Implementing *Frogger* Using Functional Programming: Undergraduate Dissertation

Author
Jack Ellis
psyje5@nottingham.ac.uk
4262333

Supervisor
Natasa Milic-Frayling
psznm@nottingham.ac.uk

Contents

0.1 Introduction

In this report I will summarise the progress made on my Undergraduate Dissertation project which focuses on re-implementing the video game *Frogger* using the Functional Programming language Haskell. Whilst this is primarily a software engineering effort, it will provide me with an opportunity to broaden and demonstrate my increased proficiency in software engineering methods, procedures, and skill. In the following sections I will first describe the origin and characteristics of *Frogger* and follow it with a discussion of relevant Haskell characteristics and capabilities.

0.1.1 *Frogger*

Frogger was first released by Konami in 1981 as an arcade machine. Since then it has become the premiere "road-crossing simulator", with over a dozen official sequels. The basic objective of the original game is to navigate a colony of frogs (the eponymous Froggers) to their homes, via a busy road and a river filled with hazards. There are so many of these hazards, in fact, that in a review of the 1982 Atari port, *Softline Magazine* stated that it had "earned the ominous distinction of being the arcade game with the most ways to die"[**?**]. Indeed, a player can die from a number of mistimed jumps, world events, and even by reaching the goal at the wrong time.

The game is highly popular, having sold in excess of 20 million copies[**?**], and there have been countless imitations created right from its inception; from 1982's *Ribbit* for the Apple II, to the iOS and Android app *Crossy Road* essentially aping the "road-crossing simulator" aspect of the original arcade game.

Indeed, such is the historical significance of the game that an entire episode of the American sitcom *Seinfeld* was dedicated to it[**?**].

0.1.2 Functional Programming and Haskell

Functional Programming (FP) has been increasingly adopted in industry of late, now to the extent that there is an entire conference dedicated to discussing its use: the Commercial Users of Functional Programming[**?**]. Currently, FP is predominantly used in the financial and research sectors, with only a couple of small game studios dedicated to using it. I am interested in extending the use of Functional Programming to games, and plan to dedicate my project to re-implementing *Frogger* using this programming paradigm.

Haskell is a functional programming language first released in 1990. By default it is a purely functional language, meaning that there are no global

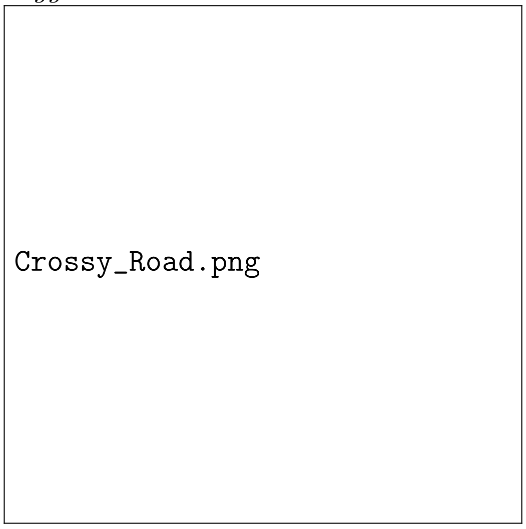
Figure 1: A screen capture of the original arcade version of *Frogger* showing the 5 road lanes, 5 river lanes, and 5 objectives (the homes).

`Frogger_Arcade.png`

variables or instructions, only functions that when applied to 0 or more arguments return a modified version of said arguments. This purely functional nature means that it is devoid of side effects, which can be a common source of error in other imperative or object-oriented (OO) languages. "Side effects" in this instance refers to any kind of effect that is not the direct manipulation of function arguments: printing to the screen, playing sound, accepting input, and so on. It is strongly typed[?], which means that all functions declared have a strict type signature, often of the form `function_name :: a -> b -> c`, where `function_name` is the name of the function, `a` and `b` are the types of the first and second argument respectively, and `c` is the type of the result, which can then be fed into other functions. This type signature is often written by the programmer, although the Glasgow Haskell Compiler (GHC) can interpret it at compile time if it is not present. It is also lazy[?], meaning that it will only calculate "as much as is needed"; this means that if a user wants the first `n` Fibonacci numbers, for instance, they can write a function that will generate all Fibonacci numbers and take the first `n` of that list. The rest of the list will simply not be generated.

Between the pure nature of the functions and the strong typing system, Haskell programs are often very easy to read and debug once a user understands the syntax and structure of a typical piece of source code. For the most part, it makes understanding the purpose and operation of functions considerably easier, with the ability to user-define types mean-

Figure 2: A screen capture of *Crossy Road*, a game which apes the basic characteristics of *Frogger*



ing that one can have a function of type `Player -> PlayerXPosition`, which would take a player "object" as an argument and return their position in X. This function's type signature would otherwise potentially be `(Float, Float, String, Float) -> Float`, which - while somewhat useful - is not as verbose as the custom types can allow. While not quite self-documenting, this approach does mean that one does not often need to refer to documentation to understand what a program does. As well as this the compiler will throw an error message detailing exactly where typine errors arise, allowing for simpler debugging.

0.2 Motivation

In my experience with the G51PGP (Programming Paradigms) and G52AFP (Advanced Functional Programming) modules, Haskell is a concise and powerful language, more so than imperative or Object-Oriented equivalents. From this experience it is my belief that if more programs can be written using the Functional Programming paradigm, they will be more extensible, maintainable, and readily understandable (primarily due to the strong typing and the fact that they can be proved correct mathematically, as opposed to being only tested, e.g. by way of some automated testing suite).

In principle, functional programming is not often used to make games. There are a number of reasons for this: performance and legacy issues being one; the sheer pervasiveness of states and effects in modern games (both concepts that Haskell and FP either have no simple implementation for or struggle with in general) being another. However, some notable individuals in video game development have commented on the potential use of FP in video games:

- John Carmack, one of the main developers of Doom and Quake, gave a talk at QuakeCon 2013 about the idea of reprogramming Wolfenstein 3d (arguably the first popular 3D first-person shooter) in Haskell[**?**]

- The founder of Epic Games, Tim Sweeney, gave a lecture on programming languages from a developer's perspective in which he describes current failures of languages and how best to overcome them. In this talk, he discusses Haskell's approach to things at length, going into some detail with regard to how well it deals with out-of-order evaluation, option types (provided by the `Maybe` Monad), references, and list comprehensions[**?**].

The motivation behind recreating *Frogger* in particular stems from the fact that *Frogger* is a deceptively simple game. At its heart it is just a "road-crossing simulator", however in practice there are many more aspects to it. The random generation of obstacles in particular is something that Haskell, while not struggling with, requires one to implement in a particular way. Likewise the number of lives the player has remaining, the idea of pausing and resuming the game, and in general the non-deterministic nature of a game's result (which is to say, that for one specific input, there are multiple possible outputs) generally require some kind of global variable. These are all concepts which, from my experience in the aforementioned university modules and my own experience programming in Haskell, the language requires further development to implement. I find this challenge to be quite appealing in the context of a final-year project.

0.3 Aims and Objectives

The aim of this project is to recreate *Frogger*, as faithfully as possible (initially), using purely functional programming.

