"Vasile Alecsandri" University of Bacău
Faculty of Sciences
Scientific Studies and Research
Series Mathematics and Informatics
Vol. 25 (2015), No. 1, 205-224

BUILDING AN AVATAR IN HASKELL 98 PRELIMINARIES: IS HOPENGL ENOUGH ?

DAN POPA

Abstract. In this paper the author is trying to put the HOPENGL, the 3D graphic library available in Haskell to work, in order to check if it's good enough to conveniently build a 3D avatar.

1. THE IDEA

Usually, in standard structured programming languages, there is a property of the statements which is actually missing. The commands can *not* be manipulated like data pieces. So, implementing a flexible program, which is building a graphic representation according to an other description usually needs special interpretation or compiling on the fly technologies and some kind of data structures, usually trees , in order to represent the avatar. But …

2. HASKELL IS DIFFERENT

In the functional language Haskell, the IO is based on a different kind of algebraic data structure: the IO monad. The elements of the IO monad are called actions, and they are simultaneously commands which can be (apparently imperatively) used, and, in the same time, data structures (see Haskell Reference and Report....[6] or other manuals available, like [4]).

Keywords and phrases: HOpenGL, Avatar, Haskell, IO monad
(2010) Mathematics Subject Classification: 68U05

This fact had leading us to a different method of building avatars, which is not based on interpretation or compilation, but on what was called in [7] pseudoconstructors on monadic values. The notion is also presented on the internet, having a separate page on the website, at the time when this article was written [10]. We have also to mention some other properties of this kind of tree representation: it is flexible and modular, can be spread across many modules of the project and it is not depending on an unique, fixed declaration of the tree data type – the usual kind of *data* declaration
used by Haskell programmers when they have to declare the data type of a tree.

As we have already shown during the Rodin Project [11] and in some compilers like [8], [9] and papers [Building a modular compiler], when using pseudoconstructors as tree representations the compiler or the interpreter becomes highly modular and the kernel of both of them become a small, (almost generic) program, no longer than one or two lines of code.

The representation of an avatar being possibly to be made as a tree, it also become possible as a tree of pseudoconstructors on monadic actions, as a tree of pseudoconstructors on IO monadic actions, etc. And that leads us to the idea of defining an avatar as a tree of pseudoconstructors on monadic IO HOPENGL based actions.

Being a tree, a data structure, such a representation of the avatar can be defined and modified interactively by the user, as he/she wish. But being simultaneously a structure of actions in he IO() monad, it becomes immediately run-able, because this actions are similar with those used in the main() function of the program. (In Haskell, the main() function has the type IO() - exactly the IO monad). A small practical problem remain: Is the HOPENGL implementation of Swen Paine [HOPENGL implementation of Swen Paine] usable for this task ?

3. AN IMPLEMENTATION IN HOPENGL AS A TEST

To check how powerful The HOPENGL library is, to check if it is suitable for our task, we had imagine a simple test. A classic OpenGL program which is drawing an avatar, selected from a course of by J.García held at The Deusto University [1],[2],[3] was rewritten, (first of all being simply translated) then being subject of re-engineering (not only simply translated, rebuild based on pseudoconstructors). If can be done, the test is succeeding

and we can conclude that avatars can be built using pseudoconsructors over graphic monadic values using the IO() monad and actions from the HOPENGL library.

The original program, in C, which was used like a source of inspiration can be found in various versions of [1],[2],[3], chapter 3.

4. DOCUMENTATION AND DEVELOPMENT STAGES, STEP BY STEP

1. The work had begun by locating a course book in avatars design, which uses the original C library: The OpenGL.
2. The selected book was, as we had said, [1],[2] . This is freely available on the Internet and the v.1.1 version is based on a Creative Common Licence. Reimplemented in C all the programs from the first 5 chapters of the book was the second step. As a secondary result, a translated edition of J.García's book [3] was published by Alma Mater Publishing House.
3. Running an testing some HOPENGL demo programs by Swen Paine, which are closely matching the examples from the [12] was an other step. The examples was a bit more complicated than García's examples, because the Red Book itself are treating more complex questions than García introductory manual. That helped us to find some technical details, to notice some differences between OpenGL and HOPENGL and also makes us confident in the success of our work. Such examples can be found accompanying the Hugs distribution, nowadays.
4. The examples made by J.García for his course was then rewritten in Haskell. The task was not trivial: Haskell is a pure functional programming language and C was a de facto standard in imperative programming. Finaly, what J.García called the *simpleguy.c* example was rebuilt and tested. It was a success. Some pictures of those tests are included here – in fact we was able to reproduce García's results and examples in Haskell. But we do not make a course here, and as a consequence, all those source codes are not included in this paper. The notations was, more or less, preserved, in order for the reader to spot the differences and similarities between OpenGL programing in C [12] and HOPENGL programming in Haskell.

