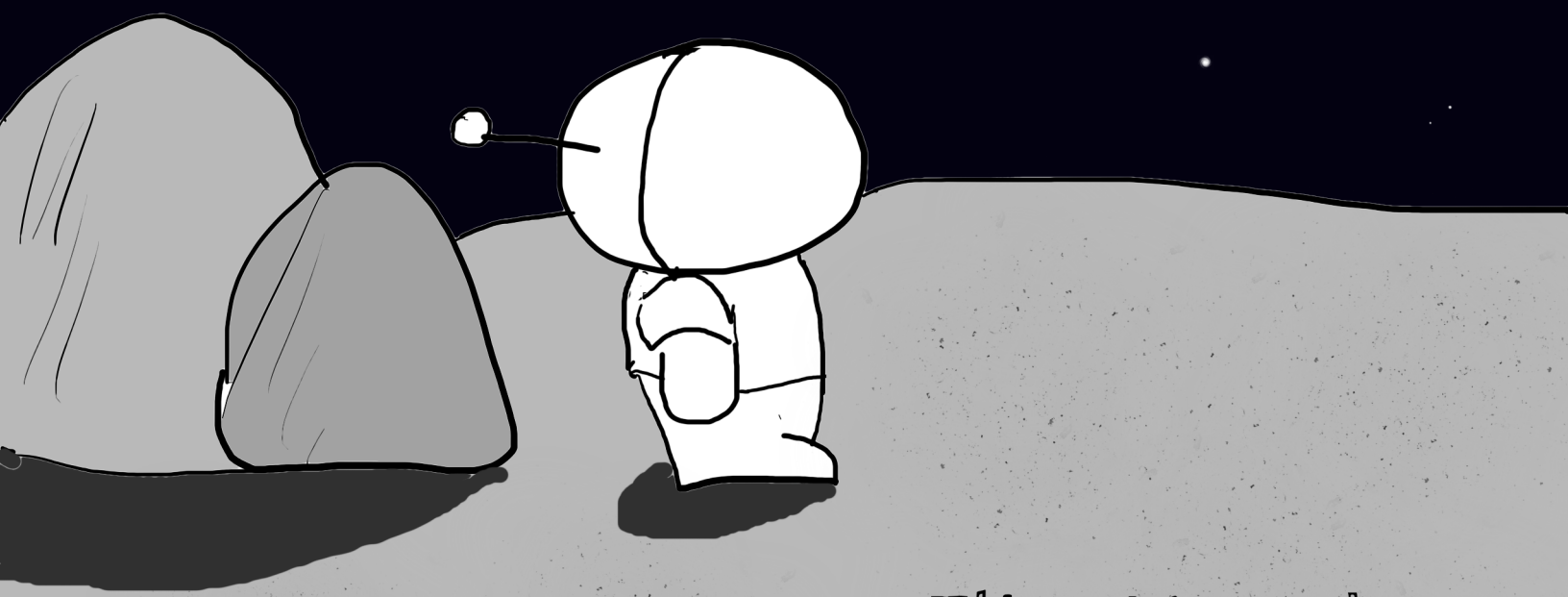
Game programming in Haskell



Elise Huard

Game programming in Haskell

Elise Huard

This book is for sale at http://leanpub.com/gameinhaskell

This version was published on 2015-06-26

Contents

Introduction

Doing things the hard way

A couple of months ago, I decided to write a game in Haskell.

In doing this, I approached the project from the opposite angle from what is common sense in a work situation: technology push instead of technology pull. I picked the language (Haskell) I wanted to build the project in, instead of defining the project in broad strokes and then picking the technology that would be most suited to perform the job. If I'd picked the opposite approach, I'd probably have ended up with a game in C++, flash, or js, which is what most games seem to be written in. Starting from a technology and doing a project is not an approach I would necessarily recommend in general, but it worked out well here.

I repeat: this is not the easiest way to develop a game. But it's a fun, rewarding way to improve your Haskell knowledge, and it's one way to get acquainted with what constitutes the necessary set of elements to construct a game.

