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 2015-02-22

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.

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: testing, start screen, window size, score, levels, settings.

The sixth chapter describes the use of Chipmunk, a 2D physics engine. 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, 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.

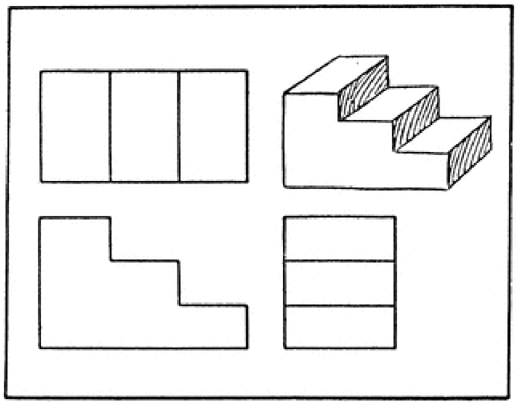
The projection

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

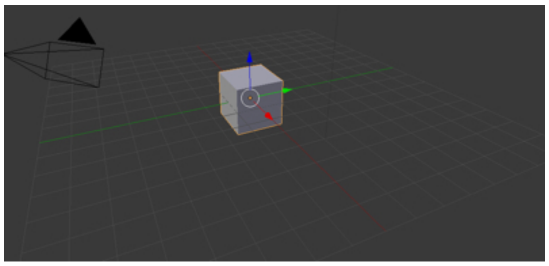
Under the hood OpenGL instructions in Gloss:

```
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:

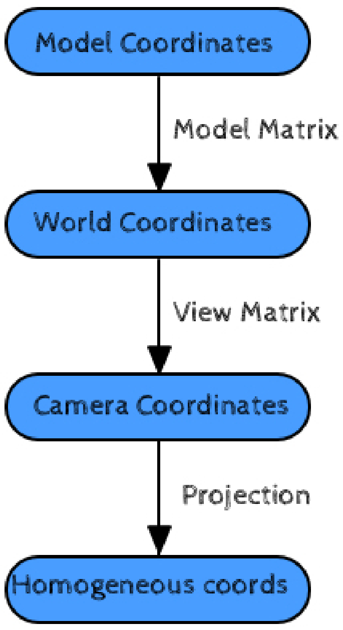


OpenGL coordinates (blender screenshot)

- *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:

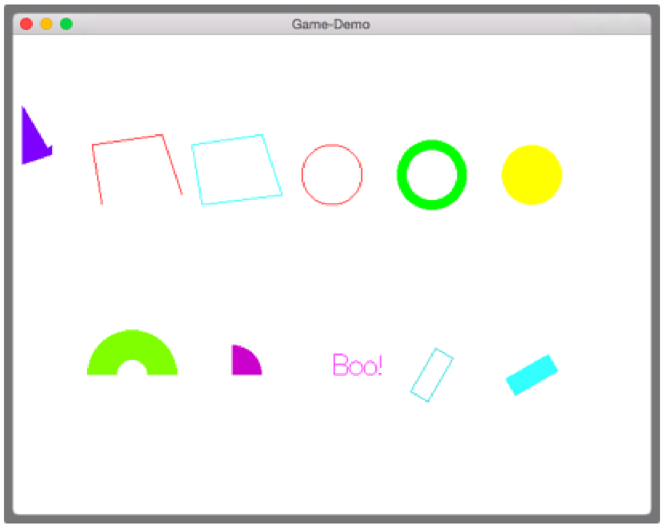
```
Color (bright magenta) $ translate 100 100 $ rotate 30 $ rectangleSolid 20 50
```

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:

```haskell
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.

Chapter 2: State with FRP

The code for this chapter can be found on https://github.com/elisehuard/game-in-haskell in the State.hs and StateFRP.hs files.

No life without state

State is this cumbersome, yet unavoidable thing: in a game, nothing can happen without at least keeping track of, say, the position of your character, or the state of the world, the score, health etc.

In Haskell you would usually encode the state in a type (or a number of types), and work with either a state monad, or to pass it around as an argument to relevant functions.

The code below shows simple implementation, with a player having x-y coordinates. Haskell types can be very expressive to describe states:

```
type Pos = Point -- gloss Point type (Float, Float)
data Player = Player {position :: Pos}

initialPlayer = Player (200,200)
playerSize = 20

  main :: IO ()
  main = do
    let width  = 640
        height = 480
    glossState <- initState
    withWindow width height "Game-Demo" $ \win -> do
          initGL width height
          loop win initialPlayer
          exitWith ExitSuccess
    where loop window state glossState =  do
            threadDelay 20000
            pollEvents
            k <- keyIsPressed window Key'Escape
            l <- keyIsPressed window Key'Left
            r <- keyIsPressed window Key'Right
            u <- keyIsPressed window Key'Up
            d <- keyIsPressed window Key'Down
```

```
              let newState = movePlayer (l,r,u,d) state 10
              renderFrame newState window glossState
              unless k  $ loop window newState
                then return ()
                else loop window newState glossState
```

The loop function gets a state argument - and the state is modified, using the *movePlayer* function, which takes the captured direction keys as an argument (the last numerical argument, *increment*, determines the speed of the player).

```
  movePlayer (True, _, _, _) (Player (xpos, ypos)) increment =
      Player ((xpos - increment), ypos)
  movePlayer (_, True, _, _) (Player (xpos, ypos)) increment =
      Player ((xpos + increment), ypos)
  movePlayer (_, _, True, _) (Player (xpos, ypos)) increment =
      Player (xpos, (ypos + increment))
  movePlayer (_, _, _, True) (Player (xpos, ypos)) increment =
      Player (xpos, (ypos - increment))
  movePlayer (False, False, False, False) (Player (xpos, ypos)) _ =
      Player (xpos, ypos)
```

The *movePlayer* function illustrates how pattern matching can provide crystal clear descriptions of state transitions - pattern matching happens on the tuple of direction key presses, and in function of this, the player gets moved by *increment*.

Actually, we may want to limit the movements of our player to the visible screen, so let's add conditions - a guard - to check whether the coordinates are close to the edge of our world:

```
  movePlayer :: (Bool, Bool, Bool, Bool) -> Player -> Float -> Player
  movePlayer direction player@(Player (xpos, ypos)) increment
    | outsideOfLimits (position (move direction player increment)) playerSize = \
player
    | otherwise = move direction player increment

  -- width and height = the size of our window
  outsideOfLimits :: (Float, Float) -> Float -> Bool
  outsideOfLimits (xmon, ymon) size =
      xmon > fromIntegral width/2 - size/2 ||
      xmon < (-(fromIntegral width)/2 + size/2) ||
      ymon > fromIntegral height/2 - size/2 ||
      ymon < (-(fromIntegral height)/2 + size/2)
```

```
move (True, _, _, _) (Player (xpos, ypos)) increment =
    Player ((xpos - increment), ypos)
move (_, True, _, _) (Player (xpos, ypos)) increment =
    Player ((xpos + increment), ypos)
move (_, _, True, _) (Player (xpos, ypos)) increment =
    Player (xpos, (ypos + increment))
move (_, _, _, True) (Player (xpos, ypos)) increment =
    Player (xpos, (ypos - increment))
move (False, False, False, False) (Player (xpos, ypos)) _ =
    Player (xpos, ypos)
```

If we're in danger of wandering off, the player doesn't allow the movement but stays in place.

Our *renderFrame* function is also modified to represent the current state - after all moving the player coordinates would be a pointless exercise without the visual feedback.

```
renderFrame (Player (xpos, ypos)) window = do
   displayPicture (width, height) white glossState 1.0 $
       translate xpos ypos $ rectangleSolid playerSize playerSize
   swapBuffers window
```

We simply draw a little black square in around the player's position.

In drawing individual frames we don't need to worry about the rest of the game, we just need to make sure that right now, at this moment, we represent the state of the world like we want it to look. It's handy to have such a clear separation between the graphical code and the actual mechanics of the game. You could completely swap out the graphical side of your game while still handling state the same way.

The code we're seen so far works: you can execute the example program, and using the arrow keys will navigate the little square up, down, left or right.

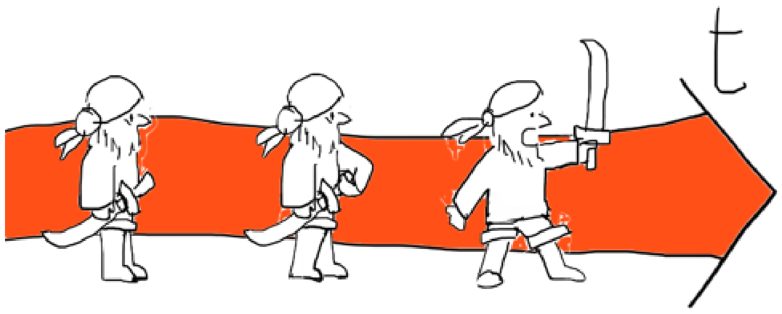
With me so far? Let's introduce functional reactive programming, a different way to approach state.

FRP

General definition

Functional Reactive Programming integrates time flow with events to make state evolve, in a functional, composed way. It can be used for domains like games, robotics, user interfaces, anything where you're looking to combine external inputs with state *over time* to produce a result.

Your state is not a single variable being modified by the loop, but becomes a combination of "Signal"/"Behaviour"/"Streams" over time ("signal" in the rest of this text). You could think about a signal as the history of a component, one long, linear timeline of states. You can combine those signals, and/or make them dependent on external inputs, also to be represented as signals.



Since its original conception by Conal Elliot and Paul Hudak[6], several variants of FRP have been formulated. They express the same general concept with different abstractions, which have each their own trade-offs. There are libraries in Haskell for pretty much each of the variants (see Evan Czaplicki's Strange Loop 2014 talk[7] for a good explanation).

The original formulation of FRP can be termed first-order FRP, where signals are known from start to finish, and defined combinations of signals are static. In most current variants of FRP, combinations of signals are dynamic, which means you can change, add or remove signals on the fly (which can be quite useful, as we will see later in the chapter).

For practical purposes, reactive-banana is a library that is more aimed at GUI development, Yampa and Netwire, both in the category of Arrowized FRP, have both been used for games, and Yampa has also been used in the context of music (synthesizers). I used yet another library called Elerea.

FRP with Elerea

Elerea, is, I quote[8], "referentially transparent ('compositional') higher-order streams without space or time leaks ('efficient')". The last bit (no space or time leaks) practically means that we don't keep track of the whole history of the signal, because this would make the memory useage grow over the time of the application.

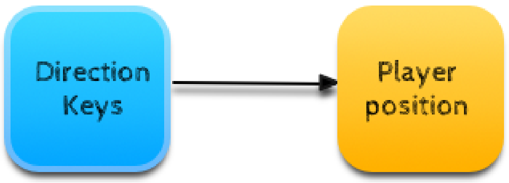
Every state is only dependent on what went on just before in the system. This means that if we need a cumulative value over time, we need to create a signal performing that operation.

Why use this library? Well, one reason (for me) is that the author, Patai Gergely, is a game developer,

[6]http://conal.net/papers/icfp97/
[7]http://www.youtube.com/watch?v=Agu6jipKfYw
[8]http://sgate.emt.bme.hu/documents/patai/publications/PataiWFLP2010.pdf

and he open sourced nice examples of his library (like elerea-examples[9] and dow[10]). A second reason is that I like the apparent simplicity of the concept - everything can be expressed with Signals and Signal Generators, both of which are monadic.

A game is expressed as a Signal Generator generating a graph of Signals. What do I mean by a graph (or network) of signals? We define at the very onset how our signals depend on each other. To take the example we described in the first paragraph of this chapter: the position of our Player in the earlier example is directly dependent on the direction key inputs.



The best way to illustrate how it all works is by showing a code example.

[9]https://github.com/cobbpg/elerea-examples.git
[10]https://github.com/cobbpg/dow.git

Transform our example

Let's take our previous example, and express it with FRP:

```
main :: IO ()
main = do
    let width  = 640
        height = 480
    (directionKey, directionKeySink) <- external (False, False, False, False)
    glossState <- initState

    withWindow width height "Game-Demo" $ \win -> do
          initGL width height
          network <- start $ do
            player <- transfer initialPlayer
                                (\s dK -> movePlayer s dK 10)
                                directionKey
            return $ renderFrame win glossState <$> player
          fix $ \loop -> do
               readInput win directionKeySink
               join network
               threadDelay 20000
               esc <- keyIsPressed win Key'Escape
               when (not esc) loop
          exitSuccess

readInput window directionKeySink = do
    pollEvents
    l <- keyIsPressed window Key'Left
    r <- keyIsPressed window Key'Right
    u <- keyIsPressed window Key'Up
    d <- keyIsPressed window Key'Down
    directionKeySink (l, r, u, d)
```

The state types, the renderFrame function and the movePlayer function don't change.

The first thing that does change, is that *external* is used to generate a signal, and a sink to feed that signal, for the direction key. The signal is given an initial value of (False, False, False, False).

```
external :: a                              -- initial value
         -> IO (Signal a, a -> IO ()) -- the signal and an IO function to feed it
```

Next, we describe how the state evolves in function of input. The key line here is

```
player <- transfer initialPlayer (\s dK -> movePlayer s dK 10) directionKey
```

transfer expresses the dependency between the player state and directionKey, and how the latter makes the player position change, starting from the initial state. Transfer's signature:

```
transfer :: a                         -- ^ initial internal state
         -> (t -> a -> a)             -- ^ state updater function
         -> Signal t                  -- ^ input signal
         -> SignalGen (Signal a)
```

The return value is a signal generator (needless to say, Elerea sports a *transfer2*, *transfer3*, *transfer4* for combining the information of 2, 3 or 4 other signals into the given signal).

The line with renderFrame expresses the output as a result of the calculation of a step in time of this network. *renderFrame*'s signature is *IO ()*, as said, so *return renderFrame* yields *SignalGen (IO ()).*

The *start* function executes the signal generator:

```
start :: SignalGen (Signal a) -> IO (IO a)
```

In this case, the type of *network* is IO (IO ()) since we use '<-' to draw from Signal, which is a monad. The role of start is to embed the signal graph into an IO environment by giving us a sample-stepping operation - to be used in the loop. The line of code defining *network* has done nothing but define how the game will unfold depending on inputs.

The actual action, unsurprisingly, happens in the loop, which is a recursive function pretty much just like the previous example. First the input is collected in *readInput* and fed to the sink, so that the directionKey signal has the necessary value. Then we do a *join* on the network. Join is a monadic function, which will remove one layer of the monadic structure. We were talking about IO (IO ()), and we need an IO (). This join operation could be expressed in do notation as follows:

```
join network == do { renderFrame <- network
                     renderFrame }
```

The rest of the loop is pretty much the same as in the conventional state example.

Worth remembering from this example:

- capture external inputs with *external*
- define a network of dependencies between external inputs and signals with *transfer*
- feed it to *start* to express the generator and get the desired IO (IO ()) out of it
- in the loop, capture all the necessary inputs and do a join operation on the resulting network

Monsters!

Let's make this a slightly more complex example to showcase more of the functionality, because the previous one is a little trivial. We'll add a little monster, which tries to hunt down the player when it's close-ish, or starts wandering randomly when it's on its own. And oh look, it changes color when it's on the hunt (deja-vu)! This will allow us to illustrate a couple of other features of Elerea.

First off, we'll add to the state. We're expressing our state as a network of moving objects, which each get their own data type - no need to pass it around as one monolythic state structure.

```
type Pos = Point
data Player = Player { position :: Pos }
data Monster = Monster Pos MonsterStatus
               deriving Show
data MonsterStatus = Wander Direction Int
                   | Hunting
               deriving Show
data Direction = WalkUp | WalkDown | WalkLeft | WalkRight
                 deriving (Show, Enum, Bounded)
```

We now have a monster type, which contains the position of the monster, but also a status expressing whether it's hunting or not, and which direction it's wandering in (the Wander type in particular is linked to the implementation details, as will become clear soon).

