Hygame: Teaching Haskell Using Games

Zachi Baharav and David S. Gladstein

Cogswell Polytechnical College,
1175 Bordeaux Drive, Sunnyvale, California, USA
{zbaharav,dgladstein}@cogswell.com
http://www.cogswell.edu

Abstract. This paper follows two dominant trends, and describes an effort to adapt the education and learning of Functional Programming to these trends. The first trend is that of human-computer interaction being dominated by graphical means; the second trend is the gamification of education. Toward this aim, we describe a new package we developed that allows users to program pure functions and assimilate the principals of functional programming, while producing visual interactive programs right from step one. This is similar to what is afforded to Python teachers by using Pygame. We further describe the envisioned usage of such a program in a course, the implied technical tradeoffs imposed on the design, and how we intend to deploy this in various channels to encourage usage.

Keywords: Hygame, Pygame, Education, Teaching, Course, Visualize, Graphics

1 Introduction

More than a quarter of a century ago, Kerninghan & Ritchie coined the use of the very simple "`hello, world`" program as the first one to try in a programming course [1]:

```
#include <stdio.h>
main()
{
  printf("hello, world\n");
}
```

Fig. 1. Original "`hello, world`" program from K&R book [1].

Today, though many things have changed in the programming arena, it is still very common to have "`hello, world`" as the first program in many languages and settings, just dipping your toe in the water. However, many of those first programs now produce new windows appearing on the screens, or new buttons showing on a browser, mobile device, and so on. Moreover, whereas the original C programming language book focused on simple text-based programs (copying and handling files, manipulating arrays, etc.), today most of the beginners' programming courses are done through game programming and interactive applications. This is due to two underlying trends, involving human-computer interaction and gamification of education:

Human-computer interface - Most of the user interaction with computers nowadays is through graphic interfaces, most notably mobile devices. The students are used to having graphics in applications, and want their program exercises to feel the same. This is true for whether the interaction is on Facebook using a browser, or slides preparation on a laptop, or especially so for applications on a mobile device.

Gamification - Gamification of learning refers to the method of motivating students to learn by being involved in a game setting. This can be by presenting the various tasks as part of a game, or letting the students design a relevant game, or compete with other teams (real or virtual) in a game setting. The main goal of this approach is to increase the enjoyment and engagement of students. Many students are much more engaged once they can produce interactive games with the programming exercises [7–10].

However, in many programming languages, creating graphics is still a cumbersome process. Creating a "window" on the screen, drawing or placing items on it, and interacting with the operating system, are still not simple things. The way many of those languages have overcome the burden for the user is by using tailored libraries or modules that hide this complexity for simple and common cases. An example of such a module is Pygame [2, 4, 3, 6], used in Python. With only a few instructions (see Fig. 2) , users new to Python can set up a window, draw on it, and interact with mouse clicks (or touches) on the screen! And thanks to the design of Pygame, this can be implemented on different platforms (e.g., Windows, iOS). This has been a great boon to teaching such languages, and Python in this case.

Haskell, especially, as a language that tries to separate the pure from the impure (aka IO), faces a big challenge in this regard. If one wants to create user-interaction, the need for explaining Monads and rather advanced issues comes to the forefront. This is not the kind of thing you want to happen when trying to demonstrate a "`hello, world`" program.

In this work we describe the development of Hygame (pronounced 'Hi' - 'Game'), which is motivated by the popularity of Pygame. We discuss the design tradeoffs, as dictated by the envisioned application of this package. We will further describe how this can be deployed on various platforms, and extended.

The paper is organized as follows: We start with describing the general system architecture. We also highlight tradeoffs and design decisions that were made. The following four sections describes in detail examples of rising complexity. Starting with 'just open a window' and writing "`hello, world`", we proceed to loading an image (asset) and displaying it, then to plotting user data, and

Pygame based

```
import pygame
pygame.init()

screen = pygame.display.set_mode ((640, 480))

fontName = pygame.font.get_default_font ()
font = pygame.font.Font (fontName, 24)
white = (255, 255, 255)
surf = font.render ("Hello, world!", True, white)

screen.blit (surf, (300, 200))
pygame.display.flip ()
pygame.time.delay (5000)
pygame.quit ()
```

Hygame based

```
module HelloWorld where
import HyGame

main = do set_mode (640, 480)
          let graphics = draw_text "Hello, world!" 24 (300, 200)
          render graphics
          delay (seconds 5)
          quit
```

Fig. 2. Pygame "`hello, world`" program on the top, and the equivalent Hygame program on the bottom.

finally demonstrate a full game, which includes game-loop and event handling capabilities. We close with remarks on our plan for deployment, as well as future directions and challenges for development.