5. SOME NOTICES CONCERNING HOPENL REENGINEERED CODE

Due to the differences in the nature of languages and some decisions made by the designers of HOPENGL libraries and functions, some things should be noticed and noted:

a) The names of the GL and GLUT libraries are, for Haskell programmers: *Graphics.UI.GLUT* and, also, *Graphics.Rendering.OpenGL*. Both should be included using the *import (there is no #include in Haskell, but import is used).*

b) Even if theoretically the Haskell programs did not have a context of variables,(a condition of being pure functional programming) a set of zero-arguments functions can be used to define the constants we need. As a result we can define constants like:

```
body_height=4.0
```

even if Haskell did not have an assignement statement in the common sense of this command in imperative languages.

c) The "gl" prefix of the functions from OpenGL libraries was removed when HOPEGL was implemented. Also a $= operator was introduced, in order to set up some parameters of the OpenGL environment from Haskell's IO() monad.

d) Some parameters of the OpenGL – which can or cannot be used by a specific application – are implemented via a specific Haskell data type: The Maybe monad. For example:

```
depthFunc $= Just Lequal
```

Remember: The Haskell programmer should know the names : *Lequal / Greater / Gequal* which are used in that instance of the Maybe monad. And, also notice that Haskell programmers can use *Nothing* as a value, in similar contexts:

```
depthFunc $= Nothing
```

e) Because Haskell language did not use commas, a 4 value color is immediately represented by simply writing the values of the RGBA components preceded by a data constructor, like:

```
Colour4  0.0  0.0  0.0  0.0 --  this means non transparent black.
```

f) Sometime, especially when writing simple programs, due to some overloaded definitions of functions like *vertex, colour, rotate* some *let ... in* ... keywords shoud be used for dissambiguation. Swen P. noticed that in this package of examples. And sometimes this let … in … can be used to speed up a program, by computing some values just one time. The reader may want to rebuild the "recusive_hierarchy.c" program from [1],[2],[3]. There

m=lat/2 and ml=-lat/2 can also be defined in that *let... in...* simplifying the next recursive calls.

g) *PushMatrix()* and *PopMatrix()* which are always working in pairs are replaced in HOPENGL by a single function: *preservingMatrix* which is usually applied to a do {…} sequence or an IO() action of any kind.

h) Sometimes, the programmer should work carefully when using types. Haskell is a well typed and strongly typed language. An example: When approximating a sphere by a sliced polyhedral object, the number of slices should be integer (for example 10). So it should be written as:

renderObject Solid (Sphere' Radius 10 10).

In our opinion, even if this differences exists, only two are of importance and should be carefully handled: d) and g). The d) point requires the programmer to learn one more times the names of some usual operators. But, in fact, the g) point makes the life of the programmer easier because he/she should not check if every pushMatrix() call have an corresponding popMatrix() call. But the curve brackets {…} still should be correctly paired. Functions like glBegin() and glEnd() are no longer needed.

