Vorlesung vom 24.01.2005: Grafikprogrammierung II Animation und Interaktion

Inhalt

- Erweiterung der Geometriebücherei
- Bewegte Grafiken: Animationen
- Labelled Records
- Haskell on the road

Refaktorierung der Geometriebücherei

- Basis (des Vektorraums) `Double` statt Int

  ```
  type Dimension = Double
  ```

- Rotation im math. positiven Drehsinn.
- Erweiterung Signature von `draw`:

  ```
  draw  :: G.Point-> Double-> Figure-> Graphic
  ```

- . . . und Kollisionserkennung
- Zeigen.

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`

Kollisionserkennung: Fälle

- Kreis und Kreis
  - Kollision, wenn Entfernung der Mittelpunkte kleiner als Summe der Radien
- Polygon und Kreis
  - Kollision, wenn ein Eckpunkt im Kreis
- Polygon und Polygon
  - Kollision, wenn ein Eckpunkt des einen im anderen
  - Voraussetzung: Polygone konvex
- Zuerst: Punkt und Figur, danach Figur und Figur

Kollisionserkennung: Punkte und Kreise

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

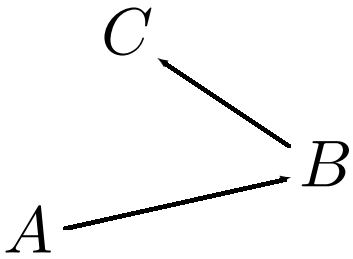
```
inC :: Point-> Double-> Point-> Bool
inC (mx, my) r (px, py) =
    len (px- mx, py- my) <= r
```

- Abstand ist Länge (Betrag) des Differenzvektors:

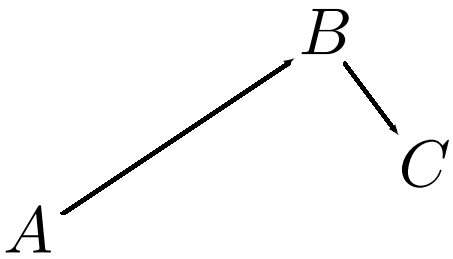
```
len :: Point-> Double
len (x, y)= sqrt (x^2+ y^2)
```

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)) =
  round (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.

Kollisionserkennung für Punkte

- Einfache Fallunterscheidung

```
contains :: Figure-> Point-> Bool
contains (Polygon pts)= inP pts
contains (Circle c r) = inC c r
```

Kollisionserkennung von Formen

- Kreise: Distanz Mittelpunkte kleiner als Summe Radien

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

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

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

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

```
intersect (Polygon p) (Circle c r)=
   inP p c || any (inC c r) p
intersect (Circle c r) (Polygon p)=
   inP p c || any (inC c r) p
```

Polarkoordinaten

- Polarkoordinaten: $P = (r, \phi)$
- Konversion in kartesische Koordinaten: $(r, 0)$ um Winkel ϕ drehen.

```
data Polar = Polar { angle :: Angle, dist  :: Dimension }
cart :: Polar-> Point
cart p = rot (angle p) (dist p, 0)
```

- Konversion in Polarkoordinaten: r Länge des Vektors, $\phi = \sin^{-1}(\frac{y}{r})$

```
polar :: Point-> Polar
polar (x, y)
  | (x, y) == (0, 0) = Polar {dist= r, angle = 0} -- beliebi
  | otherwise        = Polar {dist= r, angle= asin (y/r)}
                       where r = len (x, y)
```

Benutzereingaben

- Benutzerinteraktion:
  - 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)
    ```

Benutzereingaben: Events

- 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 umfangreichen 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 String
```

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

Probleme mit umfangreichen 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 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= (x, y)}) =
  withColor Red (ellipse (x- 10, y-10) (x+10, y+10))
```

