# Vorlesung vom 13.01.2003: Grafikprogrammierung II Animation und Interaktion

# Inhalt

- Erweiterung der Geometriebücherei
- Bewegte Grafiken: Animationen
- *Labelled Records*
- Space — the final frontier

# Kollisionserkennung

- Ziel: Interaktion der Figuren erkennen
  - . . . zum Beispiel Überschneidung
  - Eine Möglichkeit: `Region` (aus HGL)
  - Elementare Regionen:

    ```
    emptyRegion   :: Region
    polygonRegion :: [Point]-> Region
    ```

  - Vereinigung, Schnitt, Subtraktion:

    ```
    unionRegion     :: Region-> Region-> Region
    intersectRegion :: Region-> Region-> Region
    ```

  - aber leider nur Funktion nach `Graphic`, nicht `isEmpty :: Region-> Bool` oder `contains :: Region-> Point-> Bool`

- Deshalb: eigene Implementation
  - Idee: zur Darstellung wird Normalform berechnet
  - Normalform auch zur Kollisionserkennung nutzen
- Normalform: Elementare Figuren (`Shape`)
  - einfach zu zeichnen
  - einfache Kollisionserkennung
- Konservative Erweiterung des `Geometry`-Moduls
  - Alte Schnittstelle bleibt erhalten.

# Erweitung der Schnittstelle

- Abstrakter Typ `Shape`
- Konversion von Figur in Form:

```
shape :: Figure-> Shape
```

- Form zeichnen:

```
drawShape :: Shape-> Graphic
```

- Kollisionserkennung:

```
contains  :: Shape-> Point-> Bool
intersect :: Shape-> Shape-> Bool
```

- Gesamte Schnittstelle

# Implementierung

- Ein `Shape` ist
  - ein geschlossenes Polygon oder ein Kreis
  - mit normalisierten Koordinaten.

```
data Shape = Poly [Point]
           | Circ Point Double
           deriving (Eq, Show)
```

- Konversion `Figure` nach `Shape`:
  - Translation, Rotation, Skalierung herausrechnen;
  - Adaption von draw (Letzte VL)

```
shape :: Figure-> Shape
shape = fig' ((0, 0), 1, 0) where
  fig' :: (Point, Double, Angle)-> Figure-> Shape
  fig' (m, r, phi) (Translate t f) =
    fig' (add m (smult r (rot phi t)), r, phi) f
  fig' (m, r, phi) (Scale s f) =
    fig' (m, r* s, phi) f
  fig' (m, r, phi) (Rotate w f) =
    fig' (m, r, phi+ w) f
  fig' c (Rect a b) =
    poly c [(x2, y2), (-x2, y2),
                (-x2, -y2), (x2, -y2)] where
      x2= a `div` 2; y2= b `div` 2
  fig' c (Triangle l1 a l2) =
    poly c [(0, 0), (0, l1), rot a (0, l2)]
```

```
fig' c (Polygon pts) = poly c pts
fig' (m, r, _) (Circle d) =
  Circ m (r*fromInt d)
poly :: (Point, Double, Angle)-> [Point]-> Shape
poly (m, p, w) = Poly. checkclosed.
                   map (add m. smult p. rot w) where
```

- Prüfung ob Polygon geschlossen

```
checkclosed [] = []
checkclosed x  = if (head x) == (last x)
                 then x else x++ [head x]
```

# Formen zeichnen

- Form zeichnen ist trivial:

```
drawShape :: Shape-> Graphic
drawShape (Poly pts) = polygon pts
drawShape (Circ (mx, my) r) =
    ellipse (mx-r', my- r') (mx+ r', my+ r') where
      r'= round r
```

- Alte Funktion `draw :: Figure-> Graphic`

```
draw :: Figure-> Graphic
draw = drawShape . shape
```

# Kollisionserkennung

- Fallunterscheidung:
  - Polygon und Polygon
    - ▷ Kollision, wenn ein Eckpunkt des einen im anderen
    - ▷ Voraussetzung: Polygone konvex
  - Polygon und Kreis
    - ▷ Kollision, wenn ein Eckpunkt im Kreis
  - Kollision Kreis und Kreis
    - ▷ Kollision, wenn Entfernung der Mittelpunkte kleiner als Summe der Radien
- Erster Schritt: Kollision von Formen und Punkten.