Haskell

The choice of Haskell is something that made sense to me at the beginning of this project:

- the static type system: games can be complex beasts. Defining sensible types and function signatures will let the type checker weed out at least a portion of the errors. Types can also be surprisingly expressive to describe the possible states of your game.
- pattern matching: once you have the types, pattern matching is a great way to describe state machines - how functions make one particular state transition into the next.
- Haskell contains interesting abstractions, like functors, applicatives, monads, arrows and lenses. These come in useful when having to tackle and decompose complex situations.
- Haskell does garbage collection (a generational GC[1] to be precise), which means the programmer does not need to manage the allocation and freeing of memory.
- Haskell is optimized for you by ghc and gcc (or llvm if that is the back-end), and compiles down to native machine code (i.e. an executable). The end result doesn't require the JVM, which (may) improve the memory and CPU consumption, and the real-time execution of the game.

[1] http://www.haskell.org/haskellwiki/GHC/Memory_Management

A caveat to the nice picture painted here is that for this project, a lot of bindings to C libraries were used. This means that the libraries under the nice glossy varnish of Haskell still belong to a different world, one that may contain memory leaks and code that isn't necessarily thread-safe. Fortunately, the libraries used (OpenAL, FTGL, ...) have been stared at long and hard by a good subsection of the gaming world, and are doing their job well (don't blame the tools ...).

I didn't have to start completely from scratch on Haskell games, there have been projects before mine. I'll list the projects (with thanks) that helped me make sense of it all at the end of this introduction. Still, none of the projects were extremely recent, and as far as I could tell, none were running on Mac OS X, so I had to spend a fair bit of time fiddling to get things working. All the instructions to get the various libraries to run on Mac are detailed in Appendix A. I'll attempt (hopefully with some help of friends) to do the same for Linux and Windows.

For those amongst you who are Haskell beginners, I'll use asides to explain about abstractions used as I go. I'll attempt to dig deeper into the pros and cons of using Haskell for a game at the end of the book.

Game

I also came to realize that writing "a game" is as vague as it could possibly get: the ecosystem is vast, the search space infinite. Classical games (first person shooters, platform games, adventure games, ...) are but a subset. Games are a medium. Games as an art form, as a social statement, as a rage outlet - the possibilities are endless. A game, in the end, is a universe with clearly defined limits and rules, and (usually) a goal, but those limits, rules and goals can be anything at all.

I had a lot of fun discovering this, and I'll try to convey the creative process of designing a game as I understand it in the last chapter.

Aim of this book

The aim of this book is to help you, the reader, get started with writing a game in Haskell. I'm writing the manual I would have liked to have had when I started this project - there were a lot of a-ha moments and long hours lost in yak shaving. My hope is that this book will allow you to concentrate on the actual game instead of having to waste time figuring out the finicky technical details!

This book will allow you to create a 2D game with bells and whistles. Extending this to 3D means altering the graphical side to use a different framework - Gloss, the graphical framework used in this book, is focused on 2D. 3D is more complex by an order of magnitude: extra dimension, using 3D geometrical shapes, perspective, and lighting. Starting with 2D gives you the opportunity to get acquainted with the basics of writing a game before making the next step.

Prerequisite

At least a passing knowledge of Haskell is assumed. If you haven't done any Haskell yet, Learn You a Haskell for Great Good[2] is a very good introduction.

Contents

Each chapter covers an aspect of creating a game. It gives enough background to get you started in your own game, while describing the gotchas and challenges you might face.

The first chapter talks about the graphics, more specifically using Gloss and GLFW. Gloss is a nice functional layer over OpenGL. It offers the necessary features to build a 2D game, without exposing you to all the boilerplate of OpenGL - OpenGL, while powerful, is so general-purpose that doing anything takes setting a lot of options. Another reason not to use OpenGL: OpenGL is a giant state machine, which breaks the flow of a Haskell program somewhat. Gloss abstract all of this away, and offers a genuinely joyful API. GLFW provides an interface to your computers' input devices, the mouse, keyboard (and even joystick if you wish).

In the second chapter functional reactive programming is discussed, as a way to handle state which is very elegant, if a little bit of a mind twist at the beginning.

