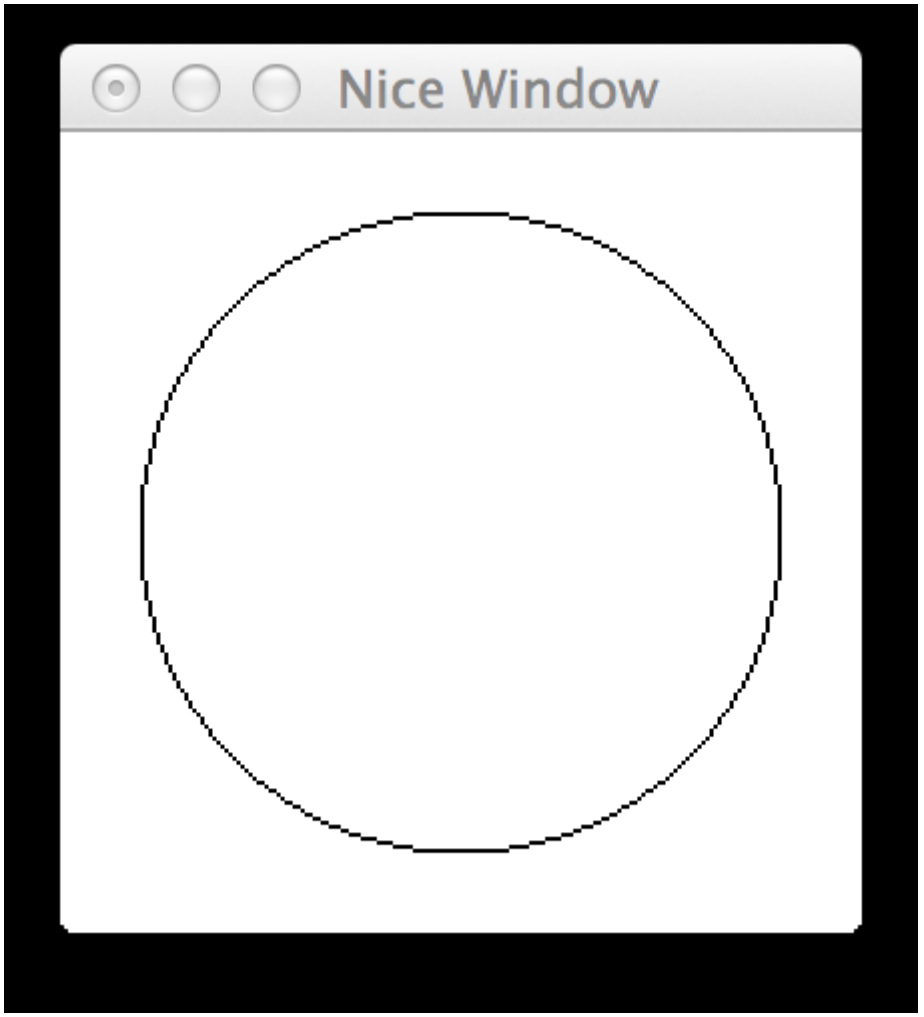
Desenhos com Gloss

Gloss

- Útil para desenhos, animações, simulações – jogos 2D.
- Simples – projetada para ensino
- Sítio oficial: http://gloss.ouroborus.net/
- Documentação: http://hackage.haskell.org/package/gloss
- Um tutorial:
  Your First Haskell Application (with Gloss)
  http://andrew.gibiansky.com/blog/haskell/haskell-gloss/#_gloss

Exemplo inicial

```
module Main(main) where

import Graphics.Gloss

window :: Display
window = InWindow "Nice Window" (200, 200) (10, 10)

background :: Color
background = white

drawing :: Picture
drawing = circle 80

main :: IO ()
main = display window background drawing
```


Nice Window

Tipos definidos no Gloss

```
type Path = [Point] -- percurso

type Point = (Float,Float) -- coordenada x,y

type Vector = Point
```

O tipo `Picture`

Valores: figuras geométricas e combinação delas

Funções construtoras

```
circleSolid :: Float -> Picture
        -- círculo dado o raio

rectangleSolid :: Float -> Float -> Picture
        -- rectangulo dada largura, altura

line :: Path -> Picture -- linha poligonal

polygon :: Path -> Picture -- polígono cheio
…
```

