3-Dimensional Lindenmayer Systems in Haskell

Eric O'Connell

March 2012

Lindenmayer Systems (L-Systems) are recursive, self-rewriting grammars which can approximate growth patterns observed in nature, most commonly known for the branching structures in plants and trees. The grammars generate strings which can be interpreted as part of a sophisticated Logo/Turtle-graphics command, generating 2 and 3-dimensional images in the process.

L-Grammars

A Lindenmayer grammar (L-Grammar) is a 3-tuple composed of an alphabet, an axiom or initial string, and a list of productions. To produce the 2nd iteration of an L-grammar, one walks the start string, replacing each existing character with a matching production rule, or itself if no productions match, into a new string. Each production consists of a predecessor, to be matched against the current generation, and a successor, which replaces the match in the next generation.

There are several types of L-Grammars, classified along a few axes according to the types of productions:

- deterministic vs. stochastic
- context-free vs. context-sensitive
- non-parametric vs. parametric

Deterministic productions, the only ones supported by this software, are of the form:

predecessor $\rightarrow$ successor

These will always generate the successor whenever the predecessor is in the next generation.

And now, for some code!

```
module Main
where


import qualified Graphics.UI.GLFW as GLFW
import Graphics.Rendering.OpenGL.Raw
import Graphics.Rendering.GLU.Raw ( gluPerspective )
import Data.Bits ( (.|.) )
import Data.IORef ( IORef, newIORef, readIORef, writeIORef )
import System.Exit ( exitWith, ExitCode(..) )
import Control.Monad ( forever )
```

LGrammar Implementation

An LGrammar is simply an axiom and a list of Producions, each of which is a predecessor and a successor String. Again, to implement either parametric or context-sensitive productions, I would change at least the type of the predecessor, and likely the type of the successor, to a more structured datatype. I also considered a more functional representation, along the lines of the Image combinators presented in class. I believe they would work particularly well with Parsec, as each production could be built up by composing functions, if I were to add support for an external file format.

```
data Production = Pr { pre, suc  :: String }
                  deriving Show

data LGrammar   = Lg { axiom       :: String,
                       productions :: [Production] }
                  deriving Show
```

Generating iterations of LGrammars

`generate` takes an LGrammar `lg` and an Int `n`, and returns the string representing the *nth* generation of *lg*. `generate` is a essentially a wrapper for the recursive `produce` function, which takes an input string `s` and the generation `n'`; it in turn relies on `next` to find the next generation of `s` in `lg`. `next` prepends the successor of the currently matching production to the successors of the rest of the string recursively. The final piece of the puzzle is `applicableProduction` which, given an input string and a list of productions, will find the matching production or default to an identity production if none match.

```
generate                :: LGrammar -> Int -> String
generate  lg n           = produce (axiom lg) n
    where produce s 0    = s
          produce s n'   = produce (next s) (n'-1)
          next []        = ""
          next (x:xs)    = (suc' x) ++ (next xs)
          suc' x = suc (applicableProduction [x] (productions lg))
          applicableProduction s []     = Pr s s
          applicableProduction s (p:ps)
              | (pre p) == s = p
              | otherwise    = applicableProduction s ps
```

It is here in generate where applicableProduction could be fairly-easily modified to find all possible matching productions and choose randomly from them.

On the other hand, in order to implement context-sensitive productions, some very expensive string matching would be necessary, or, alternately, the strings could be replaced with a structured datatype. This would be necessary in order to account for the fact that right contexts can occur *after* any branches at the same level. So, for instance, the string-matching algorithm would need to balance over the bracketed branches.

Turtle Graphics

The Turtle is represented as a TurtleStack, a stack of individual TurtleStates, each of which contains the current position, heading, and pen information for the turtle. Position is a 3D point, while heading is represented as a 3D matrix, consisting of Heading, Up, and Left vectors (HLU). The Pen includes information like the default turning radius for the heading-modifying commands, the current line width, current color index, and list of colors available.

```
type Scalar = GLfloat
type Point = (Scalar, Scalar, Scalar)
type Color = Point
type Heading = (Point, Point, Point)

data Pen = Pen {
      theta  :: Scalar,
      width  :: Scalar,
      color  :: Int,
      colors :: [Color]
      } deriving Show

data TurtleState = S {
      position :: Point,
      heading  :: Heading,
      pen      :: Pen
    } deriving Show

type TurtleStack = [TurtleState]
```

