Abstract

Functional programming offers new ways to think about software in general and has become more popular in the recent decade. However, its applications are still limited. Games, for example, are still often developed in imperative languages. In this project, we present the design and implementation of an online multiplayer, 3D first-person shooter game, Hamster Ball, in Haskell, a purely functional programming language. Yampa, a domain-specific language embedded in Haskell based on concepts of Functional Reactive Programming, is used to develop simulated worlds in a concise, clear and modular way.

1 Introduction

Games are commonly programmed in imperative languages, especially in object oriented style. While object oriented programming is a great paradigm to model the world of objects interacting with each other, game codebases usually grow large very quickly. The overhead in adding any new class of objects to the game causes the code to become difficult to manage.

Functional programming has become more popular in the recent decade, but is still limited in its applications. Functional programming offers a new way to think about software in general. In a purely functional language like Haskell, we de-emphasize code that modifies data, rather focus on functions that take immutable values as input and produce new values as output. This makes it easier to organize, reason and test the code. (SSG09)

This project investigates the merits of functional programming in Haskell when applied to developing a multiplayer 3D game that is played over the network. There are a few examples of games written in Haskell as proofs of concepts, but very few in 3D. The best 3D example is Frag which depends on the Quake 3 BSP level format. To our best knowledge, Hamster Ball is the first 3D game written in Haskell that is multiplayer. For this project, I upgraded the game Hamster Ball v0.1 to v0.2 and explain the design and implementation of the game.
The first version of the game was written by David Costanzo, Alexander Thompson, Matthew Sills and myself as the final project for a Computer Graphics course in Spring 2009 at Yale University. The game only depends on the Yampa framework, GLFW and OpenGL binding for Haskell. The v0.1 code suffered from some drawbacks, redundant data structures and poor design decisions. In developing v0.2, I reviewed the game design and drastically improved the codebase as well as the stability and playability of the game.

2 Hamster Ball

Besides providing the details of the implementation of Hamster Ball v0.2, this paper also highlights the drastic changes from Hamster Ball v0.1.

2.1 Hamster Ball v0.1

The original version was written as part of the final project of the Spring 2009 Computer Graphics course at Yale University. In the game, each player controls a hamster armed with a 10-terawatt laser and a 100-terawatt protective hamster ball. Mouse-look and standard A-S-D-W controls are used to aim and move. Left mouse-click shoots. The game’s objective is to shoot and destroy other players. Every laser hit a player takes reduces that player’s protective hamster ball strength by the wattage of the laser fired, and when a laser hit takes a hamster’s protection down to 0 strength, the hamster dies a gory death and is respawned at its original start location. The scoreboard keeps track of each player’s achievements by the number of kills and deaths.

There were various bugs and drawbacks with this first version:

- Other clients crash when a client quits out of the game, cutting off connection with the server.
- The game is slow and rather skippy, especially when more than 4 people join the game.
- It only runs on the Zoo machines in the Yale Computer Science lab via Local Area Network.
- The scoreboard only reports a tally of kills so far, but cannot report who killed whom.
- The skill set required is too limited. For example, moving velocity is limited, the mouse’s middle-click and right-click are not utilized, etc.
- Collision detection was buggy.
- The code was not portable.
- We modified Yampa library for our own use. Including the modified Yampa library is also cumbersome.
- There was no documentation on the code.
2.2 Issues with dependencies in Hamster Ball v0.1

The core of Hamster Ball v0.1 is based on Functional Reactive Programming (FRP). (EH97) For this, it uses Yampa, a domain-specific embedded language (DSL) for the programming of hybrid systems along the concepts of FRP. Although Yampa is no longer maintained and is not well optimized to run efficiently, (CNP03) it should be good enough for a game of this scale. However, we had to modify the Yampa library directly and include the source code within our code. One of the challenges in the next version is to get rid of this dependency and use the original Yampa library.

Hamster Ball v0.1 also uses an outdated OpenGL 2.2.3.0 Haskell binding and does not work with the newest OpenGL 2.4.0.1 library.

Another external library used is GLFW module for Haskell which is used for creating OpenGL contexts and managing input, including keyboard, mouse, joystick and time. Although GLFW seems to work better than the GLUT binding that we first used, GLFW doesn’t work well with GHC threads, forkIO and threadDelay. Unfortunately, we used threads and forkIO in our network code. Another challenge in this upcoming rewrite is to separate graphic rendering and network codes more cleanly.