The key objectives of this project are:

1. To establish appropriate mapping methodologies, using *Frogger* as an example.

2. To establish a set of criteria for comparisons of two *Frogger* implementations.
   This will be done in two key areas:

   - Software Architecture
     Does the Functional Programming implementation essentially imitate the exact nature of the Object-Oriented one?
   - User Experience
     Does the Functional Programming implementation correctly mimic the function of the Object-Oriented one?

3. To design and conduct a user study in which a user will compare two implementations on an experience level.

4. To demonstrate benefits of Functional Programming by either extending or refining the implementation.

0.4 Related Work

University of Nottingham Professor Henrik Nilsson has done a large amount of work with respect to using FP in a time-based setting, most notably in the Haskell library *Yampa*. In a paper he co-published, "The Yampa Arcade"[**?**], he cites video games as a key implementation of this system, detailing how it can be used to create a functional implementation of Space Invaders. Ex-UoN student Ivan Perez launched Keera Studios in 2013, a games studio focussed on creating Haskell-based video games for mobile use on the iOS and Android platforms.

Much has been done with respect to the porting of a video game to a new architecture or language. While more recent programming languages (C++, etc.) can output byte code that will work on a number of machines, it used to be the case that a game had to be reworked from the ground up for compatibility with other systems. Indeed, *Frogger* itself was the subject of a number of ports in the early 80s, with external companies publishing editions for home consoles. More recently, Michael Tonks recreated Atari's *Pac-Man* using Java as part of his undergraduate dissertation; whilst not formalising the specifications and requirements of the game itself, he chose to discuss and implement the experiential features of the game (the ghosts' pathfinding algorithms, point value of objectives, and so on)[**?**]. This is the approach I will likely take for developing a plan to create my version of *Frogger*.

0.5 Description of Work

In the initial stages (i.e. until the end of December 2018) I intended to research the game, investigating extant open-source implementations which make use of OO styles, and planning how I would go about mapping some of those OO features to an FP style. This research will extend to an experiential level, in which I will investigate the "feel" of these current versions of the game. After this I will plan out the initial stages of development and, over the Christmas break, I hope to have a first "alpha" version of the game for user testing. This will likely be a somewhat naive implementation, which may not leverage the benefits of the FP style properly and, consequently, I will ask my external sponsor, Henrik Nilsson, for feedback on this "alpha" version. From that feedback the game will be refined until such a point that it meets the specifications laid out below.

0.6 Methodologies

Broadly the project will be implemented as a cabal program, in principle for ease of testing. Cabal is Haskell's system for building programs and libraries[?]. It provides a "cleaner" method of installation than downloading and compiling all of the relevant `.hs` files. As well as this, the cabal system allows for easier configuration of system requirements (i.e. requiring a user to have at least GHC version 4.1.1), dependency management, and declarations of the project's author and maintainer. It also allows testing systems such as Travis CI to work[?], which I intend to use as part of the development process. This CI testing system will ensure that any changes I make via pull requests will not break the broader game as a whole.

I intend to use the OpenGL window system, in particular the Haskell GLUT library, for drawing windows. This is because the G53GRA Computer Graphics module has increased my knowlege of the way this library functions, in particular the translation and scaling features, and because it works on most platforms (Windows and Mac specifically) with a good degree of performance. Haskell's GLUT library operates based on a system of callbacks[?], which abstract many of the issues Haskell usually faces when trying to natively deal with time-dependent and global "variables". Eventually I would hope to move away from this GLUT system to a model based on Functional Reactive Programming. However in the "early stages" I feel that using GLUT is a logical way of ensuring that I understand and appreciate the key concepts required for creating a game using Haskell, before progressing to a more complex rendering system.

In terms of replicating the class hierarchy present in OO systems, Haskell can make use of a system of Modules, ultimately feeding functions from one file into another at the compilation stage. I intend to use this to essentially replicate the various "objects" that will be required by the game (the player, other moving objects, etc.), eventually feeding into the main executable file. I believe it will be possible, at least in the initial stage, to make use of Haskell's `data` declaration; this differs from the `type` declarations described above, in that `data` allows one to make a new algebraic data type as opposed to `type` which does little more than make your new type a "shortcut" to the old one (not dissimilar to a `#define` in C). `data` declarations can take many forms, with the following record-style syntax being the one I intend to use for the project due to its readability, and the fact that it automatically creates "methods" (to use Object-Oriented terminology) to access "member data".

```
data Object = p {x :: Float,
                 y :: Float,
                 velocity :: (Float, Float),  -- (x,y)
                 spritePath :: String}
```

From this I intend to create an "environment" data type which will encapsulate the state of the game at any given point and be fed through the callback functions in GLUT.

The main challenge I foresee in this is determining exactly what files to create and what functions should be amalgamated with others. Given Haskell's dislike of polymorphism I can envisage some amount of "boilerplate" code - many almost identical functions created for slightly different input types - with respect to updating values for the player "object" compared to other moving objects. In order to skirt this issue there are two main options: type classes and guarded type definitions. Type classes are a system in which multiple types can have a version of a function applied to them given the proper definitions. An example of a type class is that of `Show`, the `Show` instance of a type being how a variable of that type is printed to the screen. So the `Show` instance of the `Object` type above would look as follows:

```
instance Show Object where
  show p = "x:␣" ++ (show . x) p
         ++ "y:␣" ++ (show . y) p
         ++ "vel:␣" ++ (show . velocity) p
         ++ "sprite:␣" ++ (show . spritePath) p
```

This idea of type classes can be thought of as superclass methods in Object-Oriented languages, however in my view they are somewhat "clunky" to use: they must be instantiated for each new type in the class, and new fields can lead to deprecated definitions.

Guarded type definitions involve multiple definitions of the same class with different fields. The `Object` definition above is for the player "object", so if we wanted some enemy `Object` with a `target` field - but otherwise the same fields - we can do so as follows:

```
data Object = p {x :: Float,        -- Player
                 y :: Float,
                 velocity :: (Float, Float),  -- (x,y)
                 spritePath :: String}
            | e {x :: Float,        -- Enemy
```

```
                y :: Float,
                velocity :: (Float, Float),  -- (x,y)
                spritePath :: String,
                target :: (Float, Float)} -- target x and y
```

I believe this is an easier option for some level of polymorphism and reduction of "boilerplate" code.

0.6.1 Mapping Types

From the report I wrote detailing the classes and methods of the extant *Frogger* implementation, the following classes are defined:

- `AudioEfx`
- `Car`
- `CopCar`
- `CollisionObject`
- `Frogger`
- `FroggerCollisionDetection`
- `FroggerUI`
- `Goal`
- `HeatWave`
- `LongLog`
- `Main`
- `MovingEntity`
- `MovingEntityFactory`
- `Particle`
- `ShortLog`
- `Truck`
- `Turtles`
- `Windgust`

It is my view that data types should be able to encapsulate `MovingEntity` and its subclasses, `Car`, `CopCar`, `Truck`, `Turtles`, `ShortLog`, `LongLog`, and `Frogger`, using the guarded data type definitions described above. With respect to the environmental effects (`HeatWave` and `WindGust`), I believe these can be modelled using the callback functions in freeGLUT with random delays.

0.7 Design

The proposed specifications and requirements for this program are as follows:

0.7.1 Specifications

- The game shall take place in a 640x480 pixel (VGA resolution) window.
- The game shall be written in Haskell.
- The game shall make use of the OpenGL window system.