Now we need to describe the state of the monster as it wanders around, or actively hunts the player. Our signal graph changes a little:

```
    randomGenerator <- newStdGen
    network <- start $ do
           player <- transfer initialPlayer (\p dK -> movePlayer p dK 10) direc\
tionKey
           randomNumber <- stateful (undefined, randomGenerator) nextRandom
           monster <- transfer2 initialMonster wanderOrHunt player randomS
           return $ renderFrame win glossState <$> player <*> monster <*> rando\
mS
           where nextRandom (a, g) = random g
```

We have not one, but two extra signals - one is the series of random directions we'll use to have a random aspect to the wandering, the other the monster signal containing the monster state, which depends both on the player and the random directions.

Random

Unlike with imperative languages and most functional languages, I had to do some soul-searching before actually using random numbers. In other languages, you just use a rand function/method without skipping a beat.

A purely functional language like Haskell, though, makes you intensely aware of side-effects. And using a pseudo-random number generator means the state of that random number generator changes under the hood - it is a deterministic series of numbers (that are as unrelated to eachother as posssible) after all. If you run the same generator twice, it will progress along the same numbers. This state has to be conveyed either through a monad or through keeping track of state as an output of a function.

In the case of our game with Elerea, this means using a stateful signal, which will generate the next number using the random number generator we work with. It's worth noting that FRP makes randomness pure, since describing the evolution of the random generator in time makes it a composable entity.

To have a random direction, we need to define an instance of Random for the typeclass.

```
import System.Random
...
data Direction = WalkUp | WalkDown | WalkLeft | WalkRight
                 deriving (Show, Enum, Bounded)
instance Random Direction where
  randomR (a, b) g = case randomR (fromEnum a, fromEnum b) g of
                        (x, g') -> (toEnum x, g')
  random g = randomR (minBound, maxBound) g
```

The Direction type already derives the Enum typeclass, which has sensible defaults. *fromEnum* maps 0..3 to WalkUp..WalkRight, and toEnum does the opposite transformation. *minBound* is WalkUp, and *maxBound* returns WalkRight.

The result is that random g, with g being the random generator, gives a random direction. The generator, as said, contains the state, and should be kept, which is why we're using an Elerea function we haven't used so far, *stateful*:

```
randomNumber <- stateful (undefined, randomGenerator) nextRandom
```

stateful is for pure stateful signals, that is initial state is the first output, and every next state is transformed from that first state (without any external inputs).

```
stateful :: a                          -- ^ initial state
         -> (a -> a)                   -- ^ state transformation
         -> SignalGen (Signal a)
```

nextRandom just provides the next tuple in the sequence, since *random* returns a tuple of the given random number and the generator with its new state.

```
nextRandom (a, g) = random g
```

OK, let's move the monster:

```
  -- constants
  wanderDist = 40
  huntingDist = 100
  -- initial monster, peacefully wandering for wanderDist iterations
  initialMonster = Monster (200, 200) (Wander WalkUp wanderDist)

  -- wander or hunt
  wanderOrHunt player (randomDirection, _) monster =
                             if close player monster
                                then hunt player monster
                                else wander randomDirection monster

  close player monster = distance player monster < huntingDist^2

  distance (Player (xpos, ypos)) (Monster (xmon, ymon) _) =
      (xpos - xmon)^2 + (ypos - ymon)^2
```

So if the player and the monster are close (within a certain distance), hunt, otherwise wander.

Small optimizations

the Euclidian distance is normally the square root of the sum of squares - it's a typical optimization not to carry out this operation but to work with the squared distance instead, which works fine in this case.

Another tip is not to use the standard library random generator like I do here (newStdGen), which is a little slow, but to use a random generator of the mersenne-random[11] package.

Simple hunting: if the player is in the upper left quadrant relatively to the monster, move left diagonally, if in the upper right quadrant, right diagonally, etc (*signum* looks for the sign of the given operation).

[11]https://hackage.haskell.org/package/mersenne-random

```haskell
-- if player is upper left quadrant, diagonal left, etc
hunt :: Player -> Monster -> Monster
hunt (Player (xpos, ypos)) (Monster (xmon, ymon) _) =
    Monster
        ( (xmon + (signum (xpos - xmon))*monsterSpeed),
          (ymon + (signum (ypos - ymon))*monsterSpeed) )
        Hunting
```

Now we only need to define the wandering:

```haskell
  -- turn in random direction because either it's time,
  -- or we just transitioned from hunting state
  wander :: Direction -> Monster -> Monster
  wander r (Monster (xmon, ymon) (Wander _ 0)) =
    Monster (xmon, ymon) (Wander r wanderDist)
  wander r (Monster (xmon, ymon) Hunting) =
    Monster (xmon, ymon) (Wander r wanderDist)

  -- go straight
  wander _ (Monster (xmon, ymon) (Wander direction n)) = do
      let currentDirection = continueDirection direction (outsideOfLimits (xmon,\
 ymon) monsterSize)
      Monster
           (stepInCurrentDirection currentDirection (xmon, ymon) monsterSpeed)
           (Wander currentDirection (n-1))

  continueDirection :: Direction -> Bool -> Direction
  continueDirection WalkUp True = WalkDown
  continueDirection WalkDown True = WalkUp
  continueDirection WalkLeft True = WalkRight
  continueDirection WalkRight True = WalkLeft
  continueDirection direction False = direction

  stepInCurrentDirection WalkUp (xpos, ypos)    speed = (xpos, ypos + speed)
  stepInCurrentDirection WalkDown (xpos, ypos)  speed = (xpos, ypos - speed)
  stepInCurrentDirection WalkLeft (xpos, ypos)  speed = (xpos - speed, ypos)
  stepInCurrentDirection WalkRight (xpos, ypos) speed = (xpos + speed, ypos)
```

The use of the number (Int) in the *Walking* type is becomes clearer: the monster goes straight ahead for a certain number of iteration (*wanderDist* constant), which is decremented at every frame, and if it’s zero the monster makes a random turn and starts again. This is also the point where the random direction is used.

continueDirection expresses that we use guards to bounce back the monster if he's in danger of wandering off-limits (*outsideOfLimits* returns True), which is done by doing an about-face (WalkUp -> WalkDown etc) if are too close to the limits of the world. Otherwise the monster wanders straight ahead and the number is decremented (until we hit 0 and then the direction changes randomly).

Hey, this is starting to look like a game! The player needs to move to escape the monster!

But we need one more thing to make it real: if the monster catches the player, the game ends. How do we do this?

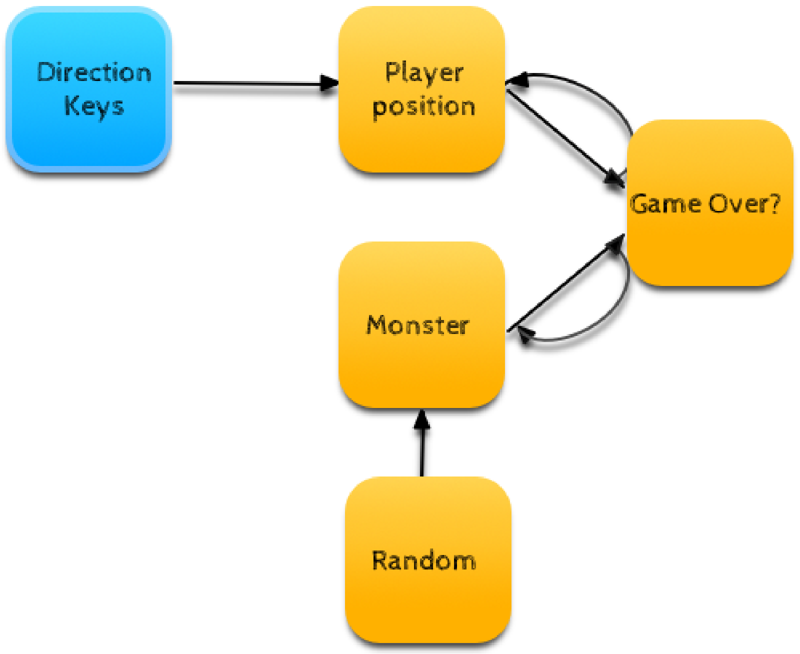
We'll need one more signal, which is the *gameOver* signal. We're illustrating a different technique again: how to generate a signal which is purely the product of one or more existing signals.

```haskell
import Control.Applicative ((<*>), (<$>))
...
let gameOver = playerEaten <$> player <*> monster
...
playerEaten player monster = distance player monster < 10^2
```

If you're not familiar with the use of <$> and <*>: we're using the fact that the Signal monad (like all monads) is also an applicative functor. These operators apply the function to the 'values' wrapped in the Signal, and output a Signal of the result (*gameOver* is of type Signal Bool).

Our Signal graph needs to change some more! We need to feed back the fact that the game is over to the player and the monster, so that they stop moving, and a "Game Over" text is displayed.

But wait, that means we have a looping graph: from player and monster signals to gameOver signal and gameOver signal back to player and monster! What to do?



Monster and Player both have a mutual dependency with GameOver. In vanilla Haskell, defining forward references (using a value which is defined on a later line in the code) in a do block causes compilation errors.

We are saved by the fact that we are speaking about signals in time. While it would be problematic to have *gameOver* be depending on *monster* and *player* at time t and vice-versa, it's fine to have *gameOver* depending on *monster* and *player* at time t, and *monster* and *player* depending on *gameOver* from previous iteration (t-1) - this way there's a step-wise calculation in time, instead of a loop of definitions that can never be resolved. How to encode this: use the *delay* function of Elerea. *delay* will generate that delayed signal we're looking for.

```
gameOver' <- delay False gameOver
```

Which is what we'll then use in the player and monster definitions.

Gotcha

Do *not* forget that delay step in using signals in a "loop". If you do, it really will be a loop, and your game will hang, understandably.

To put it another way: if your game hangs without any further feedback, check whether you haven't forgotten to use delayed signals in loops.

But we're not completely done, as Haskell will still think there's something not quite right in this picture, and refuse to compile. We solve this by using the language extension *RecursiveDo*, and mentioning this at the start of the file. *mdo* instead of do indicates where the resursive definitions occur.

```
{-# LANGUAGE RecursiveDo #-}
 ....
    network <- start $ mdo
        player <- transfer2 initialPlayer
                            (\p dead dK -> movePlayer p dK dead 10)
                            directionKey
                            gameOver'
        randomS <- stateful randomSeries pop
        monster <- transfer3 initialMonster wanderOrHunt player randomS gameOver'
        let gameOver = playerEaten <$> player <*> monster
        gameOver' <- delay False gameOver
        return $ renderFrame win glossState <$> player <*> monster <*> gameOver
 ...
```

We can do one extra thing: since gameOver' is used in 2 different places, we can cache the values so that they don't need to be reevaluated every time - with the memoizing function *memo*.

```
gameOver <- memo (playerEaten <$> player <*> monster)
```

movePlayer and wanderOrHunt change to accept the new Bool parameter, and in particular freeze the player and monster when the game is over:

```
movePlayer (_, _, _, _) player True _ = player
....
wanderOrHunt _ _ True monster = monster
```

Final step: draw the monster - in our case, a triangle which is green when wandering, and red when hunting, and print gameOver when state tells us it's over.

```
renderFrame :: Window -> State -> Player -> Monster -> Bool -> IO ()
renderFrame window
            glossState
            (Player (xpos, ypos))
            (Monster (xmon, ymon) status)
            gameOver = do
   displayPicture (width, height) white glossState 1.0 $
     Pictures $ gameOngoing gameOver
                       [renderPlayer xpos ypos,
                        renderMonster status xmon ymon]
   swapBuffers window

renderPlayer :: Float -> Float -> Picture
renderPlayer xpos ypos =
    Color black $ translate xpos ypos $ rectangleSolid playerSize playerSize

renderMonster :: MonsterStatus -> Float -> Float -> Picture
renderMonster status xpos ypos =
    Color (monsterColor status)  $ translate xpos ypos $ circleSolid playerSize
                          where monsterColor Hunting = red
                                monsterColor (Wander _ _) = green

-- adds gameover text if appropriate
gameOngoing :: Bool -> [Picture] -> [Picture]
gameOngoing gameOver pics =
  if gameOver
    then pics ++ [Color black $ translate (-100) 0 $ Scale 0.3 0.3 $ Text "Gam\
e Over"]
    else pics
```

Note that we want the “Game Over” text at the *end* of the array, since we want the text to be visible on top of everything else.

We can refactor our graph into its separate function when it gets a little complex, not to litter our main function:

```
  network <- start $ hunted win font directionKey randomSeries
....
  hunted win font directionKey randomSeries = mdo
      player <- transfer2 initialPlayer
                          (\p dead dK -> movePlayer p dK dead 10)
                          directionKey
                          gameOver'
      randomS <- stateful randomSeries pop
      monster <- transfer3 initialMonster wanderOrHunt player randomS gameOver'
      let gameOver = playerEaten <$> player <*> monster
      gameOver' <- delay False gameOver
      return $ renderFrame win font <$> player <*> monster <*> gameOver
      where playerEaten player monster = distance player monster < 10^2
            pop [] = []
            pop (x:xs) = xs
```

And this hunted function contains the whole definition of the game.

The player now needs to evade the monster. It’s not much of a game, but it’s a game already :)

Other modules of Elerea

So far we've been using FRP.Elerea.Simple. There are two other modules, FRP.Elerea.Param and FRP.Elerea.Clocked, which export variants of elerea, enabling extra features.

FRP.Elerea.Param allows you to pass certain external parameters to every single signal generator. The parameter this is most used with is the system time, or rather the increment of system time since the last time the network was evaluated. This can be very useful to (for instance) display time, or have time limits in your level, or even to measure the actual frame rate.

The readInput step of the loop changes to capture the clock increment:

```
readInput window closed directionKey resurrectKey killKey nextKey = do
    threadDelay 20000
    direction <- (,,,)
        <$> keyIsPressed window Key'Left
        <*> keyIsPressed window Key'Up
        <*> keyIsPressed window Key'Down
        <*> keyIsPressed window Key'Right
    directionKey direction

    t <- getTime
    resetTime

    k <- keyIsPressed window Key'Escape

    return $ if k ||  (t == Nothing) then Nothing else Just t
```

And every transfer function gets passed in the time increment (dt) as a first parameter. The signatures of all the Elerea functions change accordingly.

I've not used FRP.Elerea.Clocked, but the documentation implies that it adds a way to 'freeze' entire subnetworks when they are not being used. A 'withClock' function with a boolean condition will just recycle the same output value of the signal (without doing extra processing) if the boolean condition parameter returns true, and false otherwise. This is an optimization strategy, to be used when the network is starting to be large and slows down significantly.

Dynamic graphs, and levels

But what about levels? We now have state that exists for the entire lifetime of the game, but we don't really want that for level-specific data.

We could hack around this problem by making our signal state transition in value - say from world of level 1 to world of level 2 - and keep the same signals around like that, but that means a fair amount of hackery, which is always a little risky (what if you forget one signal from previous level? what if you don't really need this particular piece of state anymore at all?). We should just be able to tell the program that we don't need a particular signal anymore, and it can safely be garbage collected.