2.3 Major upgrade to Hamster Ball v0.2

Keeping the above limitations in mind, in this project, the updated Hamster Ball game has various improvements:

- The code has been rewritten to use the original Yampa library, and various functions are rewritten to use the Yampa Event library.

- The game is now packaged with cabal and is portable. It runs on Windows, Mac and Linux. It can also be installed via makefile.

- The game allows more flexible arguments to be passed into the game, such as fullscreen mode, screen size, a custom tracker address, etc.

- More game features were also added, which include
  - Powerups are randomly spawned in the game to make it more fun: “eating” these will have temporary effect on hamster ball size or laser power.
  - Mouse-wheel accelerates in the current direction.
  - Right mouse-click stops movement.
  - The scoreboard reports more statistics.

- A server tracker is added. The game is now playable over the internet, not just limited to the Yale Computer Science lab’s local area network.
• The biggest bug of the earlier version was fixed: players can now exit and rejoin the game without causing segfault or affecting other players. The server handles lost network connection gracefully. The game also scales better as more players join the game.

• The more functional code now has 3100 lines of code, including detailed inline documentation, down from the original 3600 lines of code that included virtually no comments. Most of the warning flags were turned on (except for the really unnecessary ones) for v0.2 and consequent warnings were fixed; this helped write more robust code.

3 High level game design

This section describes the high level architecture of the game. The next section will go deeper into the implementation.

The game has two main components, Client and Server. To put it simply, there is a single server and as many clients as the number of players in the game. The client is responsible for rendering the game with the updated game state, one at each player’s computer. The server communicates with many clients and is responsible for maintaining a consistent game state across the clients.

Because client-server communication is done over the network, each client needs to know where the server is and vice versa. In Hamster Ball v0.1, we had to explicitly tell the client the server’s network address upon instantiating any client: the information was shared in a public file on a local area network. In Hamster Ball v0.2, a Server Tracker is introduced to keep track of this online, making this easier, automatic and allowing various multiplayer capabilities.

3.1 Server Tracker

Our server tracker is a web server, always standby, that keeps track of the active game server instances. The tracker, implemented in Ruby, “talks” to servers and clients via a web API.

When a new server is instantiated, it sends an HTTP GET request to inform the server tracker. When a client runs, it sends an HTTP GET request to ask the server tracker for the network address of an active server and attempts to connect to that server. If the connection is unsuccessful, the client asks the server tracker for a new active server; at the same time the server tracker marks the reportedly nonaccessible server as closed.

The chart below presents this communication process between the server and the clients through the Server Tracker.
3.2 Server

A Server starts with two main threads: ACCEPTCLIENT thread and PROCESSGAME thread. A thread-safe, unbounded FIFO channel is used to store and obtain messages for each thread.

The AcceptClient thread spawns a new ListenToClient thread whenever a new client joins the game – the \( n^{th} \) ListenToClient thread listens for new messages from the \( n^{th} \) client responsible for the \( n^{th} \) player.

The ProcessGame thread loops and whenever the message queue is not empty, it aggregates all client messages queued by ListenToClient threads, processes them, and then dispatches respective server messages to each client.

To ensure consistency between the game states in the Server and its connected clients, the ProcessGame thread also loops every \( t \) milliseconds to check on the state of the game: if there is any collision, it sends messages announcing that to all clients.
3.3 Client

A Client has two main threads: MainGame thread and ListenToServer thread.

The ListenToServer thread is spawned by MainGame to listen for messages from the server and queue up the messages waiting to be processed by the MainGame thread.

The MainGame thread processes clear up the messages every 1 millisecond. If there are more than one messages in each iteration, all are applied to the current game state to compute a new game state which is then passed to its internal Yampa react loop, gameSF, which simulates the world in the game. MainGame also does other housekeeping work such as computing number of frames per second, listening for keyboard events.

4 Implementation

The Yampa framework is used to program the game. In version v0.2, I removed a large number of manual functions in favor of Yampa’s library functions where they fit better. Despite being dated, Yampa code comes with extensive test and is reliable.

4.1 Yampa

Yampa is a language embedded in Haskell for describing reactive system. A detailed account is given in the Yampa programming tutorial. (HCNP02) A good implementation of Yampa in the game Space Invaders can be found in (CNP03). Yampa is also introduced in the
implementation of the game Frag in (MC05). Our game makes the most extensive use of Yampa and understanding Yampa is essential to understand how the implementation works.

