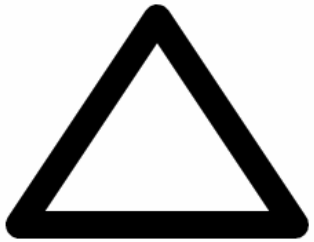
# Geschlossene Grafik-Objekte

- Beispiel 1

```
20 setlinewidth
1 setlinejoin
100 100 moveto
300 100 lineto
200 250 lineto
100 100 lineto
stroke
```



```
20 setlinewidth
1 setlinejoin
100 100 moveto
300 100 lineto
200 250 lineto
closepath
stroke
```



- Beispiel 2

```
10 setlinewidth
0.5 setgray
100 100 moveto
300 100 lineto
200 250 lineto
closepath
stroke
```

```
1 0.5 0
 setrgbcolor
100 100 moveto
300 100 lineto
200 250 lineto
closepath
fill
```

# Lineare Transformationen

- Verändern des lokalen Koordinatensystems
- Verschieben aller nachfolgenden Objekte um (x, y)

```
x y translate
```

- Drehen um den Winkel s gegen den Uhrzeigersinn

```
s rotate
```

- Skalieren um die Faktoren a und b in x- und y-Richtung

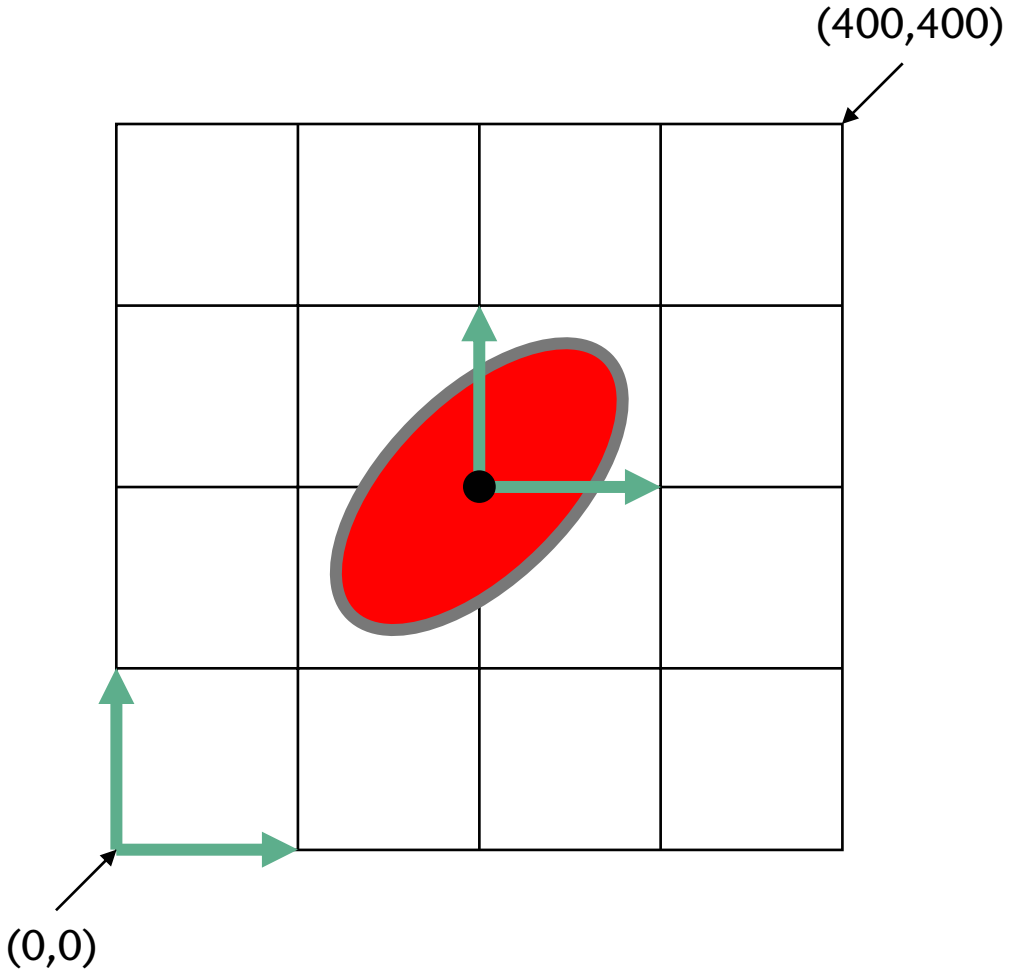
```
a b scale
```

  - wirkt sich auch auf die Liniendicke aus
  - Kreise werden zu Ellipsen

# Lineare Transformationen

- Beispiel