Figuras como círculos e retângulos são construídas no centro

Cores

Podemos mudar a côr de uma figura:

```
color :: Color -> Picture -> Picture
```

As cores usuais estão pré-definidas na biblioteca:

```
red, green, blue, yellow, cyan, magenta, …
```

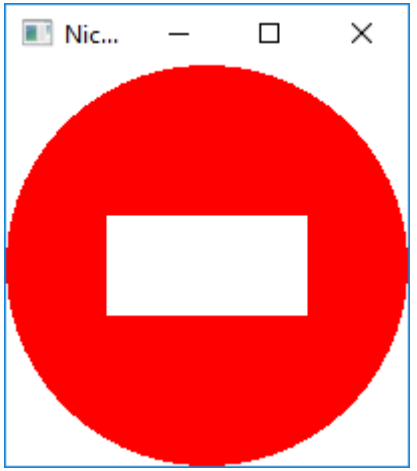
Funções

```
light :: Color -> Color
dark :: Color -> Color
bright :: Color -> Color
```

Podemos sobrepor várias figuras numa só:

```
pictures :: [Picture] -> Picture
```

```
ex2 = pictures [
        color red (circleSolid 100),
        color white (rectangleSolid 100 50) ]

main :: IO ()
main = display window background ex2
```



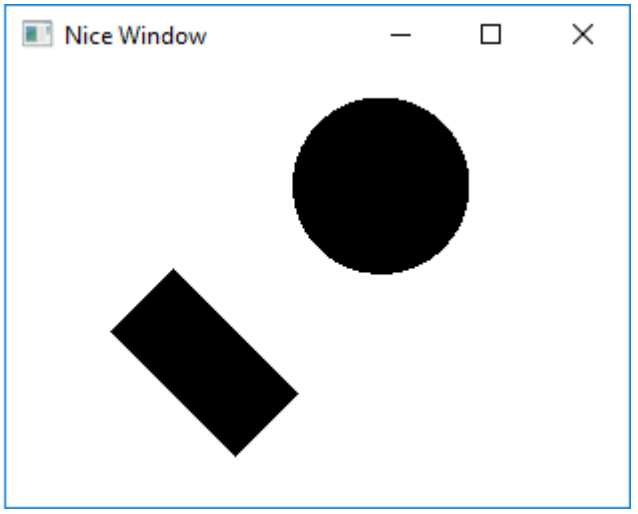
Podemos mover uma figura de posição

```
translate :: Float -> Float -> Picture -> Picture
```

Também podemos fazer rotações por um ângulo (em graus):

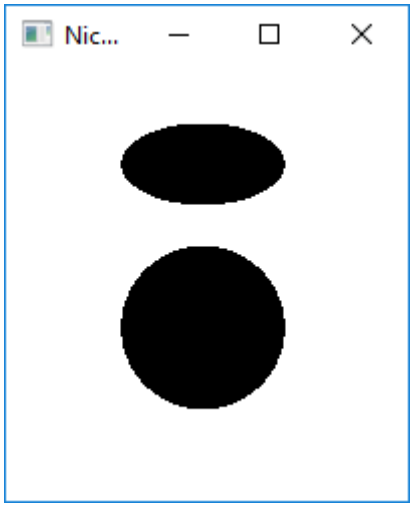
```
rotate :: Float -> Picture -> Picture
```

```
ex3 = pictures [translate 100 100 (circleSolid 50),
                rotate 45 (rectangleSolid 100 50)]
```