4.1.1 Signal Functions

Yampa is based on two central concepts: signals and signal functions. A signal is a function from time to a value (and can be thought of as a time-varying value)

\[ Signal \alpha \approx Time \rightarrow \alpha \]

A signal function is a function from Signal to Signal:

\[ SF \alpha \beta \approx Signal \alpha \rightarrow Signal \beta \]

When a value of type \( SF \alpha \beta \) is applied to an input signal of type \( Signal \alpha \), it produces an output signal of type \( Signal \beta \).

4.1.2 Composing signal functions

Yampa provides a library of primitive signal functions and a set of combinators. Yampa’s signal functions are an instance of the arrow framework proposed by Hughes. (H2000)

Three combinators from that framework are

— lifts an ordinary function to a stateless signal function

\[ \text{arr} :: (a \rightarrow b) \rightarrow SF\ a\ b \]

— composition combinators

\[ (<<<) :: SF\ b\ c \rightarrow SF\ a\ b \rightarrow SF\ a\ c \]

\[ (>>>) :: SF\ a\ b \rightarrow SF\ b\ c \rightarrow SF\ a\ c \]

Yampa also provides a the arrow-compose combinator

\[ (^>>>) :: (b\rightarrow c) \rightarrow SF\ a\ b \rightarrow \]

4.1.3 Events and Event Sources

Mouse position in the game is one example where we define as a continuous signal. To model mouse clicks, however, it is more natural to model as discrete events. The Event type, isomorphic to Haskell’s Maybe type, addresses that:

\[ \text{data Event}\ a = \text{NoEvent} \mid \text{Event}\ a \]

A signal function whose output signal is of type \( \text{Event}\ T \) for some type \( T \) is called an event source. The value carried with an event occurrence may be used to carry information about the occurrence. One of the operators provided in the Yampa library is \text{tag}, which associates such a value with an occurrence:

\[ \text{tag} :: \text{Event}\ a \rightarrow b \rightarrow \text{Event}\ b \]
4.1.4 Switching

In our game, the objects of the world evolves over time. Yampa simulates such structural changes via mode switches. There are a family of switching primitives that use events to trigger changes in the connectivity of a system. The simplest such primitive is

\[
\text{switch :: SF a (b, Event c) \to (c\rightarrow SF a b)} \to SF a b
\]

The first argument of switch is the signal function that is initially active. It outputs a pair of signals. The first defines the overall output while the initial signal function is still active. The second signal is the event that will cause the switch to take place. When this event occurs, switch applies its second argument (c \rightarrow SF a b) to the value tagged to the event and switches into the resulting function (SF a b).

Our game uses a more complicated switch \( dpSwitch \), which will be explained later.

4.1.5 Animating Signal Functions

To actually execute a Yampa program we need to connect the program’s input and output signals to the external world (IO). The implementation of Space Invader uses Yampa’s reactimate function, slightly simplified as following:

\[
\text{reactimate :: IO (DTime, a) \rightarrow sense, actuate \rightarrow SF a b \rightarrow IO ()}
\]

The first argument (sense) is an IO action that will return the next input sample paired with the amount of time elapsed (DTime) since the previous sample. The second argument (actuate) is a function that converts an output sample to IO action that will be used to process the output sample. The third argument is the signal function to be animated itself.+

Reactimate simulates the continuous-time model by performing discrete sampling of the signal function, invoking sense at the start and actuate at the end of each time step.

In the context of our game on the client side, we would use sense function to read input event from the window system and network event from client-server communication, and actuate function to render an image on the screen and output the network messages to send back to the game server.

For our implementation, however, we use the functions reactInit, react and actuate. It is useful to base the understanding of this set of functions on the reactimate function above. This is the alternative API provided by Yampa to animate a signal function when some other library needs to own the top-level control flow. In our case, we want the control to yield to graphic and networking routines where we can obtain input samples for each sampling.

reactInit is called once to initialize and returns a ReactHandle which is the reactimate’s state maintained across samples. This gives us more power than sense and also gives us the reactHandle to call react on later.

\[\text{— initialize top-level reaction handle} \]

8
reactInit :: IO a — init
  -> (ReactHandle a b -> Bool -> b -> IO Bool) — actuate
  -> SF a b
  -> IO (ReactHandle a b)

**type** ReactHandle a b = IORef (ReactState a b)

*react* processes a single input sample and is called in a loop to animate the signal function
that was passed in as the third argument to *reactInit*.