```
200 200 translate
45 rotate
2 1 scale
10 setlinewidth
0.5 setgray
newpath
0 0 50 0 360 arc
closepath
stroke
1 0 0 setrgbcolor
0 0 50 0 360 arc
fill
0.5 1 scale
-45 rotate
-200 -200 translate
0 setgray
200 200 10 0 360 arc
fill
```



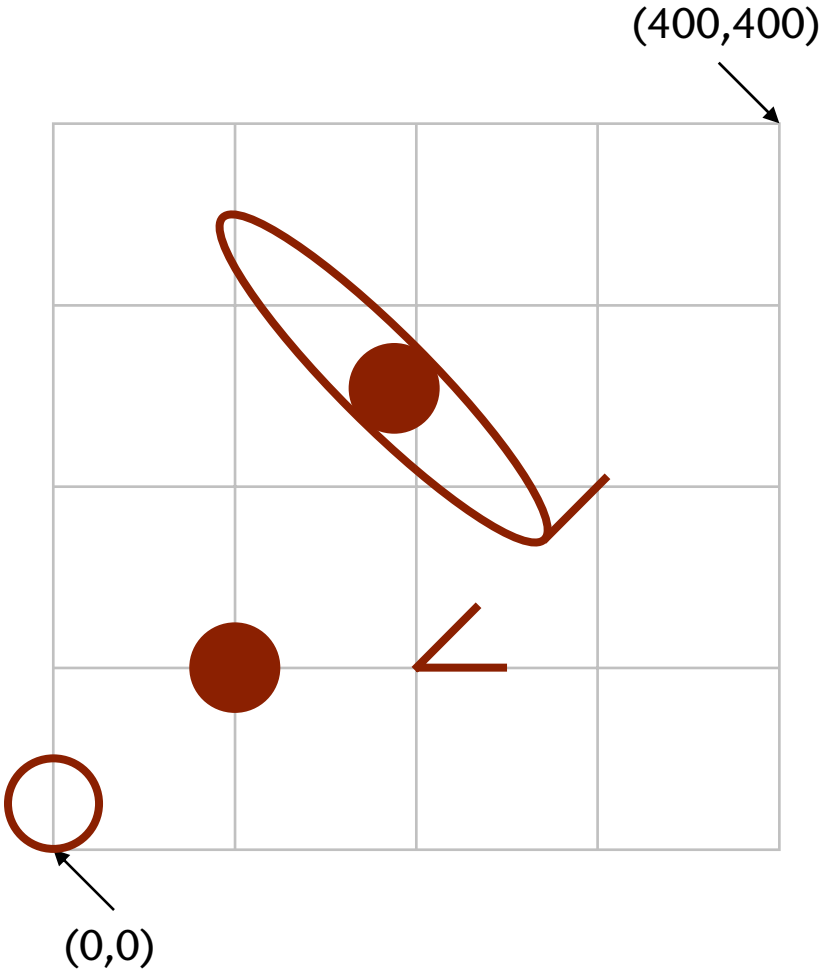
# Lineare Transformationen

- Zwei Sichtweisen möglich
- lokal
  - zeichne Objekte im aktuellen lokalen Koordinatensystem
  - Transformationen verändern das lokale Koordinatensystem
  - und wirken auf **alle** nachfolgenden Objekte
- global
  - zeichne Objekte in globalem Koordinatensystem
  - und wende alle vorigen Transformationen in umgekehrter Reihenfolge an

# Lineare Transformationen

- Beispiel

```
100 100 25 0 360 arc
fill
200 100 translate
0 0 moveto 50 0 rlineto
stroke
45 rotate
0 0 moveto 50 0 rlineto
stroke
100 0 translate
0 0 moveto 50 0 rlineto
stroke
1 5 scale
0 25 25 0 360 arc
stroke
1 0.2 scale
0 125 25 0 360 arc
fill
```