6. THE FIRST VERSION OF THE REWRITTEN PROGRAM

Here is a version of our test code which was keep tight enough to he standard OpenGL style of programming, even if Haskell allows more differences:

```
-- garex5v3.hs

-- Desenare umanoid
import Graphics.UI.GLUT  -- as GLUT
import Graphics.Rendering.OpenGL

body_Height=              4.0
body_Width =       2.5
body_Length=              1.0

arm_Height=        3.5
arm_Width = 1.0
arm_Length=        1.0

leg_Height=        4.5
leg_Width = 1.0
```

```
  leg_Length=        1.0

  head_Radius=       1.1

  main = do
    (name,_)<-getArgsAndInitialize
    initialDisplayMode $=[SingleBuffered,RGBMode,WithDepthBuffer]
    initialWindowSize $= Size 500 500
    createMyWindow name
    mainLoop

  createMyWindow windowname = do {
    createWindow windowname;
    clearDepth $= 1      ;              -- Va lucra cu adancimi
    depthClamp $= Enabled;              -- comparatiile fiind cu <=
    depthFunc $= Just Lequal;           --Lequal | Greater | Gequal
    displayCallback $= display;
  }

  -- Ce putem scrie in let.. ul de mai sus ca sa simplifcam notatia
--si sa acceleram calculele ? Observati ce se repeta.

  display = do {
    clearColor $= Color4 0.0 0.0 0.0 0.0;-- Culoarea de fond:negru
    clear [ColorBuffer,DepthBuffer];  -- Stergem bufferul de culori
                                        -- si cel de adancimi
    matrixMode $= Projection;       -- Trecem in mod proiectie
    loadIdentity;                   -- Incarcam matricea identitate
    perspective 60.0 1.0 1.0 100.0;

    let translate3f=translate :: Vector3 GLfloat ->IO()
        color3f    =color  :: Color3  GLfloat -> IO()
    in  do {color3f (Color3 1.0 0.0 0.0 );
            translate3f (Vector3 0.0 0.0 (-16.0));
  -- Este -16.0  matrixMode $= Modelview 1;
  -- Trecem in modul desenare model desenareUmanoid;
           } ;
    flush;                    --  Fortam trecerea la desenare
   }



  desenamCorpul= do {
```

```
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        color3f    =color  :: Color3  GLfloat -> IO()
        -- rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
  in do {
     translate (Vector3 0.0 (body_Height/2.0) 0.0) ;
  -- Translatie cf.vectorului
     preservingMatrix $ do {
        scale body_Width body_Height body_Length ;
  -- folosim scale
        color3f (Color3 0.0 0.3 0.8);
        renderObject Solid (Cube 1.0);
     };
   }
  }


  desenamManaDreapta= do{
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        color3f    =color  :: Color3  GLfloat -> IO()
        -- rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
  in do {
        translate (Vector3 0.0 (-(arm_Height+arm_Width)/2.0) 0.0) ;
        color3f (Color3 1.0 0.6 0.6);
        scale arm_Width arm_Width arm_Length ;
        renderObject Solid (Cube 1.0);
   }
  }

  desenamBratulDrept= do {
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        -- color3f    =color  :: Color3  GLfloat -> IO()
        rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
  in do {
     preservingMatrix $ do {
        translate    (Vector3    (-body_Width/2)    ((body_Height-
arm_Height)/2.0) 0.0) ;
        translate (Vector3 0.0 (arm_Height/2.0) 0.0) ;
        rotatef  (-30.0) (Vector3 0.0 0.0 1.0);
  -- rotim cu 30 de grade
        translate (Vector3 0.0 (-arm_Height/2.0) 0.0) ;
        preservingMatrix $ do {
          scale arm_Width arm_Height arm_Length ;
  -- la dimensiunile bratului
```

```
          renderObject Solid (Cube 1.0);
          };
        -- La capatul bratului este o mana
        desenamManaDreapta;
     };
   }
  }


  desenamManaStanga= do {
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        color3f    =color  :: Color3  GLfloat -> IO()
        -- rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
  in do {
        translate (Vector3 0.0 (-(arm_Height+arm_Width)/2.0) 0.0) ;
        color3f (Color3 1.0 0.6 0.6);
        scale arm_Width arm_Width arm_Length ;
        renderObject Solid (Cube 1.0);
   }
  }

  desenamBratulStang = do {
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        color3f    =color  :: Color3  GLfloat -> IO()
        rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
  in do {
     color3f (Color3 0.0 0.3 0.8);
  -- preserveMatrix nu restaureaza culoarea   preservingMatrix
  $ do {
        translate    (Vector3    (body_Width/2.0)    ((body_Height-
arm_Height)/2.0) 0.0) ;
        translate (Vector3 0.0 (arm_Height/2.0) 0.0) ;
        rotatef  (30.0) (Vector3 0.0 0.0 1.0);
  -- rotim invers 30 de grade
        translate (Vector3 0.0 (-arm_Height/2.0) 0.0) ;
        preservingMatrix $ do {
          scale arm_Width arm_Height arm_Length ;
  -- la dimensiunile bratului
          renderObject Solid (Cube 1.0);
          };
        -- La capatul bratului este o mana
        desenamManaStanga;
     };
```

```
   }
  }


  desenamTalpaDreapta= do {
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        color3f    =color  :: Color3  GLfloat -> IO()
        -- rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
  in do {
        translate    (Vector3    0.0    (-(leg_Height+leg_Width)/2.0)
leg_Length) ;
        color3f (Color3 0.3 0.4 0.4);
        scale leg_Width leg_Width (leg_Length*2) ;
        renderObject Solid (Cube 1.0);
   }
  }

  desenamPiciorulDrept= do {
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        color3f    =color  :: Color3  GLfloat -> IO()
        -- rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
  in do {
     color3f (Color3 0.0 0.3 0.8);
     -- preserveMatrix nu restaureaza culoarea
     preservingMatrix $ do {
        translate (Vector3 (-(body_Width-leg_Width)/2.0)
                           (-(body_Height+leg_Height)/2.0) 0.0) ;
        preservingMatrix $ do {
          scale leg_Width leg_Height leg_Length ;
          -- la dimensiunile piciorului
          renderObject Solid (Cube 1.0);
          };
        -- La capatul piciorului este o ... talpa
        desenamTalpaDreapta;
     };
   }
  }


  desenamTalpaStanga= do {
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        color3f    =color  :: Color3  GLfloat -> IO()
        -- rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
```

```
  in do {
        translate    (Vector3    0.0    (-(leg_Height+leg_Width)/2.0)
leg_Length) ;
        color3f (Color3 0.3 0.4 0.4);
        scale leg_Width leg_Width (leg_Length*2) ;
        renderObject Solid (Cube 1.0);
   }
  }

  desenamPiciorulStang= do {
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        color3f    =color  :: Color3  GLfloat -> IO()
        -- rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
  in do {
     color3f (Color3 0.0 0.3 0.8);
  -- preserveMatrix nu restaureaza culoarea
     preservingMatrix $ do {
        translate (Vector3 ((body_Width-leg_Width)/2.0)
                           (-(body_Height+leg_Height)/2.0) 0.0) ;
        preservingMatrix $ do {
          scale leg_Width leg_Height leg_Length ;
  -- la dimensiunile piciorului
          renderObject Solid (Cube 1.0);
          };
        -- La capatul piciorului este o ... talpa
        desenamTalpaStanga;
     };
   }
  }

  desenamCapul= do {
  let   -- vertex3f   =vertex :: Vertex3 GLfloat -> IO()
        color3f    =color  :: Color3  GLfloat -> IO()
        -- rotatef    =rotate :: GLfloat->Vector3 GLfloat ->IO()
  in do {
     color3f (Color3 1.0 0.6 0.6);
     preservingMatrix $ do {
        translate (Vector3 0.0
                           (body_Height/2.0+head_Radius*3.0/4.0)
--Head Radius
                           0.0) ;
        renderObject Solid (Sphere' head_Radius 10 10);
        --10 e nr intreg de parti
```