We use this *reactInit*, *react* and *actuate* trio in the client code which deals with both
graphic and network inputs, and also in the server code where only network input is relevant.

### 4.2 Game objects in Yampa

Almost all entities in the game are called objects. The terrain (ground and buildings), the
sky, the hamster ball (players) and the lasers. They are reactive to external events and
stimuli, such as collisions with one another and control commands from the player.

One advantage in this development is that individual reactive objects can be developed
and tested in isolation. Wiring separate objects into the whole complex system may require
further refinement but such changes are usually rather straightforward.

#### 4.2.1 The Object type

Similar to the implementation of Space Invaders in the Yampa Arcade paper, (CNP03) we
use the type Object for all game objects

**type** ObjectSF = SF ObjInput ObjOutput

This satisfies the requirement in *dpSwitch* that all signal functions in the dynamic collection
to be maintained by *dpSwitch* have a uniform type. We have the freedom, however, to
define *ObjInput* and *ObjOutput* as rich as we want to implement all necessary interactions
between game objects. Our structure is much more complicated than that of the Yampa
Arcade:

**data** ObjInput = ObjInput {
  oiGameInput :: !GameInput,
  oiColliding :: !(Maybe ObjOutput)
}

**data** ObjOutput = ObjOutput {
  ooObsObjState :: !ObsObjState, — Observable Object State — for OpenGL
    rendering
  ooNetworkMsgs :: ![CSMsg], — Messages to send to the server
  ooKillReq :: !(Event ()), — When this happens, kill the object itself
  ooSpawnReq :: ![ObjectSF], — Spawn these objects
  ooBounds :: !BoundingVolume — For collision detection on the client
    side (player vs player)
}
For graphic rendering, we define the observable object states of type

```haskell
data ObsObjState = OOSPlayer ! Player
  | OOSLaser ! Laser
  | OOSSelf ! Player
  | OOSTerrain ![ TerrainElement ]
  | OOSParticle ! Particle
  | OOSKillText ! String
  | OOSPoweUp ! PowerUp
  | OOSScoreBoard ! ScoreBoard
  | OOSNone
```

deriving Show

The ooSpawnReq and ooKillReq are events for the creation and destruction of game objects. Note that ooSpawnReq itself is not of Event type as in the Yampa Arcade because an empty list would mean the same thing as NoEvent. These two event fields are passed to killOrSpawn by dpSwitch. When a player’s life goes down to zero, it triggers ooKillReq so that the player can be removed from the collection. At the same time, it also triggers ooSpawnReq to add a particle system to simulate explosion. Another use case is when a player shoots a laser, the player object triggers ooSpawnReq to spawn a laser object. Objects such as lasers and explosion debris have their own timer to trigger ooKillReq to be removed from the game.

4.2.2 Maintaining the world of objects

As mentioned earlier, we use dpSwitch to maintain a dynamic collection of game objects.

--- Yampa 0.9.2.3 library, Yampa.hs, line 2613
--- Parallel switch with delayed observation parameterized on the routing function
dpSwitch :: Functor col => -- col is a collection
          (forall sf . (a -> col sf -> col (b, sf)))
          col (SF b c) ->
          SF (a, col c) (Event d) ->
          (col (SF b c) -> d -> SF a (col c)) ->
          SF a (col c)

The first argument is the routing function. Its purpose is to pair up each object (running signal function) in a collection maintained by dpSwitch with the input it is going to see at each point in time.

The second argument is the initial collection of objects (signal functions). In our case, they are a Terrain object which describes the surroundings, a Scoreboard object which has the initial empty statistics, and a ServerObject object that is responsible for spawning players on the screen whenever there is a network message from the server indicating players joining the game. It also spawns random releases of powerups which can change a player’s shape or boost its energy.
The third argument is a signal function that observes the external input signal and the output signals from the collection in order to produce a switching event. We pass \texttt{killOrSpawn} into this. \texttt{killOrSpawn} traverses the output from the game objects, collecting all kill and spawn events. If any event occurs, a switching event is generated that carries a function to update the signal function collection accordingly.

The fourth argument is a function that is invoked when the switching event occurs, yielding a new signal function to switch into based on the collection of signal functions previously running and the value carried by the switching event. This allows the collection to be updated and then switched back in by calling \texttt{dpSwitch} again.