0.7.2 Requirements

- The user shall control a single character on the screen (hereafter the "Frogger") using their computer keyboard.
- The objective of the game shall be to move from the starting point to one of multiple possible goals via a series of moving obstacles spread across 10 "lanes".
- These lanes will consist of 5 road lanes and 5 river lanes, with a safe place between them.
  - The road lane will contain vehicles: cars and lorries.
  - The river lane will contain three types of obstacle:
    - Logs: The basic object of the river, these will provide safe passage however will carry the *Frogger* in whichever direction they are already moving.
    - Turtles: will behave identically to the logs, however will submerge themselves on a regular cycle.
    - Crocodiles: will behave similarly to the logs, however if the *Frogger* steps onto their head the Frogger will die.

- If the *Frogger* is hit by a vehicle, moves off the edge of the screen, or touches the water the player shall die, lose a life, and have to begin the level anew.

- The game shall have no defined end-point, only becoming progressively more difficult as it goes on by way of obstacles moving more quickly.

0.8 The development process

0.8.1 Drawing A Window

The first stage of development for the game is to draw a window. Per the specifications set out in my interim report, this window is to be 640 pixels wide by 480 pixels tall. We first create a file called `Main.hs`; this can be thought of as analagous to the `main` function often found in imperative languages as the "point of entry" for a given program. Within this file we first declare it as a module called `Main`, and then declare the `main` function itself. Again similar to an imperative language, this is the function that will be called once the program is compiled and run. It is a function of type `IO()`, meaning that ultimately all it does is perform some `IO` action - some action with a side effect - it takes no arguments and returns no value that can be passed into another function. These functions often consist of multiple commands chained in a manner that could look imperative to some. However, as we shall see, this is not necessarily the case. To begin with we must initialise GLUT; this allows it to begin working and set up a session with the window system in use. The function `getArgsAndInitialize` also returns a tuple, with a string representing argument 0 (the program name) as the first element, and a list of strings representing the remaining arguments as the second. To again liken it to an imperative language, this is the `(int argc, char* argv)` seen so often in C and C-style `main` functions.

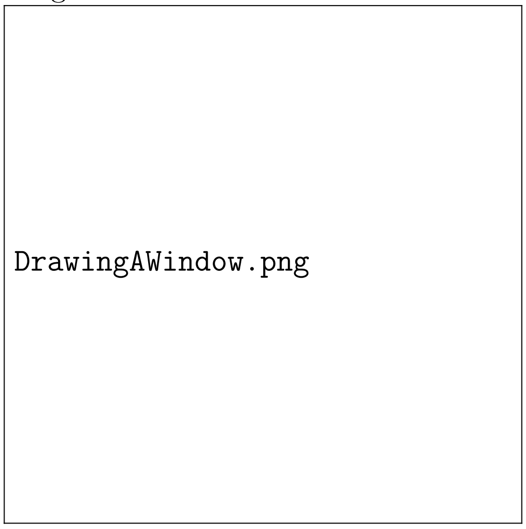
Now that the window session is initialised we need to provide it with some parameters with which to draw a window. In my motivation for doing this project I said that Haskell does not provide users with the ability to declare global variables. The GLUT library however does, via a module called `StateVar`. The library has a number of such `StateVar` variables, many of which will be explored as this program becomes more and more complex, but for the moment the one that is of use to us is the `initialWindowSize` variable. This is of type `StateVar Size`, where `Size` is a type defined in the GLUT library as little more than a wrapper for two numbers - a more readily understood tuple. We can use the `$=` operator as a "setter" for `StateVar` variables, and with it we can set the window's initial size to 640x480 pixels.

With the required settings now correct, we can call `createWindow`, passing the string "Frogger" as the window's title, and the program should compile and run. It does not, however, because we are missing something. If we try to compile the code we get the following error:

```
GLUT Warning: The following is a new check for GLUT 3.0; update your code.
GLUT Fatal Error: redisplay needed for window 1, but no display callback.
```

We need to set a `DisplayCallback`, essentially a function describing how to draw things within the created window at each refresh. In order to keep things clean this function will be declared in its own module, `Display`. Currently we're only interested in drawing the window in the first place, so we can just tell it to first clear the `ColorBuffer` - setting the whole content of the window to black - and then `flush`, which empties all command buffers, executing the commands as quickly as possible so as to ensure the image is as expected by the user.

Figure 3: The blank window drawn.



With that done, we now have a window being drawn! It is however just a blank, black-filled window, as seen in Figure **??**. In the next section we will add the game's titular *Frogger*.

0.8.2 Adding A Character

Now that we have a window to draw in, we should draw something in it. How better to start than with the hero of the game, the titular *Frogger*? Let's first represent him as a green unit square, just to get to grips with the coordinate and scaling system in Haskell's GLUT library.

First we need to set the current drawing colour to green, which is easily done using GLUT's `color` function.

```
color $ Color3 0.0 1.0 0.0 $
```

This function sets the current drawing colour to whatever is passed into it, which in our case is a `Color3` type (similar to the `Size` type described earlier), with the 3 values representing red, green, and blue values. It is worth noting that in Haskell anything that follows a lone $ is thought of as being in parentheses, so the above is semantically identical to this:

```
color (Color3 0.0 1.0 0.0)
```

Now that we have the correct colour selected we can draw a quadrilateral, which we can do with the `renderPrimitive Quads` function, as follows:

```
let p = [(1,0,0),(1,1,0),(0,1,0),(0,0,0)] :: [(Float,Float,Float)]
in renderPrimitive Quads $ mapM_ (\(x,y,z) -> vertex $ Vertex3 x y z) p
```

To expand on this code fragment somewhat, the `let` command is more or less the same as a variable declaration on the "global" scope, like declaring a variable `myVar :: Int` with a value of `5`, only limited to the scope of the function in which it is declared. It is a local variable, essentially. `p` is a list of tuples, each containing 3 `Float` values. These values are the x, y, and z coordinates for each of the points of the quadrilateral we're drawing, respectively. The reason their type is declared explicitly here is that otherwise the compiler throws an error relating to ambiguous types. From a stylistic perspective we don't want to clog up the rest of the program with values that no other function is ever going to need to know exists we append to the "local variable declaration" an explicit type declaration. The coordinates described by `p` denote a unit square extending from what would be considered the origin on a mathematical graph in positive x and y. `renderPrimitive` is the foundation for this; it does the actual drawing of what is fed into it. `Quads` is one of the options for "ways to draw": it means draw the following as a closed quadrilateral or series or separate closed quadrilaterals. Other options here include drawing something as a wire frame, triangles, a single polygon, or just points. The argument passed in here is a version of `p` that has had a lambda function[1] mapped over it. This lambda function takes a tuple of three values and, using two of the GLUT library's provided data constructors, converts them into a type accepted by `renderPrimitive`.

In theory, with all this in place GLUT should draw a unit square on the screen.

[1]lambda functions are anonymous functions written in the style of the lambda calculus. They are convenient when you need to do something slightly complicated but that will only ever get used in that one place.

Figure 4: A "unit square" - this is a result of the way in which GLUT draws and populates a window by default.

`DrawingACharacter.png`

As we can see in Figure **??** this square is much too large, and begins centered on the screen. Instead it should - in my opinion - start at the bottom left of the window. Conveniently GLUT provides functions for both scaling and translating, so this shouldn't be a problem!

