The University of New South Wales
School of Computer Science and Engineering

# Functional Programming and 3D Games

Mun Hon Cheong (3063000)
Bachelor of Engineering(Computer Engineering)
November 2005

Supervisor : Dr Manuel Chakravarty
Assessor : Ken Robinson

# Abstract

Games are commonly programmed in imperative languages. Functional languages have been known to have benefits but have rarely been used to program games.

In this thesis we implement a first person shooting game in Haskell and Yampa. The merits of this approach are examined.

# Acknowledgements

Big thanks goes to Sean Seefried who I frequently turn to for programming and writing advice.

Thanks to Dr Manual Chakravarty, who allowed me to do a thesis that is game related, which is something I have great interest in, and gave me help and advice

Thanks to Don Stewart and André Pang for the help they offered.

Finally, big, big, thanks goes to the community who write game related programming tutorials. Their tutorials that tackle topics from Alpha testing to Z-buffers helped me write this game.

# Table of Contents

# 1. Introduction

The computer gaming industry began in the 1970s with Pong, and has grown with the progress of computing technology into a billion-dollar industry. [1]

Todays commercial games are sophisticated pieces of software and may be written in hundreds of thousands of lines of code. Most commercial games require one to three years to develop in contrast to the development cycle typical of games in past. Most of the development cycle involves initial programming and then lengthy testing and changes to the initial code. [2]

Many game developers are concerned with the length of game development cycles, as longer game development cycles mean higher costs and a longer period before there is a return on investment.

Recent advances in computing have seen functional languages lead to better productivity in many industries. Ericsson have used a home-grown FPL, Erlang Language to build large telecom systems. In certain tests, they claimed to have measured improvements in productivity between 9 and 25 times greater.[3]

It is plausible the video and computer gaming industry may also benefit from the use of functional languages. Functional programming languages offer many advantages compared with the imperative languages that are widely used in this industry.

Functional programs are much more concise when compared with imperative programs. They allow for the use of powerful abstractions which can be used to improve structure and modularity of code. Functional languages also allow for polymorphism which promotes the reuse of code and less redundancy in programs.[3]

An important aspect of game development is the gameplay. In simple terms gameplay means how a game is played. Many games fail to sell well because the way they were designed to be played does not appeal to the consumer. The prototyping of a game is vital when trying to get a third party to fund or distribute a game before it is completed.

Game developers have to verify whether their ideas are viable through the playtesting of the prototype before they continue to code the game. Unfortunately this may be time consuming. With functional languages, an executable specification for a game may be written and playtesting can be completed in less time.

The potential benefits of functional languages when applied to game development form the motivation of this thesis project. It is hoped that this research will help in reducing the problems that game developers currently face and increase productivity.

## 1.1 Goals

The goal of this thesis is to program a 3D game in Haskell. Also, Yampa - an embedded DSL for modeling hybrid systems is used to program the games objects. The games graphics are programmed with HOpenGL, a Haskell binding to the OpenGL graphics library.

The genre of the game is first person shooter. A first-person shooter is a shooting game where the player's view of the game world is exactly that of the character the player assumes. The player explores the game world and shoots at objects.

The performance of the game is benchmarked and the languages used to program this game are evaluated.

## 1.2 Overview of the Thesis

The thesis is divided into the following:

| | |
|---|---|
| Background - | This chapter explains concepts used in FRPs, examines implementations of games and provides an introduction to Yampa |
| Implementation - | The design and implementation of the game is detailed here. Examples and explanations of how Yampa's signal functions were used in the game are given. |
| Benchmarks - | The performance of our game is measured here |
| Discussion - | The use of Yampa and Haskell for programming our game is evaluated here. |
| Conclusion - | "Wraps up" the thesis |

# 2. Background

Functional Reactive Programming applied to games will be a key area of research for this thesis project. DSLs will be explained briefly, examples in Fran[4] will be used to introduce key concepts in FRP.

## 2.1 DSLs

DSL is an acronym for Domain Specific Language. DSLs are programming languages tailored for use in a specific application domain. DSLs provide useful notations and abstractions to simplify programming in an application domain. Programs written with a DSL are more concise and readable than those written with general-purpose languages.

Another important aspect of DSLs is they are more declarative than they are imperative. With DSLs the focus is on specifying what something is, rather than the steps needed to do something. More examples of this concept will be seen later in this report.