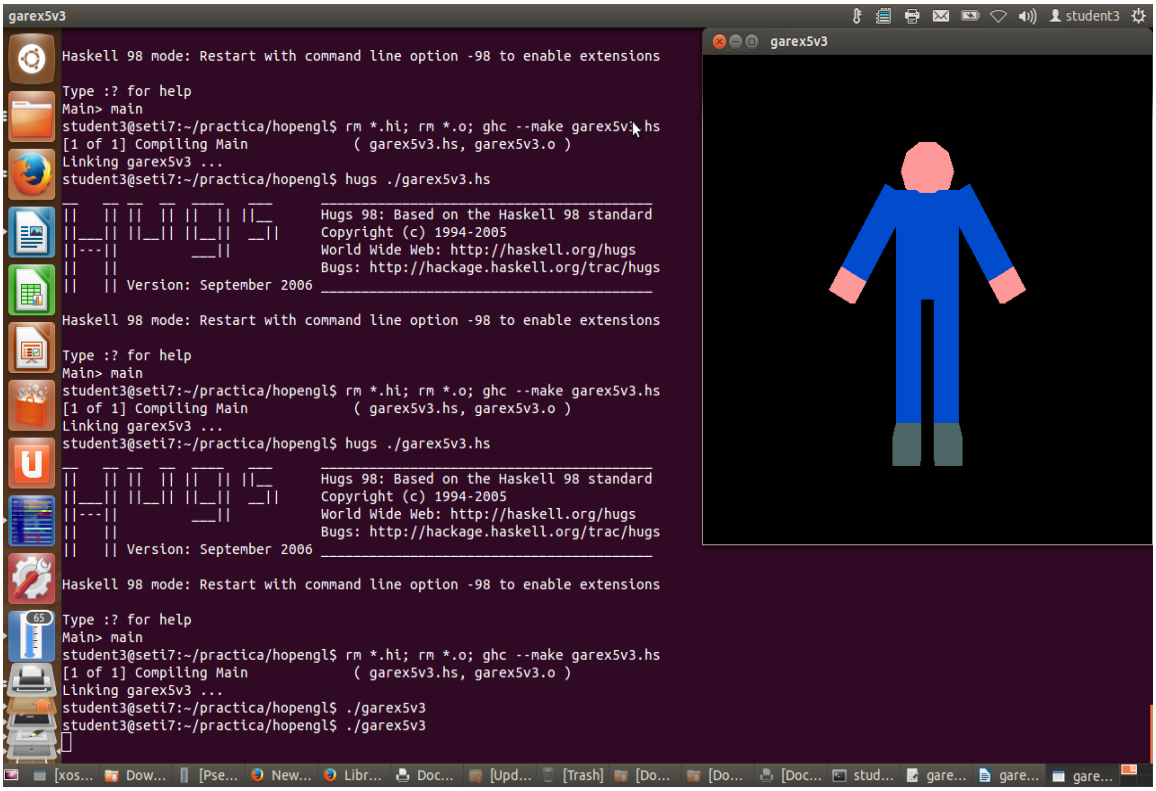
```
      };
    }
  }

  desenareUmanoid =  do {
  -- In order to draw the humanoid we should ...
     -- Desenam corpul / Draw the body
     desenamCorpul;
     -- Desenam bratul drept / Draw the right arm
     desenamBratulDrept;
     -- Desenam bratul stang / Draw the left arm
     desenamBratulStang;
     -- Desenam piciorul drept / Draw the right leg
     desenamPiciorulDrept;
     -- Desenam piciorul stang / Draw the left leg
     desenamPiciorulStang;
     -- Desenam ce ne-a mai ramas, capul / Draw the (remaining)
head
     desenamCapul;
  }
```