# Lineare Transformationen

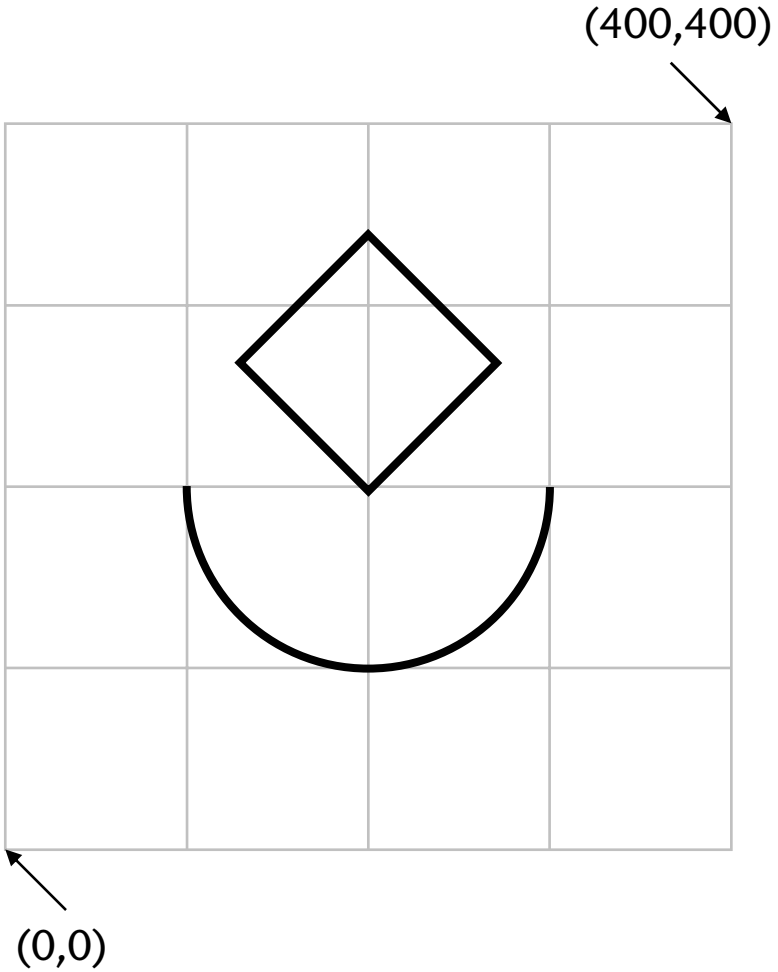
- Transformation(en) nur für ein Objekt
  - Transformation(en) ausführen
  - Objekt zeichnen
  - inverse Transformation(en) (in umgekehrter Reihenfolge) ausführen
- Alternative (besser)
  - Speichern des aktuellen Zustands `gsave`
    - inklusive aller anderen Graphikattribute
  - Wiederherstellen des Zustands `grestore`

# Lineare Transformationen

- Beispiel

```
200 200 translate
45 rotate
newpath
0 0 moveto 100 0 rlineto
0 100 rlineto -100 0 rlineto
closepath stroke
-45 rotate
0 0 100 180 360 arc stroke
```

```
200 200 translate
gsave
45 rotate
newpath
0 0 moveto 100 0 rlineto
0 100 rlineto -100 0 rlineto
closepath stroke
grestore
0 0 100 180 360 arc stroke
```



# Makros

- mehrfaches Ausführen derselben Befehlssequenz
- Beispiel
  - rotes Quadrat mit schwarzem Rand
- Sequenz als Makro zusammenfassen

```
/square {
  newpath
  0 0 moveto  100 0 lineto
  100 100 lineto  0 100 lineto
  closepath
} def
```

- allgemeine Form

  `/makroname { Befehlssequenz } def`

```
1 0 0 setrgbcolor

square

fill

0 setgray

square

stroke
```