# Kollisionserkennung: Punkte und Kreise

- Punkt ist innerhalb des Kreises gdw.
  Abstand zum Mittelpunkt kleiner (gleich) Radius

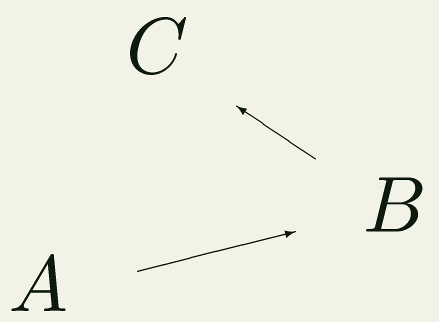
```
inC :: Point-> Double-> Point-> Bool
inC (mx, my) r (px, py) =
    len (px- mx, py- my) <= r
```

- Abstand ist Länge (Betrag) des Differenzvektors:

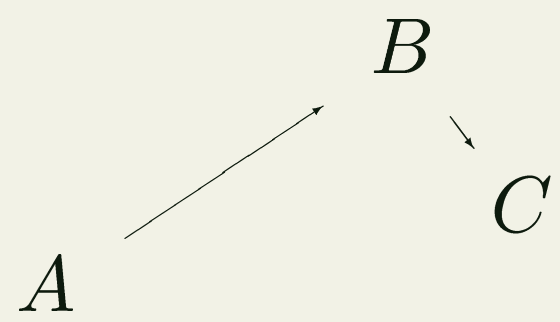
```
len :: Point-> Double
len (x, y)= sqrt (fromInt (x^2+ y^2))
```

# Kollisionserkennung: Punkte und Polygone

- Hilfsmittel: Orientierung eines Punktes
- Orientierung von $C$ bez. Strecke $\overline{AB}$:




Positive Orientierung




Negative Orientierung

- Punkt ist innerhalb eines Polygons, wenn gleiche Orientierung bez. aller Seiten.

- Orientierung ist Vorzeichen der Determinante:

$$D_{A,B,C} = (B_y - A_y)(C_x - B_x) - (C_y - B_y)(B_x - A_x)$$

  - Falls $D_{A,B,C} < 0$, dann Orientierung positiv
  - Falls $D_{A,B,C} > 0$, dann Orientierung negativ
  - Falls $D_{A,B,C} = 0$, dann Punkte in einer Flucht

```
det :: Point-> (Point, Point)-> Int
det (cx,cy) ((ax,ay), (bx,by)) =
  signum ((by-ay)*(cx-bx)-(cy-by)*(bx-ax))
```

  - `signum` ist Vorzeichen

- Punkt ist innerhalb eines Polygon, wenn gleiche Orientierung bez. aller Seiten.
  - Voraussetzung: Polygon ist konvex
  - Punkte auf den Seiten werden nicht erkannt
  - Hilfsmittel: Orientierung eines Punktes.
- Hilfsfunktion: Liste der Seiten eines Polygons
  - Voraussetzung: Polygon geschlossen

```
sides :: [Point]-> [(Point, Point)]
sides []       = []
sides (x:[])  = []
sides (x:xs)  = (x, head xs):(sides xs)
```

- Damit Hauptfunktion:
  - aus Polygon Liste der Seiten bilden,
  - Determinante aller Seiten bilden,
  - doppelte Vorkommen löschen;
  - Ergebnisliste hat Länge 1 gdw. gleiche Orientierung für alle Seiten

```
inP :: [Point]-> Point-> Bool
inP ps c =
  (length. nub. map (det c). sides) ps == 1
```

  - `nub :: Eq a=> [a]-> [a]` entfernt doppelte Elemente
  - Ineffizient — `length` berechnet immer Länge der ganzen Liste.