The solution is to add an extra signal generator. The graph we were talking about so far changes to have an extra generator which outputs a combination of two signals: one is the level state we need to represent, the other whether we need to progress to the next level. The level state can be defined in a new type (containing player, monster and level count for instance).

```
hunted font window directionKey = mdo
    let initialLevel = 1
    (levelState, levelTrigger) <-
            generate (playLevel directionKey <$> levelCount)
    levelTrigger' <- delay False levelTrigger
    levelCount <- transfer initialLevel levelProgression levelTrigger'
    return $ (renderframe font window) <$> levelState


 ...
generate gen = mdo
  trigger <- memo (snd =<< levelSignals)
  trigger' <- delay True trig -- signal bool
  -- signal of generators to run
  generatorSignal <- generator (toMaybe <$> trigger' <*> gen)
  -- signal that starts at undefined and progresses on path of generator
  levelSignals <- undefined --> generatorSignal
  return (fst =<< levelSignals, trigger)
```

If this makes your head hurt a little bit, don't worry - it took me a while to figure out as well. This is recursive signal definition in all its glory. *fst* and *snd* retrieve the first and second element of a tuple, respectively - and we expect the *playLevel* function to return a tuple of signals.

toMaybe is just a helper function which takes a Bool and a value and returns Just the value if Bool is True, and Nothing if false. *generator* is the new element here:

```
  generator :: Signal (SignalGen a) -- ^ the signal of generators to run
            -> SignalGen (Signal a) -- ^ the signal of generated structures
```

generator takes a signal of signal generators, and returns a signal generator of resulting signals (in our case the levelstate.

playLevel , the level generator we pass to 'generate', will have about the same code as our original hunted game had, it describes the dependencies of player, monster etc, and parameterizes where necessary using the level values (for instance in our monster game we could have a faster monster, a larger detection range, etc, as the levels progress). The main difference is that it returns (as in monadic 'return' to wrap in a Signal) a tuple of the level state and a boolean signalling whether we can progress to the next level.

```
playLevel directionKey level = mdo
   ...
   return (LevelState level <$> player <*> monster
          , levelCompleted) -- Bool whether to transition to next level
```

Another situation where generators can help to make the network dynamic: sometimes we want to create a variable number of entities (say - monsters, or bullets, or flying sheep). In that case, we work with an array of monsters, and the generator will generate an array of monster signals, combining existing enemies with newly spawned ones. A good example of this can be found in the Dungeons of Wor project on Github[12].

We will show complete examples of both those use cases in future chapters.

Animations

Something to think about: animations require state too. When we fade in we want to keep track of how far we are along the process to decide on the level of transparency - and when we have a cyclical animation (say a little character hopping or a shape rotating) we also need to know where we are in the cycle.

This is not the kind of thing you necessarily think about when imagining the state of the game - we prefer to think of game mechanics, AI and whatnot - but it belongs there as well.

With Elerea, I found two possible techniques to deal with this:

- create signals that deal 'just' with the animation. I've found this to be a little messy. Sometimes we don't need them to be around for the whole lifetime of the game/level, for instance in case of fade in and fade out - and in any case it feels like an artificial decoupling.
- integrate the animation state in the state of the entity concerned. This is not wildly inappropriate in my opinion, since in the end the sole aim of the state is to be represented in a visual way. This is not unlike what we do with the monster type in the example, by integrating MonsterStatus (Hunting/Walking), which is then translated to graphics in *renderFrame*.

We'll have several code examples of animations in the next chapter.

[12]https://github.com/cobbpg/dow/blob/master/src/Game.hs#L155

Ready to go

We now have the means to create a working game, with moving parts doing what they should! Isn't this amazing? There are examples of games online which don't use much more than that - addictive game mechanics will buy you a lot of forgiveness for basic graphics (think Tetris, Super Hexagon, Thomas Was Alone).

However most games go a few steps further: we want our monster to be a slavering monster, not a triangle, and the player a nice personable character, which gives a good impression of life and movement. This is what we'll look at in the next chapter.

Chapter 3: Textures and animations

Geometric shapes are often not quite enough to make our game come to life. We usually have a theme which requires more advanced graphics and animations. We want a bowling game with planets and asteroids, we want a little mage throwing around fire balls, a camel wandering around in the desert.

You'll find the code for this chapter on Github at https://github.com/elisehuard/game-in-haskell/blob/master/src/Animated.hs

Textures: using images in the game

Monsters for real now!

So far, we've had to limit ourselves to the primitives OpenGL had to offer. Textures allow us to map images onto shapes, and this makes all the difference.

For this, Gloss gives us loadBMP, to load an uncompressed bitmap as a Picture.

```
loadBMP :: FilePath -> IO Picture
```

Let's take our monsters game and add some textures.

We need an image for our player, our background and our monster. The monster can either be wandering around, or hunting (wwwwaarrrgh), so we need distinct images for those two states.

Textures

We use bitmaps for the player, the monster and a background, as Gloss works with BMPs out of the box. We store all the images in an *images* directory of our project.

It's advisable to load the images at the start of the program once and for all, instead of performing a costly load operation every time we render a frame. For the same reason, it's better to have the image at the right size from the start, instead of performing scale operations, which in Gloss will be performed every single time.

This is where glossState, the internal state we pass around while using Gloss, comes in useful: the textures are cached within OpenGL. Gloss records in the state whether texture objects have been used before, and are cached, instead of loading the image data every time.

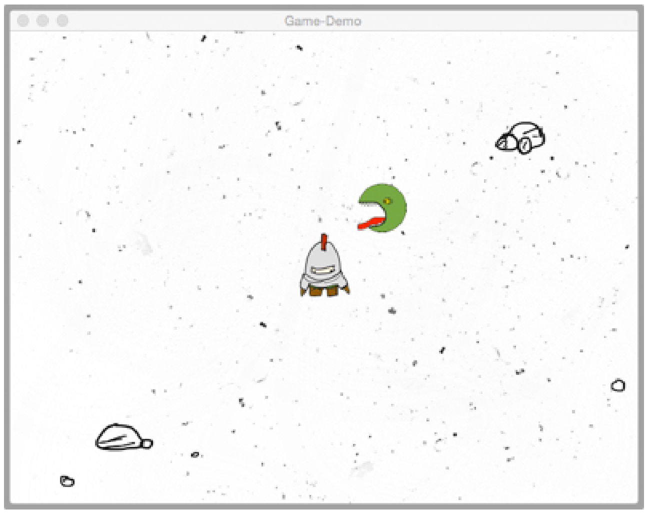
We load the textures before even creating the window:

```
      playerTexture <- loadBMP "images/alien.bmp"
      backgroundTexture <- loadBMP "images/background-1.bmp"
      monsterWalkingTexture <- loadBMP "images/monster-walking.bmp"
      monsterHuntingTexture <- loadBMP "images/monster-hunting.bmp"

      withWindow width height "Game-Demo" $ \win -> do
          let textures = [playerTexture, monsterWalkingTexture, monsterHuntingTe\
xture, backgroundTexture]
          network <- start $ hunted win directionKey randomGenerator textures gl\
ossState
```

The signature of the *hunted* function gets an array of [Picture] added, as does renderFrame, to pass in all the images.

```
  hunted win directionKey randomGenerator textures = mdo
      ....
      return $ renderFrame win glossState textures <$> player
                                                   <*> monster
                                                   <*> gameOver

 -- rendering
  renderFrame
      window
      glossState
      [playerTexture, monsterWalkingTexture, monsterHuntingTexture, backgroundTe\
xture]
      (Player (xpos, ypos)) (Monster (xmon, ymon) status) gameOver = do
     displayPicture (width, height) white 1.0 $
       Pictures $ gameOngoing gameOver
                              [backgroundTexture,
                               renderPlayer xpos ypos playerTexture,
                               renderMonster status xmon ymon
                                 (monsterWalkingTexture, monsterHuntingTexture)]
     swapBuffers window

  renderPlayer :: Float -> Float -> Picture -> Picture
  renderPlayer xpos ypos texture = translate xpos ypos $ texture

  renderMonster :: MonsterStatus
                -> Float
                -> Float
                -> (Picture, Picture)
                -> Picture
  renderMonster Hunting xpos ypos (_, monsterHuntingTexture) =
       translate xpos ypos $ monsterHuntingTexture
  renderMonster (Wander _ _) xpos ypos (monsterWalkingTexture, _) =
      translate xpos ypos $ monsterWalkingTexture
```

The background picture is the first element of the Picture array, then we render the player and the monster on top of the background.



Textures in the game

Fonts in Gloss

As already mentioned in Chapter 1, Gloss only has one font at the moment. As any designer will tell you, we need more than that for a unified feel. Textures can be a workaround: for fixed texts especially (for instance “Game Over”) we can use an image instead of generating the text with Gloss.

The real solution is to improve the font support, which might happen some way down the road, if I have my say in it.

Interlude: draw your own

Non-technical advice: if you’re starting on a game, don’t use a designer. Either draw your own, which can be lots of fun, or get Creative Commons licensed images off the internet (or if you’re not releasing it right away, any images will do).

Drawing your own may be the best option - buy a cheap graphic tablet and get down to it. Really. Even if you’re not a natural talent (I’m not) having an “undo” feature goes a long way to experiment your way towards something viable. For drawing software I use Pixelmator on Mac OS X, but Gimp is also an option - free, full-featured software that works on both Mac and Linux.

relatively cheap Wacom Intuos tablet

Using a designer will cost you money. It's something you don't want to get into before a later stage: * you want to prototype your game and ensure your concept actually has legs - if it's not fun to play with half-baked drawings, it probably won't be much better with professionnal assets. It's possible that you'll have to overhaul a lot of things, and then you'll have spent money for nothing. * your drawings will help you communicate with the designer later on, give them an idea of what you're imagining, the general spirit of things if you will.

And drawing is fun! Programming a game is a creative endeavour, you're tapping into that mysterious part of your mind which comes out of left field with new concepts an ideas - a part we're admittedly not using that much in our day-to-day professional life as programmers. Prodding that part by visualizing and then drawing parts of your game can only be beneficial. Ask any creative professional: that muscle needs training.

If you want some inspiration, watch "Indie games, the movie"[13]. You'll see that the creator of the amazing game Braid, Jonathan Blow, does the same. He starts of with wireframes, sketchy textures, because it allows him to test if the game mechanics work, before iterating to more advanced graphics.

If you really feel unable to draw simple prototypes, you can always go with free assets made available on the web - looking for Creative Commons licenced pictures (for reuse,even in commercial applications, at least if you think of distributing your game commercially).

Another intermediate solution (spend some money, but not too much) is to buy ready-made game assets - there are a variety of websites offering them (super game asset[14], Graphic Buffet[15] … I can't vouch for the quality, but googling should yield some more).

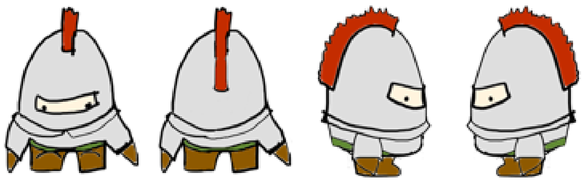
[13]http://buy.indiegamethemovie.com/

[14]http://www.supergameasset.com/

[15]http://www.graphic-buffet.com/

Bringing life to the characters

With our previous example, we have something that looks a little better, but feels stunted: the player floats across the screen, looking the same wherever it goes. Same goes for the monster. A first step at making this feel better is to have a player facing in each of the 4 directions. When the player moves away from us, we should see its back.



Several views

A consequence of having to store several perspectives for each moving object is that we're starting to have a load of textures - it's a little inelegant to store everything in an array. In a larger game a map might be appropriate - however, since we have a game with few characters, we can get away with defining record types.

```
  data TextureSet = TextureSet { front :: Picture, back :: Picture, left :: Pict\
ure, right :: Picture }
  data Textures = Textures { background :: Picture
                           , player :: TextureSet
                           , monsterWalking :: TextureSet
                           , monsterHunting :: TextureSet }
```

Loading the textures at the start of the loop:

```
  loadTextures :: IO Textures
  loadTextures = do
      playerTextureSet <- TextureSet <$> loadBMP "images/knight-front.bmp"
                                     <*> loadBMP "images/knight-back.bmp"
                                     <*> loadBMP "images/knight-left.bmp"
                                     <*> loadBMP "images/knight-right.bmp"
      monsterWalkingSet <- TextureSet
                                <$> loadBMP "images/monster-walking-front.bmp"
                                <*> loadBMP "images/monster-walking-back.bmp"
                                <*> loadBMP "images/monster-walking-left.bmp"
                                <*> loadBMP "images/monster-walking-right.bmp"
      -- moves diagonally, so only 2 textures needed technically
      monsterHuntingSet <- TextureSet
```

```
                                <$> loadBMP "images/monster-hunting-left.bmp"
                                <*> loadBMP "images/monster-hunting-right.bmp"
                                <*> loadBMP "images/monster-hunting-left.bmp"
                                <*> loadBMP "images/monster-hunting-right.bmp"
    backgroundTexture <- loadBMP "images/background-1.bmp"
    return Textures { background = backgroundTexture
                    , player = playerTextureSet
                    , monsterWalking = monsterWalkingSet
                    , monsterHunting = monsterHuntingSet }
```

To render the appropriate image, we should be able to pass on the moving direction to the rendering of the frames: at the moment the rendering doesn't get any information about which direction the player is going in, and when the monster is hunting, there is also no indication of direction. We can change this by adding the relevant information to the state of both the Player and the hunting Monster. We probably also want an extra piece of information about the player: whether it is moving or not. A good way to express this is to make the direction field can be a Maybe field for the Player. The MonsterStatus for hunting gets a direction parameter.

```
  data Player = Player { position :: Pos, dir :: Maybe Direction }
                deriving Show
  ....
  data MonsterStatus = Wander Direction Int
                     | Hunting Direction
                deriving Show
```

Now those directions need to be provided in the movement handling:

```
-- player
  move (True, _, _, _) (Player (xpos, ypos) _) increment =
      Player ((xpos - increment), ypos) $ Just WalkLeft
  move (_, True, _, _) (Player (xpos, ypos) _) increment =
      Player ((xpos + increment), ypos) $ Just WalkRight
  move (_, _, True, _) (Player (xpos, ypos) _) increment =
      Player (xpos, (ypos + increment)) $ Just WalkUp
  move (_, _, _, True) (Player (xpos, ypos) _) increment =
      Player (xpos, (ypos - increment)) $ Just WalkDown
  move (False, False, False, False) (Player (xpos, ypos) _) _ =
      Player (xpos, ypos) Nothing

  -- monster: hunting
  hunt :: Player -> Monster -> Monster
  hunt (Player (xpos, ypos) _) (Monster (xmon, ymon) _) =
      Monster ((xmon + (signum (xpos - xmon))*monsterSpeed),
          (ymon + (signum (ypos - ymon))*monsterSpeed))
          (Hunting $ huntingDirection (signum (xpos - xmon)) (signum (ypos - ymo\
n)))

  huntingDirection (-1) (-1) = WalkLeft
  huntingDirection (-1) 1 = WalkLeft
  huntingDirection 1 (-1) = WalkRight
  huntingDirection 1 1 = WalkRight
  huntingDirection _ _ = WalkRight
```

And then we need to use the direction parameter passed in the rendering, and use the appropriate element of the TextureSet (front, back, left, right textures).

```
  renderFrame window
              glossState
              textures
              (Player (xpos, ypos) playerDir)
              (Monster (xmon, ymon) status) gameOver = do
     displayPicture (width, height) white glossState 1.0 $
       Pictures $ gameOngoing gameOver
                             [background textures,
                              renderPlayer xpos ypos
                                           playerDir
                                           (player textures),
                              renderMonster status
                                            xmon ymon
                                            (monsterWalking textures)
                                            (monsterHunting textures) ]
     swapBuffers window


  renderPlayer :: Float -> Float -> Maybe Direction -> TextureSet -> Picture
  renderPlayer xpos ypos (Just WalkUp) textureSet =
      translate xpos ypos $ back textureSet
  renderPlayer xpos ypos (Just WalkDown) textureSet =
      translate xpos ypos $ front textureSet
  renderPlayer xpos ypos (Just WalkRight) textureSet =
      translate xpos ypos $ right textureSet
  renderPlayer xpos ypos (Just WalkLeft) textureSet =
      translate xpos ypos $ left textureSet
  renderPlayer xpos ypos Nothing textureSet =
      translate xpos ypos $ front textureSet

  renderMonster :: MonsterStatus
                -> Float
                -> Float
                -> TextureSet
                -> TextureSet
                -> Picture
  renderMonster (Hunting WalkLeft) xpos ypos _ textureSet =
      translate xpos ypos $ left textureSet
  renderMonster (Hunting WalkRight) xpos ypos _ textureSet =
      translate xpos ypos $ right textureSet
  renderMonster (Wander WalkUp _) xpos ypos textureSet _ =
      translate xpos ypos $ back textureSet
  renderMonster (Wander WalkDown _) xpos ypos textureSet _ =
      translate xpos ypos $ front textureSet
  renderMonster (Wander WalkLeft _) xpos ypos textureSet _ =
      translate xpos ypos $ left textureSet
  renderMonster (Wander WalkRight _) xpos ypos textureSet _ =
      translate xpos ypos $ right textureSet
```

Our monster and player now duly present front, profile or back as appropriate.