The third chapter goes into further graphics, textures, some animation. These topics (like pretty much every topic in game development) deserve whole books of their own, but I attempt to outline a "getting started", including where you can go from here.

The fourth chapter covers the use of sound. Sound is what makes games come alive, and I show one way to implement sound in the game.

The fifth chapter handles some topics that have fallen by the wayside in previous chapters: start screen, window size, score, levels.

In the sixth chapter, we talk about testing. For a game we require more than just automated testing - some aspects of the game just can't be tested automatically but need to be eyeballed with full user tests. That's why we need to create tools to make the tester's job sa painless as possible.

The seventh chapter describes the use of Chipmunk, a 2D physics engine, or rather Hipmunk, its Haskell bindings. For simple cases, you can program the physics yourself, but it's often handy to use a third-party library when things get a little more complex, which lets you focus on the game dynamics.

The last chapter discusses game design, which is arguably the hardest part of all. Game design is usually an iterative process, because only a prototype can show you what works and what doesn't.

We conclude the ebook by discussing the pros and cons of using Haskell for games.

[2] http://learnyouahaskell.com/

magic

Thanks

I want to thank my husband, Joseph Wilk, for all the advice and the enthusiasm for the project. His insights and resources were invaluable. He also paid the bills for some of the time I was experimenting away (he'd rather have been working with me on this, I'm sure :)). I'm lucky to have him.

Thanks to the authors and committers of the following projects: OpenGL, Gloss, GLFW-b, FTGL, ALUT and Elerea - especially the last one (Patai Gergely) for answering questions about how to use his library.

Chapter 1: Introducing graphics with Gloss and GLFW

The code for this chapter is on Github: https://github.com/elisehuard/game-in-haskell/blob/master/src/Shapes.hs

Why Gloss and GLFW

Gloss is a graphical framework for painless 2D vector graphics, animations and simulations. It has a friendly, functional interface for graphical display.

OpenGL is a graphics hardware API for rendering 2D and 3D vector graphics. Most of the common graphical libraries like Gloss, SDL2 and Qt use OpenGL under the hood. OpenGL has the following advantages:

- open source
- very portable - it has extensions for nearly all graphics cards, and works on mobile platforms
- powerful: converting your graphics to use the GPU of your graphics card directly to generated 3D graphics on the fly - one could say it is completely overkill for a 2D game, like hammering in a nail with a bulldozer.

The downsides of OpenGL is that it's *too* powerful - it has too many options, which leads to lots of boilerplate which makes writing what should be simple - a 2D game - a little cumbersome. That's where Gloss offers a friendly alternative, while being built on top of OpenGL, which lets us concentrate on building the game.

It is possible to program an entire game with only Gloss, no other explicit libraries. However, I chose to restrict my use of it to just the super-friendly rendering. This book aims to offer an overview of all the moving parts of a game, while Gloss hides a lot of those under the cover. I want you to be able

to decide to replace any of the given libraries by another similar library, knowing the principles, and still being able to program a game.

The part of Gloss we'll be using concerns itself with only the graphical part of things, which is why we're using GLFW as a back-end for the more platform dependent aspects, like interacting with the window system and the input devices like keyboard and mouse.

This chapter is not meant as extensive documentation, but rather as a quick 'getting started' of the most essential features.

Dependencies to add in the cabal file: gloss, gloss-rendering GLFW-b.

The Canvas

GLFW[3] is a portable library which will handle the interaction with the operating system - installing the library on your OS is a prerequisite to using the Haskell bindings GLFW-b (see Appendix A for instructions on Mac OS X).

First and foremost, we need GLFW to create a window for us.

A convenience function you'll want to use for your project goes like this:

```
import "GLFW-b" Graphics.UI.GLFW as GLFW
....
withWindow :: Int -> Int -> String -> (GLFW.Window -> IO ()) -> IO ()
withWindow width height title f = do
    GLFW.setErrorCallback $ Just simpleErrorCallback
    r <- GLFW.init
    when r $ do
        m <- GLFW.createWindow width height title Nothing Nothing
        case m of
          (Just win) -> do
              GLFW.makeContextCurrent m
              f win
              GLFW.setErrorCallback $ Just simpleErrorCallback
              GLFW.destroyWindow win
          Nothing -> return ()
        GLFW.terminate
  where
    simpleErrorCallback e s =
        putStrLn $ unwords [show e, show s]
```

[3] http://www.glfw.org/

This function is copied from the GLFW-b-demo[4] project on Github, which shows an example on how to use the features of GLFW. *withWindow* starts GLFW's context and takes care of the boilerplate error handling and cleanup.