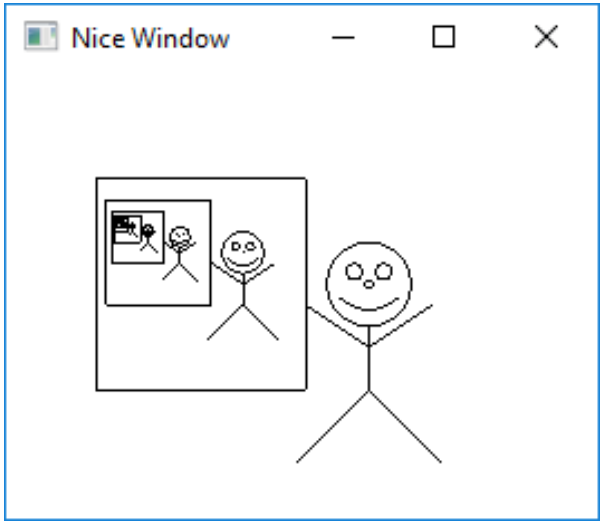


Ampliar ou reduzir

```
ex4 = pictures [circleSolid 50,
                translate 0 100
                   (scale 1 0.5 (circleSolid 50))]
```



Exemplo: espelho no espelho

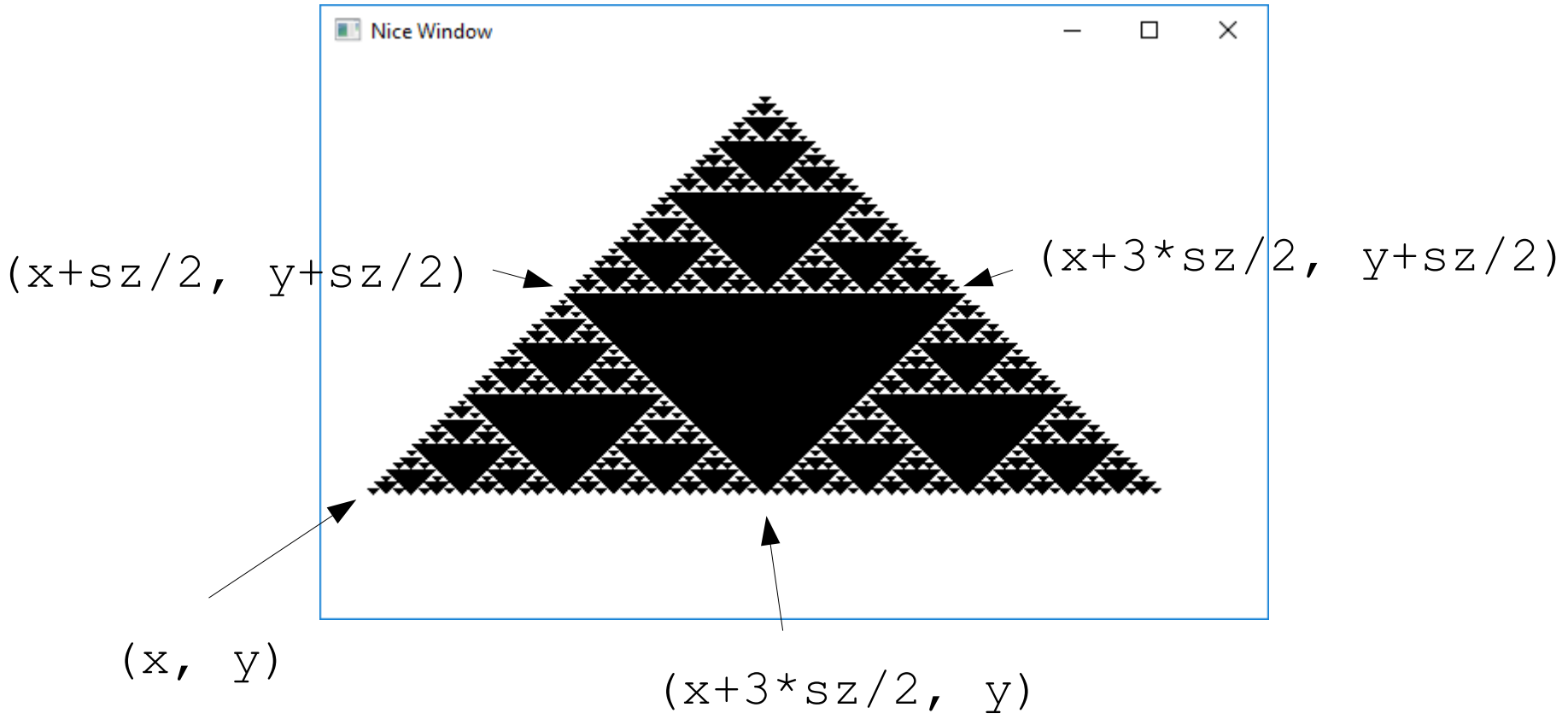


```
module Main(main) where

import Graphics.Gloss

eu :: Picture
eu = pictures [cara, bracos, pernas, corpo]
 where
  cara = pictures
              [ circle 20,
                translate (-7) 6 (circle 4),
                translate 7  6 (circle 4),
                circle 2,
                translate 0 8 (arc 225 315 20)
              ]
   bracos = translate 0 (-30)
              (line [(-30, 20), (0, 0), (30, 20)])
   pernas = translate 0 (-50)
              (line [(-35, -35), (0,0), (35, -35)])
   corpo = line [(0,-20), (0, -50)]
```

```
espelho :: Picture
espelho = translate (-80) 0 (rectangleWire 100 100)

euComEspelho :: Picture
euComEspelho = pictures [eu, espelho]

espelhoNoEspelho :: Int -> Float -> Picture
espelhoNoEspelho 0 sc = blank
espelhoNoEspelho n sc =
  pictures [ scale sc sc euComEspelho ,
             translate (-60*sc) (15*sc)
                  (espelhoNoEspelho (n-1) (sc/2)) ]
```

```
window :: Display
window = InWindow "Nice Window" (300, 200) (10, 10)

background :: Color
background = white

main :: IO ()
main = display window background
                      (espelhoNoEspelho 10 1)
```