Animations: stop motion

It's still not quite where we want it to be: to be more lifelike we'd need the characters to perform some sort of lifelike movement.

In our case, we can get away with doing some basic cartoon animations. The player walks, so we need some textures to express that walking motion. We're basically drawing one texture per movement, we'll go with 3 images for a cycle of 4 - lift feet, swing the arms a little. An animation of 4 images doesn't really make for a fluid movement yet, but you get the idea.

4 times the number of angles we see the player under (front, back, left, right). It's a combinatorial explosion of images. You start to understand why providing assets for a game could be a full-time job.

Knight animation: front

We need to add to the state of our player and monster again: which part of the animation needs to be displayed in this frame? We define a PlayerMovement type, with *step* expressing the step in the animation cycle.

```
data Player = Player { position :: Pos, movement :: Maybe PlayerMovement }
              deriving Show
data PlayerMovement = PlayerMovement { dir :: Direction, step :: WalkStage }
              deriving Show

data WalkStage = One | Two | Three | Four
              deriving (Show, Eq, Enum, Bounded)
```

In another language we'd have used an integer, and done a modulo over the number to have a cycle of 0,1,2,3,0,1,2,3. With Haskell it's actually a better idea to define a bounded, ordered type with just four elements, and cycle over that instead:

```
circular :: (Eq x, Enum x, Bounded x) => x -> x
circular x = if x == maxBound then minBound else succ x
```

Now this value needs to be managed in the player state:

```
  move (True, _, _, _)
       (Player (xpos, ypos)
       (Just (PlayerMovement WalkLeft n)))
       increment =
    Player (xpos - increment, ypos) (Just $ PlayerMovement WalkLeft (circular n))

  move (True, _, _, _) (Player (xpos, ypos) _) increment =
    Player (xpos - increment, ypos) $ Just $ PlayerMovement WalkLeft 0
```

(rince and repeat for the other 3 directions WalkUp, WalkDown, WalkRight) If the player was already moving in a certain direction, we increment the step by one - modulo 4 to cycle over the animations. When the player was moving in another direction, and starts to walk left, then we start the animation cycle from 0.

When no movement happens: {language=haskell, line-numbers=off} ~~~~~~~~~~~~~ move (False, False, False, False) (Player (xpos, ypos) _) _ = Player (xpos, ypos) Nothing ~~~~~~~~~~~~~

Rendering the appropriate textures:

```
  renderPlayer :: Maybe PlayerMovement -> TextureSet -> Picture
  renderPlayer (Just (PlayerMovement facing One)) textureSet = neutral $ playerD\
irectionTexture facing textureSet
  renderPlayer (Just (PlayerMovement facing Two)) textureSet = walkLeft $ player\
DirectionTexture facing textureSet
  renderPlayer (Just (PlayerMovement facing Three)) textureSet = neutral $ playe\
rDirectionTexture facing textureSet
  renderPlayer (Just (PlayerMovement facing Four)) textureSet = walkRight $ play\
erDirectionTexture facing textureSet
  renderPlayer Nothing textureSet = neutral $ fronts textureSet

  playerDirectionTexture :: Direction -> TextureSet -> WalkingTexture
  playerDirectionTexture WalkUp = backs
  playerDirectionTexture WalkDown = fronts
  playerDirectionTexture WalkLeft = lefts
  playerDirectionTexture WalkRight = rights
```

In other words: start from neutral, raise one foot and swing arms, back to neutral, raise the other foot and swing arms to the other direction, and start again. It isn't much, but it's much better than gliding like a ghost over the ground.

A more advanced version of this is keyframe animation[16]: in the case of keyframe animation, the key frames are defined as above, but linear or quadratic interpolation is carried out between them

[16]http://en.wikipedia.org/wiki/Key_frame

to smooth the transition, using corresponding coordinates and texture locations. This can not be done with textures in Gloss at this point in time - or only in a very limited way - but requires using different bindings (for instance OpenGL directly). It's definitely possible to do it with colors and geometric shapes primitives.

Animations: skeletal animation and transformations

The above technique is all good and well when you have a limited amount of movements, but it starts to become a little painful when the character is supposed to perform a lot of movements fluidly. That's when we want to start to decompose our character into moving parts, and then programmatically animate them - by translating, rotating, translating.

We can make our monster's wandering a very simple example: it's a round dragon, and it doesn't take itself very seriously, so we'll just make it rotate like a ball while it's wandering sideways. The state required to implement this is already available, we can use the wandering distance (wanter 10 steps and then turn) to also specify the angle of the rotation.

```
renderMonster (Wander WalkLeft n) xpos ypos textureSet _ =
    translate xpos ypos $ rotate (16* fromIntegral n) $ left textureSet
renderMonster (Wander WalkRight n) xpos ypos textureSet _ =
    translate xpos ypos $ rotate (16* fromIntegral n) $ right textureSet
```

Another example could be having our Player wave: having a separate texture for the arm, and having that rotate and translate as needed. This is closer to what is called skeletal animation[17], as we're picturing the arm as having a joint which rotates.

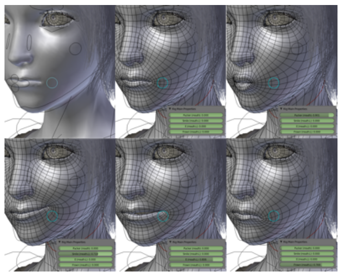
You could also play with the dimensions of your texture to create a distorted or pulsating effect, using scale, for example.

Advanced animation

The above techniques are mostly enough for a 2D game - but to be completely truthful, using Gloss limits the use of more advanced techniques. For move advanced animation, it is necessary to use a *mesh* of vertices with their texture coordinates (mapping from the texture image to the figure). The texture image gets stretched and crunched together as needed.

This is what happens in big studio games - the 3D object have a texture meshed over them, and the texture moves with the character or object.

[17]http://en.wikipedia.org/wiki/Skeletal_animation



Sintel face morph (Wikipedia)

When reaching that stage, generating all the vertex coordinates and texture coordinates manually probably won't cut it any more - you need software to generate it. Blender[18] is free and open source software to model 3D shapes. It can export OBJ files, which can then be converted to a readable format and loaded in OpenGL.

[18]http://www.blender.org/

The world is a bigger place: viewport

Coming back to our 2D game, we usually would like our player to move around in a bigger place, exploring a world that is larger than just the visible window.

That's where we use a *viewport*: the viewport is a window on our 2D world, which moves around with our player - a kind of animation in itself. What does this mean in practice? We zoom in onto a particular rectangle of our world, and this rectangle needs to change as action requires it.

Gloss provides a module for that, which we can use in our code. ViewPort gets its own signal in our FRP framework. The signal is influenced by the player position, so we definitely need that as an input, which means we use a transfer function. *viewportMove* expresses how the viewport evolves in function of the player: it gets the negative of the player's movement (when the player moves left, the background does actually move right).

```haskell
import Graphics.Gloss.Data.Viewport
....
  initialViewport :: ViewPort
  initialViewport =
     ViewPort { viewPortTranslate = (0, 0),
                viewPortRotate = 0,
                viewPortScale = 4 }

  hunted win directionKey randomGenerator textures glossState = mdo
      player <- transfer2 initialPlayer (\p dead dK -> movePlayer p dK dead 10)
                              directionKey gameOver'
      (...)
      viewport <- transfer initialViewport viewPortMove player
      return $ renderFrame win glossState textures <$> player
                                                   <*> monster
                                                   <*> gameOver
                                                   <*> viewport
      (...)

  viewPortMove :: Player -> ViewPort -> ViewPort
  viewPortMove (Player (x,y) _)
               (ViewPort { viewPortTranslate = _,
                           viewPortRotate = rotation,
                           viewPortScale = scaled }) =
      ViewPort { viewPortTranslate = ((-x), (-y)),
                 viewPortRotate = rotation,
                 viewPortScale = scaled }
```

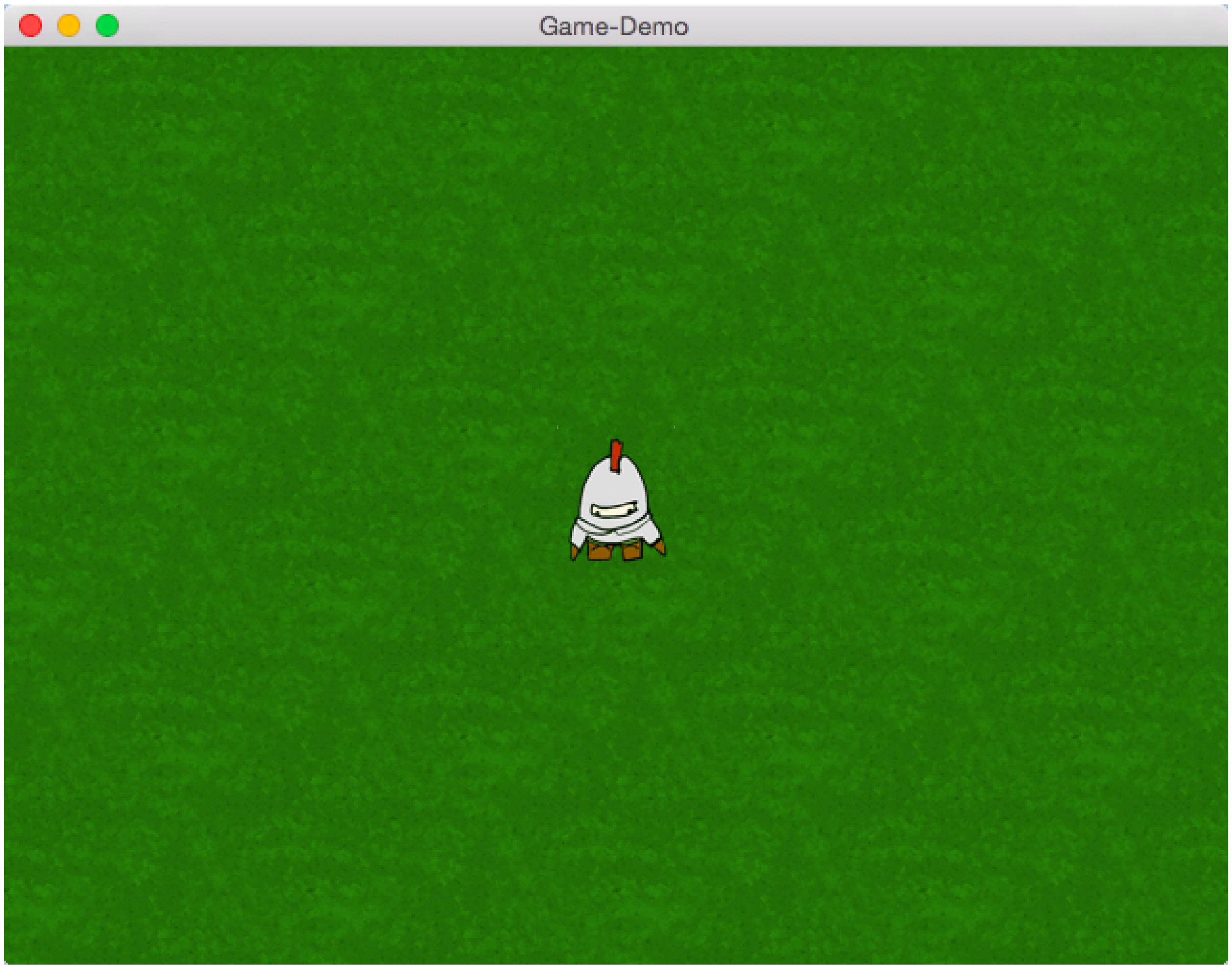
We now pass in the viewport data to *renderFrame*. Interestingly, our viewport approach means that our player doesn't move any more, visually but stays at the (0,0) of our screen representation - so we don't need the player coordinates for the rendering anymore.

When we move, the background moves: *applyViewPortToPicture*, a Gloss function, applies the viewport to the background, scaling and translating it.

Note that this is also the approach to use in a platform game: your point of view moves with the character, and you only see the part of the world directly surrounding the character.

The monster continues to roam as usual, but the coordinates it's moving on are subject to the same transformation as the background is, at least the translation (no need to scale it to make it four times bigger). It also disappears out of our viewport when it's not in the same window as the player. Fortunately, OpenGL takes care of that for us in function of its coordinates, we don't need to do anything special.

```
-- rendering
renderFrame window
            glossState
            textures
            (Player (xpos, ypos) playerDir)
            (Monster (xmon, ymon) status) gameOver viewport = do
 displayPicture (width, height) black glossState (viewPortScale viewport) $
   Pictures $
      gameOngoing gameOver
           [applyViewPortToPicture viewport (background textures),
            renderPlayer playerDir (player textures),
            uncurry translate (viewPortTranslate viewport) $
                renderMonster status
                              xmon ymon
                              (monsterWalking textures)
                              (monsterHunting textures) ]
 swapBuffers window
```



Game through a viewport

We need to add new constants to our game: the world width and the world height, which are no longer the same as our window width and height. These are also the dimensions that should be use when deciding whether the player or monster are getting out of bounds.

```
worldWidth :: Float
worldWidth = 2560

worldHeight :: Float
worldHeight = 1920

outsideOfLimits :: (Float, Float) -> Float -> Bool
outsideOfLimits (xpos, ypos) size = xpos > worldWidth/2 - size/2 ||
                                    xpos < ((-worldWidth)/2 + size/2) ||
                                    ypos > worldHeight/2 - size/2 ||
                                    ypos < ((-worldHeight)/2 + size/2)
```

A finishing touch is to have the viewport stop at the edges of the world (instead of showing the void outside it), and leave the player to move to the edges if desired. I leave this as an exercise to the reader.

One gotcha here: the resolution. Using the formula above actually doesn't work with our texture, because it gets blown up. Blowing up a bitmap means a grainy effect, so unless you're going for the vintage 8-bit effect, that's not what you want. This is not a problem when you use the OpenGL primitives, which amount to SVG, and are scaleable at will.

An intuitive solution is to have a massive texture (four times the width, four times the height), but that's a huge memory hog: you're loading 16 times more pixels, and it slows everything right down. So that's not an option.

A better solution is actually to split up the background with tiles, potentially layering textures to add, say, a tree, a rock, or a castle. The functional API of gloss is a nice help with that, since we can define our background as an array of pictures (*Pictures []* constructor), and define that array using functional composition (which beats the hell out of doing it with OpenGL directly, believe me).