The implementation of \texttt{dpSwitch} is almost identical to that in the Yampa Arcade, thus I will skip the rest of the details here. A more detailed explanation on \texttt{dpSwitch} can be found in (CNP03).

### 4.3 Server-Client message queue

As illustrated on the high level diagram of the server in Section 3.2 above, we need a thread-safe message queue for the server to handle client’s network messages in a parallel fashion. In each time step, the server needs to consolidate all the happenings reported from the clients. Instead of queuing the game states, we queue the function that modifies a game state as input. This illustrates the advantage of a functional approach:

```haskell
type ReactChan a = Chan (a -> a)

addToReact :: ReactChan a -> (a -> a) -> IO ()
addToReact rch f = writeChan rch f

getReactInput :: ReactChan a -> a -> IO a
getReactInput rch old = do
  f <- readChan rch
  return $ f old

Server's loop
```

#### 4.3.1 Message processing on the Server

The Server and Client modules is one aspect of the implementation of Hamster Ball that sets itself apart from the Yampa Arcade and Frag examples, where everything is done at the client side because there is no network communication involved.

```haskell
runServer :: PortID -> SF ServerInput (IO()) -> IO ()
runServer port serverSF = withSocketsDo $ do
  — open port
  sock <- listenOn port
  — initialize message queue
```
rch ← newChan

— spawn AcceptClient thread to listen for new players joining
forkIO $ acceptClient rch sock

— write 'id' to chan once in a while to keep the message queue non-empty.
— so that server keeps updating its state and notifies the clients
  if it detects any collision
forkIO $ do
  let loop = do
    addToReact rch id
    threadDelay 1000 — Microseconds
    loop

— initialize Yampa's top-level reaction handle
rh ← reactInit (return dummyServerInput) (_ _ sendmsgs -> sendmsgs
  >> return False) serverSF

startTime ← getCurrentTime
— main thread process.
— this readChan/unGetChan makes it more efficient (server idle
  instead of looping)
let loop lTime lastA = do
  newA ← getReactInput rch lastA
  curTime ← getCurrentTime
  — animate the next time step, passing in the amount of time
  elapsed to updated new positions of objects
  react rh (fromRational . toRational $ diffUTCTime curTime
    lTime, Just newA)
  — loop until the queue is empty
  loop curTime newA
loop startTime dummyServerInput

where acceptClient rch sock = do
  (hand, _) ← accept sock
  open ← hIsOpen hand
  printFlush ("Accepting, _verify_opened:_" ++ show open)
— fork a ListenToClient thread for each connecting client
forkIO $ do
  let loop = do
    succ ← hWaitForInput hand (-1)
    when succ $ fetchCSMsg rch hand
    loop
    — When player quits, handle becomes invalid (closed by
      main thread), thus exception thrown
    catch loop (\e -> print "Player quit."
      acceptClient rch sock

12
4.3.2 Message processing on the Client

While the server code only handles network messages as input and outputs messages, the client needs to handle both network messages and game controls from mouse and keyboard as input and output both network messages and graphic rendering routines. However, the same approach is used to animate the signal functions:

---

excerpts of function runGame in RunGame.hs, some variables renamed for expressiveness

runGame :: GameConfig -> Maybe Handle -> SF GameInput (IO (), IO ()) -> IO ()
runGame gameConfig networkHandle gameSF = do
  -- initialize current time, drawing time, callback functions handling mouse, mouse wheel, keyboard and window closing
  ...
  -- initialize message queue, see Control.Concurrent.Chan
  rch <- newChan

  -- connect to the server through networkHandle and
  -- spawn ListenToServer thread to update rch
  networkInit rch networkHandle

  -- initialize Yampa’s top-level reaction handle
  rh <- reactInit (return initGameInput) actuateFunction gameSF

  -- process the game by applying Yampa’s react repeatedly until the message queue is empty
  loop rh rch quit startTime initGameInput

  -- finish up GLFW
  closeWindow
  terminate

where

loop rh rch quit lTime curA = do
  sleep 0.001
  pollEvents
  curTime <- getCurrentTime
  let reactLoop cA = do
        empty <- isEmptyChan rch
        let dt = fromRational.toRational $ diffUTCTime curTime lTime
            if empty
              then (react rh (dt, Nothing) >>= return cA)
              else (do
                      newA <- getReactInput rch cA
                      react rh (dt, Just newA)
                      reactLoop newA)
    nA <- reactLoop curA
  q <- readIORef quit
  unless q $ loop rh rch quit curTime nA
4.4 Server’s implementation