`initialState` is a convenience function, used to generate the original stack containing a turtle at the origin, headed north. Ideally the color list would be separated from this function and somehow attached to a specific L-System, perhaps as part of an external data file, but for now it's hard-coded here.

```
initialState      :: Scalar -> Scalar -> TurtleStack
initialState th w  = [ S ( 0, 0, 0 )
                         (( 0, 1, 0 ),
                          ( 1, 0, 0 ),
                          ( 0, 0, 1 ))
                         (Pen (d2r th) w 0 [(0.53, 0.50, 0.40),
                                            (0.15, 0.80, 0.25),
                                            (0.95, 0.92, 0.95)]) ]
```

I use a simple helper function to convert from degrees to radians:

```
d2r    :: Scalar -> Scalar
d2r d   = (d/180 * pi)
```

Linear Algebra

`matMult` is a very bone-headed (but effective!) implementation of 3d matrix multiplication. I would have liked to have looked into some 3rd-party libraries, particularly `vect` on Hackage, but appreciate the convenience of being able to explicitly specify my Scalar type as GLfloat, avoiding any type inconsistencies across the OpenGL boundary. the `rH`, `rL`, and `rU` functions generate rotation matrices around the H, L, and U vectors respectively, allowing for yaw, roll, and pitch.

```
matMult :: Heading -> Heading -> Heading
( (a1, b1, c1), (d1, e1, f1), (g1, h1, i1)) `matMult`
 ((a2, b2, c2), (d2, e2, f2), (g2, h2, i2)) =
    (((a1*a2 + b1*d2 + c1*g2), (a1*b2 + b1*e2 + c1*h2), (a1*c2 + b1*f2 + c1*i2)),
     ((d1*a2 + e1*d2 + f1*g2), (d1*b2 + e1*e2 + f1*h2), (d1*c2 + e1*f2 + f1*i2)),
     ((g1*a2 + h1*d2 + i1*g2), (g1*b2 + h1*e2 + i1*h2), (g1*c2 + h1*f2 + i1*i2)))

rU       :: Scalar -> Heading
rU alpha  = ((  cos alpha, sin alpha, 0),
             ( -sin alpha, cos alpha, 0),
             (          0,         0, 1)) :: Heading

rL       :: Scalar -> Heading
rL alpha  = (( cos alpha, 0, -sin alpha),
             ( 0,         1,          0),
             ( sin alpha, 0,  cos alpha)) :: Heading

rH       :: Scalar -> Heading
rH alpha  = (( 1,         0,          0),
             ( 0, cos alpha, -sin alpha),
             ( 0, sin alpha,  cos alpha)) :: Heading
```

Turtle movement / manipulation functions

To interpret a string of turtle commands generated by an L-Grammar, `interpret` takes a String and some initial configuration parameters (currently only the default theta by which to modify the turtle's heading) and passes along each character to the dispatcher, `actOn`.

```
interpret         :: String -> GLfloat -> IO [TurtleState]
interpret s th     = interpret' s (return $ initialState th 3)
    where interpret' [] state     = state
          interpret' (x:xs) state = return (actOn x state)
                                    >>= interpret' xs
```

All turtle commands are ultimately dispatched via `actOn`, which takes a Char and calls the appropriate function. To add new commands, implement the behavior in a function from TurtleStack to IO TurtleStack, and then add the appropriate command here in the case statement.

```
actOn     :: Char -> IO TurtleStack -> IO TurtleStack
actOn c s  = do states <- s
                let states' = case c of
                          'F'  -> forward True  states
                          'f'  -> forward False states
                          '+'  -> rotateX (theta' states) states
                          '-'  -> rotateX (-theta' states) states
                          '&'  -> rotateY (theta' states) states
                          '^'  -> rotateY (-theta' states) states
                          '\\' -> rotateZ (theta' states) states
                          '/'  -> rotateZ (theta' states) states
                          '|'  -> turnAround states
                          '['  -> push      states
                          ']'  -> pop       states
```

```
                        '!'  -> decrWidth states
                        '{'  -> polyBegin states
                        '}'  -> polyEnd   states
                        '\'' -> nextColor states
                        _    -> return    states
          states'
      where theta' ss = (theta . pen . head) ss
```