For instance, we remove the scaling from the viewport application altogether, and use the following for the background rendering:

```
uncurry translate (viewPortTranslate viewport) $ tiledBackground (background tex\
tures) worldWidth worldHeight

tiledBackground texture width height = Pictures $ map (\a ->  ((uncurry translat\
e) a) texture)
                                                        $ translateMatrix worldWidth wor\
ldHeight
```

The *translateMatrix* functions yields an array of tuples containing the respective positions of all the tiles. We map so that every one of these position gets applied to *translate x y $ texture*, just like for the other elements. The background texture in this case is a nice uniform grass texture with a size of 160 pixels.

Number conversions

You'll undoubtedly stumble over number conversions. It pays to do them correctly, because if you don't, it will come back to bite you hard (this applies to every app, not only gaming).

Gloss works with Floats, and we have some Ints in the program, so the issue is limited in this case: *fromIntegral* will make a Float out of an Int, and *floor* or *round* will make an Int out of a Float (losing precision if there's any decimals of course).

Nearly there

Our game starts looking like a game. We also need sound - sound is surprisingly important in making a game come to life. The best games have absolute earworms for soundtracks (Tetris, Katamari Damacy, ...). Time to look into that side of things.

Chapter 4: Sound

Sound is paramount in making a game feel complete. The best games have haunting soundtracks and immersive sounds. This chapter will discuss how to add sound to the game.

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

Package to add to the cabal file: ALUT

Challenges

Obtaining sounds

First off, we're faced with a challenge not unlike the one we had with textures and animations: music and sounds are a creative endeavour and so just downloading any track and using it in a game would unfortunately be illegal, unless you negociated and paid the appropriate rights with the relevant label.

If you have sound engineer or musician friends, or are any of those yourself, you could produce your own music. Obviously, if you use a friend's band, you'll need to cut them in or negociate licensing with them if you intend to make profits on your game.

You don't even have to compose - you could take a tune that is in the public domain (which often means: old enough) and interpret it using whatever instrument or software strikes your fancy. Public domain unfortunately still doesn't mean you could just download them from somewhere, because that would breach the performers' rights.

Some of us have no skills in the musical department - fortunately, there's another option: look for sounds under a Creative Commons license (for reuse without restrictions) and use those. A good source of sounds, for instance, is freesound[19] - it may take some rooting through the samples, but you might find what you need. Soundcloud[20] also lets you filter on license (Filter results on the left, click on the © symbol). There are probably many other sources of loosely-licensed sounds.

You can also buy audio assets online, so for a slightly more professional touch, it might be the way to go. The vendors usually tailor the price to the size of the audience, so it can start with a cheap license. One thing to remember, though, is that you usually are not allowed to open source samples obtained like this (so it's out for this ebook, for instance).

Which sounds will we need?

[19]https://www.freesound.org/
[20]https://soundcloud.com/

- background music - or musics, depending on whether switching to another level or mode changes the background music.
- punctual sounds: sounds that occur for punctual events, like jumping, shooting, obtaining loot …
- variable length sounds: when a sound should continue as long as you push a key, or as long as something happens in the game, for instance. This practically means a sound you could repeat seamlessly, with potentially an initial sample for the attack and a sample at the end for the release.

With the library we're using (OpenAL), we need WAV files to work with. We can store all our sounds in a *sounds* directory in the directory structure of our project.

The loop

We have another challenge, of a sort: our main loop is really tuned to our graphical needs, based on the desired framerate. Sounds will need to be managed within that framework - triggering when needed but also making sure we don't start, say, 60 (or framerate) samples per second. Continuously playing samples, for instance, need to be handled carefully.

Also we're not only rendering our state graphically, we need to "render" the soundscape as well - the conclusion of our loop (*renderFrame*) needs to change somewhat.

Let's start coding to show how we'll manage both those issues.

Let's code

OpenAL and ALUT

OpenAL is a cross-platform audio API[21]. It's designed for efficient rendering of multichannel three-dimensional positional audio, so it's much more powerful than we require for a 2D game. Its API style and conventions are modeled after OpenGL.

ALUT stands for Audio Library Utility Toolkit, and provides higher-level convenience functions on top of OpenAL. This is what we'll use in this book.

It needs to be said: just like with Gloss, there are many other options and libraries that could do what you need. OpenAL is just the one I found the easiest to use and install, but Your Mileage May Vary.

For more processor-intensive and geeky fun, you could always synthesize the sound yourself. An example of this can be found in Asteroids with Netwire by Oliver Charles[22].

[21]http://en.wikipedia.org/wiki/OpenAL
[22]https://ocharles.org.uk/blog/posts/2013-08-18-asteroids-in-netwire.html

Background music

We want a background music sample to loop in the background. “Looping in the background” usually means a background thread, but in this case any background processing is handled by OpenAL, and we just need to let it know when we need a sample played. In this it’s not unlike the graphical pipeline with OpenGL, which happens pretty much behind the scene as far as we’re concerned.

Playing a sound in OpenAL requires first setting up a context (similarly to GLFW-b), then creating an object in that context containing contents loaded from the WAV file, and then playing it.

A simple implementation of background music - where we only have one background tune which plays as long as the program runs - could be as follows:

```
import Sound.ALUT hiding (Static)
...
-- loading sound
loadSound path = do
    -- Create an AL buffer from the given sound file.
    buf <- createBuffer (File path)
    source <- genObjectName
    buffer source $= Just buf
    return source

backgroundMusic :: Source -> IO ()
backgroundMusic source = do
  -- ALUT initialize context
  withProgNameAndArgs runALUT $ \progName args -> do
        loopingMode source $= Looping
        play [source]

-- in main function, once window is created
music <- loadSound "sounds/oboe-loop.wav"
backgroundMusic music
```

Setting *loopingMode* of the sample source to *Looping* makes sure the sample restarts once it’s been played to the end (the default is *OneShot*). This is all good and well, but it’s a little restrictive: one tune playing over and over again for the whole duration of the game may be what you want, but maybe we want the state of our game to affect the background music too.

Music

We have to change our existing code a little to be able to affect what's going on. Our loop gets an extra output component, and becomes more general than just graphically rendering a frame. We can define appropriate types to express the "routing" of the state information to either rendering or to sound output (*RenderState* and *SoundState*).

Say we want our music to be affected by whether the monster is hunting the player - we can create an extra data type to express whether the player is being hunted or not (*StatusChange*).

```
data StatusChange = Danger | Safe

data RenderState = RenderState Player Monster Bool ViewPort
data SoundState = SoundState (Maybe StatusChange)
...
outputFunction window glossState textures music renderState soundState =
    (renderFrame window glossState textures renderState) >>
    (playSounds music soundState)
```

We use FRP to manage the music state: we're going to speed up the music when the monster starts hunting, and slow it down again when he starts wandering again. Since we don't want to update the music speed 60 (framerate) times per second, we need to be able to work with changes in monster status, not with the actual status itself. We're doing a kind of differentiation operation on the monster signal. Elerea's *delay* function comes to the rescue again, as we use not only current monster status, but the delayed monster status.

We also need to pass in the loaded background music sound to the FRP network function *hunted* to be able to affect it in the loop.

```
  hunted win directionKey randomGenerator textures glossState music = mdo
...
    -- in hunted: new delayed monster signal
    monster' <- delay initialMonster monster
    statusChange <- transfer2 Nothing monitorStatusChange monster monster'

    let renderState = RenderState <$> player
                                  <*> monster
                                  <*> gameOver
                                  <*> viewport
        soundState  = SoundState <$> statusChange

    return $ outputFunction win glossState textures music <$> renderState
                                                          <*> soundState

  monitorStatusChange (Monster _ (Hunting _)) (Monster _ (Wander _ _)) pace =
    Just Danger
  monitorStatusChange (Monster _ (Wander _ _)) (Monster _ (Hunting _)) pace =
    Just Safe
  monitorStatusChange _ _ pace =
    Nothing
```

Note how we compose the *renderState* and the *soundState* signals out of the relevant signals before passing them to the outputFunction.

monitorStatusChange expresses whether the monster changed status by returning a *Maybe StatusChange* structure. Nothing if there's nothing to report, and the relevant desired pace if the monster starts hunting or goes back to wandering.

What about the output function for sound, *playSounds*? Well, the only thing we need it to do is to change the speed of playback with OpenAL's *pitch* function when necessary - *pitch* set to two means play the source two times faster than normal.

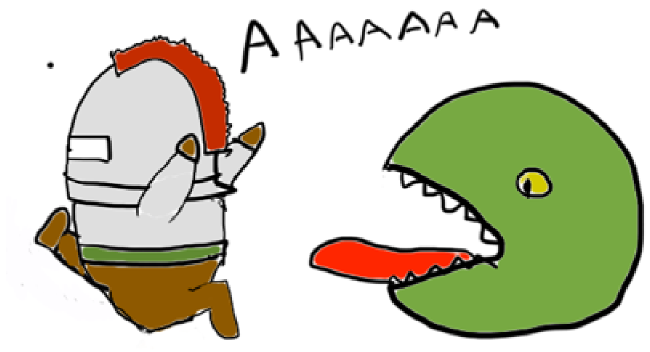
```
  playSounds ::Source -> SoundState -> IO ()
  playSounds music (SoundState (Just pace)) = pitch music $= (paceToPitch pace)
  playSounds music (SoundState Nothing)     = return ()

  paceToPitch Quiet = 1
  paceToPitch Danger = 2
```

That's all! When our monster starts hunting, the music speeds up, and we also laid the groundwork for extra sounds to come in with the *SoundState* and the *RenderState* being passed to the generalized *outputFunction*.

This could easily be reworked to play a different tune when the monster is hunting by for instance using two different sound sources.

Punctual events

What about punctual events? We can imagine a couple for our game, first and foremost the player screaming when he gets caught by the monster (note: should also have a tasteful graphical animation to be complete).



We're going to use an Elerea function we haven't used before, but is perfect for one-off events like this one: *until*. *until* is True exactly once: the first time the input signal is true. This requires a little syntactic wriggling, because predictably the name *until* conflicts with Prelude's *until* so we need to qualify it as belonging to Elerea.

```
import FRP.Elerea.Simple as Elerea

-- SoundState changes
data SoundState = SoundState (Maybe StatusChange) Bool

-- in the hunted network function
endOfGame <- Elerea.until gameOver
let renderState = RenderState <$> player <*> monster <*> gameOver <*> viewport
    soundState  = SoundState <$> statusChange <*> endOfGame
```

Since loading sounds from file is not the fastest operation, it's a good idea to load all the sounds at the start of the game (or the level). We create a *Sounds* data structure, similar to the *Textures* one created in chapter 3. For this little game a Haskell record suffices, but if we start to have many more sounds, something like a Map (Data.Map.Strict) might be advisable to keep things civilized. This *Sounds* data structure then needs to be passed in to the network function *hunted*, to *outputFunction* and ultimately to *playSounds*.

For anything to happen in the sounds department, the ALUT context needs to be initialized. The best policy is to just initialize the ALUT context just after opening the window, to have it ready in any case, as shown below.

```
data Sounds = Sounds { backgroundTune :: Source, shriek :: Source }

loadSounds :: IO Sounds
loadSounds = do
    music <- loadSound "sounds/oboe-loop.wav"
    shriek <- loadSound "sounds/shriek.wav"
    return $ Sounds music shriek

-- in main
    withWindow width height "Game-Demo" $ \win -> do
      withProgNameAndArgs runALUT $ \progName args -> do
          sounds <- loadSounds
          backgroundMusic (backgroundTune sounds)
          network <- start $ hunted win
                                     directionKey
                                     randomGenerator
                                     textures
                                     glossState
                                     sounds
```

The *playSounds* output function also changes to accomodate the new sound. It now needs to both adapt the background music and play the shriek when appropriate. A new singular *playSound* function takes care of some ALUT boilerplate:

```
import System.IO ( hPutStrLn, stderr )
import Data.List (intersperse)


playSounds :: Sounds -> SoundState -> IO ()
playSounds (Sounds music shriek) (SoundState mbPace endOfGame hunting) = do
  changeBackgroundMusic music mbPace
  when endOfGame $ playSound shriek

changeBackgroundMusic source (Just pace) = pitch source $= (paceToPitch pace)
changeBackgroundMusic source Nothing     = return ()

playSound source = do
  play [source]
```

```
    -- Normally nothing should go wrong above, but one never knows...
    errs <- get alErrors
    unless (null errs) $ do
        hPutStrLn stderr (concat (intersperse "," [ d | ALError _ d <- errs ]))
    return ()
```

When the monster catches the little knight, a blood-curdling shriek can be heard.

Variable-length sound events

To add to the tension, we'll make the monster make appropriately hungry sounds when it's hunting.

```
data Sounds = Sounds { backgroundTune :: Source
                     , shriek :: Source
                     , bite :: Source }

-- in main: initialize ALUT here
withWindow width height "Game-Demo" $ \win -> do
  withProgNameAndArgs runALUT $ \progName args -> do
    sounds <- loadSounds

loadSounds :: IO Sounds
loadSounds = do
    music <- loadSound "sounds/oboe-loop.wav"
    shriek <- loadSound "sounds/shriek.wav"
    bite <- loadSound "sounds/bite.wav"
    sourceGain bite $= 0.5
    return $ Sounds music shriek bite
```

Just after loading we use the *sourceGain* function from OpenAL, which basically lowers the volume of the bite somewhat, since the sample is a bit loud compared to the rest of the soundscape.

Functional meets non-functional

The OpenAL code stands out like a sore tooth in our Haskell code: there's lots of state under the covers, and it's clearly a departure from purely functional code (the async action on *play*, for instance, or getting and setting state variables on sound sources).

The same thing happens when working with OpenGL bindings directly instead of Gloss - lots of imperative code. Gloss shields us from the gigantic state machine that is OpenGL by giving us a nice, composable functional API. Maybe the world needs a Gloss for OpenAL.

We need a new function, since we're playing a sound continuously as long as the monster hunts. Let's conveniently call it *playContinuousSound*. We can't just set OpenAL's setting to *Looping* anymore, since it needs to stop when we need it to. We can either change the loopingMode according to the given variable, or we can play the sound every time we need it to.

```
playContinuousSound source = do
        state <- get (sourceState source)
        unless (state == Playing) $ play [source]
```

Now to convey the right signal (monster status = Hunting) to the *playSounds* output function. The monster signal contains the information we need. Wait … we also need the gameover signal to get the signal to stop the biting, as well! So a composed signal, *hunting* is in order.

```
data SoundState = SoundState (Maybe StatusChange) Bool Bool

-- in hunted network function:
let hunting = stillHunting <$> monster <*> gameOver
    renderState = RenderState <$> player <*> monster <*> gameOver <*> viewport
    soundState  = SoundState <$> statusChange <*> endOfGame <*> hunting

stillHunting _                      True  = False
stillHunting (Monster _ (Hunting _)) False = True
stillHunting _                      False = False

playSounds :: Sounds -> SoundState -> IO ()
playSounds (Sounds music shriek bite) (SoundState mbPace endOfGame hunting) = \
do
    changeBackgroundMusic music mbPace
    when endOfGame $ playSound shriek
    when hunting $ playContinuousSound bite
```

If the repeating sample is a little long, and we want it to stop on cue when it's no longer needed, we can substitute the when by an if .. else and use OpenAL's *stop* function (stop [bite] in our case).

Advanced Sound

We've used some nice features of OpenAL: *loopingMode* which indicates whether a sample should be looped, *pitch*, which counter-intuitively manages playback speed, *sourceGain* which manages volume. They are the tip of the iceberg.