The second argument of runServer in the listing above, serverSF, is of type SF ServerInput (IO()) and is responsible for the processing input messages, updating the internal game state, and outputing network messages to clients.

```
data ServerInput = ServerInput { msg :: !CSMsg, handle :: !(Maybe Handle) } — handle of the client deriving (Show, Eq)

server :: SF ServerInput (IO())
server = proc si -> do
  msgs <- loopPre (emptyServerState, emptyServerState) objSF <- si
  return A <- msgs
```

A useful way to explain signal function is through diagrams:

![Diagram](image)

4.5 Client’s implementation

Similar to the role of serverSF as an argument to runServer above, on the client side, gameSF, the third argument of runGame, is responsible for the internal processing.

```
data GameInput = GameInput { key :: Maybe Key, keyState :: Maybe KeyButtonState, leftClick :: Bool, mWheel :: Int, posMouse :: Position, rightClick :: Bool, message :: SCMsg }
```

The overall structure can be represented as follows:
**renderObsObjStates** function is used to translate the object information to 3D drawings, each object has its own rendering function

\[
\begin{align*}
\text{renderObsObjState} &:: \text{ObsObjState} \rightarrow \text{IO} () \\
\text{renderObsObjState} (\text{OOSPlayer p}) &= \text{renderPlayer p} \\
\text{renderObsObjState} (\text{OOSLaser l}) &= \text{renderLaser l} \\
\text{renderObsObjState} (\text{OOSSelf p}) &= \text{renderSelf p} \\
\text{renderObsObjState} (\text{OOSTerrain ts}) &= (\text{foldr} \ (>>) \ (\text{return} () ) \ $ \map \ \\
\text{renderTerrainElement ts}) \\
\text{renderObsObjState} (\text{OOSParticle p}) &= \text{renderParticle p} \\
\text{renderObsObjState} (\text{OOSKillText str}) &= \text{renderKillText str} \\
\text{renderObsObjState} (\text{OSScoreBoard sb}) &= \text{renderScoreBoard sb} \\
\text{renderObsObjState} (\text{OOSPowerrUp pow}) &= \text{renderPowerUp pow} \\
\text{renderObsObjState} \text{OOSNone} &= \text{return} ()
\end{align*}
\]

**sendNetworkMsgs** function serializes the client to server messages to string that can be sent over the network. This is a separate module and can be replaced with one that sends binary data instead of string because string comparisons are expensive, especially when this is done every millisecond. However, I found that the game performance is acceptable in v0.2 and decided that serializing to string makes the code more readable and maintainable, especially useful when we need to add more object states to the game.

5 Discussion

This section discusses the result and lessons learned. We will address the choice of Haskell the language, the frameworks, the aspects of functional programming and non functional programming in the design and implementation of the game Hamster Ball, both v0.1 and v0.2.
5.1 Haskell

My teammates and I picked Haskell to implement Hamster Ball v0.1 as a challenge and an opportunity to explore Haskell’s possibilities. In a few weeks, we were able to implement a sufficiently complicated game using some of the self developed libraries. If we had decided to use C++ while using as few external C++ libraries as possible, we believed we would not have accomplished as much.

In this project, one great advantage of Haskell is the ease of refactoring. Naturally, it’s impossible to make all best implementation decisions, especially in the first version. In fact, later in the development of v0.1 we became increasingly aware of some costly mistakes. In some other areas, we wrote something we felt uncomfortable but did it anyway due to the lack of a better solution. We were more feature-driven than perfectionist. In fact, we settled with “it works so that’s fine for now!” and “the code is expressive enough, no need for writing comments!” mentality.

Haskell really shone in this project when I tore the v0.1 code apart and refactored a large chunk of it to use more elegant data types and easily add more features. Haskell’s strict syntax and type checking helps reduce programming errors, both syntactical and logic ones. I owe it to Haskell’s type signatures and pattern matching that help me write more readable and less error prone code. The code was also maintained with a purpose of making it open source. I added documentation generously to aid any reader, including commit messages in the git version control system. All these cleanups turned out to make it easier for me to quickly implement more features later on.

The main setback with Haskell is the lack of available libraries and frameworks. OpenGL and GLFW binding for Haskell make it convenient to translate the knowledge of OpenGL from the object oriented C++ to Haskell. However, not all functions are ported and the documentation is incomplete, resulting in a waste of time looking these functions up. From the purely functional programming standpoint, the OpenGL and GLFW codes are the ugliest part because they are imperative. We resort to using variables (in the form of IORefs) and callback functions, which we are familiar with in imperative programming.

