

Praktische Informatik 3
Einführung in die Funktionale Programmierung
Vorlesung vom 28.01.09:
Grafikprogrammiererung

Christoph Lüth

WS 08/09



Organisatorisches

- Tutorium Dominik Lücke morgen im Cartesium 2.043
- Fachgespräche
- Scheine

Fahrplan

- Teil I: Grundlagen
- Teil II: Abstraktion
- Teil III: Beispiele, Anwendungen, Ausblicke
 - Datenmodellierung mit XML
 - Effizienzerwägungen
 - Grafik
 - Schluss

Inhalt

- **Grafikprogrammierung** mit HGL (Haskell Graphics Library)
- **Einführung** in die Schnittstelle, kleine **Beispiele**.
- Bewegte Grafiken: **Animationen**
- Fallbeispiel: **Space** — the final frontier

Die Hugs Graphics Library HGL

- Das kanonische Beispielprogramm ([Zeigen](#)):

```
module Main where
import Graphics.HGL
main :: IO ()
main = runGraphics $ do
    w <- openWindow "Hallo Welt?" (300, 300)
    drawInWindow w (text(100, 100) "Hallo, Welt!")
    drawInWindow w (ellipse (100,150) (200,250))
    getKey w
    closeWindow w
```

Verwendete Funktionen

- `runGraphics :: IO () -> IO ()`
führt Aktion mit Grafik aus;
- `openWindow :: Title -> Point -> IO Window`
öffnet Fenster;
- `drawInWindow :: Window -> Graphic -> IO ()`
zeichnet Grafik in Fenster;
- ADTs `Window` und `Graphic`:
Fenster und darin darstellbare Grafiken;
- `getKey :: Window -> IO Char` wartet auf Taste
- `closeWindow :: Window -> IO ()` schließt Fenster

Die Haskell Graphics Library HGL

- **Kompakte Grafikbücherei** für einfache Grafiken und Animationen.
- **Gleiche Schnittstelle** zu X Windows (X11) und Microsoft Windows.
- Bietet:
 - Fenster
 - verschiedene Zeichenfunktionen
 - Unterstützung für Animation
- Bietet nicht:
 - Hochleistungsgrafik, 3D-Unterstützung (e.g. OpenGL)
 - GUI-Funktionalität
- Windows-Unterstützung nur für **alte Version** (2.0.5)

- **Grafik:** wird gezeichnet
 - Atomare Grafiken
 - Modifikation mit Attributen:
 - Pinsel, Stifte und Textfarben
 - Farben
 - Kombination von Grafiken
- **Fenster:** worin gezeichnet wird
- Benutzereingaben: **Events**

Basisdatentypen

- Winkel (Grad, nicht Bogenmaß)

```
type Angle      = Double
```

- Dimensionen (Pixel)

```
type Dimension = Int
```

- Punkte (Ursprung: links oben)

```
type Point      = (Dimension, Dimension)
```

Atomare Grafiken

- **Ellipse** (gefüllt) innerhalb des gegebenen Rechtecks

```
ellipse :: Point -> Point -> Graphic
```

- **Ellipse** (gefüllt) innerhalb des Parallelograms:

```
shearEllipse :: Point-> Point-> Point-> Graphic
```

- **Bogenabschnitt** einer Ellipse (**math. positiven** Drehsinn):

```
arc :: Point-> Point-> Angle-> Angle-> Graphic
```

- Beispiel (**Zeigen**):

```
drawInWindow w (arc (40, 50) (340, 250) 45 270)
```

```
getKey w
```

```
drawInWindow w (arc (60, 50) (360, 250) (-45) 90)
```

Atomare Grafiken

- Strecke, Streckenzug:

```
line      :: Point -> Point -> Graphic
polyline :: [Point] -> Graphic
```

- Polygon (gefüllt)

```
polygon :: [Point] -> Graphic
```

- Text:

```
text :: Point -> String -> Graphic
```

- Leere Grafik:

```
emptyGraphic :: Graphic
```

Modifikation von Grafiken

- Andere **Fonts**, **Farben**, Hintergrundfarben, ...:

```
withFont           :: Font      -> Graphic -> Graphic
withTextColor     :: RGB       -> Graphic -> Graphic
withBkColor       :: RGB       -> Graphic -> Graphic
withBkMode        :: BkMode    -> Graphic -> Graphic
withPen           :: Pen       -> Graphic -> Graphic
withBrush         :: Brush     -> Graphic -> Graphic
withRGB           :: RGB       -> Graphic -> Graphic
withTextAlignment :: Alignment -> Graphic -> Graphic
```

