Vorlesung vom 17.01.2005: Grafikprogrammerierung I Die Hugs Graphics Library

Inhalt

- Grafikprogrammierung mit HGL (Hugs Graphics Library)
- Einführung in die Schnittstelle, kleine Beispiele.
- Abstraktion über `HGL`:
  - Entwurf und Implementation einer kleine Geometriebücherei.
  - Kleine Beispiele.
- Literatur: Paul Hudak, The Haskell School of Expression.

Grafik — erste Schritte.

- Das kanonische Beispielprogramm:

```
module Hello where
import GraphicsUtils
hello :: IO ()
hello = 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)
```

- `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 Hugs 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
- Achtung: Benötigt altes Hugs Nov 2002!

Übersicht `HGL`

- 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 gegeben 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:

```
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
```

- Modifikatoren sind kumulativ:

```
withFont courier (
  withTextColor red (
    withTextAlignment (Center, Top)
      (text (100, 100) "Hallo?")))
```

- Unschön — Klammerwald.
- Abhilfe: `($) :: (a-> b)-> a-> b` (rechtsassoziativ)

```
withFont courier $
withTextColor red $
withTextAlignment (Center, Top) $
text (100, 100) "Hallo?"
```

Attribute

- Konkrete Attribute (Implementation sichtbar):
  - Farben: Rot, Grün, Blau

```
data RGB        = RGB Int Int Int
```

  - Textausrichtung, Hintergrundmodus

```
type Alignment = (HAlign, VAlign)
data HAlign = Left' | Center   | Right'
data VAlign = Top   | Baseline | Bottom
data BkMode = Opaque | Transparent
```

- Abstrakte Attribute: `Font`, `Brush` und `Pen`

Attribute: `Brush` und `Pen`

- `Brush` zum Füllen (Polygone, Ellipsen, Regionen)
  - Bestimmt nur durch Farbe

```
mkBrush :: RGB -> (Brush-> Graphic)-> Graphic
```

- `Pen` für Linien (Bögen, Linien, Streckenzüge)
  - Bestimmt durch Farbe, Stil, Breite.

```
data Style = Solid| Dash| Dot| DashDot| DashDotDot| ...
mkPen :: Style -> Int-> RGB-> (Pen-> Graphic)-> Graphic
```

Schriften

- Fonts:

```
createFont :: Point-> Angle-> Bool-> Bool-> String-> IO Font
```

- Größe (Breite, Höhe), Winkel (nicht unter X), Fett, Kursiv, Name.
- Portabilität beachten — keine exotischen Kombinationen.
- Portable Namen: `courier`, `helvetica`, `times`.
- Wenn Font nicht vorhanden: Laufzeitfehler
- Beispiel.

Quelle dazu

```
main :: IO ()
main = runGraphics $ do
  w <- openWindow "Fonts" (300, 150)
  f1<- createFont (7,14) 0 True False "Helvetica"
  drawInWindow w $
    withFont f1 (text (10, 10) "Helvetica (bold, 14 pt)")
  f2<- createFont (10, 18) 0 False True "Courier"
  drawInWindow w $
    withFont f2 (text (40, 50) "Courier (italic, 18 pt)")
  f3<- createFont (12,24) 0 False False "Times"
  drawInWindow w $
    withFont f3 (text (80, 100) "Times (24 pt)")
```

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 ()
```

- Abgeleitetete Funktionen:
  - In Fenster zeichnen:

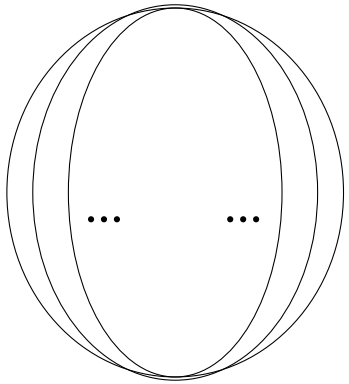
```
drawInWindow :: Window -> Graphic -> IO ()
drawInWindow w g = do
  old <- getGraphic w
  setGraphic w (g `overGraphic` old)
```

  - Grafik löschen

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

Ein einfaches Beispiel

- Ziel: einen gestreiften Ball zeichen.
- 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.



- 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
```