# Makros

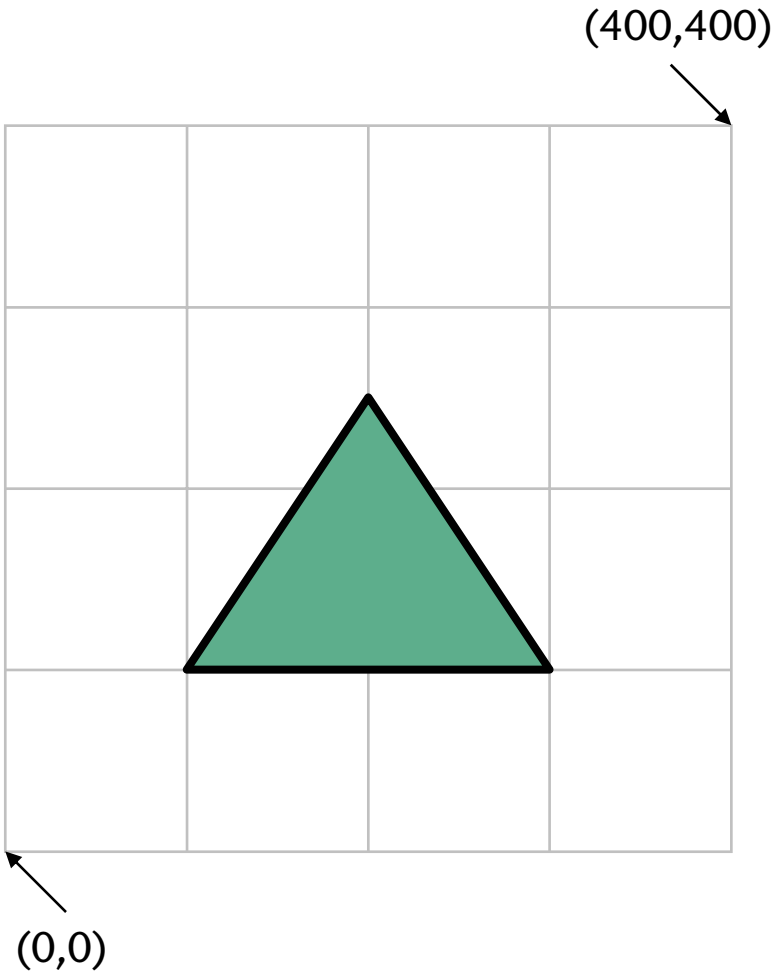
- Beispiel

```
/green {0 1 0 setrgbcolor} def
/black {0 setgray} def

/triangle {
  newpath
  0 0 moveto
  200 0 lineto
  100 150 lineto
  closepath
} def

%%EndProlog
```

```
100 100 translate
green triangle fill
black triangle stroke
```



# Makros

- Nachteil des Quadrat-Makros
  - feste Größe von 100 × 100
- wünschenswert wäre ein Makro für Quadrate beliebiger Größe n × n
- 1. Lösung
  - definiere Makro „n“ je nach Bedarf

    `/n 50 def` oder `/n 100 def` etc.
  - benutze Makro „n“ im Quadrat-Makro
- eleganter wäre allerdings der Aufruf

  `50 square` oder `100 square` etc.

```
/square {
  newpath
  0 0 moveto
  n 0 lineto
  n n lineto
  0 n lineto
  closepath
} def
```

```
100 100 translate
/n 50 def
square fill

100 0 translate
/n 100 def
square stroke

0 150 translate
/n 50 def
square fill
```

# Makros mit Argumenten = Funktionen

- genaue Funktionsweise von `def`
  - holt die beiden obersten Elemente vom Stapel
  - definiert zweitoberstes durch oberstes
- neuer Befehl `exch`
  - vertauscht die beiden obersten Elemente des Stapels
- 2. Lösung
  - verschiebe die Definition des Makros „n“ in das Quadrat-Makro
- benutze `1 dict begin` ... `end` zur Vermeidung von Namenskonflikten