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
```

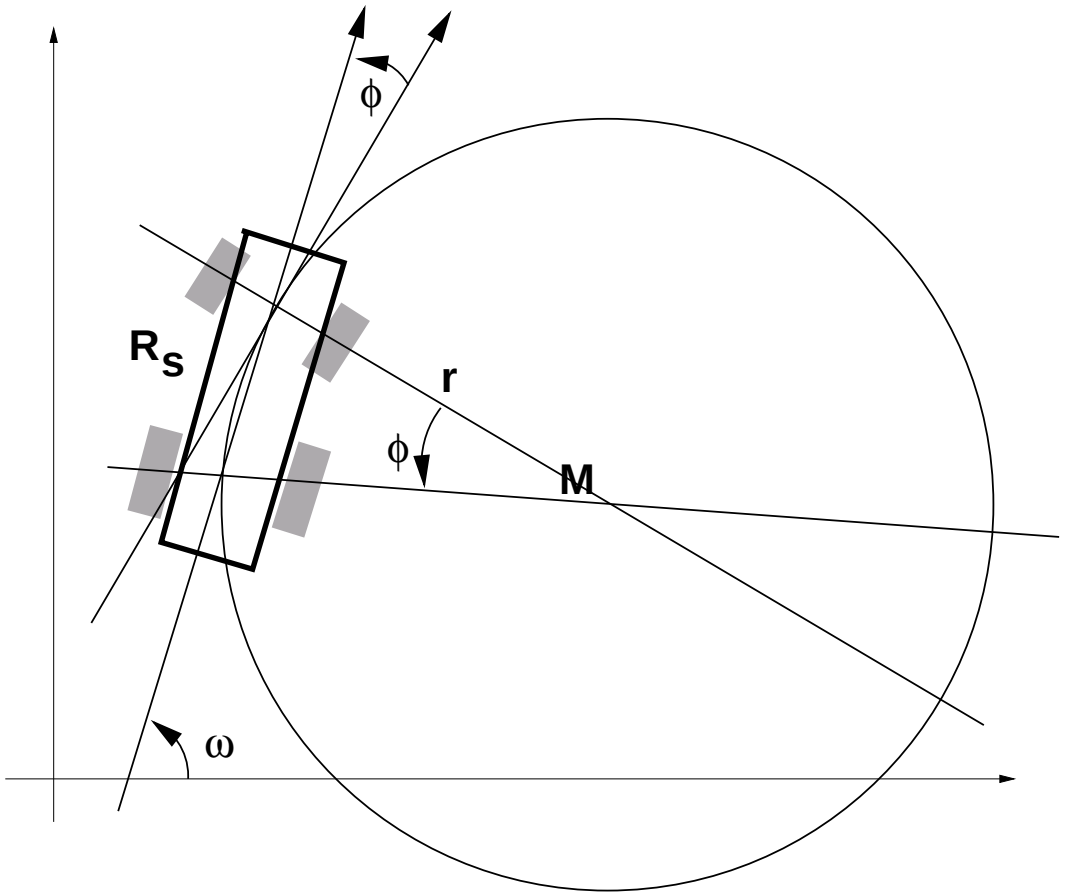
Hauptprogramm

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

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

Ein Autosimulator

Ziel: Simulation des Fahrverhaltens eines Autos



Zustand des Autos: b

- Radstand R_s
- Position $\vec{p} \in \mathbb{R}^2$
- Orientierung $\omega \in \mathbb{R}$
- Geschwindigkeit $v \in \mathbb{R}$
- Lenkwinkel $\phi \in \mathbb{R}$

Daraus

- Lenkradius $r = \frac{R_s}{\tan \phi}$
- Lenkmittelpunkt M
- Drehwinkel um M

Zustandsübergang

- Lenkradius $r = \frac{R_s}{\tan \phi}$
- Lenkwinkel $\omega_A = \omega + \phi + \frac{\pi}{2}$
- Lenkmittelpunkt $\vec{M} = cart(r, \omega_A) + \vec{p}$
- Drehung im Lenkmittelpunkt: $\zeta = \frac{v \Delta t}{r}$

  Bewegung um $l = v\Delta t$ auf Kreisbahn mit $\frac{l}{2\pi r} = \frac{\zeta}{2\pi}$
- Neue Position: $\vec{p}' = rot(\vec{M}, \zeta, \vec{p})$

  Rotation um Winkel ζ und Mittelpunkt $\vec{M}$
- Neue Orientierung: $\omega' = \omega + \zeta$
- Lenkwinkel und Beschleunigung: durch Benutzer

Benutzerinteraktion

- Konstanten a_{max} für Beschleunigung, b für Bremsbeschleunigung
- Tasten für $Schneller$, $Langsamer$, $Bremsen$
- $Schneller$ drücken: Beschleunigung auf a_{max} setzen
- $Schneller$ loslassen: Beschleunigung auf 0
- $Langsamer$ drücken: Beschleunigung auf $-a_{max}$ setzen
- $Langsamer$ loslassen: Beschleunigung auf 0 setzen
- $Bremsen$ drücken: Beschleunigung auf Bremsbeschleunigung
- $Bremsen$ loslassen: Beschleunigung auf 0
- Neuer Zustand:
  Zustandsübergang plus Benutzerinteraktion.

Zustände

- Zustand des Autos, Gesamtzustand:

```
data Car = Car { a     :: !Double
               , brake :: !Bool
               , phi   :: !Angle
               , w     :: !Angle
               , pos   :: !Point
               , v     :: !Double
               } deriving (Show, Eq)

data State = State { car    :: !Car
                   }
```

- Zustände sind explizite Parameter

Zustandsübergang

- Sonderfall: $\phi = 0$ — Geradeausfahrt.

```
moveCar :: Car-> Car
moveCar (Car {a= a, brake= brake, phi= phi,
              w= w, pos= pos, v= v})
  | phi == 0  =
    let pos' = winMod (add pos (cart (Polar {angle= w,
                                             dist= v* deltaT})))
    in  Car {a= a, brake= brake, phi= 0, w= w,
             pos= pos', v= v'}
```

Zustandsübergang

- Normalfall: Kurvenfahrt

```
  | otherwise =
    let r    = wheelBase /tan phi -- Lenkradius
        wa   = w+ phi+ pi/2  -- Lenkwinkel
        m    = add (cart (Polar{angle= wa, dist= r})) pos
                                   -- Lenkmittelpunkt
        zeta = v*deltaT / r -- Drehwinkel um m
        pos' = winMod (rotAround m zeta pos) -- neue Pos'n
        w'   = w+ zeta -- neue Orientierung
    in  Car {a= a, brake= brake, phi= phi,
             w= w', pos= pos', v= v'}
```

Neue Geschwindigkeit

- Sonderfall: Bremsen

```
where v'= if brake then
              if v> 0 then max 0 (v- brakeAcc*deltaT)
                      else 0
          else if abs v <= vMax then v+ a*deltaT
                                else v
```

- Ansonsten: linearer Anstieg bis v_{max}

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
```

- Alle vorliegenden Events lesen:

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

- Neuer Zustand: Events bearbeiten, Auto bewegen

```
nextState :: [Event]-> State-> IO State
nextState evs s = return
  (setCar moveCar (foldl (flip procEv) s evs))
```

Event bearbeiten:

```
procEv :: Event-> State-> State
procEv (Key {keysym= k, isDown=down})
     | k `isKey` ' ' && down     = setBrake True
     | k `isKey` ' ' && not down = setBrake False
     | isUpKey k && down         = setAcc accMax
     | isUpKey k && not down     = setAcc 0
     | isDownKey k && down       = setAcc (- accMax)
     | isDownKey k && not down   = setAcc 0
procEv (Button{isLeft= True, isDown= True})  = setBrake True
procEv (Button{isLeft= True, isDown= False}) = setBrake Fals
procEv (MouseMove {pt= (x, y)}) = setAng (mouseToAngle x)
procEv _ = id
```

- Hilfsfunktion: prüft `Key`

```
isKey :: Key-> Char-> Bool
isKey k c = isCharKey k && keyToChar k == c
```

- Hilfsfunktion: Mausposition in Lenkwinkel

```
mouseToAngle :: Int-> Double
mouseToAngle x =
  let x2= fromInt (winX `div` 2)
  in  asin ((fromInt x - x2) / x2)* phiMax / pi
```

Hauptfunktion

- Window öffnen, Hauptschleife starten

```
main :: IO ()
main = runGraphics $
  do w<- openWindowEx "Haskell Racer!" Nothing
          (winX, winY) DoubleBuffered (Just 30)
     i<- initialState
     loop w i
     closeWindow w
```

- Zeigen!

Zusammenfassung

- Refaktorierung der Geometriebücherei
  - Basis `Double`, Polarkkoordinaten, 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 einer einfachen Autosimulation