- Farbveränderung, zyklisch:

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

- Hauptprogramm:

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

Eine Geometrie-Bücherei

- Ziel: Abstrakte Repräsentation von geometrischen Figuren
- Aufbauend auf HGL.
- Unterstützung von Translation, Skalierung, Rotation
- Basisfiguren: Rechtecke, Dreiecke, Polygone, Kreise
  - Alle Basisfiguren liegen im Nullpunkt
- Keine Mengen, keine grafischen Attribute (Farben)

Schnittstelle

- Repräsentation der Figuren:

```
data Figure  -- abstract
```

- Figuren konstruieren;

```
rect     :: Dimension-> Dimension-> Figure
triangle :: Dimension-> Angle-> Dimension-> Figure
poly     :: [Point]-> Figure
circle   :: Dimension-> Figure
```

- Figuren manipulieren:

```
translate :: Point-> Figure-> Figure
scale     :: Double-> Figure-> Figure
rotate    :: Angle-> Figure-> Figure
```

Beispiele

- `rect 10 20` — Rechteck mit Höhe 10, Länge 20
- `rotate (pi/2) (triangle 10 (pi/3) 10)`
  Gleichschenkliges Dreieck, auf dem Kopf stehend.
- `rotate pi (circle 20)` — rotierter Kreis
- Beispiel für abgeleitete Funktion: Drehung um einen Punkt:

```
rotAt :: Point-> Angle-> Figure-> Figure
rotAt (px, py) w = translate (px, py) .
                   rotate w .
                   translate (-px, -py)
```

Implementation

- Repräsentation von `Figure`:
- Rechtecke und Dreiecke sind spezielle Polygone.
  - Alles ist ein Polygon oder eine Ellipse.
- Rotation, Translation und Skalierung herausrechnen.
- Problem: Nicht kommutativ!
  - d.h. Reihenfolge relevant.

Mathematische Grundlagen

- Lineare Algebra: Vektorraum $\mathbb{R}^2$
- Skalierung mit Faktor F: Skalarmultiplikation in $\mathbb{R}^2$
- Translation um Punkt $\vec{t}$: Addition in $\mathbb{R}^2$
- Rotation um den Winkel ω: Multiplikation mit der Rotationsmatrix

$$M_\omega = \begin{pmatrix} \cos\omega & \sin\omega \\ -\sin\omega & \cos\omega \end{pmatrix}$$

- Es gelten folgende Gleichungen:

$$(\vec{p}+\vec{q})M_\omega = \vec{p}M_\omega + \vec{q}M_\omega \quad (1)$$

$$r\cdot(\vec{p}+\vec{q}) = r\cdot\vec{p}+r\cdot\vec{q} \quad (2)$$

$$(r\cdot\vec{p})M_\omega = r\cdot(\vec{p}M_\omega) \quad (3)$$

Implementation von `Figure`

- (1) – (3) erlauben Reduktion zu einer Normalform

$$E(\vec{p}) = \vec{t} + s \cdot (\vec{p} M_\omega)$$

  - Zuerst Rotation um Vektor ω
  - Dann Skalierung um Faktor s
  - Dann Translation um Vektor t

- Intern: Grundfiguren `Polygon` oder `Circle`
  - Rotation einrechnen
  - Translation und Skalierung akkumulieren

```
data Figure = Polygon Point Double [Point]
            | Circle  Point Double
```