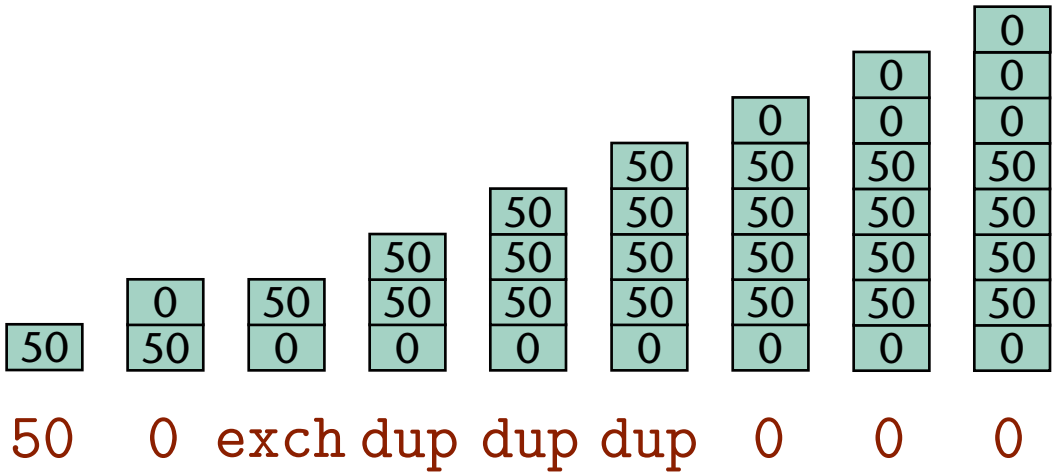
```
/square {
 1 dict begin
  /n exch def
  newpath
  0 0 moveto
  n 0 lineto
  n n lineto
  0 n lineto
  closepath
 end
} def
/n 50 def
100 100 translate
50 square fill

100 0 translate
100 square stroke

0 150 translate
50 square fill
```

# mehr Stapel-Operationen

- neuer Befehl `dup`
  - legt ein Kopie des obersten Stapelelements auf den Stapel
- 3. Lösung
  - geschicktes Verdoppeln und Vertauschen
  - kommt ohne lokale Variable aus




```
/square {
  newpath
  0 exch dup dup
  dup 0 0 0
  moveto lineto
  lineto lineto
  closepath
} def
```

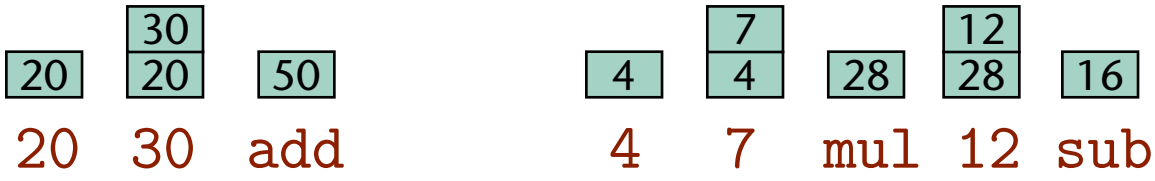
```
50 square
```

=

```
newpath
0 0 moveto
50 0 lineto
50 50 lineto
0 50 lineto
closepath
```

# Rechnen in PostScript

- Arithmetische Operatoren
  - holen Operanden vom Stapel
  - legen Ergebnis wieder auf den Stapel
- Beispiel




- wichtigste Operatoren

`x y add` → $x+y$, `x y sub` → $x-y$, `x neg` → $-x$,

`x y mul` → $x*y$, `x y div` → $x/y$, `x abs` → $|x|$,

`x sin` → $\sin(x)$, `x cos` → $\cos(x)$, `x sqrt` → $\sqrt{x}$,

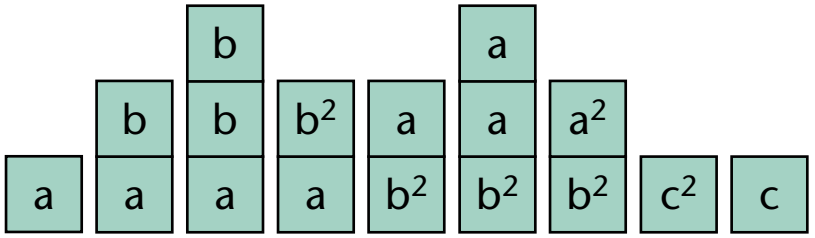
`x y atan` → $\arctan(x/y)$, `x floor` → $\lfloor x \rfloor$

# Rechnen in Postscript

- Beispiel 1

```
/hypotenuse {
  dup mul exch dup mul add sqrt
} def

a b hypotenuse
```