OpenAL offers features to situate sound in 3D - if the monster moves from the left of the screen to the right, and you have stereo audio, you could have the sound move from your left speaker to the right. *sourcePosition* and *sourceVelocity* can be set for any sound (using a *Vector3* type for 3D coordinates). *sourceRelative* indicates whether the position and velocity are relative to a listener, or whether they are absolute.

We can also get or set the offset of the sample in seconds with *secOffset*. We can play several samples in a queue (*play* takes an array of sources) and jump to a *sampleOffset* depending on that. Then there's self-explanatory *pause* and *rewind*. And there's probably lots of other features I'm missing.

Plenty of room to play!

Next: Miscellanea

We now have a good basic toolset to start developing a game. There's still lots of things to implement to have a game you'd even consider showing to a friend: a start screen and ways to restart the game when it ends, levels of difficulty, HUD, some sort of character progression (scoring being the simplest), window size handling.

We'll touch on those in the next chapter, as well as mention the elephant in the room: testing.

Following Creative Commons licenced samples from freesound were used in the code: oboe loop by Thirsk[23], shriek by jorickhoofd[24], bite by dan2008ds[25]

[23]https://www.freesound.org/people/Thirsk/sounds/121035/
[24]https://www.freesound.org/people/dan2008ds/sounds/175169/
[25]https://www.freesound.org/people/dan2008ds/sounds/175169/

Chapter 5: Making it into a real game

Many topics left untouched in the last few chapters - the next few chapters will do some mopping up and get your game more production-ready.

In this chapter we'll tackle growing the game:

- directory structure: time to get some structure in.
- the knight fights back: shooting bolts at the monster.
- having a start screen, levels and scoring.
- window size management: bigger window would be nice, sometimes.

You'll find the code for this chapter on github in the same repository, under https://github.com/elisehuard/game-in-haskell/tree/master/src/Hunted

Directory structure

Divide and conquer

Our code is growing, and will keep doing so as we add more features. It's time to do some housekeeping.

The sensible thing is to split up the code in such a way that the contact area is minimal, and that changing small details will usually be restricted to the minimum number of files. In Haskell we can measure the first (minimal contact area) by using explicit exports - by indicating which functions or constructors we expose to the external worlds - and the number of internal imports. If our split is right, there are not too many exports or imports. The second one, checking which files will be affected by new features or bugs, is much harder, and is often obvious only *after* having actually worked on the restructured code base.

For our code the best split is by the part of game it handles: Game.hs for the game mechanics and FRP network, Sound.hs for the sound, Graphics.hs for all the rendering, Backend.hs for the GLFW-b stuff, and a Main.hs to bind them all. If necessary, either of those can have sub-files again. It's unavoidable to have a separate GameTypes.hs file for the state that is being conveyed back and forth, to avoid circular dependencies. We make a namespace, Hunted, to put them under.

```
|-- Hunted
|    |-- Backend.hs
|    |-- Game.hs
|    |-- GameTypes.hs
|    |-- Graphics.hs
|    |-- Main.hs
|    `-- Sound.hs
```

It’s also the right time to make sure the dependencies to external libraries are exclusively where they belong: Graphical frameworks, sound, etc should as much as possible only be referred to in the relevant files. This makes it easier to have an overview of what happens where, and also to swap out the relevant library if the need arises.

Reminder: all new modules have to be declared in the cabal file under *other-modules*, and the module namespace should correspond to the directory structure.

Refactoring

It’s a good time to look at the code and see if some aspects could be teased out because they’re used all over the place. One obvious one is the operations around our Pos type, which is effectively a geometric vector:

```haskell
type Pos = (Float, Float)
```

The distance between two points is also one we can move to this file, since it’s fairly general purpose.

Moving something in a certain direction means adding a vector - wait, you said adding? what about a monoid for this? A tuple has a monoid instance defined in Data.Monoid of base:

```haskell
  instance (Monoid a, Monoid b) => Monoid (a,b) where
          mempty = (mempty, mempty)
          (a1,b1) `mappend` (a2,b2) =
                  (a1 `mappend` a2, b1 `mappend` b2)
```

But then we hit the fact that Float, as a Num, has a couple of possible Monoids attached to it, so we need to use the right one (Sum).

```haskell
plus :: Pos -> Pos -> Pos
plus (a,b) (c,d) = getTupleSum $ (Sum a, Sum b) <> (Sum c, Sum d)
                   where getTupleSum (x, y) = (getSum x, getSum y)
infixl 6 `plus`
```

Scalar multiplication is simpler.

```
times :: Float -> Pos -> Pos
times a (x,y) = (a*x, a*y)
infixl 7 `times`
```

The *infixr* keywords sets the fixity (associativity) of the operator, with an Integer from 0 to 9 for the precedence. To simplify things, I checked the fixity of the '*' and '+' operators (:info in the repl), and replicated their fixity. It matches usual conventions to have multiplications have a higher precedence than addition.

This means that we now can define the movements using operators on the Pos "vectors":

```
stepInCurrentDirection direction (xpos, ypos) speed = speed `times` (stepInDirec\
tion direction) `plus` (xpos, ypos)

stepInDirection WalkLeft  = (-1, 0)
stepInDirection WalkRight = (1, 0)
stepInDirection WalkUp    = (0, 1)
stepInDirection WalkDown  = (0, -1)
```

We'll add all these nice utility functions to src/Hunted/GameTypes.hs.

The knight fights back

The game will only amuse for a minute or so at the moment, because there's not much you can do, except run away as fast as you can. We need to give the knight a fighting chance.

Let's give the knight a crossbow, so he can try and shoot the monster. The arrows only have limited reach, otherwise it would be too easy. The knight has a limited supply of bolts, but he can try and gather more.

The monster now has a limited health, which should also be visible in the game. The usual visualization is a health bar floating somewhere in the vicinity of said monster.

New keyboard keys passed in

Unavoidably, we need some more controls to control the shooting. We use the a, d, w, s keys for left, right, up, down respectively (assuming a QWERTY keyboard) - this kind of thing should be settings, which requires another layer of indirection, but we'll go with this set for now.

Like for the direction keys in Chapter 2, we can define a sink for the shoot keys, and make sure this signal is then used in the FRP network function. The *readInput* function also needs to handle the new keys, capturing them with GLFW's function and feeding them into the sink. Let's also sprinkle a little bit of syntactic sugar to have a couple of one-liners instead of 5 lines per sink. In *src/Hunted/Backend.hs* :

```
readInput :: Window
          -> ((Bool, Bool, Bool, Bool) -> IO ())
          -> ((Bool, Bool, Bool, Bool) -> IO ())
          -> IO ()
readInput window directionKeySink shootKeySink = do
    pollEvents
    directionKeySink =<< (,,,) <$> keyIsPressed window Key'Left
                               <*> keyIsPressed window Key'Right
                               <*> keyIsPressed window Key'Up
                               <*> keyIsPressed window Key'Down
    shootKeySink =<< (,,,) <$> keyIsPressed window Key'A
                           <*> keyIsPressed window Key'D
                           <*> keyIsPressed window Key'W
                           <*> keyIsPressed window Key'S
```

There is one additional gotcha: since the signal is polled 60 times per second, we probably need to fix the use of our signal, since we want one bolt per press, not 60 times per second times the length of time our finger presses the relevant keys. Let's throttle our signal, by comparing every shoot signal with its previous value, and only using it when it transitions from False to True.

```
-- in network function
shoot <- edgify shootKey

edgify :: Signal (Bool, Bool, Bool, Bool)
       -> SignalGen (Signal (Bool, Bool, Bool, Bool))
edgify s = do
  s' <- delay (False, False, False, False) s
  return $ s' >>= \x -> throttle x s

throttle :: (Bool, Bool, Bool, Bool)
         -> Signal (Bool, Bool, Bool, Bool)
         -> Signal (Bool, Bool, Bool, Bool)
throttle shot sig
   | hasAny shot = return (False, False, False, False)
   | otherwise = sig

hasAny :: (Bool, Bool, Bool, Bool) -> Bool
hasAny (l, r, u, d) = l || r || u || d
```

OK, now we have the input data to start creating bolts as required.

FRP: Signal of array of bolts

Our network is starting to be more dynamic: we have new entities (crossbow bolts) getting created and removed to and from our state as a matter of course. This kind of feature is essential for any more advanced games, where for instance you will need to manage a varying number of monsters, or rolling spiky balls, or …

With our flavour of FRP, there are two ways to handle this:

- having a signal on an array of crossbow bolts: we have our transfer function adding and pruning to an array as required by the shoot keys.
- having an array of signals, with every signal representing a crossbow bolt, and they each manage their own state.

Both approaches work, so it's really a matter of choice. The first approach is maybe easier for small data structures like our crossbow bolts, the second approach makes more sense when working with more complex entities (say spawning more monsters). We'll start with the first approach, which we're more familiar with, and show the other approach in a later paragraph.

```
type Range = Int
data Bolt = Bolt Pos Direction Range Bool
              deriving Show
```

The Pos and Direction parameters speak for themselves. We add a Range because it adds a bit of spice to force the player to approach the monster to effectively shoot it - we don't want it to be too easy after all. The last bool expresses whether the bolt has hit the monster or not, which means the bolt can now cease to exist.

If we work with an array of bolts, we'll need a "bolts" signal, depending on the shoot signal we generated, as well as the player (for the originating position) and the monster (the bolt stops when it hits the monster). This is fairly similar with what we've been doing so far.

```
bolts <- transfer3 [] (manageBolts worldDimensions) shoot player monster
...
manageBolts worldDimensions shoot player monster bolts =
    map moveBolt $ filter (boltIsAlive fdimensions monster)
                 $ addNew bolts
    where addNew bolts =
      if hasAny shoot
        then (Bolt (position player) (dirFrom shoot) boltRange False):bolts
        else bolts

boltIsAlive worldDimensions monster bolt =
  (not (hasHit monster bolt)) && boltStillGoing worldDimensions bolt

hasHit (Monster (xmon, ymon) _ _) (Bolt (x, y) dir range hit)
  | dist (xmon, ymon) (x, y) < ((monsterSize/2)^2) = True
  | otherwise = False
```

Managing involves:

- moving the already existing bolts one step further as required, and decrementing the range.
- boltIsAlive: indicating whether the bolt has not hit the monster (hitMonster) and whether the bolt is still going based on range and world dimensions.
- add new bolts depending on the shoot value.

This is all done by manipulating the bolts array contained in the signal.

That's not all however: we also need to reflect the effect of said bolts. Once again, we need looping signals, so we need to feed the delayed signal bolts' back into the monster signal. The monster also gets modified to have a Health argument, since we have to record how often it gets hit!

In *src/Hunted/GameTypes.hs*:

```
type Health = Float
data Monster = Monster Pos MonsterStatus Health
               deriving Show
```

In *src/Hunted/Game.hs*:

```
monster <-
  transfer4 initialMonster (wanderOrHunt worldDimensions) player
                                                          randomNumber
                                                          gameOver'
                                                          bolts'

...

-- when monster dies, all movement stops
wanderOrHunt _ _ _ _    _ monster@(Monster _ _ 0) = monster

-- register hits immediately in the health of the monster
wanderOrHunt dimensions player (r, _) Nothing bolts monster = do
    let monsterHit = hitOrMiss bolts monster
    if close player monsterHit
```

```
     then hunt player monsterHit
     else wander r monsterHit dimensions

-- hit or miss: depending on whether a bolt is on the monster
hitOrMiss bolts monster@(Monster (xmon, ymon) status health) =
    Monster (xmon, ymon) status (health - (hits monster bolts))
    where hits monster bolts =
      fromIntegral $ length
                   $ filter (<= (monsterSize/2)^2) (boltDistances monster bolts)
      boltDistances (Monster (xmon, ymon) _ _) bolts =
        map (\(Bolt (xbolt, ybolt) _ _ _) -> dist (xmon, ymon) (xbolt, ybolt)) b\
olts
```

Note that in *hitOrMiss* we need to be slightly more permissive than in the pruning of the bolts: we use <= instead of < to decide whether the bolt has hit the monster, since we want the hit to be registered before the bolt disappears.

But wait, that's not the only thing! The game has to end when the monster dies, but it's definitely a different ending than if the knight gets gobbled up. Our *gameover* signal type needs a different definition. The type of the signal value becomes *Maybe Ending* instead of a straight *Bool*.

Data structure:

```
data Ending = Win | Lose
              deriving (Show, Eq)
```

Define what constitutes winning and losing in the game network:

```
gameOver <- memo (gameEnds <$> player <*> monster)

where playerEaten player monster
            | distance player monster < (playerSize^2  :: Float) = Just Lose
            | otherwise                                          = Nothing
        monsterDies (Monster _ _ health)
            | health == 0 = Just Win
            | otherwise   = Nothing
        gameEnds player monster = maybe (monsterDies monster) Just (playerEaten \
player monster)
```

We now have a slightly more complex definition for ending the game. We could also define a win as something different than just killing monsters: carry the drunk grizzly bear to his cave before the monster gets either of you, or find all the treasures, ...

The *RenderState* structure changes accordingly, so that we can render both the flying bolts and the appropriate ending.

```
data RenderState = RenderState Player Monster (Maybe Ending) ViewPort [Bolt]
```

We need to render the bolts, the ending, and let's not forget the monster's health status, floating above its head. Again, thanks to Gloss's nice functional API, we can map over the bolts to render the bolts, and compose with gameOngoing to present the right characters.

Note that this time I replaced the "Game Over" text by a texture, to have nice fonts - as long as Gloss doesn't have better font support that's an acceptable compromise for static texts.

When the monster dies, we can show this by using the appropriate texture.

```
renderFrame window
            glossState
            textures
            (worldWidth, worldHeight)
            (RenderState player
                         monster
                         gameOver
                         viewport
                         bolts
                         lives
                         score
                         dimensions) = do
   displayPicture dimensions black glossState (viewPortScale viewport) $
        Pictures $ gameOngoing gameOver lives (texts textures)
                 $ gameStats lives score dimensions $
                  [ uncurry translate (viewPortTranslate viewport) $
                      tiledBackground (background textures) worldWidth worldHeight
                  , Pictures $ map (uncurry translate (viewPortTranslate viewport) .
                                    (renderBolt (boltTextures textures))) bolts
                  , renderPlayer player (playerTextures textures)
                  , uncurry translate (viewPortTranslate viewport) $
                      renderMonster (monsterWalking textures)
                                    (monsterHunting textures)
                                    (deadMonster textures)
                                    monster
                  , uncurry translate (viewPortTranslate viewport) $
                      renderHealthBar monster ]

renderBolt :: TextureSet -> Bolt -> Picture
renderBolt textureSet (Bolt (xpos, ypos) facing _ _) =
  translate xpos ypos $ directionTexture facing textureSet
```

```
gameOngoing :: Maybe Ending
            -> Int
            -> Map.Map String Picture
            -> [Picture]
            -> [Picture]
gameOngoing (Just Lose) 1 textTextures pics =
  pics ++ [translate (-50) 0 $ (textTextures Map.! "game-over")]
gameOngoing (Just Win)  _ _            pics =
  pics ++ [Color black $ translate (-100) 0 $ Scale 0.3 0.3 $ Text "You win!"]
gameOngoing Nothing     _ _            pics =  pics

-- healthbar above the monster's head
renderHealthBar (Monster (xmon, ymon) _ health) =
  Pictures
    [ translate xmon (ymon + 30) $
        Color black $
        rectangleSolid healthBarLength healthBarWidth
    , translate
         (xmon - healthBarLength/2 + health*healthBarLength/(numberOfLives*2))
         (ymon + 30) $
        Color red $
        rectangleSolid (health*healthBarLength/numberOfLives) healthBarWidth ]
```

FRP: array of Signal Bolt

Hold on to your suspenders, because this is going to be a slightly more challenging paragraph. Say we now want to keep track of the bolts as signals in their own right, instead of just having a Signal of Bolts as we did before …

Why would we do that? Two reasons I can think of:

- conceptually: we want a description of what the bolts do by themselves, having their own signals
- we could think of situations where the entities have complex interactions with the outside world, but also amongst themselves (for instance little monsters instead of bolts). It becomes more convenient to express those signals as their own little network of dependencies.