Skalierung und Translation

- Skalierung eines Punktes (x, y) um den Faktor f:

$$(x', y') = (fx, fy)$$

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

- Translation eines Punktes (x, y) um einen Vektor (a, b):

$$(x', y') = (x + a, y + b)$$

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

Rotation

- Rotation eines Punktes (x, y) um den Winkel ϕ:

$$(x', y') = (x \cos \phi + y \sin \phi, -x \sin \phi + y \cos \phi)$$

```
rot :: Angle-> Point-> Point
rot w (x, y)
  | w == 0     = (x, y)
  | otherwise = (round (x1* cos w+ y1* sin w),
                 round (-x1 * sin w + y1* cos w))
                 where x1 = fromInt x
                       y1 = fromInt y
```

Implementation der Konstruktoren

- Rechtecke:

```
rect a b = let x2= a `div` 2; y2= b `div` 2
  in Polygon (0,0) 1
             [(x2, y2), (-x2, y2), (-x2, -y2), (x2, -y2)]
```

- Dreiecke:

```
triangle l1 a l2 =
  Polygon (0,0) 1 [(0, 0), (l1, 0), rot a (l2, 0)]
```

- Polygone:

```
poly  ps = Polygon (0, 0) 1 ps
```

- Kreise:

```
circle r = Circle (0, 0) (fromInt r)
```

Implementation der Manipulationsfunktionen:

- Rotation:

```
rotate a (Polygon t r ps) = Polygon (rot a t) r
                                    (map (rot a) ps)
rotate a (Circle c r)     = Circle (rot a c) r
```

- Skalierung:

```
scale s (Polygon t r ps) = Polygon t (r* s) ps
scale s (Circle  c r)    = Circle  c (r* s)
```

- Translation:

```
translate p (Polygon q r ps) = Polygon (add p q) r ps
translate p (Circle c r)     = Circle (add p c) r
```

Implementation der Zeichenfunktion

- Zusätzliches Argument: Verschiebung zur Fenstermitte

```
draw :: Point-> Figure-> Graphic
```

- Polygone zeichnen:

```
draw m (Polygon q r ps) =
  polygon (map (add m. add q. smult r) ps)
```

- Kreise zeichnen:

```
draw (x, y) (Circle (cx, cy) r) =
  ellipse (cx+x- rad, cy+y-rad) (cx+x+ rad, cy+y+rad)
     where rad= round r
```

Ein kleines Beispielprogramm

- Sequenz von gedrehten, skalierten Figuren.
- Im wesentlichen zwei Funktionen:
  - `drawFigs :: [Figure]-> IO` zeichnet Liste von Figuren in `wechselnden Farben`
  - `swirl :: Figure-> [Figure]` erzeugt aus Basisfigur unendliche Liste von gedrehten, vergrösserten Figuren

- Liste von Figuren zeichnen:

```
drawFigs :: [Figure]-> IO ()
drawFigs f = runGraphics $ do
  w<- openWindow "Here's some figures!" (500, 500)
  drawInWindow w
    (overGraphics (zipWith withColor
          (cycle [Blue ..])
          (map (draw (250, 250)) f)))
  getKey w
  closeWindow w
```

  - `(cycle [Blue ..]` erzeugt unendliche Liste aller Farben.
  - NB: Figuren werden im Fenstermittelpunkt gezeichnet.

- Figuren erzeugen

```
swirl :: Figure-> [Figure]
swirl = iterate (scale 1.123. rotate (pi/7))
```

  - Nutzt `iterate :: (a-> a)-> a-> [a]` aus dem Prelude:
    `iterate f x = [x, f x, f f x, f f f x, ...]`

- Hauptprogramm:

```
main = do
  drawFigs (take 30 (swirl (rect 15 10)))
  drawFigs (take 50 (swirl (triangle 6 (pi/3) 6)))
```

Zusammenfassung

- Die Hugs Graphics Library (HGL) bietet abstrakte und portable Graphikprogrammierung für Hugs.
  - Handbuch und Bezugsquellen auf PI3-Webseite oder `http://www.haskell.org/graphics`
- Darauf aufbauend Entwicklung einer kleinen Geometriebücherei.
- Nächste Woche: Interaktion und Animation.