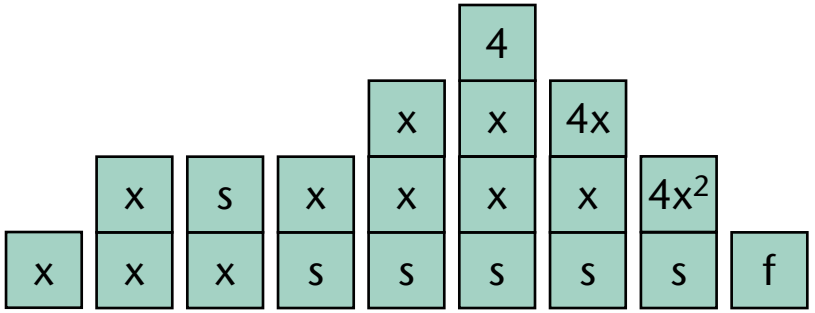



berechnet $c = \sqrt{a^2 + b^2}$

- Beispiel 2

```
/f {
  dup sin exch dup 4 mul mul add
} def

x f
```




berechnet $f(x) = 4x^2 + \sin(x)$

# Zusammenfassung

- mit den Makros

```
/red   {1 0 0 setrgbcolor} def
/green {0 1 0 setrgbcolor} def
/blue  {0 0 1 setrgbcolor} def
/black {0 setgray} def
```

und den Funktionen

```
/square { ... } def

/fsquare {
  dup square fill
  black square stroke
} def

/hypotenuse { ... } def
```

können wir nun ganz leicht die Pythagoras-Figur erstellen

```
/pythagoras {
 3 dict begin
  /b exch def
  /a exch def
  /c a b hypotenuse def
  gsave
   0 c neg translate
   green c fsquare
   0 c translate
   b a atan rotate
   red a fsquare
   a 0 translate
   -90 rotate
   blue b fsquare
  grestore
 end
} def

200 300 translate
100 150 pythagoras
```

# Schleifen in PostScript

- einfachste Art der Schleife

  `n { Befehlssequenz } repeat`

  führt die gegebene Befehlssequenz n-mal aus

- Beispiel

```
/square {
  newpath
  0 exch dup dup dup 0 0 0
  moveto lineto lineto lineto
  closepath
} def
```

=

```
/square {
  newpath
  0 exch dup dup dup 0 0 0
  moveto 3 {lineto} repeat
  closepath
} def
```

- weitere Schleifen- und Verzweigungsbefehle

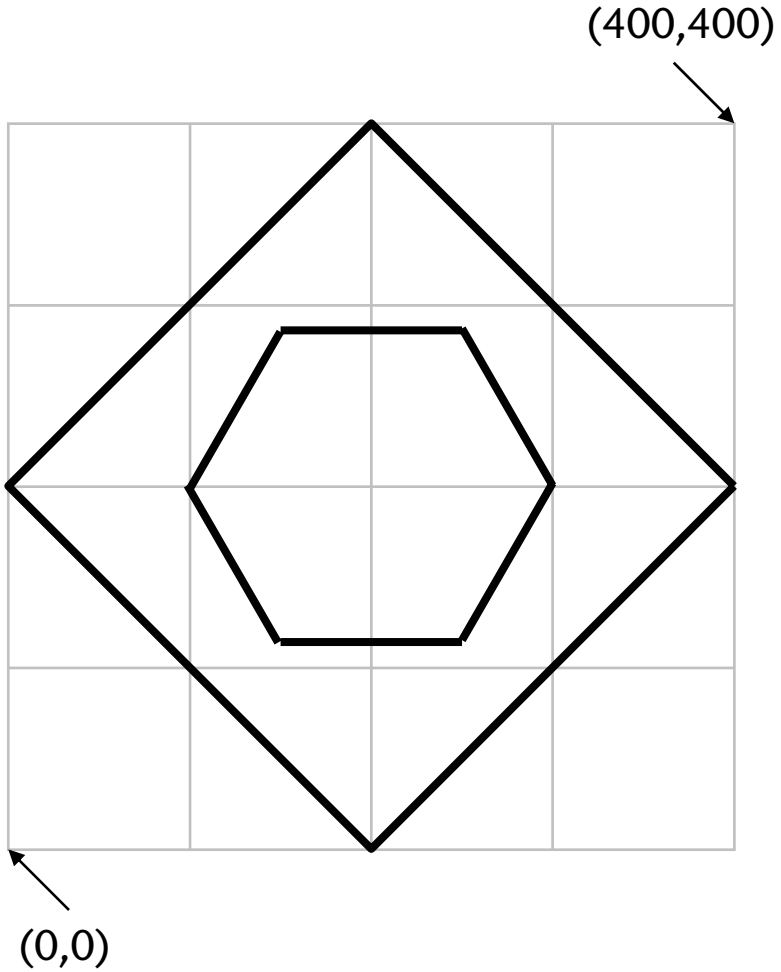
  `for`, `if`, `ifelse`, `loop`, `exit`, ... fast wie in C/C++