5.2 Yampa

Yampa utilizes great aspects of functional programming on top of Haskell. First, as noted by (CNP03), signal functions provide the programmer with a natural way to factor the game development task. Furthermore, being able to test the signal functions separately before composing them is a big win for an incremental approach. This also enabled effective collaboration between my teammates back in v0.1.

Another aspect that promotes purely functional programming practice is that signal functions can accumulate internal state, protected from input and output signals. This modularity promotes de-coupling and extensibility. For example, when a player passes through a powerup object, it temporarily shrinks its size (making it easier to dodge lasers). The player object itself maintains its internal state to return to normal. The overall game state does not have store that information.
Its events and switching combinators allow event driven programming which is suitable for game programming.

Yampa’s reactimate API for animating signal functions encourages the programmer to think about the design of the signal functions more carefully arrive at good sense and actuate function for reactimate.

Yampa’s switches provide a powerful abstraction to simulate objects and update them in parallel.

One of the known setbacks of Yampa is its performance issues. However, for a game of small scale, our game is not affected.

5.3 Future work

I am interested in further work on Hamster Ball to improve the structuring of the game and add features. Below are some of the potential areas to clean up or explore:

- Move all collision code to the server: Currently, collision detection is done partly on the server side and partly on the client side. The server detects collision between players/players and players/lasers and leave collection detection between objects with terran (surrounding, static objects) done on the client side.

- Refactor the reactInit, react, and actuate animating API to use the modular reactimate taking sense and actuate arguments instead. It is visible from the code excerpts of runServer and runGame functions that there are similar codes in the animating loop. Forcing Yampa to yield the top level control as it is currently done makes the code less modular.

- Bypass game control from client-side processing and handle it on the server instead. Currently, keyboard and mouse inputs are translated into network messages and sent from a client to the server which announces to other clients. It is more efficient to have the server handle all clients’ game control, consolidate the new game state and announces to all clients.

- Benchmark to see how playable the game is. I experimented with GHC’s Profiling options but did not get too far. This is the only way to conclude where the bottleneck of the game is.

6 Conclusion

Before this project, the existence of Hamster Ball v0.1 proved that it is possible and actually fun to implement a 3D network game in Haskell. Through this project, many lessons were learned in the process of refactoring and upgrading the game Hamster Ball from v0.1 to v0.2. It strongly suggests that game programming in Haskell is highly maintainable.
I took advantage of Haskell’s expressive data types and it is clear that this is another advantage that Haskell brings to game development.

With OpenGL and GLFW, producing 3D graphics for a first person shooter game is not difficult. However, the OpenGL code is inherently imperative, especially via the use of callbacks and state vars.

Yampa’s embedded syntax does not read functional but its use of encapsulation takes advantage of Haskell’s arrow syntax to keep things modular and extensible. Once familiar with Yampa, I found it an effective and readable framework to program objects with signature functions and wire them together with an event-driven game engine.

In Hamster Ball v0.2, I first got rid of the dependency on the modified Yampa’s library that v0.1 required. This way, Hamster Ball v0.2 can immediately benefit from any newer release of Yampa. I also removed many custom functions in favor of similar Yampa’s Event library functions and that resulted in much less code.

Overall, this game is a good proof of concept on the potential of Functional Reactive Programming in general and Yampa in particular in modelling a dynamic, reactive system like in a multiplayer 3D first-person shooter game.

7 Acknowledgements

I would like to express my heartfelt appreciation to professor Paul Hudak, who introduced me to Haskell in his two exhilarating Computer Music courses and patiently advised me on this project. I am also grateful to Paul Liu and Eric Cheng for their help with Haskell in particular.

Thanks to my teammates David Costanzo, Alex Thompson, and Matt Sills for the great teamwork on Hamster Balls v0.1 in Spring 2009.

The original project Hamsters Ball v0.1 would not have been possible without professor Julie Dorsey’s Spring 2009 Computer Graphics course, in which she permitted us to use Haskell instead of C++ as required by the syllabus.

Last but not least, thanks to professor Dana Angluin and professor Stan Eisenstat for their guidance over these four years.

References


[EH97] Conal Elliott and Paul Hudak, in ICFP 97. Functional Reactive Animation