# Kolliosionserkennung für Punkte

- Einfache Fallunterscheidung

```
contains :: Shape-> Point-> Bool
contains (Poly pts)= inP pts
contains (Circ c r)= inC c r
```

# Kollisionserkennung von Formen

- Kreise: Distanz Mittelpunkte kleiner als Summe Radien

```
intersect :: Shape-> Shape-> Bool
intersect (Circ (mx1, my1) r1)
          (Circ (mx2, my2) r2)=
   len (mx2- mx1, my2- my1) <= r1+ r2
```

- Polygone: Eckpunkt des einem im anderen
  - Beachtet Randfall nicht: Polygone berühren sich.

```
intersect (Poly p1) (Poly p2)=
   any (inP p1) p2 || any (inP p2) p1
```

- Polygon und Kreis: ein Eckpunkt im Kreis
  - Beachtet Randfall nicht: Kreis schneidet nur Seite

```
intersect (Poly p) (Circ c r)=
   inP p c || any (inC c r) p
intersect (Circ c r) (Poly p)=
   inP p c || any (inC c r) p
```

# Noch eine Hilfsfunktion

- Polarkoordinaten $P = (r, \phi)$
- Konversion in kartesische Koordinaten:

  Punkt $(r, 0)$ um Winkel $\phi$ drehen.

```
polar :: Double-> Angle-> Point
polar r phi = rot phi (round r, 0)
```

# Benutzereingaben: `Events`

- Benutzereingabe:
  - Tasten
  - Mausbewegung
  - Mausknöpfe
  - Fenster: Größe verändern, schließen
- Grundliegende Funktionen:
  - Letzte Eingabe, auf nächste Eingabe warten:

    ```
    getWindowEvent :: Window -> IO Event
    ```
  - Letzte Eingabe, nicht warten:

    ```
    maybeGetWindowEvent :: Window-> IO (Maybe Event)
    ```

- Event ist ein labelled record:

```
data Event
   = Char      { char :: Char }
   | Key       { keysym :: Key, isDown :: Bool }
   | Button    { pt :: Point,
                 isLeft, isDown :: Bool }
   | MouseMove { pt :: Point }
   | Resize
   | Closed
  deriving Show
```

- Was ist das ?!?

# Probleme mit großen Datentypen

- Beispiel Warenverwaltung
  - Ware mit Bezeichung, Stückzahl, Preis (in Cent)

```
data Item = Item String Int Int
```

  - Kommt Stückzahl oder Preis zuerst?

- Beispiel Buch:
  - Titel, Autor, Verlag, Signatur, Fachgebiet, Stichworte

```
data Book' = Book' String String String String Str
```

  - Kommt Titel oder Autor zuerst?
    Ist Verlag das dritte oder vierte Argument?

# Probleme mit großen Datentypen

- Reihenfolge der Konstruktoren.
  - Typsynonyme (`type Author= String`) helfen nicht
  - Neue Typen (`data Author = Author String`) zu umständlich
- Selektion und Update
  - Für jedes Feld einzeln zu definieren.

```
getSign :: Book'-> String
getSign (Book' _ _ _ s _) = s
setSign :: Book'-> String-> Book'
setSign (Book' t a p _ f) s = Book' t a p s f
```

- Inflexibilität
  - Wenn neues Feld hinzugefügt wird, alle Konstruktoren ändern.

# Lösung: *labelled records*

- Algebraischer Datentyp mit benannten Feldern
- Beispiel:

```
data Book = Book { author :: String,
                   title  :: String,
                   publisher :: String }
```

- Konstruktion:

```
b = Book
  {author = "M. Proust",
   title  = "A la recherche du temps perdu",
   publisher = "S. Fischer Verlag"}
```

- Selektion durch Feldnamen:

```
publisher b --> "S. Fischer Verlag"
author b    --> "M. Proust"
```

- Update:

```
b{publisher = "Rowohlt Verlag"}
```

  - Rein funktional! (`b` bleibt unverändert)

- Patternmatching:

```
print :: Book-> IO ()
print (Book{author= a, publisher= p, title= t}) =
  putStrLn (a++ " schrieb "++ t ++ " und "++
            p++ " veröffentlichte es.")
```

- Partielle Konstruktion und Patternmatching möglich:

```
b2 = Book {author= "C. Lüth"}
shortPrint :: Book-> IO ()
shortPrint (Book{title= t, author= a}) =
  putStrLn (a++ " schrieb "++ t)
```

  ◦ Guter Stil: nur auf benötigte Felder matchen.