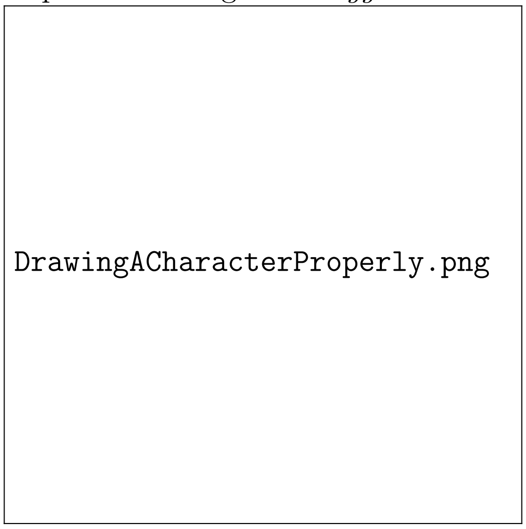
Something worth noting before I continue is that it is very important to scale and translate in the correct order; in general translation should be done first, because scaling actually affects translation. Mathematically speaking they are not commutative actions; this is because they are both examples of matrix multiplication.

The translation and scaling should be done before any other part of the display, given that they directly impact all other parts of drawing. They are both one-line functions taking similar forms.

The main difference is that `translate` takes a `Vector3` data type as an argument, whereas `scale` takes only 3 `Float`s. `translate` moves the drawing "origin" in a way described by the coordinates in its argument; before any scaling happens a GLUT window essentially consists of 4 "1x1" quadrants (these are stretched to fill the aspect ratio of the window they're in). The `scale` function takes 4 arguments: the scale factors in `x`, `y`, and `z`, and the thing to be scaled. Here the scale factors are `1/32` in x and `1/24` in y. These are both based on the window size in their respective direction and - because the window is effectively 2 units both across and tall - it should be scaled to 2/ that value, and the fractions have been reduced manually. Here one unit size is equivalent to 10 pixels, and so now we have a unit square as shown in

Figure **??**.

Figure 5: The unit square denoting our *Frogger* is now an appropriate size.



Now we have a character drawn and scaled appropriately. Next we need to give him some movement, by way of keyboard input.

0.8.3 Keyboard Input

Keyboard input is handled by way of another callback in the Haskell GLUT system, however implementing it will require some additional foundation code being added first. Specifically it will require the introduction of one of the major building blocks of this program, the `IORef`. This is Haskell's way of making use of the IO Monad to do things that it, as a language, would rather not, in this instance global variables and states. Firstly, however, we must declare what kind of variable it is we want to be global. We do that in a new file, `Type.hs`, which will be the point from which we declare all new types and functions relating to them. For now we will only have one type, the `Mover`, based on the `MovingEntity` superclass in the GitHub project reported on elsewhere.

This is a "record-style" data type with two fields: x and y, denoting the x and y coordinates of the *Frogger*. With this in place we can add the `Type` module to `Main` and, after creating the window, create a new `IORef Mover` with `x` and `y` both set to 0.0.

Finally before we set up the `keyboardMouseCallback`, we modify the `display` function to take as an argument an `IORef Mover`, and draw the character based on it. This requires that we change 3 lines of the function, resulting in the following.

loadIdentity is a very important function in all of this - it resets the modification matrix, which is to say that it resets the scaling and translation applied in the last call to the function. If this function is forgotten the screen is simply left blank with no explanation as to why, and all of the objects are translated and scaled from the previous frame. Moreover, the change from flush to swapBuffers is similarly important; the way the GLUT system works is with a pair of buffers (this is automatic on a Mac, but on other platforms will require modifying Main.hs to add the line initialDisplayMode $= [DoubleBuffered]). This pair of buffers is swapped between constantly, with one being displayed and the other "behind" it. The "behind" buffer is the next one to be drawn, and so is the updated version. swapBuffer, as the name suggests, swaps the buffers. If flush was not replaced there could therefore be no animation.

With all of that done, we can now create the file Input.hs, which will contain all of the functions required to deal with keyboard and mouse inputs. In our case we want the WASD keys to be responsible for movement, which we can achieve by defining a function input which takes as an argument an IORef Mover and returns a KeyboadMouseCallback, which we will set the global keyboardMouseCallback to in Main.hs. A keyboardMouseCallback takes 5 arguments: our Mover variable, the key being pressed, that key's state (up or down), any modifier keys (Shift, Ctrl, Alt), and a position. Our input function can thus be described as follows.

```
input :: IORef Mover -> KeyboardMouseCallback
input m c Down _ _
  | c == (Char 'w') || c == (Char 'W') = m $~! \f -> f {y = y f + step}
  | c == (Char 'a') || c == (Char 'A') = m $~! \f -> f {x = x f - step}
  | c == (Char 's') || c == (Char 'S') = m $~! \f -> f {y = y f - step}
  | c == (Char 'd') || c == (Char 'D') = m $~! \f -> f {x = x f + step}
  | otherwise                          = return ()
  where step = 5
input _ _ _ _ _ = return ()
```

With this we now have a *Frogger* who will move continuously about as long as either W, A, S, or D are being pressed.

There is an additional callback required for this all to work, however: idleCallback. This callback deals with background events and is required for continuous animation. It can handle background tasks such as updating the positions of non-player characters (a function that will have great use later on in this process). For now however it will simply take the following form:

```
idle :: IORef Mover -> IdleCallback
idle e = do e $~! id
            postRedisplay Nothing
```

0.8.4 Other Characters

Frogger is a game of avoiding obstacles, so let's now add some obstacles to be avoided. This will first require that we extend the Mover type to include some "subclasses", in the first instance a Car "subclass".

```
data Mover = Frogger {x :: Float
                     ,y :: Float
                     ,s :: Float
           }
           | Car {x :: Float
                 ,y :: Float
                 ,l :: Float
                 ,w :: Float
                 ,v :: Float
           }
```

x and y are common with the Frogger "subclass", and will be used with the drawing function we will define later. l and w are the length and width of the car respectively; width will likely remain a constant, however length could change depending on what kinds of car obstacles we wish to add to the game. v is the velocity of the car, with the positive direction being left-to-right, or increasing in x. This will be used in conjunction with a modified version of the idle function, discussed later.

We now have to create a containing data type for both the Frogger and the obstacles, such that they can all be stored as one IORef. This is to be the environment variable, and in due course will contain all relevant information about the game, including scores, number of lives, time remaining, and so on. For now, though, it only need include two things: the Frogger and a list of other obstacles. It can therefore be defined thus:

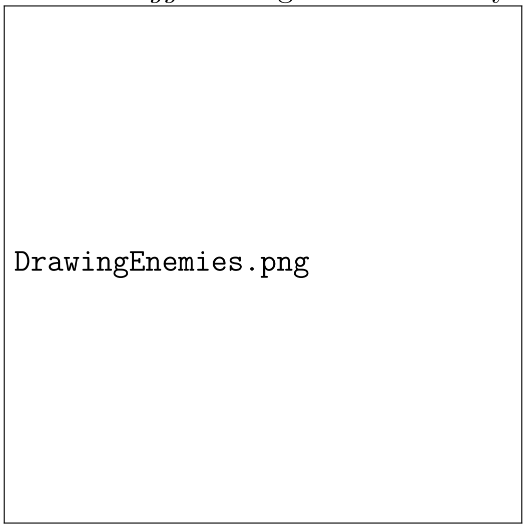
```
data Env = E {player :: Mover
             ,enemies :: [Mover]
         }
```

We can then extend Display to include a helper function to draw any Mover to the screen. This will be a two-step function, the first step being a call to preservingMatrix, a function in GLUT that stores all of the current transformation information (in this instance the translation and scaling discussed when we first drew a character to the screen). We can then call a function of our own definition, using pattern matching to ensure the appropriate version of it is called based on what kind of Mover we are dealing with - it would not work if we drew the player character identically to a car. Something to consider is that for all moving objects with a given velocity, the positivity of that velocity denotes the direction of that object's movement. Therefore we can use that value in conjunction with the Haskell Prelude function signum to appropriately "rotate" the object (in reality this is a scale being used to flip the image 180 degrees).