The tricky thing is that we still need to output, or have access to a *[Bolt]* array structure to feed into the monster signal, and also to render it, so we need to do some hackery to maintain both *[Signal Bolt]* and *Signal [Bolt]*.

We don't need to change that much in our original program. We'll have a different approach, in that the new bolts are generated with a Signal Generator (type *SignalGen*). In our *hunted* network definition function:

```
-- takes (bool, bool, bool, bool) and player, output SignalGen [Signal Bolt]
let bolt direction range position =
      stateful (Bolt position direction range False) moveBolt
    mkShot shot player =
      if hasAny shot
        then (:[]) <$> bolt (dirFrom shot) boltRange (position player)
             else return []
```

The *mkShot* function will use the shoot and the player values and generate a new signal. The *bolt* function uses *stateful*, because bolts' signals are inherently self-contained stateful signals (using the *moveBolt* function, just like in our previous code). They go straight ahead until they cease to exist, nothing will make them change their course as such. The only possible event is that they cease to exist, and that's managed by working on the array of signals, not from within the signal itself, as we'll see below. This would obviously change if we wanted a more complex interaction, in which case we'd use a transfer function instead.

```
newBolts <- generator (mkShot <$> shoot <*> player)
bolts <- collection newBolts (boltIsAlive worldDimensions <$> monster)
bolts' <- delay [] bolts
```

newBolts is the bolt source signal generator, which uses *mkShot*. It is of type *Signal [Signal Bolt]*: mkShot is applied (<$>) to a Signal, which returns *Signal (SignalGen [Signal Bolt])*. The generator function inverts this into a signal generator of signals *SignalGen (Signal [Signal Bolt])*, which is then executed to return *Signal [Signal Bolt]*. It is a signal of new bolt signals.

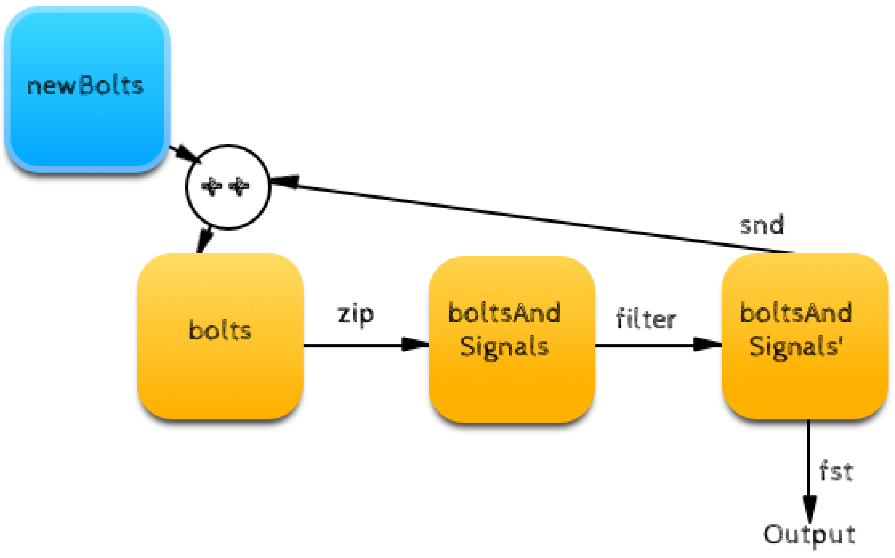
The actual bolts signal will use both the defined *newBolts* source, and the *boltIsAlive* function which we already used in our previous code as well to determine whether a bolt is still going. Since we apply it to the *monster* signal, what we pass in the *collection* function is curried and of type *Signal (Bolt -> Bool)*.

collection is where it gets a little more complicated. We need to maintain the signal of running bolts *Signal [Signal Bolt]* (as mentioned previously) to let the bolt live its life, but we also need a useable *Signal [Bolt]* to have actual bolts to use in the rest of the program. To do this, we'll use an internal tuple type, keeping track of both. We use *fst* to manage the output (the signal of bolts), and *snd* to continue grooming our internal signals (the bolts living their life).

The source we provide (of type *Signal [Signal Bolt]*) provides the new bolts over time, and the *isAlive* signal will allow the filtering of the signals when necessary.

```
collection :: Signal [Signal Bolt]
           -> Signal (Bolt -> Bool)
           -> SignalGen (Signal [Bolt])
collection source isAlive = mdo
  boltSignals <- delay [] (map snd <$> boltsAndSignals')
  -- add new bolt signals
  bolts <- memo (liftA2 (++) source boltSignals)
  -- boltsAndSignals type: SignalGen (Signal [Bolt], [Signal Bolt])
  let boltsAndSignals = zip <$> (sequence =<< bolts) <*> bolts
  -- filter out
  boltsAndSignals' <- memo (filter <$> ((.fst) <$> isAlive) <*> boltsAndSignals)
  return $ map fst <$> boltsAndSignals'
```

In short, what we have here, is a little enclosed network of signals just for the bolts (slightly simplified graph below, I find images help).



bolts network

We still output a SignalGen (Signal [Bolt]) which is usable by other parts of the game, so it functions just like the previous solution - but we have described every single bolt as a stand-alone signal.

Twang!

Let's also make sure we have appropriate sounds for firing bolts, hitting the monster, and the monster dying! This paragraphs builds on what we saw in the previous chapter about how to use sound in the game.

```
data Sounds = Sounds { backgroundTune :: Source
                     , shriek :: Source
                     , bite :: Source
                     , groan :: Source
                     , twang :: Source
                     , thump :: Source }

loadSounds :: IO Sounds
loadSounds = do
    bite <- loadSound "sounds/bite.wav"
    sourceGain bite $= 0.5
    Sounds <$> loadSound "sounds/oboe-loop.wav"
           <*> loadSound "sounds/shriek.wav"
           <*> return bite
           <*> loadSound "sounds/groan.wav"
           <*> loadSound "sounds/twang.wav"
           <*> loadSound "sounds/thump.wav"
```

I found a convincing “twang” for the crossbow shooting (interestingly, they don’t actually twang, it’s more like “toc” - but we deal here in fantasy sounds, what the player imagines a crossbow might sound like, so twang it is). Hitting the monster should result in a muffled “thump”, and the monster groans when it dies. We obviously also need to change our *SoundState* type to convey this information.

```
data SoundState = SoundState { mood :: (Maybe StatusChange)
                             , playerScreams :: Bool
                             , hunting :: Bool
                             , monsterDies :: Bool
                             , shoot :: Bool
                             , hit :: Bool }
```

And the sound state is calculated like so in the network function *hunted*:

```
let soundState = SoundState <$> statusChange
                            <*> playerScreams
                            <*> hunting
                            <*> monsterScreams
                            <*> (hasAny <$> shoot)
                            <*> (boltHit <$> monster <*> bolts)
```

Which is then used in the *playSounds* function (which is one giant side-effect, obviously, we’ve left he purely functional part of the program there).

```
playSounds :: Sounds -> SoundState -> IO ()
playSounds sounds soundState = do
  changeBackgroundMusic (backgroundTune sounds) (mood soundState)
  when (playerScreams soundState) $ playSound (shriek sounds)
  when (monsterDies soundState) $ playSound (groan sounds)
  when (shoot soundState) $ playSound (twang sounds)
  when (hit soundState) $ playSound (thump sounds)
  if (hunting soundState) then playContinuousSound (bite sounds)
                          else stop [bite sounds]
```

A full game: start screen, levels, scoring

Our game is sadly lacking in features we traditionally expect from casual games: a start screen, levels, score, and the ability to start again.

To enable this, we'll need to have a network of signals on top of our network, signals keeping track of the stage in the game.

Extra tiers to our signal network

Let's start from the bottom: our network function describing the monster, player, bolts signal now becomes the network for one single level.

Let's first group our *RenderState* and *SoundState* signal under one type, *GameState* - which will make the following part clearer.

```
data GameState = GameState RenderState SoundState
```

This groups all the state information in one type. This doesn't require a lot of change in our output function

```
outputFunction window
               glossState
               textures
               dimensions
               sounds
               (GameState renderState soundState) =
  (renderFrame window
               glossState
               textures
               dimensions
               (worldWidth, worldHeight)
               renderState) >>
  (playSounds sounds soundState)
```

First off, we need to keep track of the game state: are we in the start screen, or are we playing?

```
data GameStatus = Start | InGame
```

The upper level of our game now becomes the following simple network:

```
hunted window
       dim
       directionKey
       shootKey
       randomGenerator textures glossState sounds = mdo
  let mkGame = playGame directionKey shootKey randomGenerator
  (gameState, gameTrigger) <- switcher $ mkGame <$> gameStatus'
  gameStatus <- transfer Start gameProgress gameTrigger
  gameStatus' <- delay Start gameStatus
  return $ outputFunction win glossState textures dim sounds <$> gameState
  where gameProgress False s      = s
        gameProgress True  Start  = InGame
        gameProgress True  InGame = Start
```

Which manages whether we're in a game, or on the start screen. The change from one to the other is managed using the gameTrigger Signal. playGame returns the type *SignalGen (Signal GameState, Signal Bool)* - so when applied to *Signal GameStatus* we get a signal of generators using the game status as a parameter.

The secret sauce here is in the *switcher* function. The *switcher* function takes a *signal of generators* - imagine a timeline of generator functions - and when the boolean signal generated by the given generator becomes true, we poll the next generator signal to continue. This effectively means that we switch from one generator to the next, based on the feedback boolean signal (I know this is not easy, bear with me).

```
switcher :: Signal (SignalGen (Signal GameState, Signal Bool))
         -> SignalGen (Signal GameState, Signal Bool)
switcher gen = mdo
  trigger <- memo (snd =<< gameSignal)
  trigger' <- delay True trigger
  maybeSignal <- generator (toMaybe <$> trigger' <*> gen)
  gameSignal <- transfer undefined store maybeSignal
  return (fst =<< gameSignal, trigger)
  where store (Just x) _ = x
        store Nothing x = x
        toMaybe bool x = if bool then Just <$> x else pure Nothing
```

So what happens here? It may take a few reads to get an understanding. Let's do a line by line explanation.

```
  trigger <- memo (snd =<< gameSignal)
```

The definition of trigger is using the looped signal *gameSignal* which is defined a little lower in the function (*mdo* in action, as described in chapter 2), or rather its second component, which happens to be of type *Signal Bool*. The next line defines a delayed Signal, which can be used right away, since the first value, *True*, is already ligned up, regardless of what follows.

```
  maybeSignal <- generator (toMaybe <$> trigger' <*> levelGen)
```

Here we start using the signal of generators, *gen*, together with the *trigger'* signal. If the value of *trigger'* is True (and it is the first time round, as we just saw), then *toMaybe* returns *Just <$>* the value of the signal generator. In other words, something of the type *Signal (SignalGen (Maybe (Signal GameState, Signal Bool)))*. Generator makes a generator of this, swapping around a signal of signal generators to a generator of signals *SignalGen (Signal (Maybe (Signal GameState, Signal Bool)))*, and executing that (<-) returns *Signal (Maybe (Signal GameState, Signal Bool))*.

```
  gameSignal <- transfer undefined store maybeSignal
```

This is the crucial line of this function: *store* will return the value of the Maybe if there is any, or otherwise it will continue on the current signals. If we remember from the line before, the Maybe value will contain something if the trigger was True. This means that we use the signal of generators to move one generator down the line *if* the trigger is fired.

```
  return (fst =<< gameSignal, trigger)
```

The returned value contains the first component of the gameSignal, which is (Signal GameState), and the trigger value, which was defined earlier as *snd =<< gameSignal*. This is what we ened to render the output function.

Maybe things will become clearer when we look at the playGame, which is the generator fed into the switcher function.

The start screen:

```
playGame _ shootKey _ Start = mdo
    let startGame = sIsPressed <$> shootKey
    return (pure (GameState StartRenderState StartSoundState), startGame)
    where sIsPressed (_,_,_,s) = s
```

The start screen returns a fixed start screen state, and the trigger is based on whether the s key is pressed "Press

```
playGame directionKey shootKey randomGenerator InGame = mdo
  (gameState, levelTrigger) <- switcher $ playLevel directionKey
                                                    shootKey
                                                    randomGenerator
                                                    <$> levelCount'
                                                    <*> score'
                                                    <*> lives'
  levelCount <- transfer2 initialLevel statusProgression gameState levelTrigger
  levelCount' <- delay initialLevel levelCount
  lives <- transfer initialLives decrementLives gameState
  lives' <- delay initialLives lives
  score <- memo (stateScore <$> gameState)
  score' <- delay 0 score
  let gameOver = isGameOver <$> gameState
  return (gameState, gameOver)
  where isGameOver (GameState (RenderState {renderState_lives = l}) _) = l == 0
        isGameOver (GameState StartRenderState _) = False
        stateScore (GameState (RenderState {renderState_score = s}) _) = s
        stateScore (GameState StartRenderState _) = 0
        decrementLives (GameState (RenderState {renderState_ending = Just Lose})\
 _)
                      l = l - 1
        decrementLives (GameState _ _) l = l
```

We have several signals here, which manage the flow of one game. The level count, of type LevelStatus, the lives and the score.

```
data LevelStatus = Level Int
                   deriving Show
```

Lives are managed in this network at the end of a level, and depend on the value of the *Ending* returned in the render state: if we lose, then lives are decremented. Scores have a different deal: the current score is passed in to the *playLevel* generator, and extracted back out of the *gameSignal*: the score is calculated within the level itself.

And last but not least: another ‘switched’ function to manage the generators for different levels!

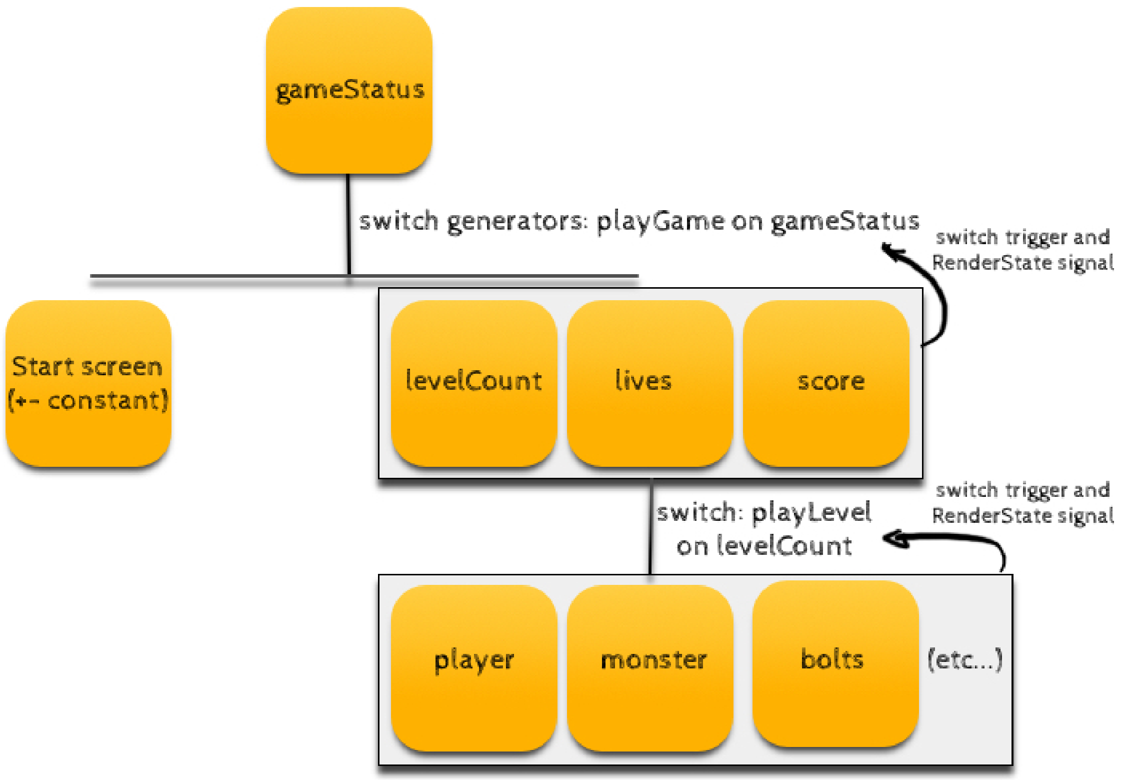
What would the trigger be here? Why, anything that actually ends the game, as expressed by the *isGameOver* function: if the number of lives is 0, then the game ends and we go back to the start screen.