The program of a simple avatar, (J.García's C code is rewritten in Haskell).

We have used the Haskell Platform as it was included in the Ubuntu 12.04 for 32 bits machine distribution. In order to compile the source we had used the ghc compiler. But the Hugs interpreter can also be used. (The .o and .hi files remaining from a previous compilation should be deleted every time, before recompiling the source).

And here is the expected result, which looks exactly like the example from [1],[2],[3].



From statements in brackets to lists of actions

Fig 1. Runing the Haskell program to draw a simple avatar from J.García's book.
(we have also reuse the colors)

Usually, the computer graphic programmer creates images by executing statements over the available API. In this case OpenGL (here actually wrapped in Haskell). As an effect, in Haskell, the program have to be written using what is called do notation. We used it above. Here is a simple example, where the actual do notation used for drawing is marked using bold.

```
  -- Ex: desenarea unui patrulater; Dan Popa dupa Swen Eric Panitz [5] si
J.Garcia

  import Graphics.UI.GLUT
  import Graphics.Rendering.OpenGL

  main = do
    (name,_)<-getArgsAndInitialize
    createMyWindow name
```

```
  mainLoop

createMyWindow windowname = do
  createWindow windowname
  clear [ColorBuffer]
  displayCallback $= display

display = do {
  clearColor $= Color4 0.0 0.0 0.0 0.0;  -- Culoarea de fond: negru
  clear [ColorBuffer];            -- Stergem bufferul de culori
  matrixMode $= Projection;       -- Trecem in mod proiectie
  loadIdentity;                   -- Incarcam matricea identitate
  ortho (-1.0)  1.0 (-1.0)  1.0  (-1.0)  1.0;
  matrixMode $= Modelview 0;     -- Trecem in modul desenare model
  let vertex3f =vertex :: Vertex3 GLfloat -> IO()
      color3f  =color  :: Color3  GLfloat -> IO()
  in
   renderPrimitive Quads  $        -- nu uita 's'
   do  {
       color3f (Color3 0.0 1.0 1.0); -- Culoarea varfului dintai: vernil
       vertex3f (Vertex3 (-0.5) 0.5 (-0.5));-- Coordonatele primului varf
       vertex3f (Vertex3 (-0.5) (-0.5) 0.5);--Coordonatele varfului #2
       vertex3f (Vertex3 0.5 (-0.5) 0.5); -- Coordonatele varfului #3
       vertex3f (Vertex3 0.5 0.5 (-0.5)); -- Coordonatele varfului #4
       };
  flush;                    --  Impunem trecerea la desenare
 }
```

An other example from García's book, rewriten in Haskell using the *do-notation*

In order to run the previous example uisng GHC, the following commands was used:

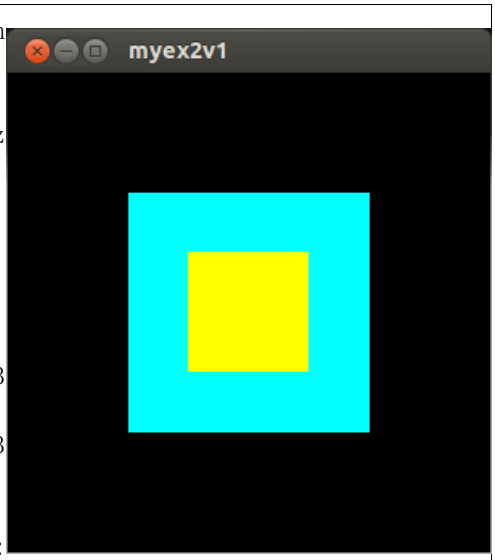
```
radacina@Seti:~/practica/hopengl$ ghc -package GLUT -o garex2v1 garex2v1.hs
[1 of 1] Compiling Main             ( garex2v1.hs, garex2v1.o )
Linking garex2v1 ...
radacina@Seti:~/practica/hopengl$ ./garex2v1
```