`forward` moves the turtle forward in space, optionally either drawing a line segment (when used by an `F` command), or simply setting a point in a polygon, as when used by an `f` command. `forward` moves in steps of `delta`, which is a configuration parameter in need of a better home.

```
forward                                 :: Bool -> TurtleStack -> IO TurtleStack
forward line ((S (x, y, z) ((hx, hy, hz), l, u) p):ss) = do
  if line
    then
      glBegin gl_QUADS >>
      glVertex3f (x  + ddx) (y  - ddy) z  >>
      glVertex3f (x  - ddx) (y  + ddy) z  >>
      glVertex3f (x' - ddx) (y' + ddy) z' >>
      glVertex3f (x' + ddx) (y' - ddy) z' >>
      glEnd
    else
      glVertex3f x' y' z'
  return ((S (x', y', z') ((hx, hy, hz), l, u) p) : ss)
    where x'  = x + hx * delta
          y'  = y + hy * delta
          z'  = z + hz * delta
          ddx = -hy / w       -- ddx and ddy are currently used to give the lines
          ddy = -hx / w       -- some width, until I figure out gluCylinder
          w   = width p

delta :: Scalar
delta = 2
```

`rotateX`, `rotateY`, and `rotateZ` rely on the `rU`, `rL`, and `rH` functions to modify the turtle's current heading by `theta` radians. These are called when `actOn` encounters `+`, `-`, `&`, `^`, \, or `/`.

```
rotateX                                 :: Scalar -> TurtleStack -> IO TurtleStack
rotateX theta ((S pos head p):ss)        = return $ (S pos ((rU theta) `matMult` head) p):ss

rotateY                                 :: Scalar -> TurtleStack -> IO TurtleStack
rotateY theta ((S pos head p):ss)        = return $ (S pos ((rL theta) `matMult` head) p):ss

rotateZ                                 :: Scalar -> TurtleStack -> IO TurtleStack
rotateZ theta ((S pos head p):ss)        = return $ (S pos ((rH theta) `matMult` head) p):ss
```

`turnAround` causes the turtle to performan an about-face, rotating 180 degrees through the U vector. It is most commonly used in polygons such as leaves and flower petals, making it easy to complete the hull. It is triggered by a | in the input.

```
turnAround                              :: TurtleStack -> IO TurtleStack
turnAround ((S pos head p):ss)           = return $ (S pos (rU (d2r 180) `matMult` head) p):ss
```

Branching is handled using `push` and `pop`, triggered by `[` and `]` respectively. These simply implement the underlying stack behavior of the TurtleStack.

```
push                                    :: TurtleStack -> IO TurtleStack
push ss'@((S pos head p):ss)             = return $ (S pos head p) : ss'

pop                                     :: TurtleStack -> IO TurtleStack
pop (s:ss)                               = return ss
```

These pen functions modify the width and color index of the turtle:

```
decrWidth                               :: TurtleStack -> IO TurtleStack
decrWidth ((S pos head (Pen t w c cs)):ss)
                                         = return $ (S pos head (Pen t (w*0.95) c cs)):ss


setColor           :: Color -> IO ()
setColor (r, g, b)  = do
  glColor3f r g b

nextColor ((S pos head (Pen t w c cs)):ss) = do
  let c' = (c + 1) `rem` (length cs)
  setColor (cs !! c')
  return $ (S pos head (Pen t (w*0.95) c' cs)):ss
```

And finally, { and } trigger `polyBegin` and `polyEnd` to capture convex polygons which can be filled in, such as leaves and flower petals.

```
polyBegin                               :: TurtleStack -> IO TurtleStack
polyBegin ss                             = do
  glBegin gl_POLYGON >> return ss

polyEnd                                 :: TurtleStack -> IO TurtleStack
polyEnd ss                               = do
  glEnd >> return ss
```

Drawing the Pretty Pictures

In order to draw some pictures, first, let's have some L-Systems worth looking at:

```
tree :: LGrammar
tree  = (Lg "F"
         [Pr "F" "FF-[-F+F+F]+[+F-F-F]"])


bush :: LGrammar
bush  = (Lg "A"
          [Pr "A" "[&FL!A]/////'[&FL!A]///////'[&FL!A]",
           Pr "F" "S ///// F",
           Pr "S" "F L",
           Pr "L" "['''^^{-f+f+f-|-f+f+f}]"])
```