In another paper (to be presented at "The 4th International Workshop on Trends in Functional Programming in Education", TFPIE 2015), we describe a specific course plan, with projects and work examples, for a first course in Haskell deploying the Hygame package.

2 System Architecture

Just before we embark on describing the new system, we will spend the next section describing briefly the workings of an existing, and very successful, system: Pygame. Although Pygame is deployed in another langauge, Python, we will follow many of its design decisions, as these proved very successful (by the sheer popularity of Pygame).

2.1 Pygame: Simplicity

Quoting directly from the Pygame.org website [2]:

> “Pygame is a set of Python modules designed for writing games. Pygame adds functionality on top of the excellent SDL library. This allows you to create fully featured games and multimedia programs in the Python language. Pygame is highly portable and runs on nearly every platform and operating system. Pygame itself has been downloaded millions of times, and has had millions of visits to its website. Pygame is free.”

Note that Pygame is aiming for graphics and audio, whereas in the sequel we will be concerned only with graphics.

In Fig. 3 we include a program that creates a bouncing ball, moving around in a box represented by the window on the screen. The program is taken from [5]. Let us now explain the relevant Pygameparts of the code in Fig. 3.

```
import sys, pygame
pygame.init()

size = width, height = 320, 240
speed = [2, 2]
black = 0, 0, 0

screen = pygame.display.set_mode(size)

ball = pygame.image.load("ball.bmp")
ballrect = ball.get_rect()

while 1:
    for event in pygame.event.get():
        if event.type == pygame.QUIT: sys.exit()

    ballrect = ballrect.move(speed)
    if ballrect.left < 0 or ballrect.right > width:
        speed[0] = -speed[0]
    if ballrect.top < 0 or ballrect.bottom > height:
        speed[1] = -speed[1]

    screen.fill(black)
    screen.blit(ball, ballrect)
    pygame.display.flip()
```

Fig. 3. Pygame bouncing-ball program, that includes putting an image on the screen, and interactivity (albeit just the user closing the window).

1. **Module loading** - Line (1), "`import sys, pygame`", loads the necessary modules.
2. **Module Initialization** - Line (2), "`pygame.init()`", takes care of initialziation.
3. **Creating the Screen object** - Line (8), with the code "`screen = pygame.display.set_mode(size)`", creates the screen, or window. Note that Pygame is using double buffer, so there is one (internal) screen we can draw on, and at the appropriate time (see below), we copy this into the user-viewed screen. However, for users, the interaction is through drawing on the screen, and then 'sending' it to the user.
4. **Loading assets** - Line (10), the code "`ball = pygame.image.load("ball.bmp")`", loads a bmp image.
5. **Internal Data Structure** - Line (11), "`ballrect = ball.get_rect()`", establishes the Rect class which is used to handle graphic objects in Pygame, both in terms of positioning on screen, as well as collision detection and so forth. The Rect class is a simple abstraction of the objects' location and size: (top,left,width,height). Manipulating the Rect class associated with an object will cause it to move on screen.
6. **Game Loop** - Line (13), which is a simple 'always true' loop. The loop will go on forever, until something within the loop breaks out.
7. **Event handling** - Line (16), "`pygame.event.get()`", is the part responsible for getting events into the program. Pygame handles all events, and queues them until asked to retrieve those. When "`pygame.event.get()`" is called, the queue of events is passed to the program. In our program, the only event we are interested is the user closing the window. This will be an event of type "`pygame.QUIT`". If this is indeed the event, we break the while loop and exit the program. Can't beat that for simplicity and directness...
8. **Manipulating screen objects** - Lines (17-21). This falls into the logic of the program (which in our case, moves the ball around the screen). Note that this is done by handling the Rect class associated with the object. No drawing on screen is happening in this part.
9. **Drawing on user-viewable window** - Lines (23-25). These lines contain 3 different commands: First, the screen (actually an internal buffer) is painted black using "`screen.fill(black)`", in order to clear the previous image. Then, the ball image is drawn onto the screen using "`screen.blit(ball, ballrect)`", which includes the image and its expected location. The third and last step is that of copying the internal buffer into the actual screen, using "`pygame.display.flip()`".

The methods described in this example of queueing the events, running the program in an infinite loop, and quitting by breaking out, might not be ideal for all cases. However, they afford much simplicity for users, and that is part of the reason for the popularity of Pygame, which we try to replicate here.