Note the additional helper function unitSquare - this will be replaced in due course as the game's art becomes more complex. However for now we can use it to draw a square of size 1x1 pixel, which we can translate and scale as we wish. The helper `makeVertex` on the other hand will likely only become more useful as the complexity of drawing increases - it is a useful constructor, taking 3 points and converting them into the data type used by `renderPrimitive`.

Now that we have these, we can extend `display` to take as an argument an `IORef Env` rather than an `IORef Mover`, and replace much of the drawing code with the `drawMover` function applied to parts of the environment variable.

Figure 6: The *Frogger* being closed in on by two cars



As we can see in Figure **??**, we can now draw cars to the screen. By modifying the `idle` function we can also animate them, and with that we can get to the first real glimpse of the game of *Frogger*.

The modification to `idle` is slightly involved, mostly because of the now nested records we have meaning that modifying an internal one is more or less a two-step process. Fortunately Haskell's `let` syntax makes this process considerably easier, because we can take the internal record value (i.e. the one we want to change) and store it with a different name. We can then take a helper function, whose role is to update the x position of a mover based on its velocity, and map it across the `enemies` field of the environment variable, thus:

```
idle :: IORef Env -> IdleCallback
idle e = do e $~! \env -> let es = enemies env
                          in env {enemies = map updateMover es}
            postRedisplay Nothing

updateMover :: Mover -> Mover
updateMover c@(Car {x = cx, v = cv}) = c {x = cx + cv}
```

This two-step processing does also have to be extended to the `input` functions, which leads to a great deal of boilerplate code.

```
input :: IORef Env -> KeyboardMouseCallback
input m c Down _ _
  | c == (Char 'w') || c == (Char 'W')
    = m $~! \e -> let p = player e
                  in e {player = p {y = y p + step}}
  | c == (Char 'a') || c == (Char 'A')
    = m $~! \e -> let p = player e
                  in e {player = p {x = x p - step}}
  | c == (Char 's') || c == (Char 'S')
    = m $~! \e -> let p = player e
                  in e {player = p {y = y p - step}}
  | c == (Char 'd') || c == (Char 'D')
    = m $~! \e -> let p = player e
                  in e {player = p {x = x p + step}}
  | otherwise
    = return ()
  where step = 5
input _ _ _ _ _ = return ()
```

The repetition of `let p = player e in e {...` throughout the middle of those guarded functions is both ugly and not especially useful; if that function were to change each individual instance of the function would have to, and the more instances of the function that there are, the more likelihood one will be missed. This has major negative impact on the maintainability of this code, and will have to be addressed. For now however, we are focussed on getting the game to a point where it is playable.

0.8.5 Collision Detection

This will again consist mostly of modifications to the `idle` function - at this stage in development any kind of detection is the desired goal; acting upon this detection is a task for later. The function `hasCollided` will do this; it will take as an input two `Movers` (one being the `Frogger`) and, taking into account the direction of the enemy mover, will return whether or not their two hitboxes intersect. If they do, the *Frogger* will die. This function simply determines whether or not any combination of the top/bottom and left/right boundaries of the *Frogger* intersect those of the `Mover` it is being checked against. We can then map this function over the `enemies` field of the environment, checking the *Frogger* against every enemy. We can then create a function `hitCheck` to do exactly that, returning a list of `Bools`, which we can apply `or` over to determine if any enemy has hit the *Frogger*. If any have, at the moment it simply prints "Hit!", otherwise does nothing.

As it stands I am pleased that simple rectangular hitboxes are an adequate option.

0.8.6 Game State

Next we add the ability to play/pause the game, or rather add a game state variable in general. This will allow for not only the aforementioned playing and pausing of the game but also for things such as splash screens and death events to be reflected more appropriately. To do this is first a matter of modifying `Type.hs`, firstly creating a new type `GameState` which will give us a verbose manner of specifying what state the game is in. This could be done using a simple integer, with 1 representing playing, 2 paused, and so on, but Haskell provides us with the tools to do so in a more intuitive and easier to follow manner, so we shall use them.

```
data GameState = PreStart | Playing | Paused | PlayerDead
                 deriving (Eq, Show)
```

Now we add a field to `Env` which will contain a `GameState` variable, and that should be sufficient for the game. This is not quite the case though, as of course we still have to insert code elsewhere to make sure things are actually playing and pausing as they should.

We can begin in `Idle`, where all of the updating of the objects on the screen happens. Here we can add a check, an if statement querying the current state of the game as given in the argument. If the game is currently in the `Playing` state we can continue to update everything as normal. Otherwise pause all updating, using `return()` as the empty case.

To move between the states we must update `Input.hs`, adding cases for the various new `GameStates`.

0.8.7 Collision Detection 2 - Detection on the River

Collision detection on the river section is slightly more complex than it is on the road; in this case it is a matter of the player remaining alive if they *have* collided with an object, and dying if they step onto a lane with no object present.

Hence this will require some modification of the code. We introduce a custom typeclass[2], `Drawable`. This typeclass will have two operations, `draw :: a -> IO()`, with which we can replace the `drawMover` function, and `update :: a -> a`, with which we can replace the `updateMover` function. It will also include operations to draw the object to the screen whilst preserving the current transformation matrix, and to do so to a list of objects. Both of the latter operations will have default definitions.

[2]A typeclass in Haskell can be thought of similarly to superclasses in imperative languages. They are collections of types which support certain operations, and allow for ad-hoc polymorphism. www.haskell.org/tutorial/classes.html

This will allow us to break the `Mover` type down, improving readability and giving us the ability to create functions for more specific instances of a type. In particular, we can split it into 4 new types:

- `Frogger`: the player character.
- `RoadMover`: moving objects on the road.
- `RiverMover`: moving objects on the river.
- `Goal`: the goal objects.

As a result of this the `Env` type must be modified with the `enemies` field being split into `roadEnemies`, `riverEnemies`, and `goals`.

As well as this, the collision detection function will be modified to take the x, y, length, and width values of both objects as arguments, as opposed to just taking the objects themselves.

With this we can split the `hitCheck` function into three parts:

- `roadCollision :: Frogger -> RoadMover -> GameState`
  If the Frogger has collided with the road object, it returns `PlayerDead`, otherwise it returns `Playing`. We can use this to build a function `hitCheckRoad`,
- `riverCollision :: Frogger -> RiverMover -> GameState`
  Essentially the opposite: if there is a collision it will return `Playing`, else `Playing`.
- `goalCollision :: Frogger -> Goal -> Bool`
  A simple boolean check to see if a goal has been collided with. This function can then be used in a larger function `hitCheckGoal`, which at this point simply `or`s the list of collided goals, and returns `LevelComplete` if that is `True`.