If the reader wants to run the example using the Hugs interpreter, it can be done by using:

```
radacina@Seti:~/practica/hopengl$ hugs garex2v1.hs
Main> main
```

But the mathematician, especially when he or she is coming from the field of geometry is less familiar with the do notation or with C like languages. But is familiar with the concept of set. And because Haskell may use IO() actions like elements of a set, too, and it is providing a *sequence_* function which is combining a list of type [IO()] in a single IO() action, others approaches are possible, using lists and *data structures of actions*. Here is a similar but more simple program, rewritten in order to to use lists of actions. Note how simple can we write the action of rendering:

```
done = renderPrimitive Quads $ sequence_ dots3
```




```
-- Desenarea unuor patrulatere din liste de
--  actiuni de [IO()]
-- Dan Popa dupa Swen Eric Panitz --[5] si J.Garcia

import Graphics.UI.GLUT

import Graphics.Rendering.OpenGL

vertex3f =vertex :: Vertex3 GLfloat -> IO()
color3f =color :: Color3 GLfloat -> IO()

vertex3fc :: (GLfloat,GLfloat,GLfloat) -> IO()
vertex3fc (x, y, z) = vertex3f (Vertex3 x y z)

main = do
  (name,_) <- getArgsAndInitialize
  createMyWindow name
  mainLoop

dots::[IO()]
dots = [vertex3f (Vertex3 (-0.5) 0.5 (-0.5)),-- Coordonatele primului varf
```

```
    vertex3f (Vertex3 (-0.5) (-0.5) 0.5), --Coordonatele varfului #2
    vertex3f (Vertex3 0.5 (-0.5) 0.5),    -- Coordonatele varfului #3
    vertex3f (Vertex3 0.5 0.5 (-0.5))     -- Coordonatele varfului #4
       ]

dots2::[IO()]
dots2 = map vertex3f
       [ (Vertex3 (-0.5) 0.5 (-0.5)),  -- Coordonatele primului varf
         (Vertex3 (-0.5) (-0.5) 0.5),  -- Coordonatele varfului #2
         (Vertex3 0.5 (-0.5) 0.5),     -- Coordonatele varfului #3
         (Vertex3 0.5 0.5 (-0.5))      -- Coordonatele varfului #4
       ]

dots3::[IO()]
dots3 =
       [color3f (Color3 0.0 1.0 1.0)] ++
       map (vertex3fc)
       [  ((-0.5), 0.5, (-0.5)),  -- Coordonatele primului varf
          ((-0.5), (-0.5), 0.5),  -- Coordonatele varfului #2
          (0.5, (-0.5), 0.5),     -- Coordonatele varfului #3
          (0.5, 0.5, (-0.5))      -- Coordonatele varfului #4
       ] ++
       [color3f (Color3 1.0 1.0 0.0)] ++
       map (vertex3fc)
       [  ((-0.25), 0.25, (-0.75)), -- Coordonatele primului varf
          ((-0.25), (-0.25), 0.75), --Coordonatele varfului #2
          (0.25, (-0.25), 0.75),    -- Coordonatele varfului #3
          (0.25, 0.25, (-0.75))     -- Coordonatele varfului #4
       ]

 -- Grupam actiunile de pelista  intr-o singura actiune compusa:
 --randarea
done = renderPrimitive Quads $ sequence_ dots3
-- incercati dots,dots2...

createMyWindow windowname = do
  createWindow windowname
  clear [ColorBuffer]
  displayCallback $= display

display = do {
  clearColor $= Color4 0.0 0.0 0.0 0.0;  -- Culoarea de fond: negru
  clear [ColorBuffer];                   -- Stergem bufferul de culori
```

```
    matrixMode $= Projection;              -- Trecem in mod proiectie
    loadIdentity;                       -- Incarcam matricea identitate
    ortho (-1.0)  1.0 (-1.0)  1.0  (-1.0)  1.0;
    matrixMode $= Modelview 0;          -- Trecem in modul desenare model
    -- color3f (Color3 0.0 1.0 1.0);
    -- Culoarea varfului dintai vernil
    done;
    -- Actiunea compusa de desenare
    flush;
    -- Impunem trecerea la desenare
   }
```

An other example from García's book, rewritten using lists of actions (various forms)

In order to compile this example using GHC, the following commands can be used:

```
  radacina@Seti:~/practica/hopengl$ ghc -package GLUT -o myex2v1
myex2v1.hs
  [1 of 1] Compiling Main             ( garex2v1.hs, garex2v1.o )
  Linking garex2v1 ...

  radacina@Seti:~/practica/hopengl$ ghc --make myex2v1.hs
```

If the reader wants to run the example using the Hugs interpreter, it can be done by using:

```
  radacina@Seti:~/practica/hopengl$ hugs myex2v1.hs
```

7. CONCLUSION

As a conclusion, working on Graphics using Haskell and HOpenGL have some advantages:

1) We are not stick to a set of statements in a specific order. Now the graphic statements can be placed on lists, on trees, selected from there, combined in whatever order we need, combined in a big IO() action. After that they can be drawn by running the program.
2) Graphics are made using the IO() actions which have a strong algebraic structure (the IO() monad).
3) Haskell provides a system for *type inferences*, so it is simple to check

if your graphic expression fits somewhere in the program by checking it's type. (Ex: type :t dot3 in the console to see the answer.)