- Datentyp erweiterbar:

```
data Book = Book {author :: String,
                  title  :: String,
                  publisher :: String,
                  signature :: String }
```

  Programm muß nicht geändert werden (nur neu übersetzt).

# Zusammenfassung labelled records

- Reihenfolge der Konstruktorargumente irrelevant
- Generierte Selektoren und Update-Funktionen
- Erhöht Programmlesbarkeit und Flexibilität
- NB. Feldnamen sind Bezeichner
  - Nicht zwei gleiche Feldnamen im gleichen Modul
  - Felder nicht wie andere Funktionen benennen

# Animation

Alles dreht sich, alles bewegt sich. . .

- Animation: über der Zeit veränderliche Grafik
- Unterstützung von Animationen in HGL:
  - `Timer` ermöglichen getaktete Darstellung
  - Gepufferte Darstellung ermöglicht flickerfreie Darstellung
- Öffnen eines Fensters mit Animationsunterstützung:
  - Initiale Position, Grafikzwischenpuffer, Timer-Takt in Millisekunden

```
openWindowEx :: Title-> Maybe Point-> Size->
                RedrawMode-> Maybe Time-> IO Window
data RedrawMode
  = Unbuffered | DoubleBuffered
```

# Eine einfache Animation

- Ein springender Ball:
  - Ball hat Position und Geschwindigkeit:

```
data Ball = Ball { p :: Point,
                   v :: Point }
```

  - Ball zeichnen: Roter Kreis an Position $\vec{p}$

```
drawBall :: Ball-> Graphic
drawBall (Ball {p= p}) =
  withColor Red
    (draw (Translate p (Circle 20)))
```

- Ball bewegen:
  - Geschwindigkeit $\vec{v}$ zu Position $\vec{p}$ addieren
  - In X-Richtung: modulo Fenstergröße 500
  - In Y-Richtung: wenn Fensterrand 500 erreicht, Geschwindigkeit invertieren
  - Geschwindigkeit in Y-Richtung nimmt immer um 1 ab

```
move :: Ball-> Ball
move (Ball {p= (px, py), v= (vx, vy)})=
  Ball {p= (px', py'), v= (vx, vy')} where
    px' = (px+ vx) `mod` 500
    py0 = py+ vy
    py' = if py0> 500 then 500-(py0-500) else py0
    vy' = (if py0> 500 then -vy else vy)+ 1
```

- Hauptprogramm:
  - Fenster öffnen

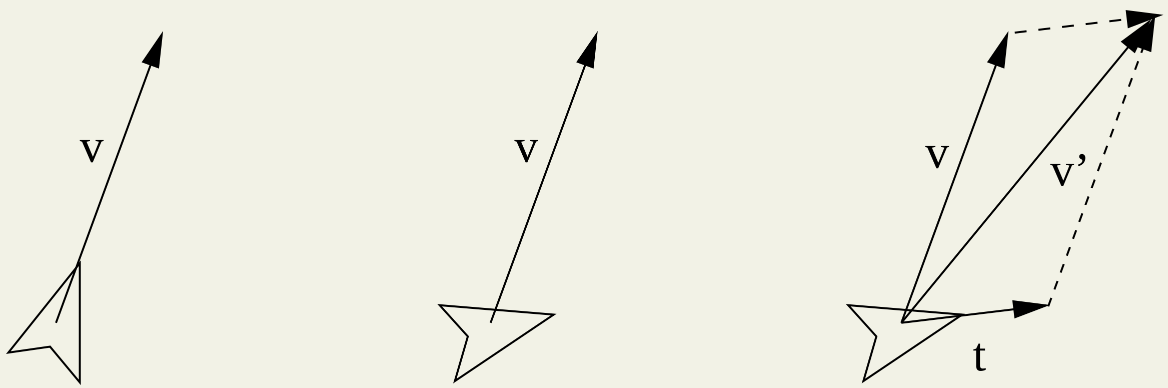
```
main :: IO ()
main = runGraphics $
  do w<- openWindowEx "Bounce!"
                      Nothing (500, 500) DoubleBuffered
                      (Just 30)
     loop w (Ball{p=(0, 10), v= (5, 0)}) where
```