`hitCheck` now becomes a ”routing” function, calling `roadCollision`, `riverCollision`, or `goalCollsion` depending upon where on the map the player is, while checking that the player is within the screen bounds (if they go beyond the screen they die).

`lanes` is a data object exported by `Type` containing the y-values of all lanes - lanes 1-5 (index 0-4) are the road lanes, and lanes 7-10 (index 6-11) are the river lanes.

We now have support for determining death both on the road and on the river. However if the player is sat on a log they currently do not move. Fixing this will be the next step.

0.8.8 Tracking River Objects

The first step to this will be updating the `Frogger` data type with a velocity element, `f_V`. We then change the `update` function in its `Drawable` instance from the identity function to something resembling the definition for the other moving objects, specifically

With that done, and updating the Frogger's position added to the idle callback, we can implement the "follow" functionality.

To do this we make use of Haskell's `Either` type, which allows a function to return results of one of two types, given as the `Left` type and `Right` type.

Therefore `hitCheck`'s type signature will now be `hitCheck :: Env -> Either GameState Fl`

Thanks to the breaking down of the `hitCheck` function into its road, river, and goal elements we need only concern ourselves with changing the `hitCheckRiver` function, simply prepending `Left` to the other returned `GameState`s.

All definitions for `riverCollision` will be updated in the same way, but we will again use the one for `Log`s as an example:

```
riverCollision (Frogger {f_X = fx, f_Y = fy, f_S = fs}) (Log {ri_X = lx, ri_Y = ly, ri_L = ll, ri_W = lw, ri_V = lv})
  = if hasCollided fx fy fs lx ly ll lw then Right lv
                                        else Left PlayerDead
```

As we can see, the function has been updated to, in the instance of a collision, return the `Right` type, a float giving the value of the Frogger's new velocity. Importantly if a `Float` is returned it can be assumed that the player is not dead.

All updating of the main `Env` variable has now been moved to a function `updateEnv`, which is only called when the `gameState` is `Playing`. Within this, `update` is applied to all applicable fields, and the `Either` returned by `hitCheck` is now dealt with as follows:

```
(f_V', gameState') = case hitCheck e of Left gs -> (0.0, gs)
                                        Right v -> (v, Playing)
```

0.8.9 Cause Of Death

It would be nice if the player knew what killed them - *Frogger* after all has the most ways to die of any game - so that will be the next thing implemented.

The first thing to do here is to modify the `GameState` data type, adding a `String` to `PlayerDead` such that it can carry a message.

Then we update `hitCheckRoad`, `hitCheckRiver`, and their associated collision functions, which conveniently return `GameState`s rather than `Bool`s. This means that we can trivially add to the `PlayerDead` options with a brief message detailing what killed the player, getting hit by a car or drowning being the current options. Finally we update the end of `hitCheck` with the fact that the player has died by jumping off the map, and it's done.

0.8.10 Different displays

First we update the `display` function to show different things depending on the current state of the game

- In `PreStart` it should show a welcome message detailing the controls of the game.
- In `PlayerDead s` it should render `s`, the string explaining why as well as the number of points the player has accrued in their game.
- In `LevelComplete` it should congratulate the player on completing the level, tell them how many points they've got, and tell them how to advance to the next level.
- Otherwise it should draw the map according to the current `Env`

This can be accomplished by adding a `case gameState e of ...` statement to `display`, with the different options being applied as necessary.

0.8.11 Numbered Levels

The next step is to add numbered levels, such that the user can keep track of their progress without needing an on-screen score. This is a simple addition, adding to the `Env` type a field `level :: Int` and initialising that to 1 in `startEnv`.

0.8.12 Entities

The code was at this point becoming quite complex, with each different type (`Frogger`, `RoadMover`, `RiverMover`, `Goal`) having a different definition and descriptor for their positions in `x` and `y`, their length and width, and their velocities in both directions. This makes the definitions for the typeclass `Drawable` somewhat more complex than is necessary, with explicit definitions having to be made for every function in every new instance of the typeclass. Haskell supports default definitions in its typeclasses, and to take advantage of this will make the code much easier to read and understand. Consequently we can implement a new custom type, `Entity`, another record-style type defined as follows:

```
data Entity = Entity { x :: Float
                     , y :: Float
                     , dX :: Float
                     , dY :: Float
                     , l  :: Float
                     , w  :: Float
              }
```

We can then assign an `Entity` to each instance of the `Drawable` class, and removed all existing instances of values representing x and y positions and velocities, length, and width. From this we are able to extend the `Drawable` typeclass, implementing a user-defined function to get the `Entity` out of the `Drawable`, and several functions with default definitions to, from that, get the position, speed, and length values out of that `Entity`.

This is effectively an implementation of `Java`'s `contains` class constructor - `Entity` can be considered the superclass of everything in the typeclass `Drawable` because every `Drawable` contains an `Entity`. With the implementation of some user-defined functions to set various values in the contained `Entity`s, the `Drawable` definition now looks like this:

```
class Drawable a where
  update :: a -> a
  update = updateX . updateY
  draw :: a -> IO()
  preservingDraw :: a -> IO()
  preservingDraw = preservingMatrix . draw
  preservingDraws :: [a] -> IO[()]
  preservingDraws = sequence . map preservingDraw
  getEntity :: a -> Entity
  getX :: a -> Float
  getX = x . getEntity
  getY :: a -> Float
  getY = y . getEntity
  getdX :: a -> Float
  getdX = dX . getEntity
  getdY :: a -> Float
  getdY = dY . getEntity
  getL :: a -> Float
  getL = l . getEntity
  getW :: a -> Float
  getW = w . getEntity
  updateX :: a -> a
  updateX d = setX (getX d + getdX d) d
  updateY :: a -> a
  updateY d = setY (getY d + getdY d) d
  setX :: Float -> a -> a
  setY :: Float -> a -> a
  setdX :: Float -> a -> a
  setdY :: Float -> a -> a
```

0.8.13 Goal Numbers

As the user progresses through the levels of Frogger, not only do the enemies get faster but also the number of goals to reach becomes greater, as follows:

- Level 1: 1 goal
- Level 2: 2 goals
- Level 3: 3 goals

- Level 4: 4 goals
- Level 5+: 5 goals

To do this we modify `startEnv` to take as an argument the level to generate, and modified the goal creator with a `case... of` statement, which - based on the level given - generates the relevant number of goals by way of a list comprehension.

0.8.14 Froggers Occupying Goals

With multiple goals now implemented, another way to die in Frogger is to try and jump on a Goal that is already occupied - i.e. one which you've already got a Frogger to. To implement this we add an additional case to the river part of the collision detection function to look up, using Haskell's `Maybe` data type, whether or not the Frogger has collided with a `RiverMover` or a `Goal`. If they have collided with a `Goal` it now goes to a second stage of processing: determining if that `Goal` is occupied. If it is, the Frogger dies, otherwise the `Goal` is marked occupied and the Frogger's position reset. If all `Goal`s in the level are marked as occupied, the level is deemed completed.