Using this function, creating a window in your 'main' function is trivial:

```
main :: IO ()
main = do
    let width  = 640
        height = 480
    withWindow width height "Resurrection" $ \win -> do
        -- graphical and application code goes here
```

Note that the last argument of *withWindow* is a function passing in the created window as an argument. This is useful for any further GLFW-related operations like capturing inputs or swapping the window buffer.

Gloss' under the hood initialization

Gloss does some initialization that is useful to know about, even if we don't have to do it explicitly. It sets the right projection for 2D and sensible defaults for the coordinate system, as well as the background color and depth function.

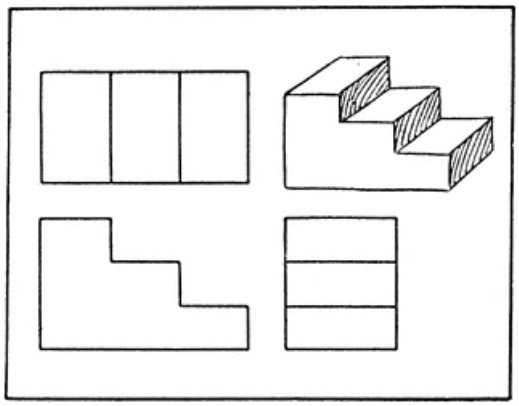
The projection

3D graphics requires projection, since our screen real estate is obviously 2D. The cubes, spheres and various other shapes need to be translated into something that fits into a plane, in a way that still conveys 3D information as much as possible. Various perspectives achieve this.

For 2D we don't really need to worry too much, we just want to render everything as is in the x-y plane: this is why we use an orthographic projection. An orthographic projection means the projection lines are all parallel and orthogonal to the projection plane (see picture below). To keep ourself working in a simple coordinate system this plane is the x-y plane. The projection is carried out only within the limits of our world, which means we can picture the edes of our projections having 'clipping planes'.

A common example of orthographic projection are home floor plans, or technical drawings (with 3 orthogonal orthographic projections to give a full specification of the object).

[4] https://github.com/bsl/GLFW-b-demo



Orthographic projection of stairs - Thomas E. French and Carl L. Svensen Mechanical Drawing For High Schools (New York: McGraw-Hill Book Company, Inc. , 1919) 29

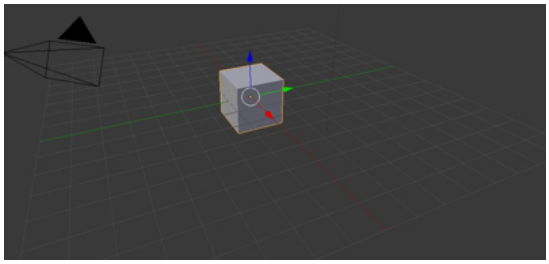
Under the hood OpenGL instructions in Gloss:

```
let (sx, sy)    = (fromIntegral sizeX / 2, fromIntegral sizeY / 2)
GL.ortho (-sx) sx (-sy) sy 0 (-100)
```

ortho means orthographic projection, the other numbers indicate the clipping planes. OpenGL allows you to define the coordinates of your world pretty much at will. The parameters of ortho are the limits of your world in the coordinate system: in this case the decision in Gloss is to have the world extend from -width/2 to width/2 on the x axis, and from -height/2 to height/2 on the y axis.