Farben

- Nützliche Abkürzung: benannte Farben

```
data Color = Black | Blue | Green | Cyan | Red
           | Magenta | Yellow | White
           deriving (Eq, Ord, Bounded, Enum, Ix, Show, Read)
```

- Benannte Farben sind einfach `colorTable :: Array Color RGB`
- Dazu Modifikator:

```
withColor :: Color-> Graphic-> Graphic
withColor c = withRGB (colorTable ! c)
```

Kombination von Grafiken

- Überlagerung (erste über zweiter):

```
overGraphic  ::  Graphic-> Graphic-> Graphic
```

- Verallgemeinerung:

```
overGraphics :: [Graphic]-> Graphic  
overGraphics = foldr overGraphic emptyGraphic
```

Fenster

- **Elementare** Funktionen:

```
getGraphic :: Window -> IO Graphic  
setGraphic :: Window -> Graphic-> IO ()
```

- **Abgeleitete** Funktionen:

- In Fenster zeichnen:

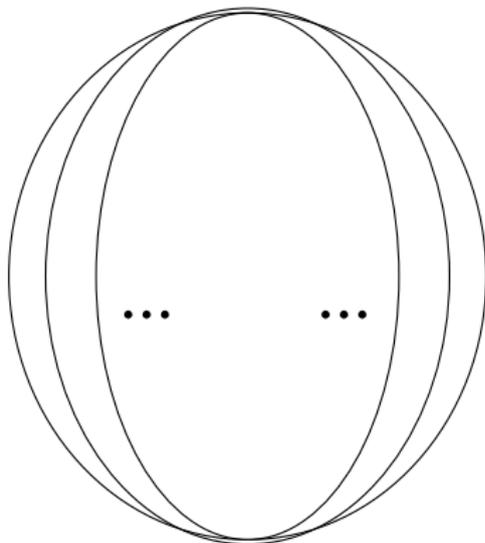
```
drawInWindow :: Window -> Graphic -> IO ()  
drawInWindow w g = do  
  old <- getGraphic w  
  setGraphic w (g 'overGraphic' old)
```

- Grafik löschen

```
clearWindow :: Window -> IO ()
```

Ein einfaches Beispiel

- **Ziel:** einen gestreiften Ball zeichnen
- **Algorithmus:** als Folge von konzentrischen Ellipsen
 - Start mit Eckpunkten (x_1, y_1) und (x_2, y_2) .
 - Verringerung von x um Δ_x , y bleibt gleich.
 - Dabei Farbe verändern.



Ein einfaches Beispiel

- **Bälle** zeichnen:

```
drawBalls :: Window -> Color -> Point -> Point -> IO ()
drawBalls w c (x1, y1) (x2, y2) =
  if x1 >= x2 then return ()
  else let el = ellipse (x1, y1) (x2, y2)
        in do drawInWindow w (withColor c el)
              drawBalls w (nextColor c)
                    (x1+deltaX, y1)
                    (x2-deltaX, y2)

deltaX :: Int
deltaX = 25
```

Ein einfaches Beispiel

- **Farbveränderung**, zyklisch:

```
nextColor :: Color -> Color
nextColor Red    = Green
nextColor Green  = Blue
nextColor Blue   = Red
```

- Hauptprogramm (**Zeigen**)

```
main :: IO ()
main = runGraphics $ do
  w <- openWindow "Balls!" (500,500)
  drawBalls w Blue (25, 25) (485, 485)
  getKey w
  closeWindow w
```

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
```

Der springende 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 (circle p 15)
```

- Kreis zeichnen:

```
circle :: Point -> Int -> Graphic  
circle (px, py) r = ellipse (px- r, py- r) (px+ r, py+ r)
```