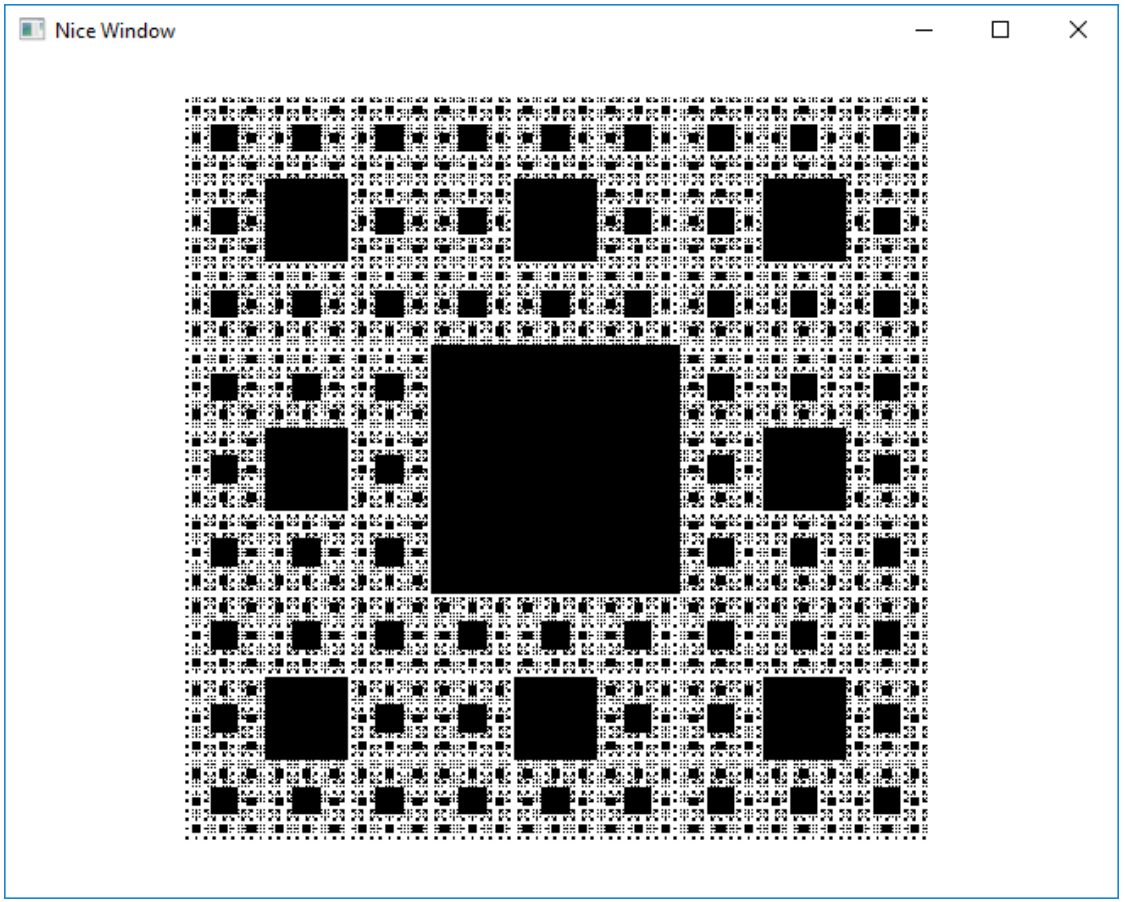
Triangulo de Sierpinski



```
sierpinsky :: Int -> Point -> Float -> Picture
sierpinsky 0 _ _ = blank
sierpinsky n (x, y) sz =
  pictures [  polygon [(x+sz/2, y+sz/2),
                       (x+3*sz/2, y+sz/2), (x+sz, y)]
            , sierpinsky (n-1) (x, y) (sz/2)
            , sierpinsky (n-1) (x+sz, y) (sz/2)
            , sierpinsky (n-1) (x+sz/2, y+sz/2) (sz/2)
            ]

main :: IO ()
main = display window background
                       (sierpinsky 5 (-100, -100) 200)
```