  - Hauptschleife: Ball zeichnen, auf Tick warten, Folgeposition berechnen

```
loop :: Window-> Ball-> IO ()
loop w b =
  do setGraphic w (drawBall b)
     getWindowTick w
     loop w (move b)
```

# Ein Raumflugsimulator

- Ziel: Simulation eines Raumschiffs
- Steuerung nur mit Schub und Drehung




- Geschwindigkeit $\vec{v}$, Schub $\vec{t}$
  - Schub operiert immer in Richtung der Orientierung

- Zustand des Schiffes zu gegebener Zeit:
  - Position $\vec{p} \in \mathbb{R}^2$
  - Geschwindigkeit $\vec{v} \in \mathbb{R}^2$
  - Orientierung $\phi \in \mathbb{R}$ (als Winkel)
  - Schub $s \in \mathbb{R}$ (als Betrag; $\vec{t}$ ist $polar(s, \phi)$)
  - Winkelbeschleunigung $\omega \in \dot{\mathbb{R}}$
- Zustandsübergang:
  - Neue Position: $\vec{p}' = \vec{p} + \vec{v}'$
  - Neue Geschwindigkeit: $\vec{v}' = \vec{v} + polar(s, \phi)$
  - Neue Orientierung: $\phi' = \phi + \omega$
  - Winkelbeschleunigung und Schub: durch Benutzerinteraktion

- Benutzerinteraktion:
  - Konstanten $W$ für Winkelbeschleunigung, $T$ für Schub
  - Tasten für $Links$, $Rechts$, $Schub$
  - $Links$ drücken: Winkelbeschleunigung auf $+W$ setzen
  - $Links$ loslassen: Winkelbeschleunigung auf $0$ setzen
  - $Rechts$ drücken: Winkelbeschleunigung auf $-W$ setzen
  - $Rechts$ loslassen: Winkelbeschleunigung auf $0$ setzen
  - $Schub$ drücken: Schub auf $T$ setzen
  - $Schub$ loslassen: Schub auf $0$ setzen
- Neuer Zustand:
  Zustandsübergang plus Benutzerinteraktion.

- Modellierung des Gesamtsystems
  - Für den Anfang nur das Schiff:

```
data State = State { ship  :: Ship }
```

  - Schiffszustand:
    - ▷ `Shape` merken für effiziente Kollisionserkennung!

```
data Ship =
   Ship { pos     :: Point,
          shp     :: Shape,
          vel     :: Point,
          ornt    :: Double,
          thrust  :: Double,
          hAcc    :: Double }
```

- Globale Konstanten
  - Das Raumschiff

```
spaceShip :: Figure
spaceShip = Polygon [(15, 0), (-15, 10),
                     (-10, 0), (-15, -10), (15, 0)]
```

  - Fenstergröße

```
winSize :: (Int, Int)
winSize = (800, 600)
```

  - Schub

```
aDelta :: Double
aDelta = 1
```

  - Maximale Geschwindigkeit

```
vMax :: Double
vMax = 20
```

- Winkelbeschleunigung

```
hDelta :: Double
hDelta = 0.3
```

- Der Anfangszustand: Links oben, nach Süden gerichtet

```
initialState :: State
initialState =
  State {ship= setShp $
     Ship{pos= (40, 40),
          vel= (0, 0), ornt= -pi/2,
          thrust= 0, hAcc= 0}}
```