# Schleifen in PostScript

- Beispiel 1

```
/n-gon {
 3 dict begin
  /l exch def
  /n exch def
  /a 360 n div def
  newpath
  l 0 moveto
  n {
    a rotate
    l 0 lineto
  } repeat
  closepath
 end
} def
```

```
200 200 translate
4 200 n-gon stroke
6 100 n-gon stroke
```

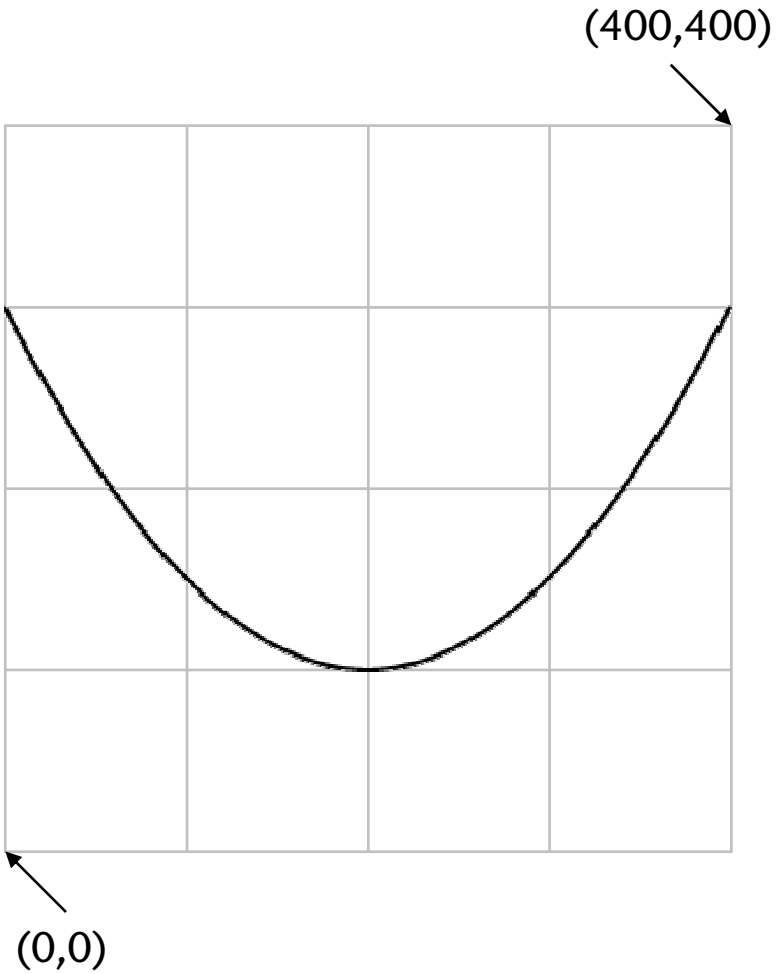


# Schleifen in PostScript

- Beispiel 2

```
/f {dup mul} def
/plot-f {5 dict begin
  /n exch def
  /max exch def
  /min exch def
  /x min def
  /dx max min sub n div def
  x dup f moveto
  n {
    /x x dx add def
    x dup f lineto
  } repeat
  stroke
end} def
```

```
200 100 translate 200 200 scale
0.01 setlinewidth
-1 1 100 plot-f
```



# Textausgabe in PostScript

- Setzen eines Zeichensatzes „Font“ in Größe p

```
/Font findfont p scalefont setfont
```

  - Verfügbare Zeichensätze: „Times“ und „Helvetica“, optional mit Ergänzung „-Oblique“ (kursiv) oder „-Bold“ (fett)

- Ausgabe der Zeichenkette „text“

```
(text) show
```

- Umwandlung einer Zahl „x“ in eine Zeichenkette

```
/str 100 string def
x str cvs
```

- Umwandlung der Zeichenkette „text“ in einen Pfad

```
(text) false charpath
```