2.2 System description

Following in the steps of Pygame, we introduce the following architecture described schematically in Fig. 4. The structure is based on four distinct parts,

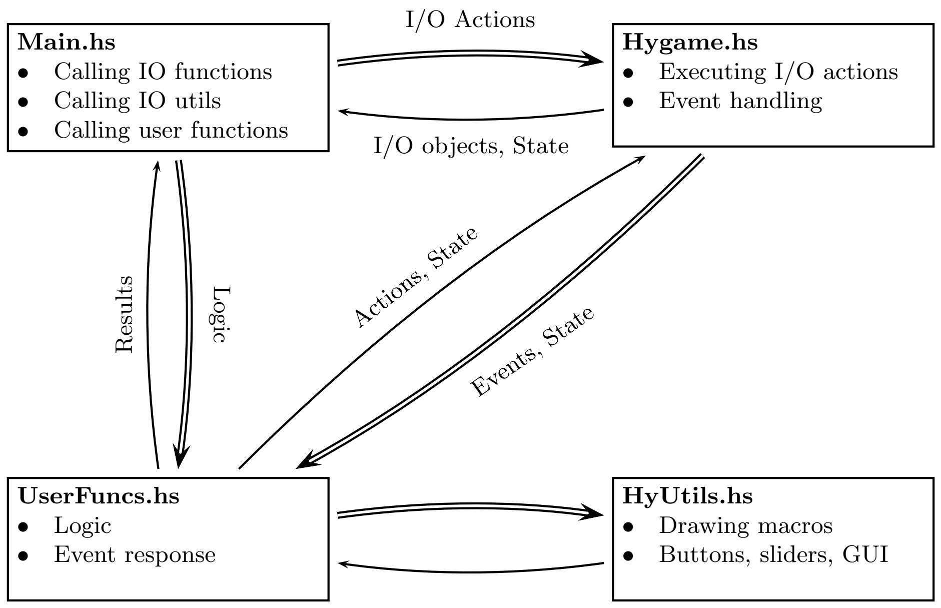

Fig. 4. Structure of Software.

though of course some items can be placed in a few of these basic blocks.

1. **Main.hs** - The main function. This function calls on Hygame to handle graphics IO, and might call to HyUtils in order to leverage on pre-defined functions to achieve easier graphics interaction. The important role of Main.hs is to serve as a buffer between the pure-logic part of the user, and the IO intensive part performed by Hygame. Making this separation allows the beginners to concentrate on the pure aspects of the language, while achieving graphic and IO interface at the same time.
2. **Hygame.hs** - The enabler for abstracting IO graphic operations from the user. This package is described in more detail through the following examples.
3. **UserFuncs.hs** - These functions should be dealing with pure logic. The caveat to this is the case when the user creates an interactive game, and needs to deal with IO actions from the event-responding function. As we will see in the examples, the user does not need to execute IO commands directly. Rather, the user defines directives on what should be done, and the IO action itself is performed in Hygame.hs

4. **HyUtils.hs** - As the name implies, this module allows us to define utilities for common tasks. Again, this module should not perform directly any IO actions. Rather, it specifies actions, which are later on chained together, and executed by Hygame.hs

If we compare this structure to the popular MVC (Model, View, Controller) model, we can see that the correspondence is not one-to-one with the described blocks. The MVC is in a different level than what we describe here. For example, the user-functions that respond to events, need to take care both of logic and directing the view. Thus, they encompass both Model and Controller role. It is left to the user to organize the code properly so things are separated. However, we wanted to clearly point out that it would be incorrect to identify Main.hs as the controller, UserFuncs as Model, and HyGame as View. Maybe the right way to partition it is calling Main.hs as the initialization block; HyGame.hs as the Viewer, and UserFuncs.hs as both the Controller and Model. Separating the last two would lie on the shoulders of the programmer.

In the following sections we will walk through examples of using Hygame, and explain the various module-roles in more details.

3 Program examples: “`hello, world`”

In Fig. 5 we give the simplest “`hello, world`” program using Hygame. This program opens a window, draws on it the text “Hello, world!”, and closes it after 5 seconds. This enables us not to require any user interaction, not even for closing the graphic window.

```
1. module HelloWorld where
2. import HyGame
3.
4. main = do set_mode (640, 480)
5.           let graphics = draw_text "Hello, world!" 24 (300, 200)
6.           render graphics
7.           delay (seconds 5)
8.           quit
```

Fig. 5. Hygame “`hello, world`” program.