Embedded Domain Specific Languages, are DSLs that are embedded into a host programming language. The advantage is the DSL would be able to inherit the properties of its host language and the task creating a new language from scratch is avoided.

### 2.1.2 Functional Reactive programming

Functional Reactive Programming was first manifested in Fran, an embedded domain specific language (EDSL) for graphics and animation developed by Conal Elliott at Microsoft Research. Fran is a high-level language for modeling reactive animations. FAL[5], Frob[6], Fvision[7], Fruit[8] and Yampa[9] are four other examples of Functional Reactive Programming that were developed for use in particular application domains.

### 2.1.3 Fran

As mentioned previously, Fran is embedded in Haskell and inherits its properties such as laziness. Technically Fran is a library of functions in Haskell but its domain specific abstractions and special notations disguises that fact and makes Fran seem like a language of its own.

The syntax used in Fran is simpler and easier to understand compared with other implementations of FRP. Thus examples of code written in Fran will be used as examples to explain the basic concepts of FRP.

#### 2.1.3.1 Composability of Functions

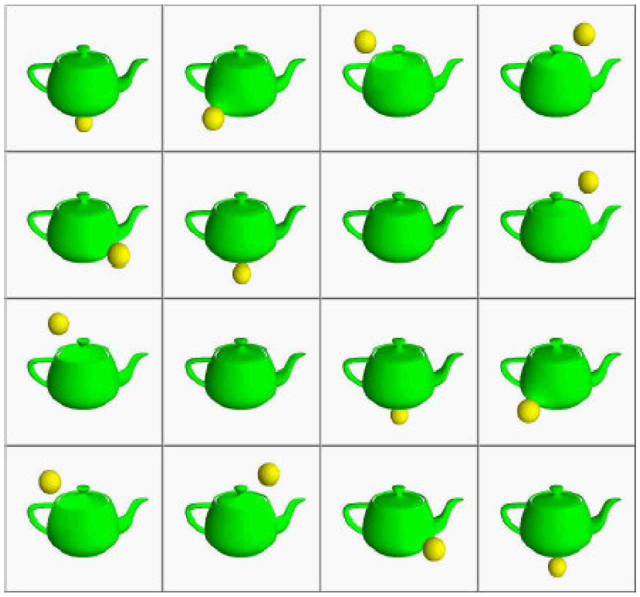

Figure 2.1: An animation of a pot being circled by a light. Figure taken from [4].

In Fran the above animation is declared with the following line:

```
potAndLight = withColorG green teapot ‘unionG‘ movingLight
```

Listing 2.1: The code for the animation in figure 2.1. Code taken from [4].

`potAndLight` is declared as the composition of a green teapot with a moving light. `unionG` is an infix function used to compose two animations together. The resulting animation – `potAndLight` is also of the same type as the animations used to compose it, thus it can be composed with other animations.

In FRP domain specific functions can be composed together which allows powerful expressions to be built.

### 2.1.3.2 Continuous time semantics

Code used to create the animation in figure 2.1 is shown here. The motion of the yellow light is described concisely with a time varying spherical coordinate.

The line, `vector3Spherical 1.5 (pi*time)(2*pi*time)`, specifies the light will be 1.5 units from the center of the teapot, while its latitude will be twice its longitude.

```
 (**%) :: Transform3B -> GeometryB-> GeometryB

movingLight =
      translate3 motion **%
      uscale3 0.1       **%
      withColorG yellow (sphereLowRes ‘unionG‘ pointLightG)
      where
        motion =  vector3Spherical 1.5 (pi*time)(2*pi*time)
```

Listing 2.2: The code for the moving light in figure 2.1. Code taken from [4].

Games make use of physical equations that use integrals with respect to time, such as those for acceleration and velocity. FRP provides a concise way of expressing these equations.

A notable detail is the use of the "`**%`" operator. It is an example of the special notations found in DSLs. It is implemented as a higher order function in Haskell.

### 2.1.3.3 Encapsulation and Abstraction

The behavior of the moving light in figure 2.1 is encapsulated in the function `movingLight`. Unlike regular functions, which take a parameter and return a value, no parameters are required in `movingLight`, the function can be used in an expression like a value. Also, the state of the function is encapsulated, it does not require the current state of the function to be used as a parameter to calculate the next iteration.