0.8.15 Crocodile Death

Continuing in the vein of "adding ways to die", it should be that if the Frogger jumps into a `Croc`'s mouth they die. The `Croc`'s mouth can be considered to be the entire front third of its body. This again requires that we modify the collision detection function, specifically adding a case for if the `RiverMover` is a `Croc`. In this case, the `Croc` is now logically split into two smaller `Croc`s, one representing the body and the other the head. If the player collides with only the head they die, otherwise they ride the `Croc` as if it is a `Log`.

0.8.16 Jumping

Currently the Frogger does not jump. Instead, when the user presses a directional button the Frogger teleports to the new location rather than jumping towards it. This makes the game considerably easier than it should be; the user can always outrun road enemies and catch up to useful `RiverMovers`. Hence the next step is for us to implement non-instantaneous jumping. The first step was to add all of the requisite record fields to the `Frogger`, specifically a two `Booleans` denoting whether or not the `Frogger` is jumping in x or y, and four `Floats` denoting the `Froggers` target position and previous speed in x and y. Next I updated the Froggers instance of update with a conditional that checks if the Frogger is jumping in a specific direction. If it is, update then checks if it has reached its target coordinate in that direction, and if that is true it stops the jump and resets the Frogger's speed in x and y. This does lead to a minor problem whereby if the Froggers speed during the jump is not a divisor of the distance it is jumping (e.g. the speed is 3 and the distance is 10) it will never stop. This is easy enough to deal with however, by simply ensuring that the speed is an even divisor of the distance. We can do this using the lanes variable defined earlier, and basing the Frogger's speed on some divisor of the difference between the first and second element of that list.

```
laneDiff :: Float
laneDiff = lanes !! 1 - lanes !! 0

baseSpeed :: Float
baseSpeed = laneDiff / 5
```

Here the speed is 1/5 of the distance between lanes - this means that the Frogger will take 5 frames to move 1 lane up or across.

Additionally we should modify the collision detection function such that the Frogger will never die while it is mid-jump - only when it lands.

The final step of this addition is to modify the input function such that when a movement key is pressed it sets the Frogger off jumping. Importantly, if the Frogger is already jumping any input is ignored, otherwise the user could just spam movement keys with potentially game-ruining consequences.

0.8.17 Migrating from GLUT to gloss

Haskell is a pure language by default - this means that every input will give the same output every time it is applied to a particular function. It does however have some impure elements, one being the IORef. IORefs are references, similar to pointers in imperative languages, which can be used to store some piece of information that is used in multiple parts of a program[?]. This is a convenient workaround for the fact that Haskell does not allow for global variables, but it has its drawbacks. In particular IORefs are impure, existing within the impure IO monad. This means that modifying and using them can have side effects that may impact future computations, leading to errors and bugs within the program.

The next step of this project is therefore to remove IORefs, which required changing the entire drawing system. Gloss is a Haskell library based on the GLUT backend, but with options for pure gameplay. This means that the game will function more predictably, and be better suited to take advantage of the benefits of Haskell as a language.

The migration process is fairly straightforward: the update functions translated almost directly, and the display functions were only slightly more difficult to translate to the new library. One major difference is that the coordinate system for drawing shapes is center-based rather than bottom-left based, so for functions like collision detection the objects now need to be logically ”translated” up and right by half of their height and width respectively.

0.8.18 Randomness

Randomly generating the speeds of the `Mover`s is the next objective; the game would become too easy for a user if it was exactly the same every time. Random number generation in Haskell relies on the IO monad to get the initial number generator, and consequently the first step of implementing this was to modify `main` to create a random number generator. This is a somewhat involved process, because the specific seeding method we want to use was to get the current time and generate the seed from that. In Haskell, this requires that you get the current time of day, which is of a type that is more or less unusable in any computation, convert that to picoseconds since midnight, and divide that value by 10^{12}. Now that we have the seed generated we can pass it as an argument to `mkStdGen`, and the random number generator is implemented.

We can then extend the `Env` type to have a space for a `StdGen` (the type of a number generator), and the `startEnv` function. `startEnv` is modified to generate a list of initial velocities for the `Mover`s to have, and those velocities are now applied to said `Mover`s on generation.

Finally we modify `inputDead` to, on player death, generate a new `Env` with a different velocity list such that a user would not get stuck on one particularly difficult seed for an extended period of time.

0.8.19 Sprites

The game is now nearly finished, the only remaining thing to do was to add sprites for a greater level of graphical fidelity. We should use `vitalius`' spritesheet for this, in the main to ensure that in the user testing phase the graphics are as similar as possible between the two versions of the game and consequently people's experience of the games is as similar as possible.

In order to do this we first have to define two new types: `Sprite` and `SpriteID`. A `Sprite` is a tuple containing a `String` and a `Picture` (gloss' representation of an image), where the `String` denotes the situation in which the `Sprite` will be used. For example the `Frogger` will have two `Sprites`, one for when it is stationary and one for when it is moving, and they would be labelled as such. A `SpriteID` is a tuple containing a `String` and a `Sprite`, where the `String` denotes what type of object the `Sprite` should be mapped to.

Then we update the `Entity` type to add a field `ss :: [Sprite]`, and `Drawable` to add a function to return the list of `Sprite`s belonging to a `Drawable`. Next we add a `SpriteID` list and two `Picture` fields to `Env`, here denoting the full list of ID'd sprites, the background of the game, and the complete spritemap. The full list of ID'd sprites will be used on each new level generation to ensure that everything has a sprite assigned to it.

We then modify the `draw` functions of all of the `Drawables` with a conditional: if there exists a sprite for this object and the current "situation" of the object (i.e. Turtles are underwater, Frogger is jumping) then use that sprite, else fall back on the primitive drawing being used previously. We add a function to assign a list of `Sprite`s to an object next, which I did by making `Drawable` a subclass of `Show`, and using the record constructor (e.g. `Frogger {...}`) as the identifier. All `SpriteID`s have the constructor as the ID `String`, so a function to assign `Sprite`s to objects now only needs to pair those up.

Finally we add to `main` a section to get the initial sprite list, chop it up into the relevant sprites for all objects, and assign them all to `SpriteID`s.

0.8.20 Haddock

The program is now finished; the game is playable and complex enough so as not to be boring. The final thing to do within the back end of the game is add proper commentary and documentation, such that another user can take the source code and extend or modify it. Normal commentary (denoted in Haskell by prepending it with `--`) is fine for most purposes, but something more intelligent is - in my opinion - necessary for a project of this scale. For more intelligent commentary we can use Haddock.

Haddock is a system used to document the core libraries of GHC[?], and can automatically generate a rich HTML website with details about all exposed (user-facing) functions, types, and classes. To implement Haddock, we can simply add a small amount of syntax to the beginning of comments, specifically a pipe (|) character, which Haddock will process as the leading comment for the function/class/type definition.

We add comments as necessary to every defined function explaining their purpose and expanding on what each argument is used for. An example of the Haddock-generated documentation can be seen in Figure ??.

0.9 The Finished Program

The final program quite closely resembles an object-oriented design. The super (type) class `Drawable` has 4 subclasses: `Frogger`, `RoadMover`, `RiverMover`, and `Goal`. There are centralised `update` and `draw` functions, with each of the above classes having their own definitions and implementations for them.