1. **Module loading** - Line (2), “`import HyGame`”, loads the necessary module. This is equivalent to the “import pygame” directive in Pygame.
2. **Creating the Screen object** - Line (4), “`set_mode (640, 480)`”. The “set_mode” command creates both the physical window that will be visible to the user, as well as the drawing screen which is used for double-buffering. This is the equivalent to the command “display.set_mode(size)” in Pygame.

However, in our case there is no return value to the user, as we rely on the fact we have only one screen, and this is the one that will be used throughout.

3. **Manipulating screen object** - Line (5), the code is "`graphics = draw_text "Hello, world!" 24 (300, 200)`". The command "draw_text" draws the text on the screen. This is the equivalent of the "blit" command in Pygame, for text. however, as you will note shortly, this is just a directive to perform the action. Nothing is being executed yet. Two things to note here: Since we are dealing with only ONE screen (and one window), there is no need to specify which one it is. In addition, maybe more importantly, the value of "`graphics`" is just an IO action to be performed. No IO operation was actually performed yet. The IO operation will be performed only through the "render" function.
4. **Drawing on user-viewable window** - Line (6), "`render graphics`". This is when the IO actions are executed, and things are drawn onto the user viewable window. This performs all the IO actions specified in graphics, and the "`flip`" command in Pygame.

To conclude, the Hygame based program (Fig. 5) is very similar to its zpygame counterpart (Fig. 2) . We took extra effort to simplify the returned values and handling of details, for example not returning the 'screen' variable to the user, nor dealing with fonts. In addition, we separated the IO actions from the actual execution of these. The only part which acts on the IO actions is the "`render`" function. This affords more simplicity in terms of explaining to newcomers to Haskell, at the price of less flexibility. Since dealing with IO types might be confusing for beginners at Haskell, we opted for this tradeoff.

4 Program examples: Loading and Displaying an image

In Fig. 6 we go beyond the previous example in two aspects. The first is loading an external graphic image for drawing on the screen. This is an example of loading external assets into the program and using them, a very common task in game programming. The second aspect is that of chaining (or sequencing) IO operations. As we mentioned, the actual execution of all graphics IO actions is done through the Hygame module. In the previous program we encountered only one IO action that was sent to the function "`render`". We expand on this below. This time, we will point only to the differences and new aspects as compared to previous programs.

1. **Asset loading** - Lines (5-6) describe loading a new (image) asset. The function "`load`", residing in Hygame, loads the file and creates a Surface object. The Surface object is explicitly names in the next line. It holds both the Rect (similar to Pygame), and the bitmap itself.
2. **Chaining IO actions** - Lines (8-10) describe chaining of two IO actions: The first is drawing an image on the screen (using "`blit`" action), followed by "`draw_text`" action. These are bound to the variable "`graphics`", which is then sent for execution in function "`render`". This formulation abstracts

```
module PictureViewer where
import HyGame

main = do let file_name = "earth.jpg"
          image <- load file_name
          let (Surface bitmap (Rect x y w h)) = image
          set_mode (w, h)
          let graphics =
               sequence_ [blit image,
                          draw_text file_name 18 white (0, h - 18)]
          render graphics
          delay (seconds 5)
          quit
```

Fig. 6. Sample program: loading and displaying an image.

the need to make multiple IO operations directly from the main program. It replaces that with creating a set of actions to be executed, and then calling one function to execute these actions.

5 Program examples: Plotting user data

In Fig. 7 we describe only small part of a program to plot a $2D$ scatter plot of a set of points, with axes. The goal is to emphasize the aspect of sequencing operations for IO. Thus, the user program does not need to execute the IO actions, but rather build the appropriate sequence, in a logical way, and then we use the already encountered "`render`" command to execute these actions.

```
plot_points = sequence_ [draw_text "*" 12 white point
                        | point <- pixel_points]
x_axis      = sequence_ [draw_text "-" 12 green (i, (height - 10)/2)
                        | i <- [0, 10.. width] ]
y_axis      = sequence_ [draw_text "|" 12 green (0, j)
                        | j <- [0, 20.. height]]
graphics    = sequence_ [x_axis, y_axis, plot_points]
```

Fig. 7. Sample program: Plotting a set of points in a $2D$ graph.