- Neuen Schiffszustand berechnen
  - Geschwindigkeit so verändern, dass Betrag Obergrenze $v_{max}$ nie überschreitet.

```
moveShip :: Ship-> Ship
moveShip(Ship {pos= pos0, vel= vel0,
               hAcc= hAcc, thrust= t, ornt= o}) =
  setShp $
    Ship{pos= addWinMod pos0 vel0,
         vel= if l> vMax then smult (vMax/l) vel1
                         else vel1,
         thrust= t, ornt= o+ hAcc, hAcc= hAcc} where
           vel1= add (polar t o) vel0
           l   = len vel1
```

- Shape berechnen und setzen:
  - `spaceShip` ist das Aussehen des Schiffes (globale Konstante)
  - Um Orientierung drehen und an Position verschieben.

```
setShp :: Ship-> Ship
setShp s = s{shp= shape (Translate (pos s)
                         (Rotate (ornt s) spaceShip))}
```

  - Vektoraddition modulo Fenstergröße:

```
addWinMod :: (Int,Int) -> (Int,Int) -> (Int,Int)
addWinMod (a, b) (c, d)=
  ((a+ c) `mod` (fst winSize),
   (b+ d) `mod` (snd winSize))
```

- Systemzustand darstellen
  - Gesamter Systemszustand

```
drawState :: State-> Graphic
drawState s = drawShip (ship s)
```

    ▷ Weitere Objekte mit `overGraphics` kombinieren

  - Schiff darstellen (Farbänderung bei Beschleunigung)

```
drawShip :: Ship-> Graphic
drawShip s =
   withColor (if thrust s> 0 then Red else Blue)
             (drawShape (shp s))
```

- Neuen Systemzustand berechnen:
  - Hauptschleife: zeichnen, auf nächsten Tick warten, Benutzereingabe lesen, Folgezustand berechnen

```
loop :: Window-> State-> IO ()
loop w s =
  do setGraphic w (drawState s)
     getWindowTick w
     evs<- getEvs
     s'<- nextState evs s
     loop w s' where
```

- Liste aller Eingaben seit dem letzten Tick:

```
getEvs :: IO [Event]
getEvs = do x<- maybeGetWindowEvent w
            case x of
                Nothing -> return []
                Just e  -> do rest <- getEvs
                              return (e : rest)
```

- Folgezustand berechnen: später IO möglich (Zufallszahlen!)

```
nextState :: [Event]-> State-> IO State
nextState evs s =
  return s1{ship= moveShip (ship s1)} where
     s1= foldl (flip procEv) s evs
```

◦ Eine Eingabe bearbeiten:

```
procEv :: Event-> State-> State
procEv (Key {keysym= k, isDown=down})
    | isLeftKey k && down        = sethAcc hDelta
    | isLeftKey k && not down    = sethAcc 0
    | isRightKey k && down       = sethAcc (- hDelta)
    | isRightKey k && not down = sethAcc 0
    | isUpKey k && down          = setThrust aDelta
    | isUpKey k && not down      = setThrust 0
procEv _ = id
sethAcc :: Double->State-> State
sethAcc a s = s{ship= (ship s){hAcc= a}}
setThrust :: Double-> State-> State
setThrust a s = s{ship= (ship s){thrust= a}}
```

- Das Hauptprogramm

  ◦ Fenster öffnen, Schleife mit Anfangszustand starten

```
main :: IO ()
main = runGraphics $
  do w<- openWindowEx "Space --- The Final Frontier"
                      Nothing winSize DoubleBuffered
                      (Just 30)
     loop w initialState
     closeWindow w
```

# Zusammenfassung

- Erweiterung der Geometriebücherei
  - Der Datentyp `Shape` und Kollisionserkennung
- Neues Haskell-Konstrukt: labelled records
  - Reihenfolge der Konstruktorargumente irrelevant
  - Generierte Selektoren und Update-Funktionen
  - Erhöht Programmlesbarkeit und Flexibilität
- Animation:
  - Unterstützung in `HGL` durch Timer und Zeichenpuffer
  - Implementation eines einfachen Raumschiffsimulators