Aside: OpenGL coordinates: the whole story

OpenGL provides three matrices to define your view of 3D space. Let's start at the beginning:

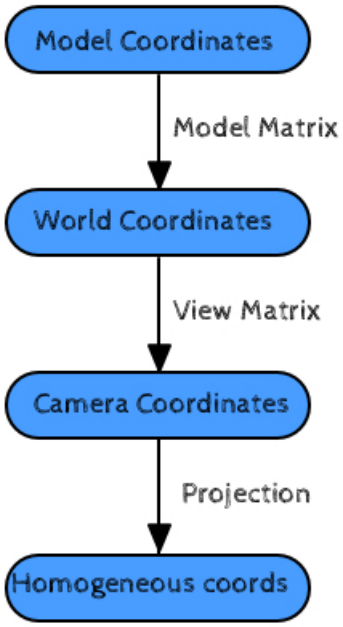


OpenGL coordinates (blender screenshot)

- *model matrix*: when you draw a shape, you implicitly center it at the origin (0, 0, 0). If you left it at that, everything would be piled up around the origin, which is not what we want. We scale, translate, rotate our object to its appropriate position, to world coordinates : the model matrix represents these operations.

- *view matrix*: another transformation is in order, since we want to view the world from a certain angle. We operate another transform, to be in camera coordinates (all vertices defined relatively to the camera).
- *projection matrix*: once we are in camera space, we need to determine where every pixel will be represented on the 2D screen, depending on its z distance from camera. For this we need a projection. This is where the orthographic projection comes in for 2D. For 3D we'd use a perspective, for a more realistic feel. This is where we use the orthographic projections mentioned above.

Our view matrix is the unity matrix, our camera is located vertically above the origin in the x-y plane.



OpenGL coordinates

In Gloss we only busy ourselves with the modelview matrix (moving our object to the right location), since our orthographic projection takes care of the projection side of things.

Color, Depth

Other parameters: the default background color and the depth buffer.

The default background color (the color the canvas is 'cleared' to) is given as a parameter to Gloss' *displayPicture* function described in one of the next paragraph. This is the color that will be displayed in absence of any other shapes.

Depth keeps track for OpenGL of how deep a pixel is supposed to be along the z-axis, and how it is to be displayed as a consequence. In 2D, this is not a concern, but there's still a matter of deciding

how shapes that occupy the same space are displayed consistently. The intuitive choice is drawing from bottom to top: background first, then player, then trees, for instance. Gloss make sure this is happens automatically (in OpenGL terms, the depht Function is GL_ALWAYS).

The Loop

Any real-time system has a never-ending loop, which can be expressed in pseudo code as follows:

```
loop:
  check input (like keyboard, mouse, ...)
  change state and process as a result of input
  perform any necessary output (render frame)
  if no exit signal was given, loop again
```

A first implementation of our loop could look like the following Haskell code:

```
  loop window =  do
      pollEvents
      renderFrame window
      threadDelay 20000
      k <- keyIsPressed window Key'Escape
      if k
          then return ()
          else loop window
```

The first line of this loop tells GLFW to poll for inputs, as expected from the earlier pseudocode. The second line calls a function to render a frame.

But what about the next line? Why do we need to let this thread sleep for 20000 microseconds? Well, the problem is that rendering a frame actually means firing up a chain of events: going from OpenGL to the GPU to the actual rendering of a screen buffer. This only takes a very short amount of time, but some time nevertheless. If the processor (or core) spins this loop as fast as it can, this means the commands to display frames will not have enough time to be executed, pile up and make the process crash.

And fortunately, we don't need that many frames anyway! A human eye will quite happily interpret a succession of images as smooth video if they are displayed at a rate of 24 images a second - which is what is known as the frame rate. Slowing the loop down with 20000 microseconds will get you about 60 frames per second - though that may have to be re-evaluated as the processing in the loop gets more computationally intensive - slower. In any case, it's more than enough for the rendering to take place in the background.

You don't want to go too high anyway - if your monitor runs at 60 Hz (refreshes the picture 60 times per second), a framerate of 100 frames per second means that 40 frames are never displayed, which is pointless. In some cases you can link your frame rate to your screen refresh rate using VSync on some systems, but we're not going to go into this here.

In fact, the whole issue is slightly more complex than this - how will we ensure that we always have a constant framerate, no matter how heavy the processing in our game is (for instance in the case of an ongoing physics simulation)? This blog post[5] is a good reference on how to do that.