4) We can use mathematics, especially *sets and trees of graphic actions*.
5) Lists of vertices can be immediately converted in lists of actions using the common *map* function and then composed in a big action using (monadic) *sequence_* operator. There exist even a monadic *map_* operator in Haskell libraries. See [6].

There are also some things to take care of:

1) Some functions and constructors are overload so, in small programs, they need specific *type signatures*. But both programmers and mathematicians used to declare the type of a function, it's domain and it's co-domain. In fact, in Haskell we did not need to declare the domains for *all* of our functions, because the type inference system (Hindley Milner Algorithm) is doing the rest.
2) Due to the need of color changes, the lists of IO actions for a vertex are concatenated with other IO actions. (See: dot3s.). But when an object have a lot of faces of the same color it's possible to use a list of faces, for example Quads or Triangles.
3) The order of the vertices on the list is important for the rendering of some primitives – that's inherited from OpenGL. This is a common problem in graphic programming.
4) Sometime some conversion functions can be needed. This one creates an IO() action from the triple of coordinates of a point in 3D space.

```
vertex3fc :: (GLfloat,GLfloat,GLfloat) -> IO()
vertex3fc (x, y, z) = vertex3f (Vertex3 x y z)
```

Next step of the experiment

There are something which have been teste dby us and should also be tested by the reader:

1) To change the projection from orthographic to perspective.
2) To comment or uncomment or even add different colors.
3) To use this one or an other sets of actions. See dots, dots2, dots3, …
4) To rebuild the classic humanoid from the first program using sets or pseudoconstructors. Pseudoconstructors, as in [7],[8],[9],[10] was usefull for the build of composed structures of actions which can be modulars and placed in different included files. Even entire languages

was built in such a modular way [11]. Let's do it:

The conversion from IO() actions like *desenamManaStanga:: IO()* to pseudoconstructors is simple. It consist in the folowing steps:

1) First of all we have to locate all named internal actions in the description of an action. For example the action of drawing the left hand *desenamManaStanga* is including an other action *desenamPalmaStanga* inside of it.

2) In the next phase, the included actions are replaced by at least one parameter (named *dps* for example), and this parameter is included in the signature of the function. As a result the new type of *desenamManaStanga* will not be *IO()* but something like *desenamManaStanga:: IO() -> IO().* This will be used with the requested parameter, like: *desenamManaStanga desenamPalmaStanga.*

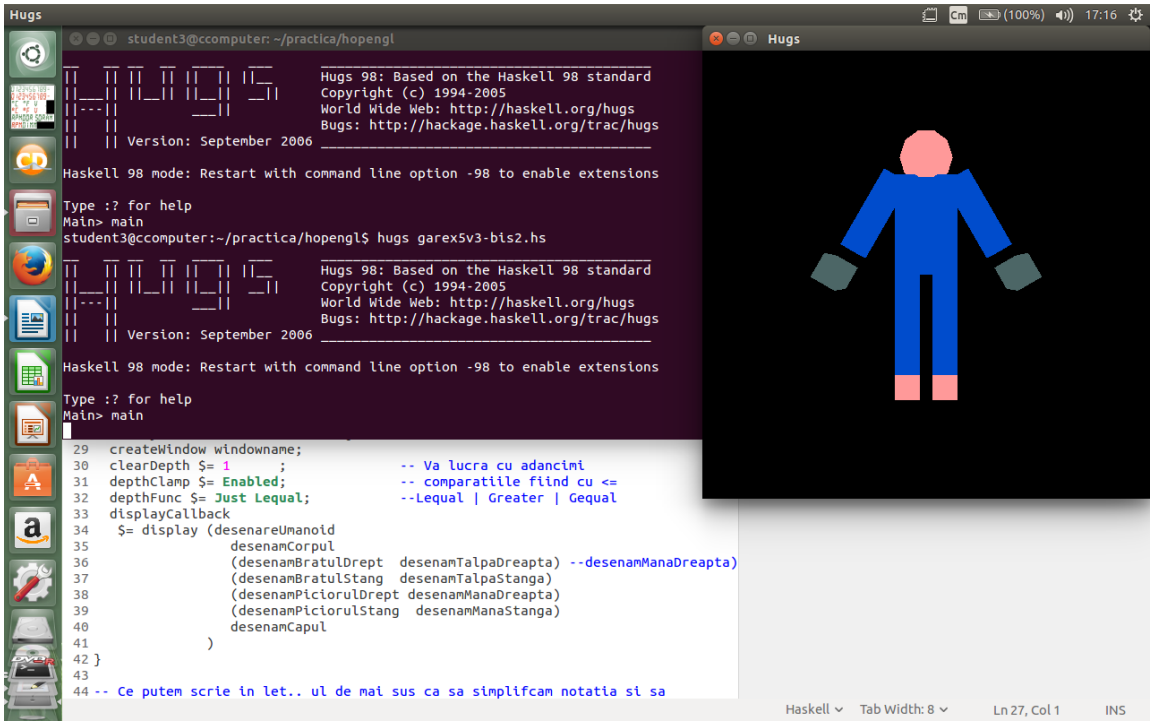
3) The *display* action will also be changed in the same way, becoming a pseudoconstructor itself.