Exercício

- Escreva um programa que desenhe o tapete de Sierpinsky



Pontos e vetores

Aritmética com pontos e vetores

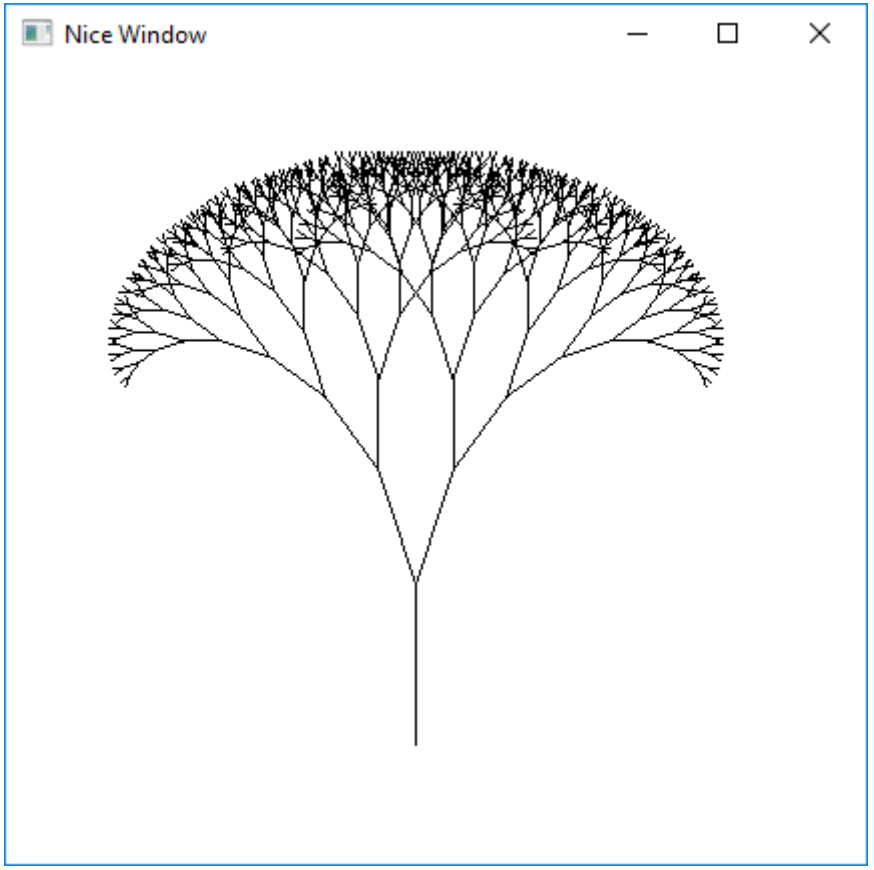
`Graphics.Gloss.Data.Point.Arithmetic`

`+`, `-`, `*` e `negate`

Funções geométricas

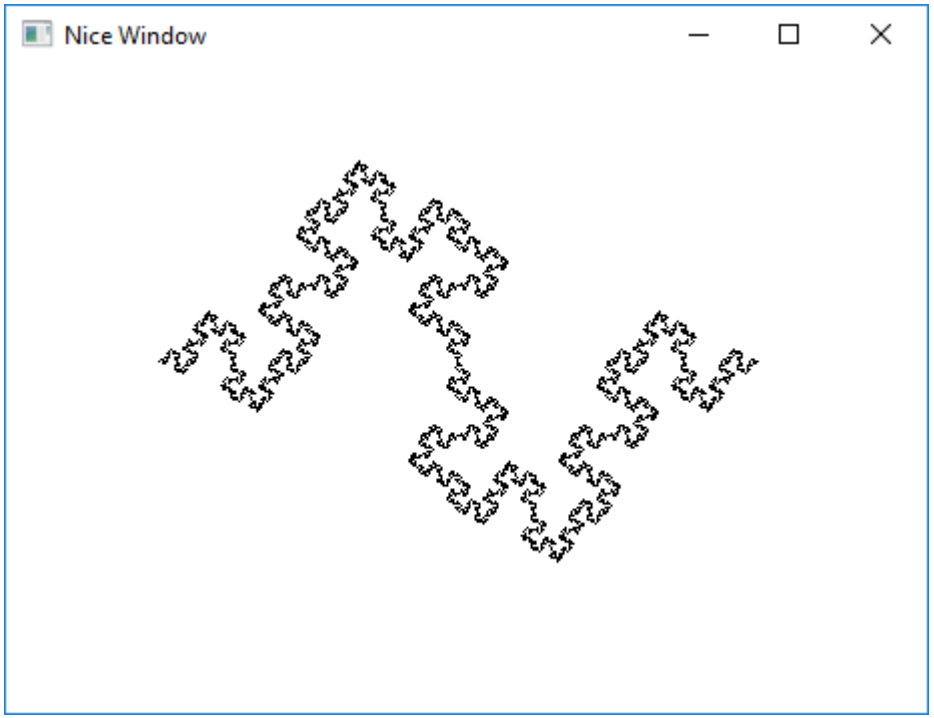
```
Graphics.Gloss.Data.Vector


magV :: Vector -> Float
argV :: Vector -> Float
angleVV :: Vector -> Vector -> Float
dotV :: Vector -> Vector -> Float
detV :: Vector -> Vector -> Float
mulSV :: Float -> Vector -> Vector
rotateV :: Float -> Vector -> Vector
normalizeV :: Vector -> Vector
unitVectorAtAngle :: Float -> Vector
```