The next few lines provide a possible exit from the loop, by capturing the escape keypress and exiting the loop if it has been pressed. The keyIsPressed function is not part of GLFW-b, but is a short little convenience function:

```
keyIsPressed :: Window -> Key -> IO Bool
keyIsPressed win key = isPress `fmap` GLFW.getKey win key

isPress :: KeyState -> Bool
isPress KeyState'Pressed   = True
isPress KeyState'Repeating = True
isPress _                  = False
```

It is possible to fine-tune this function if a distinction between the key states is needed. The signature of getKey is *Window -> Key -> IO KeyState*. The *fmap* is using the fact that the IO monad is a functor (all monads are functors), and isPress is applied directly on the KeyState value.

Let's have a look at a minimal renderFrame function:

```
renderFrame window = do
    -- all drawing goes here
    swapBuffers window
```

The swapbuffers function tells GLFW to swap the existing buffer of the window to the newly rendered one.

Let's draw something

Now we have our blank sheet, time to start thinking about how to make things more interesting.

The *displayPicture* function in Gloss has the following signature:

[5] http://gafferongames.com/game-physics/fix-your-timestep/

```
displayPicture
        :: (Int, Int)   -- ^ Window width and height.
        -> Color        -- ^ Color to clear the window with.
        -> RS.State     -- ^ Current rendering state.
        -> Float        -- ^ View port scale, which controls the level of detail.
                        --   Use 1.0 to start with.
        -> Picture      -- ^ Picture to draw.
        -> IO ()
```

The render function takes a window size, the background color, and a Picture data structure. The State argument is purely for the internal implementation of Gloss (specifically for performance), and we'll talk about its function in the third chapter. The View port scale we will talk about in the third chapter - we can set it to 1.0 for now.

The Picture data type is Gloss's DSL to shield us from OpenGL: we specify what we want drawn as a data structure, and the *displayPicture* function takes it and converts it to a sequence of OpenGL instructions.

Picture can be an individual shape, using a Circle constructor for example, or an array of Picture (using the Pictures constructor).

Gloss gives us nice functions to create Picture data with individual shapes:

- *polygon*: the most general shape, takes an array of points (points in Gloss are of type (Float, Float), for the (x,y) coordinates). By default, the polygon is a filled shape.
- *line*, *lineLoop*: draws a line, again taking an array of points. Every pair of points will form a line segment, with all segments interconnected. *lineLoop* will add a line from the last point with the first point.
- *circle*, *thickCircle*, *circleSolid*: *circle* draws a simple circle, and takes a Float for the radius. *thickCircle* will take two floats, the first the radius, the second the thickness. *circleSolid* is filled with the given color.
- *arc*, *thickArc*, *arcSolid*: draw an arc, with as parameters two angles, in degrees (0 to 360) counted counterclockwise from a vertical line pointing up, and a radius. *thickArc* takes an extra parameter for the thickness of the line.
- *text*: Text in Times New Roman, takes a string as parameter.
- *rectangleWire*, *rectangleSolid*: takes 2 Floats, for sizeX and sizeY, centered around the origin.
- *pictures*: takes an array of pictures. The first element gets drawn first, and the other elements are layered on top of it.

We can specify a color for every shape we draw. Gloss provides a default, which is black. So if we just want a solid red circle with radius 10:

```
glossState <- initState
...
displayPicture white glossState 1.0 (color red $ circleSolid 10)
```

Gloss provides a number of predefined colors: *greyN, black, white, red, green, blue, yellow, cyan, magenta, rose, violet, azure, aquamarine, chartreuse, orange*. It also defines modifiers: *dim*, *bright*, *light*, *dark* (so you can use *dark red*), and even *addColor* and *mixColor*.

Somewhat more familiarly for anyone who's programmed for the web or used drawing software, there's also the possibility to define colors yourself, by using makeColor:

```
makeColor
        :: Float        -- ^ Red component.
        -> Float        -- ^ Green component.
        -> Float        -- ^ Blue component.
        -> Float        -- ^ Alpha component.
        -> Color
```

The Red - Green - Blue floats in question should be in the range of 0..255, with the usual translation: RGB 0 0 0 is black, and RGB 255 255 255 is white. The alpha component goes for 0 to 1, sliding from fully transparent (0) to opaque (1). With *makeColor* you can generate any possible color you might need.