Despite architecturally resembling an OO program, this program does make heavy use of functional and Haskell-specific features. In particular, the use of `lookup` and the `Maybe` type in the collision detection allows for a particularly efficient handling of possible collisions.

The game works as follows: the user is first presented with a splash screen welcoming them to the game and detailing the basic control system, then they are told to press any key to proceed. Once they do they are into the main body of the game: pressing the arrow keys will move the Frogger around and they must, as per the original, avoid road obstacles and use the floating objects in the river to fill all lilypads at the top of the screen with Froggers. Once all pads are filled, the level is complete. On level completion, all objects become faster as a way of increasing difficulty. There is no "game complete" state - it is an arcade game and consequently will go for as long as the user can stay alive.

0.10 User Testing

The testing process for this project was done in the form of a user study, conducted as follows (initially):

1. User plays one version of the game for 10 minutes.
2. User writes down notes about their experience for up to 5 minutes.
3. User plays the other version of the game for 10 minutes.
4. User fills out a short questionnaire rating aspects of the two games for similarity (0 indicating that the aspect was completely different and 10 indicating that it was identical).

I planned to get around 15 users to complete this study, with the first three being used to determine what worked well with the study and what needed changing. After the first 3 rounds of testing I determined that 10 minutes was too long for a user - they would become bored and fail to pay attention to the game. Consequently for the remaining rounds of testing I cut the play time down to 5 minutes. Additionally a piece of feedback I received from all three initial participants was that my game was too hard; in particular the Frogger moved too slowly. For the remaining rounds I increased its speed by a factor of 2, enough that it can now outrun all but the fastest `Cars`.

The remaining rounds of testing were all done under these new parameters. I processed the results using R and the `ggplot2` library to visualise them. The box plots denoting the spread of ratings for the initial 5 aspects of gameplay can be seen in Figure **??**. In it we can see that there was a very wide spread of ratings for the aesthetics of the game, a surprise given that they used the same spritesheet. The controls for both games are identical, so the generally high score given there makes a good deal of sense. Difficulty resulted in a wide spread of ratings initially, however filtering the results to only take into account the Studies post-tuning shows a much narrower spread of ratings. Despite the tuning however, the difficulty was the most different aspect between the two games. Stress level is, in my view, an interesting measure; one would assume that it would be scaled roughly the same as the difficulty, however this is shown to not be true. Additionally, there is little difference in the spreads for all results and only post-tune results. Finally understandability is generally very high - this is not a surprise as the game is relatively simple.

In Figure **??** I have plotted the mean of the 5 aspects represented in the box plots in Figure **??** against the score that user assigned to the "Overall Gameplay" category.

0.10.1 Critical Evaluation

Looking at this user study critically, there are some issues present. The main one is that there is very little in the way of detail within the answers - the two games are rated as very different in terms of difficulty but from the results there is no way of determining which game was the more difficult. Similarly with aesthetics - what aesthetic differences are there? One user rated the aesthetic similarity a 1, but with no way to express themselves beyond numbers we cannot know what differences they noticed. Were I to run this study again I would have a notes section where the user could write a short paragraph explaining their answer.

0.11 Summary and Reflections

Broadly speaking I believe this project has been a success. I believe that the game of Frogger I have created is a good example of the game, and the results of the User Study suggest that it is close in feel to the original. I do believe that it could be better - implementing lives, a countdown timer, and sounds would take it from being close to being near-identical.

In my opinion also I have used some features of Haskell very effectively in creating this game - specifically the use of `Maybe`s for collisions, many list comprehensions used for generating objects, and at one point I was using the `Either` type for giving the Frogger a speed in x or y (however this has since been deprecated). The `cabal` system I feel helped greatly with development, mainly because it allowed for continuous integration testing via Travis CI.

Moving to the gloss drawing system was not something I originally planned on doing, but the process of moving to it was considerably easier than I anticipated. In particular the update function translated directly, and the display function required markedly less modification than I expected it would. I believe this migration was overall useful, in particular because of the control over the frame rate it affords, as well as abstracting away the impure parts of the code from the user.

With regard to my management of the project, I feel that it has been mostly good. The Gantt chart I showed in my proposal has by and large been followed well, with the exception of the "FP-specfic" sections - it was easier to simply add these in during the development process than to tack them on at the end. The user study was carried out later than I would have liked, which meant that some of the more in-depth analysis I had wanted to do was unfeasible, but with regard to almost every other aspect of the initial plan, it has been followed quite tightly.

In summary this project has been a success, and the experience has been a positive one. I have learned and made use of a number of useful Haskell features (in particular typeclasses, `Maybe` functions, and the cabal system), and to an extent recreated the object-oriented paradigm within a functional language.

Figure 7: A screenshot showing the Haddock-generated documentation for the file Update.hs

Haddock.png

Figure 8: Screenshots showing the stages of the game. Clockwise from the top-left: the splash screen, main gameplay, the screen shown when the user dies, and the screen shown when the user completes a level.

`Screenshots.png`

Figure 9: Box plots showing the ratings for the 5 aspects of gameplay recorded in the User Study

`../../Testing/Processing/boxes.png`

`../../Testing/Processing/boxesPost.png`

Figure 10: Scatter graph showing the means of the 5 aspects compared to the general gameplay rating

`../../Testing/Processing/meanVGeneral.png`

Bibliography

[1] Commercial Users of Functional Programming homepage
`http://cufp.org`
Accessed 2018-18-11

[2] Softline Magazine, November 1982
PDF available at `www.cgwmuseum.org/galleries/issues/softline_2.2.pdf`
Accessed 2018-11-15

[3] ”The Frogger”, *Seinfeld*, NBC, 23rd April 1998
Television

[4] The Haskell website
`www.haskell.org/`
Accessed 2018-11-15

[5] John Carmack’s keynote at Quakecon 2013
`www.youtube.com/watch?v=1PhArSujR_A&feature=youtu.be&t=127`
Accessed 2018-11-15

[6] Slides from Tim Sweeney’s talk ”The Next Mainstream Programming Language: A Game Developer’s Perspective”
`www.st.cs.uni-saarland.de/edu/seminare/2005/advanced-fp/docs/sweeny.pdf`
Accessed 2018-11-15

[7] The Yampa Arcade
A. Courtney, H. Nilsson, and J. Peterson, 2003
`www.antonycourtney.com/pubs/hw03.pdf`
Accessed 2018-11-15

[8] The Haskell website, Cabal page
`www.haskell.org/cabal/`
Accessed 2018-11-22

[9] GLUT's Hackage page
hackage.haskell.org/package/GLUT
Accessed 2018-11-22

[10] Konami's Frogger and Castlevania Nominated for Walk of Game Star
www.gamespot.com/news/konamis-frogger-and-castlevania-nominated-for-walk-of-g
Archive available at web.archive.org/web/20130202065907/http://www.gamespot.com/ne
Accessed 2018-12-04

[11] The Travis CI home page
travis-ci.com
Accessed 2018-12-04

[12] Tonks, M 2018
PUC-MAN
Undergraduate Dissertation
University of Nottingham
Nottingham, England

[13] The Hackage page for IORefs
hackage.haskell.org/package/base-4.12.0.0/docs/Data-IORef.html
Accessed 2019-04-08

[14] The Haddock homepage
www.haskell.org/haddock/
Accessed 2019-04-21