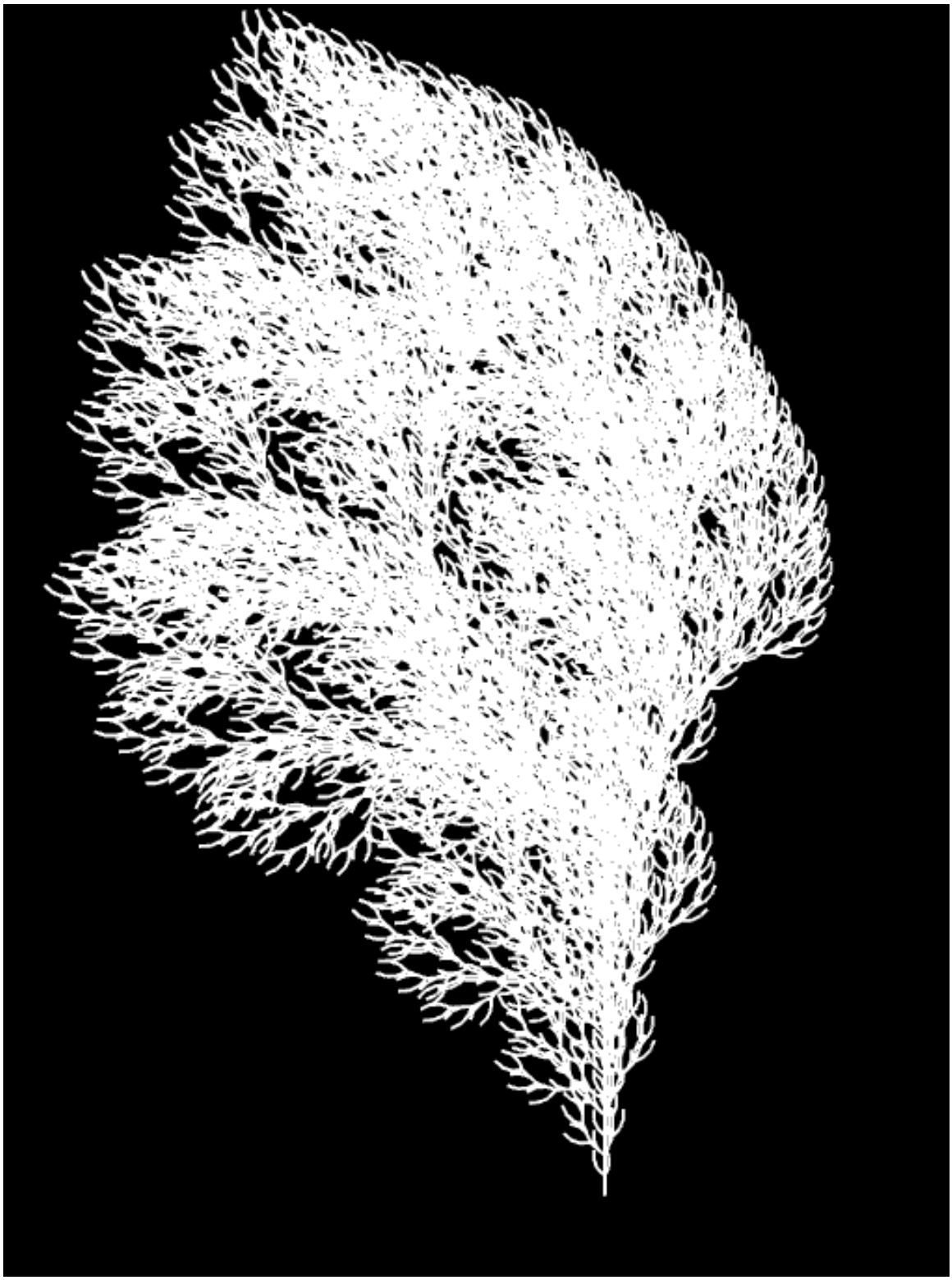

Figure 1: tree

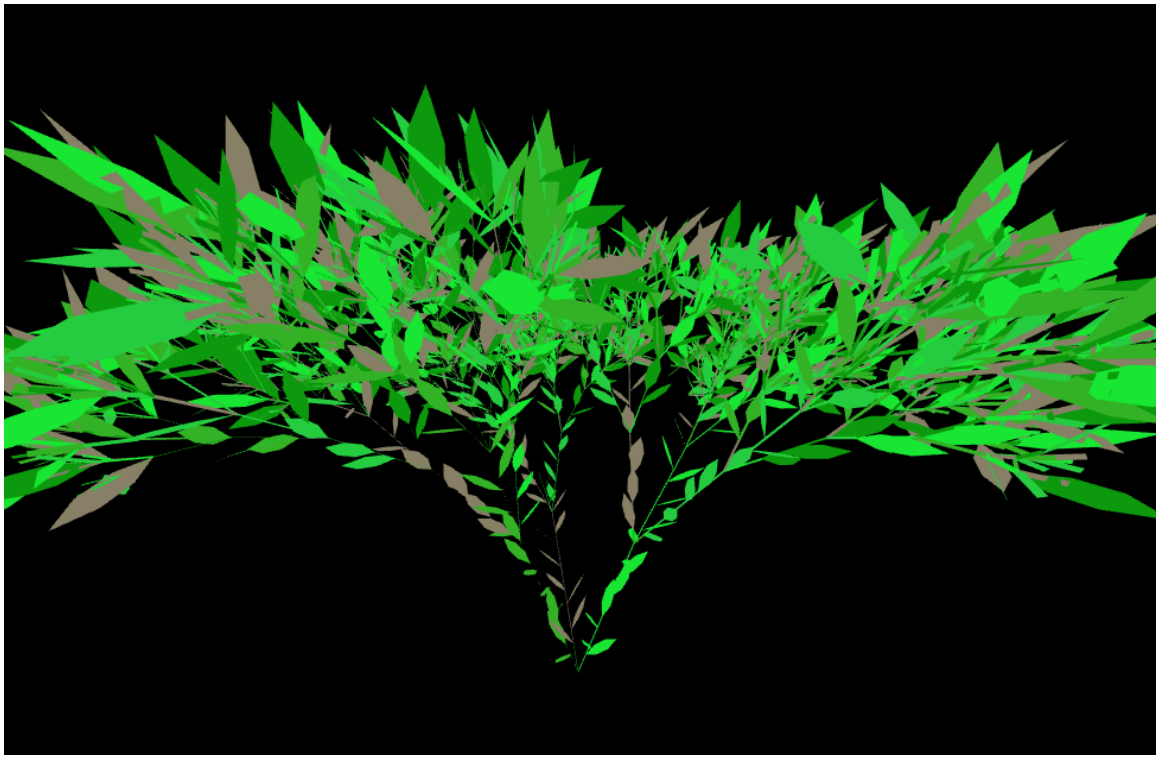

Figure 2: bush

```
plant :: LGrammar
plant  = (Lg "P"
          [Pr "P" "i+[P+o]--//[--l]i[++l]-[Po]++PF",
           Pr "i" "Fs[//&&l][//^^l]Fs",
           Pr "s" "FsF",
           Pr "l" "['{+f-ff-f+|+f-ff-f}]",
           Pr "o" "[&&&c'/w////w////w////w////w]",
           Pr "c" "FF",
           Pr "w" "['^F][{&&&&-f+f|-f+f}]"])
```

Now, `drawScene` is the core loop of the program. It takes a String of turtle commands and an IORef which represents the current angle of rotation at which to display the LSystem. It works by clearing the OpenGL context, translating and rotating the drawing matrix, and then `interpret`ing the string of commands to actually do the drawing. It then updates the rotation angle and flushes commands to OpenGL.

```
drawScene :: String -> IORef Scalar -> IO ()
drawScene s rtheta = do
  glClear $ fromIntegral  $  gl_COLOR_BUFFER_BIT
                         .|. gl_DEPTH_BUFFER_BIT
  glLoadIdentity
  th <- readIORef rtheta
  glTranslatef (0) (-100) (-80)
  glRotatef th 0 1 0
  _ <- interpret s 22.5
  writeIORef rtheta $! th + 0.5
  glFlush
```

Figure 3: plant

Summary

That, barring some OpenGL glue code, is the main body of work for this project. In many ways, I found this deeply satisfying, because it is by far the most complete implementation of L-Systems and turtle graphics I have ever successfully written. For many years I have wanted to build this, and I'm proud of what I've achieved.