Bewegung des Balles

- 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
```

Der springende Ball

- **Hauptprogram:** Fenster öffnen, Starten der Hauptschleife

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

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

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

Modellierung von Benutzereingaben

- **Benutzereingabe:**

- Tasten
- Mausbewegung
- Mausknöpfe
- Fenster: Größe verändern, schließen

- Grundlegende **Funktionen:**

- Letzte Eingabe, auf nächste Eingabe **warten:**

`getWindowEvent :: Window -> IO Event`

- Letzte Eingabe, **nicht warten:**

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

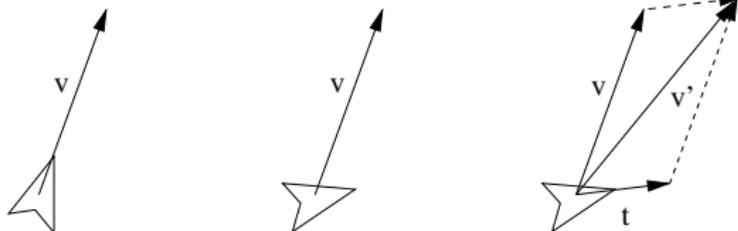
Benutzereingaben: Event

- 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
```

Fallbeispiel: 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**

Modellierung des Schiffszustandes

- **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

Punkte als Vektoren

- **Skalarmultiplikation**

```
smult :: Double -> Point -> Point
smult f (x, y)
  | f == 1    = (x, y)
  | otherwise = (round (f * fromIntegral x),
                 round (f * fromIntegral y))
```

- **Vektoraddition**

```
add :: Point -> Point -> Point
add (x1, y1) (x2, y2) = (x1 + x2, y1 + y2)
```

- **Betrag** eines Vektors

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

Punkte als Vektoren

- Konversion aus Polarkoordinaten

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

- Rotation (um den Nullpunkt)

```
rot :: Angle -> Point -> Point
rot w (x, y)
  | w == 0    = (x, y)
  | otherwise = (round (x' * cos w + y' * sin w),
                 round (-x' * sin w + y' * cos w)) where
    x' = fromIntegral x; y' = fromIntegral y
```

Modellierung der 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.

Datenmodellierung

- Modellierung des Gesamtsystems
 - Für den Anfang nur das Schiff

```
type State = Ship
```

- Schiffszustand:

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

Globale Parameter

- Fenstergröße

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

- Schub

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

- Maximale Geschwindigkeit

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

- Winkelbeschleunigung

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

Globale Parameter

- Das Raumschiff (als Polygon)

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

- Der Anfangszustand: Links oben, nach Süden gerichtet

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

Berechnung des neuen Schiffszustandes

- 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}) =
  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
```

Systemzustand visualisieren

- Gesamter Systemzustand

```
drawState :: State -> Graphic
```

```
drawState s = drawShip 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) $
```

```
    polygon (map (add (pos s). rot (ornt s)) spaceShip)
```

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<- getEvents w
     loop w (nextState evs s)
```

Folgezustand berechnen

- Folgezustand berechnen

```
nextState :: [Event]-> State-> State
```

```
nextState evs s =
```

```
    moveShip (foldl (flip procEv) s evs)
```

- Liste aller Eingaben seit dem letzten Tick:

```
getEvents :: Window-> IO [Event]
```

```
getEvents w =
```

```
    do x<- maybeGetWindowEvent w
```

```
        case x of Nothing -> return []
```

```
                Just e  -> do rest <- getEvents w
```

```
                    return (e : rest)
```

Benutzereingabe verarbeiten

```
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{hAcc= a}
setThrust :: Double-> State-> State
setThrust a s = s{thrust= a}
```

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
```

- Zeigen!

Zusammenfassung

- Abstrakte und portable **Grafikprogrammierung** mit HGL
 - **Animation**: Unterstützung in HGL durch Timer und Puffer
 - Implementation eines einfachen **Raumschiffsimulators**
- **Stuktur** der Programme:
 - Zustand mit **Modell** M der Welt und Übergang $M \rightarrow M$
 - **Darstellung** der Welt $M \rightarrow \text{Graphic}$
 - Benutzerinteraktion I **kontrolliert** Übergang $I \times M \rightarrow M$
- **Nächste Woche**: Wiederholung