Let’s go one more level down the rabbit hole, to the *playLevel* function, which looks much more familiar. It’s basically the *hunted* function we know from previous chapters. The parameters are different: we don’t need to pass in anything to do with the output (window, textures, sounds) since that's managed at the actual output level (the hunted function gets passed the state and calls the *outputFunction*) - and we add parameters for the level, the lives and the score, which in turn will be needed.

```
playLevel directionKey
          shootKey
          randomGenerator
          (Level _)
          currentScore
          lives = mdo
    let worldDimensions = (worldWidth, worldHeight)
    player <- transfer2 initialPlayer
                        (movePlayer 10 worldDimensions)
                        directionKey
                        levelOver'
    randomNumber <- stateful (undefined, randomGenerator) nextRandom
    ...
    score <- transfer currentScore accumulateScore hits
    levelOver <- memo (levelEnds <$> player <*> monster)
    levelOver' <- delay Nothing levelOver
    viewport <- transfer initialViewport viewPortMove player
    ...
    return (GameState <$> renderState <*> soundState, isJust <$> levelOver)
```

Our previous *gameOver* becomes *levelOver* - and we return the game signal and the trigger, which checks whether the level is over in any way.

Note that the score is being incremented at this level, while the lives are just used for calculation purposes, and the actual decrementation of lives happens in *playGame* where we return when we *Just Lose*.


gameStatus
switch generators: playGame on gameStatus
switch trigger and RenderState signal
Start screen (+- constant)
levelCount
lives
score
switch: playLevel on levelCount
switch trigger and RenderState signal
player
monster
bolts
(etc...)

levels

Not completely playable yet

There's (at least) one big flaw in the above structure: the switching happens instantaneously. The level ends, boom! we switch to the next one. The game ends and suddenly we're on the start screen, no transition.

We need to slow things down a bit and let the user savour their victory or wallow in despair for at least 5 seconds.

Simplest possible solution: use a very delayed version (51 times in this case) of our levelOver signal and use that as a trigger:

```
delayedLevelOver <-
  (foldr (>=>) (delay Nothing) (replicate 50 (delay Nothing))) levelOver
...
return (GameState <$> renderState <*> soundState, isJust <$> delayedLevelOver)
```

This is already better, as we see the messages displayed, at least, before going on.

Usually, you want something better though, like an animation to express how the level ended. Instead of delayedLevelOver, we can go with an animation signal of type *Maybe Animation*.

The new type in *src/Hunted/GameTypes.hs*:

```
data Animation = DeathAnimation Float | NextLevelAnimation LevelStatus Float
                 deriving Show
```

The signal itself basically represents a countdown of cycles to execute the animation - the animation stops when the number reaches 0. To be fair, at this point I'm just using Float by default because it's more convenient to use with Gloss, but an Integral type would be more appropriate. It needs to be fed into the *RenderState*, which acquires a new record member *renderState_animation :: Maybe Animation*.

```
animation <- transfer Nothing (endAnimation level) levelOver
...
return (GameState <$> renderState <*> soundState, animationEnd <$> animation)

-- transfer function for animation
endAnimation :: LevelStatus -> Maybe Ending -> Maybe Animation -> Maybe Animation
-- animation is finished: static
endAnimation _ _ (Just (DeathAnimation 0)) = Just (DeathAnimation 0)
endAnimation _ _ (Just (NextLevelAnimation l 0)) = Just (NextLevelAnimation l 0)
-- animation is ongoing (n -> n-1)
endAnimation _ _ (Just (DeathAnimation n))     = Just (DeathAnimation (n - 1))
endAnimation _ _ (Just (NextLevelAnimation l n)) = Just (NextLevelAnimation l (n\
 - 1))
-- start animation
endAnimation _ (Just Lose) _ = Just $ DeathAnimation 50
endAnimation (Level n) (Just Win)_ = Just $ NextLevelAnimation (Level (n+1)) 50
-- no animation
endAnimation _ _ Nothing = Nothing

-- we can leave level at animation's end
animationEnd (Just (DeathAnimation 0)) = True
animationEnd (Just (NextLevelAnimation _ 0)) = True
animationEnd _ = False
```

The level trigger is no longer *isJust levelOver*, but becomes based on whether the animation actually has ended (the countdown is down to 0).

Then this can be used when drawing the end of the level (say just indicating the transition to the next level by doing a fadeout - fadein):

```
renderFrame window
            glossState
            textures
            dimensions
            (worldWidth, worldHeight)
            (RenderState (Player _ playerDir)
                         monster
                         gameOver
                         viewport
                         bolts
                         lives
                         score
                         mbAnimation) = do
   displayPicture dimensions black glossState (viewPortScale viewport) $
     Pictures $ animation mbAnimation dimensions $ ...

animation (Just (NextLevelAnimation l n)) (w, h) pics =
  pics ++
    [ Color (animationColor n) $ rectangleSolid (fromIntegral w) (fromIntegral h)
    , Color white $ translate (-100) 0 $ scale 0.3 0.3 $ Text $ show l ]
    where animationColor n
            | n > 25 = makeColor 0 0 0 (0.04*(50-n))
            | otherwise = makeColor 0 0 0 (0.04*n)
```

We're using makeColor to generate a more or less transparent overlay to fade out the level.

Multiple levels

We obviously need to vary the levels as we progress! As a rule you want progression in difficulty, in character abilities and look, maybe big bosses, potentially an end goal to reach. Several possible approaches to this:

- use pattern matching and have a distinct network function for every level (*playLevel ... (Level 1) ...*)
- load level structures from file and define textures, parameters, by hand in those files, and base the *playLevel* function on the loaded level.
- make a random level generator so that every level becomes a surprise (like the *Binding of Isaac* does very well).

My understanding of the last option, a random level generator, which seems the simplest in itself, is that it takes a fair amount of fine-tuning to get it right. It might be better to start with the first option, and then move down to get a good feel for what constitutes a good progression of levels.

A simple addition to every level could be having a world specific to every level. This world could contain obstacles (trees, castles, etc) which both the monster and the player cannot pass through. This effectively means redefining the *outsideLimits* function to take the world as an argument and forbid any movement into solid structures.

Another is adding a monster in every level, and dumping them in different places.

Ideally we would have different weapons, monsters, evolution through the levels where one gains new abilities, etc. These are just extensions of what we've described so far.

An easy level variation could be to add one monster per level. This is actually not too much work, since it mostly requires the mapping (fmapping) of existing functions over an array of monsters instead of using them straight. We use the random number generator to generate their starting position. We can keep a fixed number of monsters throughout the level, keeping their corpses around for the duration of the level when they die instead of pruning them like we did with the bolts.

```
initialMonster :: (Float, Float) -> Monster
initialMonster pos = Monster pos (Wander WalkUp wanderDist) 4
...
playLevel windowSize
          directionKey
          shootKey
          randomGenerator
          level@(Level n)
          currentScore
          lives = mdo
  let worldDimensions = (worldWidth, worldHeight)
    randomWidths = take n $
      randomRs (round ((-worldWidth)/2 + monsterSize/2),
                round ((worldWidth/2) - monsterSize/2)) randomGenerator :: [Int]
    randomHeights = take n $
      randomRs (round ((-worldHeight)/2 + monsterSize/2),
                round ((worldHeight/2) - monsterSize/2)) randomGenerator :: [Int]
    monsterPositions = zip (map fromIntegral randomWidths)
                           (map fromIntegral randomHeights)
    ...
    monsters <- transfer4 (fmap initialMonster monsterPositions)
                          (monsterWanderings worldDimensions)
                          player
                          randomNumber
                          levelOver'
                          hits
...
```

Obviously *RenderState* now needs to convey that array of *Monster* instead of a single monster, and rendering similarly requires mapping over that array:

```
  uncurry translate (viewPortTranslate viewport) $
    Pictures $ map (renderMonster (monsterWalking textures)
                                  (monsterHunting textures)
                                  (deadMonster textures))
                     monsters
```

This should spice things up a little for higher level numbers. But even this simple example illustrates how random generation in a game actually needs a fair amount of thinking and tuning, as referred to earlier: the program as it is will drop the monster just anywhere, which could also be *on* the player, which would not give a very good game.

Window size management

We've worked with a fixed window size of 640*480 - we'll want to be able to change that window size at will, perhaps even have fullscreen. This is not possible without some changes.

GLFW-b provides a callback to send the window size to. Our job is to feed the window size in to a Signal sink, so that it's just another signal we can work with, not unlike keyboard keys.

```
(windowSize,windowSizeSink) <- external (width, height)
```

Our convenience *withWindow* function gets fed that sink as a parameter, so that we can use the callback (in *src/Hunted/Backend.hs*):

```
setWindowSizeCallback win $ Just $ resize windowSizeSink
...
resize :: ((Int, Int) -> IO()) -> Window -> Int -> Int -> IO()
resize windowSizeSink _ w h = windowSizeSink (w, h)
```

We now need to pass this down to the graphical output layer through the network function - the parameter *dimensions* as a constant disappears, replaced by the varying state provided by a signal.

The *RenderState* data type changes accordingly:

```
data RenderState = RenderState { renderState_player :: Player
                                 ...
                               , renderState_windowSize :: (Int, Int) }
```

The signal is passed down to various generators in the network functions.

The changes in *src/Hunted/Graphics* are surprisingly few: it turns out Gloss handles the change of size splendidly, we only need to pass the varying dimensions into the *displayPicture* function. THe main change being that the window dimensions are now passed in with the *RenderState* type instead of using a constant parameter.

```
renderFrame window
            glossState
            textures
            (worldWidth, worldHeight)
            (RenderState (Player _ playerDir)
                         ...
                         dimensions) = do
   displayPicture dimensions black glossState (viewPortScale viewport) $ ...
```

That's all it takes! You can now scale your window up and down, and the window will act as, well, a window onto our world.

Next: testing

We now have a working game, with levels, scoring, etc. At this point, I hope you have all the clues to take your game wherever you want it to go - imagination is the limit.

In the next chapter I'll address a glaring omission of this book so far: tests. Especially if the game grows any more complex, automated tests will help keep sanity - to a certain extent, with a game a lot of testing still has to happen by hand.

The chapter will also include concepts that are useful for any haskeller to know, like a little bit about QuickCheck, one of Haskell's great assets in testing.

Appendix A: Installation on Mac

Homebrew

Homebrew[26] is probably the best package manager on Mac OS X at this point in time. It is not a prerequisite for any of the code in this book, but it will help you install the native libraries that it requires. Macports is another option, but I've found it to be much less easy to work with.

The instructions to install Homebrew are on the website.

GLFW

To use GLFW-b the underlying library GLFW[27] must be installed on your system first. It is possible to use homebrew for this:

```
brew install glfw3
```

Another option is to install from source, which is more work.

FTGL

FTGL requires the underlying installation of, yes, you've guessed right, FTGL[28], a C++ library to manage fonts. Homebrew to the resque:

```
brew install ftgl
```

ALUT

ALUT on Mac OS X Yosemite doesn't work on the preinstalled ALUT (explanations I've found online mention the removal of critical headers). The solution is to install freealut:

[26] http://brew.sh/
[27] http://www.glfw.org/
[28] http://sourceforge.net/projects/ftgl/

```
brew install freealut
```

Appendix B: useful tips for Haskell development

This does not necessarily belong in the book, but I wanted to give you a few tips on how to use cabal in a practical way

Sandboxes

There is such a thing as dependency hell in Haskell: if you have several packages that use different versions of the same library, and you install all your dependencies globally, things will get messy very rapidly.

Fortunately, since Cabal version 1.18.0, we have sandboxes.

```
cabal sandbox init
cabal install --only-dependencies
```

This will make sure that all your dependencies are installed locally, under .cabal-sandbox, and there is a local configuration file, cabal.sandbox.config. They don't belong in version control, so they should be added to .gitignore (if you use git).

This will make your life much, much easier.

Warnings

Warnings are extremely useful, and the Haskell ones actually often point out involuntary errors in a program, or things that should be cleaned up, like unused parameters or imports.

How to activate the warnings: in your cabal file, add a compilation option for ghc under the relevant *library* or *executable* section. For instance:

```
executable frp-demo
  main-is:              StateFRP.hs
  ghc-options:        -Wall
  build-depends:        base >=4.7 && <4.8
                      , GLFW-b
                      , OpenGL
                      , elerea
                      , random
                      , FTGL
  hs-source-dirs:       src
  default-language:     Haskell2010
```

Linter

When you're learning Haskell, extra tips are useful - and when you don't have a friendly and experienced Haskell developer at hand to tell you what you're doing wrong, the Haskell linter can help.

A nice way to do this is to add a test suite to your project which will exclusively run the linter. This means that if the linter is not happy with your code, the tests fail!

In your cabal file:

```
test-suite hlint
  type: exitcode-stdio-1.0
  main-is: hlint.hs
  hs-source-dirs: test

  build-depends:
    base,
    hlint >= 1.7
  default-language:     Haskell2010
```

The hlint file itself, in the test directory (test/hlint.hs):

```
module Main where

import Control.Monad
import Language.Haskell.HLint
import System.Environment
import System.Exit

main :: IO ()
main = do
    args <- getArgs
    hints <- hlint $ ["src", "--cpp-define=HLINT", "--cpp-ansi"] ++ args
    unless (null hints) exitFailure
```

This assumes your source is located in the src directory (which is a personal convention) - adapt where necessary.

If you have somewhat outlandish code in your project, the linting itself can be configured. I found this trick in Edward Kmett's Lens project[29]. For most cases though, the basic linter is fine.

To activate a test suite, a cabal.config file can be used:

```
tests: True
```

This means that your *cabal build* will build the tests as well.

Note: at the time of writing, there was some trouble installing hlint because of its dependency to pandoc. A solution to this is the following pre-install:

```
cabal install hsb2hs
cabal install pandoc -fembed_data_files
```

Staying up to date

Haskell is very much a rapidly evolving ecosystem. Things to do regularly: keeping cabal up to date:

```
cabal update
```

And reinstalling cabal-install when appropriate. Cabal is nice enough to warn us when the update hasn't happened for more than 15 days. It may be worth automating.

The second thing is to try and keep up with the latest version of the Haskell language. Your local packaging system may keep up, but the best way to check is to have a look at the ghc website[30] once in a while. At the time of writing GHC 7.8.3 is the current version.

[29]https://github.com/ekmett/lens/blob/master/HLint.hs
[30]https://www.haskell.org/ghc/

An amazing resource

A great practical online post on programming in Haskell is the (long, long) blog post by Stephen Diehl What I wish I knew when I learned Haskell[31].

[31]http://dev.stephendiehl.com/hask/