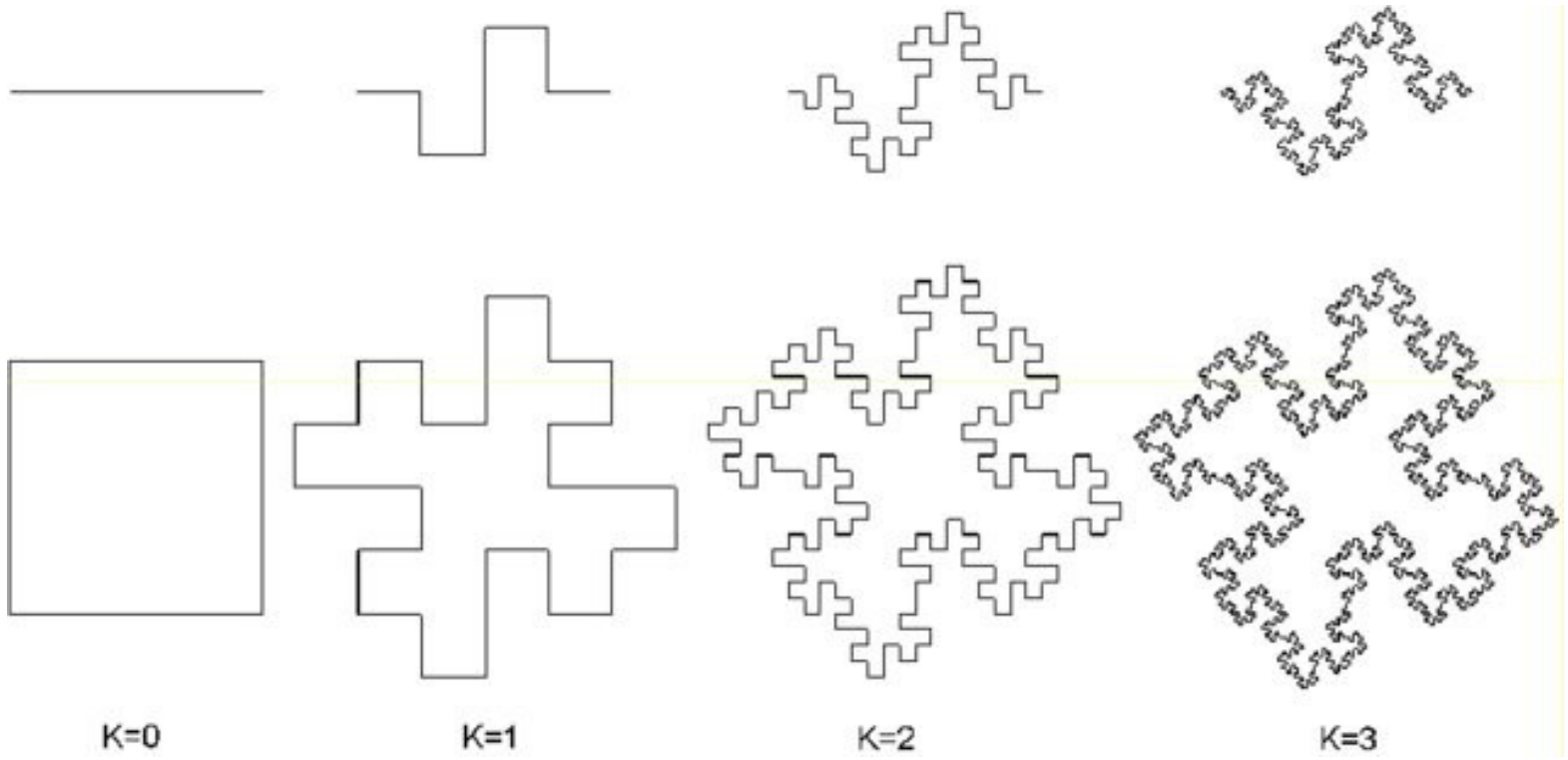
Tree



```
import qualified
    Graphics.Gloss.Data.Point.Arithmetic as V
...
tree ::     Int   -- recursion level
         -> Point -- starting point
         -> Point -- starting direction
         -> Float -- starting segment size
         -> Float -- angle  (** pi/10 **)
         -> Picture
tree 0 _ _ _ _ = blank
tree n p d sz a =
   pictures [  line [p, p1]
             , tree (n-1) p1 d1 (0.75*sz) a
             , tree (n-1) p1 d2 (0.75*sz) a
            ]
 where
   p1 = p V.+ mulSV sz d
   d1 = rotateV a d
   d2 = rotateV (-a) d
```

Minkowski Sausage




K=0
K=1
K=2
K=3

```
minkowskiSausage :: Int -> Point -> Point -> Picture
minkowskiSausage 0 p q = line [p, q]
minkowskiSausage n p q =
     pictures [  minkowskiSausage (n-1) p p1
               , minkowskiSausage (n-1) p1 p2
               , minkowskiSausage (n-1) p2 p3
               , minkowskiSausage (n-1) p3 p4
               , minkowskiSausage (n-1) p4 p5
               , minkowskiSausage (n-1) p5 p6
               , minkowskiSausage (n-1) p6 p7
               , minkowskiSausage (n-1) p7 q
              ]
  where v = mulSV (1/4) (q V.- p)
        p1 = p V.+ v
        p2 = p1 V.+ rotateV (pi/2) v
        p3 = p2 V.+ v
        p4 = p3 V.+ rotateV (3*pi/2) v
        p5 = p4 V.+ rotateV (3*pi/2) v
        p6 = p5 V.+ v
        p7 = p6 V.+ rotateV (pi/2) v
```

Exercício: curva de Koch