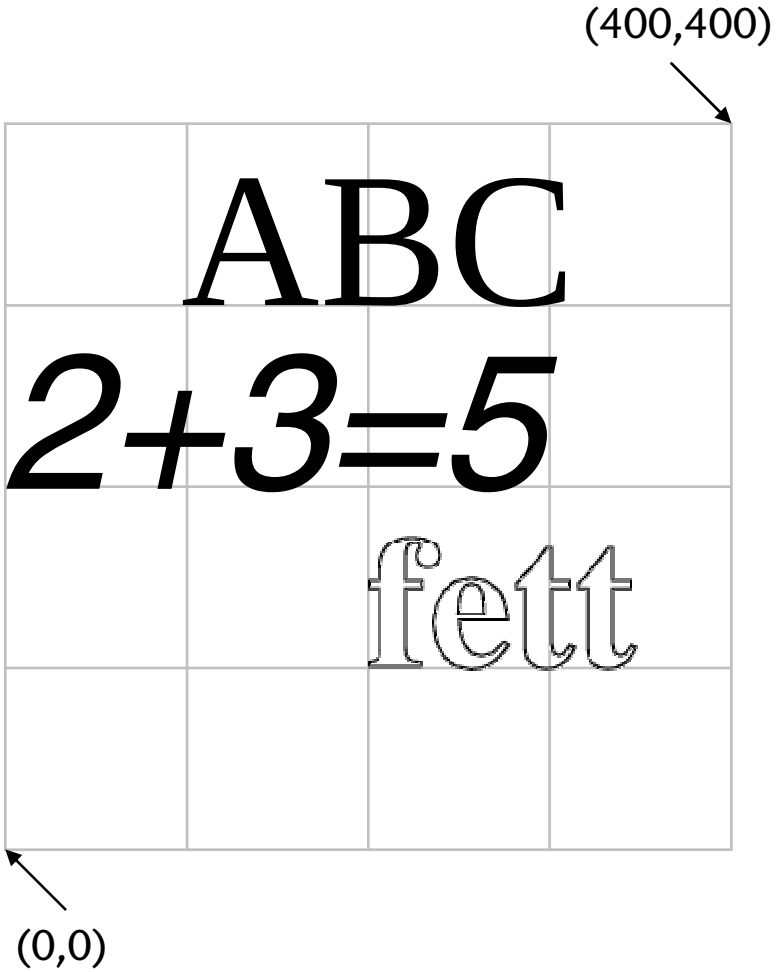
# Textausgabe in PostScript

- Beispiel

```
/Times findfont
100 scalefont setfont
100 300 moveto
(ABC) show

/Helvetica-Oblique findfont
100 scalefont setfont
0 200 moveto
/str 10 string def
2 str cvs show (+) show
3 str cvs show (=) show
5 str cvs show

/Times-Bold findfont
100 scalefont setfont
100 100 moveto
(fett) false charpath
stroke
```



# Zusammenfassung

- Programmiersprache PostScript
  - Beschreiben von 2D-Vektorgrafiken
  - auflösungs- und geräteunabhängig
  - Interpretersprache
- Aufbau eines PostScript-Programms
  - Header
  - Prolog
  - Skript
  - Tailer
- einfache Syntax

# Zusammenfassung

- Befehle mit $n \geq 0$ Operanden $o_i$ in Postfixnotation

  `o1 o2 ... on befehl`

- elementare Zeichenbefehle

  `moveto`, `lineto`, `rmoveto`, `rlineto`, `arc`, `arcn`, `curveto`, `strike`, `fill`, `newpath`, `closepath`

- Verändern der Zeichenattribute

  `setlinewidth`, `setdash`, `setlinecap`, `setlinejoin`, `setgray`, `setrgbcolor`

- Verändern des Koordinatensystems

  `translate`, `rotate`, `scale`

# Zusammenfassung

- Definition von Makros

  `/makroname { Befehlssequenz } def`

- wichtigste Stapeloperatoren

  `exch`, `dup`

- arithmetische Operatoren

  `add`, `sub`, `mul`, `div`, `sin`, ...

- einfachste Art der Schleife

  `n { Befehlssequenz } repeat`

- Text- und Zahlenausgabe

  `(text) show` bzw. `n 100 string cvs show`