4) Finaly the new *display* action will be called with parameters (which are describing the whole structured image) when the *displayCallback* of the HOpenGL is defined:

```
createMyWindow windowname = do {
  createWindow windowname;
  clearDepth $= 1      ;              -- Va lucra cu adancimi
  depthClamp $= Enabled;               -- comparatiile fiind cu <=
  depthFunc $= Just Lequal;           --Lequal | Greater | Gequal
  displayCallback
   $= display (desenareUmanoid
               desenamCorpul
               (desenamBratulDrept  desenamTalpaDreapta)
               (desenamBratulStang  desenamTalpaStanga)
               (desenamPiciorulDrept desenamManaDreapta)
               (desenamPiciorulStang  desenamManaStanga)
               desenamCapul
             )
}
```

The display Callback is now including the entire structure of the drawing, represented as a tree creating using pseudoconstructors over IO() actions.

Exchanging the positions of the body's parts.



References

[1] García Jorge (Bardok), Curso de introduccion a OpenGL (v1.0), 2003

[2] García Jorge (Bardok), Curso de introduccion a OpenGL (v1.1), 2003

[3] García Jorge , Manual introductiv de OpenGL, Alma Mater Publishing House, ISBN 978-606-527-349-8

[4] Hudak Paul , John Peterson, Joseph H. Fasel, A Gentle Introduction to Haskell 98, 2000

[5] Paniz Sven Eric, HopenGL – 3D Graphics with Haskell, A small tutorial (Draft), version 9th October 2003, TFH Berlin

[6] Peyton Jones,Simon : "Haskell 98 language and libraries: the Revised Report", Cambridge University Press, 2003, ISBN 0521826144

[7] Popa Dan , Adaptable Software – Modular extensible monadic evaluator and typechecker based on pseudoconstructors, ARA35-119, ARA Congress, Timisoara, 2011, http://www.haskell.org/wikiupload/7/78/Popa_Dan_fullpaper_template.pdf.zip (draft)

[8] Popa Dan , How to build a modular monadic extensible compiler using The State Monad and pseudoconstructors over monadic values, Scientific

Studies and Research. Series Mathematics and Informatics, vol. 21, no.2, 2011, pag. 97-116

[9] Popa Dan , Modular Monadic Compilers for Programming Languages

http://www.haskell.org/haskellwiki/Modular_Monadic_Compilers_for_Programming_Languages

[10] Popa Dan, Pseudoconstructors over monadic values.
http://www.haskell.org/haskellwiki/Pseudoconstructors

[11] Popa Dan, The Rodin Language Project, http://www.haskell.org/haskellwiki/Rodin

[12] Shreiner Dave, OpenGL programming guide: The official guide to learn OpenGL versions 3.0 and 3.1, Pearson Education Inc., 2010 http://www.opengl-redbook.com/ .

Extra references and resources

Panne Swen - HOpenGL implementation -
http://www.haskell.org/haskellwiki/Applications_and_libraries/Games
http://www.haskell.org/haskellwiki/OpenGLTutorial1
http://www.haskell.org/haskellwiki/OpenGLTutorial2 [HOPENGL implementation of Swen Panne]

http://www.ics.uci.edu/~gopi/CS211B/opengl_programming_guide_8th_edition.pdf

OpenGL Code Resources, http://www.opengl.org/wiki/Code_Resources

Department of Mathematics, Informatics and Education Sciences
Faculty of Sciences
"Vasile Alecsandri" University of Bacău
157 Calea Mărăşeşti, Bacău, 600115, ROMANIA
e-mail: popavdan@yahoo.com