6 Program examples: Game-loop and Interaction

In Fig. 8 we again show only a portion of the code for the game, in order to highlight new and interesting aspects.

```
data State = State { healthPoints, stepsLeft :: Int,
                     goalReached :: Bool,
                     player :: Surface,
                     fixedAssets :: FixedAssets
                   }

data FixedAssets = FixedAssets {background, enemies,
                                healthFood, treasure :: Surface}

main :: Action
main = do background <- load "background.png"
          enemies    <- load "enemies.png"
          healthFood <- load "healthFood.png"
          treasure   <- load "treasure.png"
          player0    <- load "player.png"

          let (Surface _ (Rect _ _ width height) _) = background
          set_mode (width, height)

          let fixedAssets = FixedAssets background enemies
                                        healthFood treasure
              state0 = State { healthPoints = 5,
                               stepsLeft = 30,
                               goalReached = False,
                               player = player0,
                               fixedAssets = fixedAssets
                             }

          transitionLoop state0 transition

-- Transition function
transition :: State -> Event -> (State, Action)
transition state Quit = (state, quit)
transition state (VideoResize _) = (state, redraw state)
transition state (KeyDown _ _ _ keyName) | not (gameOver state) =
                   --omitted details here ...
transition state _ = (state, nop)
```

Fig. 8. Sample program : Full game, including assets loading, state setting, event handling, and game loop.

The heart of this program is the game loop, which is performed by the call to "`transitionLoop`" on Line (29). The "`transitionLoop`" function takes as inputs the current State, and a function. In our case the function is called "`transition`". This function serves as the 'callback' function, and takes a State and an Event as inputs, and creates a new State and an Action. The State is initialized on lines (20-27), and includes the obvious parameters of the game. Lines (32-37) describe the "`transition`" function: depending on the input State and the Event, the next State is determined, and the proper Action is assigned. Note that the Action is not executed in this function, but rather in in the "`transitionLoop`" function.

The transition loop function "`transitionLoop`" is described in detail in Fig. 9.

```
-- two clarifying statements
type StateTransition state = state -> Event -> (state, Action)
type Action = IO ()
-- the function itself
transitionLoop :: state -> StateTransition state -> Action
transitionLoop state0 transitionFunction =
 do evs <- events
    let (state, actions) = mapAccumL transitionFunction state0 evs
    sequence_ actions
    transitionLoop state transitionFunction
```

Fig. 9. Sample program : Game loop in details.

This function resides in the Hygame module, and is hidden from the user in the sense that a programmer does not need to be familiar with its implementation in order to write simple games. We bring it here for the benefit of explaining how Hygame abstracts the subject of events and actions for the user. The user needs only to implement the call to the "`transitionLoop`" function, establish the State, and write the event handling function.

7 Summary and Future directions

We described Hygame, a framework for enabling the use of Haskell in introductory courses to programming. The use of Hygame in various graphics and game-related programs was demonstrated, with details about the various trade-offs taken.

The biggest challenge for moving this framework forward is, of course, getting wide adoption by users. To this aim, we are setting up a web page `www.hygame.org` [11] that will facilitate sharing tutorials, sample programs, lesson plans, and other tools. In addition, we will work on developing this framework for various platforms, and possibly even integrating with other packages (like haste [12]) to make it adapted to HTML5 and display on webpages. One more venue for future work is the addition of Audio capabilities, and leveraging on a stronger back-end engine for the multimedia operations.

References

1. Kernighan, B., Ritchie, D: The C Programming Language. Prentice Hall, New Jersey (1988)
2. PygameWebsite, `http://www.pygame.org/`
3. Best 10-tutorials for Pygame, `http://inventwithpython.com/blog/2010/09/01/the-top-10-pygame-tutorials/`
4. McGugan W.: Beginning Game Development with Python and Pygame: From Novice to Professional. Prentice Hall, New Jersey (1988)
5. Pygame introduction game, `http://www.pygame.org/docs/tut/intro/intro.html`
6. Shein E.. Python for Beginners. In : Communications of the ACM, Vol. 58 No. 3, Pages 19-21. ACM, New York, NY, USA (2015)
7. Gamification on Wikipedia, `http://en.wikipedia.org/wiki/Gamification`
8. Reiners T., Wood L.: Gamification in Education and Business. Springer e-book (2014)
9. Kumar B., Khurana P.: Gamification in education - learn computer programming with fun. International Journal of Computers and Distributed Systems, Vol 2.(1), 46–53 (2012)
10. Deterding S., Dixon D., Khaled R., and Nacke L. . From game design elements to gamefulness: defining “gamification”. In : 15th International Academic MindTrek Conference: Envisioning Future Media Environments (MindTrek ’11). ACM, New York, NY, USA (2011)
11. Hygame Website, `http://www.hygame.org/`
12. Haste Website, `http://haste-lang.org/`