Vorlesung vom 06.01.2003: Grafikprogrammerierung I Die Hugs Graphics Library

Inhalt

- Organisatorisches:
  - Scheinanmeldung (bis Semesterende)
- Grafikprogrammierung mit HGL (Hugs Graphics Library)
- Einführung in die Schnittstelle, kleine Beispiele.
- Abstraktion über `HGL`:
  - Entwurf und Implementation einer kleine Geometriebücherei.
  - Kleines Beispiel.
- 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 X11 und Win32 Graphics Library.
- Bietet:
  - Fenster
  - verschiedene Zeichenfunktionen
  - Unterstützung für Animationen
- Bietet nicht:
  - Hochleistungsgrafik, 3D-Unterstützung (e.g. OpenGL)
  - GUI-Funktionalität

Übersicht `HGL`

- Grafik
  - Atomare Grafiken
  - Modifikatoren
  - Attribute
  - Kombination von Grafiken
  - Pinsel, Stifte und Texfarben
  - Farben
- Fenster
- 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

- Größte Ellipse (gefüllt) innerhalb des gegeben Rechtecks

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

- Größte Ellipse (gefüllt) innerhalb des Parallelograms:

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

- Bogenabschnitt einer Ellipse im math. positiven Drehsinn:

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

- Beispiel:

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

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

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

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

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

- Fonts
  - Punktgröße, Winkel (nicht unter X11), Fett, Kursiv, Name.
  - Portabilität beachten — keine exotischen Kombinationen.
  - Portable Namen: `courier`, `helvetica`, `times`. Beispiel, Quelle.

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

- Farben
  - Nützliche Abkürzung: benannte Farben

```
data Color = Black | Blue | Green | Cyan | Red
           | Magenta | Yellow | White
```

  - Dazu Modifikator:

```
withColor :: Color-> Graphic-> Graphic
```

- Kombination von Grafiken
  - Überlagerung (erste über zweiter):

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

Fenster

- Elementare Funktionen:

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

- Abgeleitetete Funktionen:
  - In Fenster zeichnen:

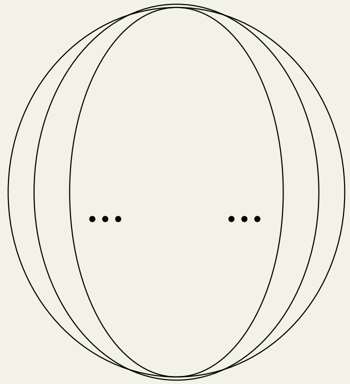
```
drawInWindow :: Window -> Graphic -> IO ()
drawInWindow w g = do
  old <- getGraphic w
  setGraphic w (g `overGraphic` old)
```

  - 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
- Komplementär zu HGL.
- Unterstützung von Translation, Skalierung, Rotation
- Basisfiguren: Rechtecke, Dreiecke, Polygone, Kreise
  - Alle Basisfiguren liegen im Nullpunkt
- Später: Unterstützung von Kollisionserkennung
- Keine Mengen, keine grafischen Attribute (Farben)

- Repräsentation durch algebraischen Datentyp

```
type Dimension = Int
data Figure = Rect Dimension Dimension
            | Triangle Dimension Angle Dimension
            | Polygon [Point]
            | Circle Dimension
            | Translate Point Figure
            | Scale Double Figure
            | Rotate Angle Figure
          deriving (Eq, Ord, Show)
```

- 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 Funktionen:
  - Drehung um einen Punkt:

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

Darstellung einer geometrischen Figur

- Rechtecke und Dreiecke sind spezielle Polygone.
- Alles ist ein Polygon oder eine Ellipse.
- Nullpunkt $(0, 0)$ des Koordinatensystems links oben.
- 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:

$$(\vec{p} + \vec{q})M_\omega = \vec{p}M_\omega + \vec{q}M_\omega \tag{1}$$
$$r \cdot (\vec{p} + \vec{q}) = r \cdot \vec{p} + r \cdot \vec{q} \tag{2}$$
$$(r \cdot \vec{p})M_\omega = r \cdot (\vec{p}M_\omega) \tag{3}$$

- Implementation von `draw`:
  - (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

- 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 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 Zeichenfunktion

- Damit Zeichenfunktion:

```
draw :: Figure-> Graphic
draw = draw' ((0, 0), 1, 0) where
  draw' :: (Point, Double, Angle)->
                Figure-> Graphic
```

- Hauptarbeit findet in `draw'` statt:
  - Translation, Skalierung, Rotation in Argumenten kumulieren
  - Basisfiguren entsprechend zeichnen

- Translation, Skalierung, Rotation:
  - Translation, Skalierung, Rotation aufrechnen
  - Dabei Gleichungen (1), (2) beachten

```
draw' (m, r, phi) (Translate t f) =
  draw' (add m (smult r (rot phi t)), r, phi) f
draw' (m, r, phi) (Scale s f) =
  draw' (m, s+ r, phi) f
draw' (m, r, phi) (Rotate w f) =
  draw' (m, r, phi+ w) f
```

- Basisfiguren zeichnen:

```
draw' ((mx, my), r, _) (Circle d) =
  ellipse (mx- rad, my-rad) (mx+ rad, my+rad)
  where rad= round (r*fromInt d / 2)
draw' c (Rect a b) =
  poly c [(x2, y2), (-x2, y2),
          (-x2, -y2), (x2, -y2)]
  where x2= a `div` 2; y2= b `div` 2
draw' c (Triangle l1 a l2) =
  poly c [(0, 0), (0, l1), rot a (0, l2)]
draw' c (Polygon pts) = poly c pts
```

- Hilfsfunktion: Polygon zeichnen

```
poly :: (Point, Double, Angle)->
          [Point]-> Graphic
poly (m, p, w) =
  polygon . map (add m. smult p. rot w)
```

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. Translate (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 20 (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: Kollisionserkennung und Animation.