I would very much like to do more – especially in the way of expanding the supported grammars, such as context-sensitive, stochastic, and parametric productions. Some of these seem within reach (such as stochasticism) and others seem like they may have to wait a while – like fully context-sensitive productions.

Creating this software in Haskell has been a very pleasant experience overall. I enjoy that Haskell has such strong opinions about code and how it should be structured, because it pushes me toward that ideal and encourages me to think harder about what I'm writing and how it should be expressed. I have several lingering refactors and enhancements I'd like to undertake, such as:

- splitting up the `generate` function and promoting some of the helper functions to the top level
- attempting a combinatoric approach to productions rather than a data- driven approach
- using Parsec to add support for an external file format
- adding support for user interaction with the 3d model

Bibliography

The Algorithmic Beauty of Plants, by Aristid Lindenmayer and Przemyslaw Pruszinkiewicz.

OpenGL Utiltity Functions

These were borrowed rather wholesale from a Haskell port of the very useful Neon-Helium OpenGL tutorials of yore (for values of yore in the neighborhood of 1999).

I have modified them slightly from the code in Hackage, though not substantially.

```
setupGraphics              :: Int -> Int -> IO ()
setupGraphics w h = do
  r <- newIORef 0
  True <- GLFW.initialize
  -- select type of display mode:
  -- Double buffer
  -- RGBA color
  -- Alpha components supported
  -- Depth buffer
  let dspOpts = GLFW.defaultDisplayOptions
                { GLFW.displayOptions_width  = w
                , GLFW.displayOptions_height = h
                -- Set depth buffering and RGBA colors
                , GLFW.displayOptions_numRedBits   = 8
                , GLFW.displayOptions_numGreenBits = 8
                , GLFW.displayOptions_numBlueBits  = 8
                , GLFW.displayOptions_numAlphaBits = 8
                , GLFW.displayOptions_numDepthBits = 8
```

```
                , GLFW.displayOptions_numFsaaSamples = Just 8
                -- , GLFW.displayOptions_displayMode = GLFW.Fullscreen
                }
  -- open a window
  True <- GLFW.openWindow dspOpts
  -- window starts at upper left corner of the screen
  GLFW.setWindowPosition 0 0
  GLFW.setWindowTitle "ls-hs"
  -- register the function to do all our OpenGL drawing
  let s = (generate tree 5)
  GLFW.setWindowRefreshCallback (drawScene s r)
  -- GLFW.setWindowRefreshCallback (drawScene rt rq)
  -- register the funciton called when our window is resized
  GLFW.setWindowSizeCallback resizeScene
  -- register the function called when the keyboard is pressed.
  GLFW.setKeyCallback keyPressed
  GLFW.setWindowCloseCallback shutdown
  -- initialize our window.
  initGL
  forever $ do
    drawScene s r
    GLFW.swapBuffers

initGL :: IO ()
initGL = do
  glShadeModel gl_SMOOTH -- enables smooth color shading
  glClearColor 0 0 0 0 -- Clear the background color to black
  glClearDepth 1 -- enables clearing of the depth buffer
  glEnable gl_DEPTH_TEST
  glDepthFunc gl_LEQUAL  -- type of depth test
  glHint gl_PERSPECTIVE_CORRECTION_HINT gl_NICEST

resizeScene :: GLFW.WindowSizeCallback
resizeScene w     0      = resizeScene w 1 -- prevent divide by zero
resizeScene w h = do
  glViewport 0 0 (fromIntegral w) (fromIntegral h)
  glMatrixMode gl_PROJECTION
  glLoadIdentity
  gluPerspective 120 (fromIntegral w / fromIntegral h) 0.01 200
  glMatrixMode gl_MODELVIEW
  glLoadIdentity
  glFlush

shutdown :: GLFW.WindowCloseCallback
shutdown = do
  GLFW.closeWindow
  GLFW.terminate
  _ <- exitWith ExitSuccess
  return True

keyPressed                  :: GLFW.KeyCallback
keyPressed GLFW.KeyEsc True  = shutdown >> return ()
keyPressed _           _     = return ()
```

```haskell
main :: IO ()
main = do
  setupGraphics 1000 750
```