We can now draw shapes ... but we also need to be able to place them in the right positions!

Transforms

As explained in the paragraph about OpenGL coordinates, Gloss just requires of us to specify the transforms requires by the first step, from model space to world space. This is done by specifying a few transforms:

- *translate*: drag to the correct coordinates. *translate* takes two Float, for the x and y movements - the center of your defined shape is moved to the given position.
- *rotate*: rotate the picture by the given angle, in degrees (0 to 360) - parameter type Float
- *scale*: most useful in the case of text and bitmaps (which we'll start using in next chapter).

So if we wanted to define a rectangle, move its center to (100, 100), and rotate it 30 degrees:

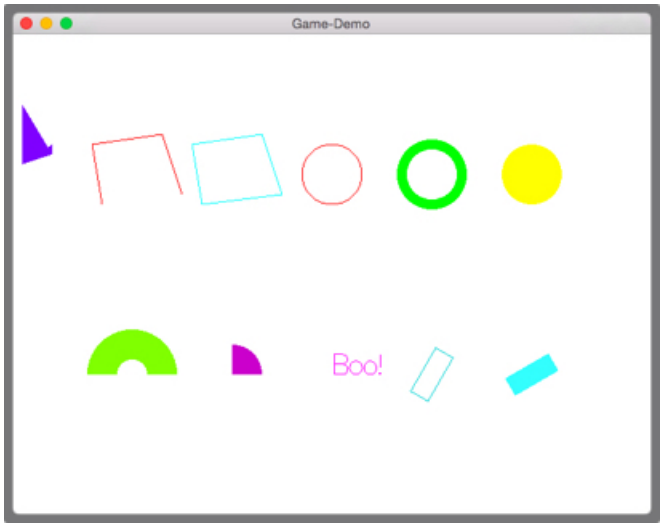
```
Color (bright magenta) $ translate 100 100 $ rotate 30 $ rectangleSolid 20 50
```

Small gotcha

The order of the transformation within the picture matters. It's probably easiest to first translate, then scale or rotate. The reason for this is that every transformation will affect the coordinate system locally: when you rotate, you actually rotate the whole coordinate system! This means that if you rotate 30 degrees, and then translate, you will be translating long an x and y axis that are rotated 30 degrees as well, which can lead to confusing situations. Same goes for scaling, you suddenly find yourself in a coordinate system which has scaled with your figure.

All together now:

```
import Graphics.Gloss.Rendering
import Graphics.Gloss.Data.Color
import Graphics.Gloss.Data.Picture
(...)
  renderFrame window glossState = do
     displayPicture (width, height) white glossState 1.0 $
       Pictures
         [ Color violet $ translate (-300) 100 $
              polygon [((-10), 10), ((-10), 70), (20, 20), (20, 30)]
         , Color red $ translate (-200) 100 $
              line [(-30, -30), (-40, 30), (30, 40), (50, -20)]
         , Color (makeColor 0 128 255 1) $ translate (-100) 100 $
              lineLoop [(-30, -30), (-40, 30), (30, 40), (50, -20)]
         , Color red $ translate 0 100 $
              circle 30
         , Color green $ translate 100 100 $
              thickCircle 30 10
         , Color yellow $ translate 200 100 $
              circleSolid 30
         , Color chartreuse $ translate (-200) (-100) $
              thickArc 0 180 30 30
         , Color (dark magenta) $ translate (-100) (-100) $
              arcSolid 0 90 30
         , Color (bright magenta) $ translate 0 (-100) $ scale 0.2 0.2 $
              text "Boo!"
         , Color (dim cyan) $ translate 100 (-100) $ rotate 30 $
              rectangleWire 20 50
         , Color (light cyan) $ translate 200 (-100) $ rotate 60 $
              rectangleSolid 20 50 ]
```



shapes-demo

What next?

We now have the ability to open a window and draw static shapes onto that window … which is not yet anywhere close to an actual, playable game. In the next chapter we introduce the use of state, and in particular Functional Reactive Programming to manage the state, which is where things start moving, literally.