`potAndLight` and `movingLight` are functions that are written in a declarative style as opposed to an imperative style. The operations needed to perform the transformations are abstracted. Programming animations in an imperative language would require these operations be defined.

### 2.1.3.4 Reactivity and Events

```
grow :: User -> RealB
grow u = integral (bSign u)

bSign :: User -> RealB
bSign u = selectLeftRight 0 (-1) 1 u

selectLeftRight :: a -> a -> a -> User-> Behavior a
selectLeftRight none left right u =
       ifB (leftButton u)
            (constantB left)
                  (ifB (rightButton u)
                   (constantB right)
                   (constantB none).


spin2 :: User -> ImageB
spin2 = withSpin potSpin1
   withSpin f u = growHowTo u ‘over‘
      renderGeometry (f (grow u) u) defaultCamera

potSpin1 angle u = spinPot red angle

spinPot :: ColorB -> RealB -> GeometryB
spinPot potColor potAngle =
   rotate3 zVector3 potAngle **% withColorG potColor teapot
```

Listing 2.3: Code for the spinning teapot in figure 2.2. Code taken from [4].

In FRP, behaviors can change in response to events.

`bSign` creates a stream of values over time, based on the users input. The `constantB` operator used in `selectLeftRight` is used to repeat the discrete values that are arguments to `selectLeftRight`. Depending on the mouse button that is pressed, `selectLeftRight` will continuously generate the value `1` or `-1`. If neither are pressed a stream of 0s will be generated. This is an example of a behavior that responds to events.

The function `integral` is an example of a stateful function used in FRP. It sums the values generated by `bSign`. The ‘reactive’ value from `integral`, is used as an argument to `spin2` that rotates the teapot horizontally.

Aspects of a game that are event driven, such as artificial intelligence, can be modeled with reactive behaviors.

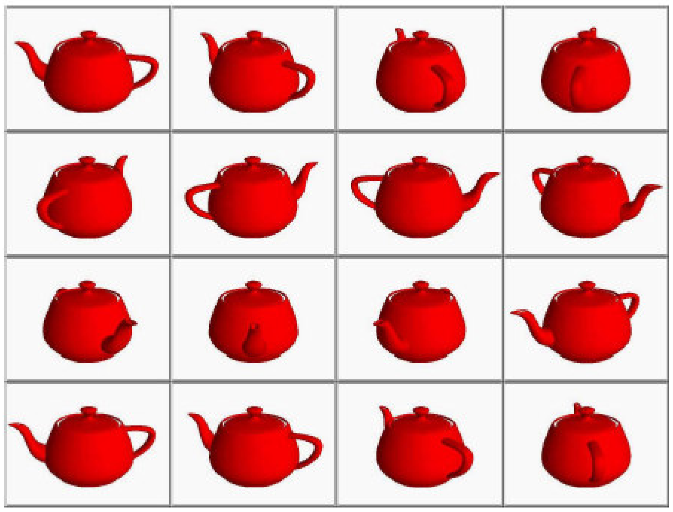
Figure 2.2: The resulting teapot from listing 2.3. Figure taken from [4].

## 2.2 Games Implemented with FRP

Paul Hudak implemented PaddleBall with 17 lines of code in FAL, a language similar to Fran[5]. One may speculate that Functional Reactive Programming may allow for games to be developed with less lines of code and less time compared with other languages.

However, initially there were some problems in programming more complex games with the older implementations of Functional Reactive Languages such as Fran, Fal and Fruit.

Antony Courtney and Henrik Nilsson and John Peterson addressed these problems in the paper “The Yampa Arcade” [9]. The introduction of this paper details the difficulties of implementing a Space Invaders like game with Functional Reactive Programming.

> “This paper was inspired by some gentle taunting on the Haskell GUI list by George Russell: I have to say I’m very sceptical about things like Fruit which rely on reactive animation, ever since I set our students an exercise implementing a simple space-invaders game in such a system, and had no end of a job producing an example solution. Things like getting an alien spaceship to move slowly downward, moving randomly to the left and right, and bouncing o. the walls, turned out to be a major headache. Also I think I had to use ”error” to get the message out to the outside world the aliens had won. My suspicion is that reactive animation works very nicely for the
> examples constructed by reactive animation folk, but not for my examples.[10]

Two problems were identified in that paper. The first problem was there was no support for switching over dynamic collections of reactive objects in the earlier FRP incarnations. This was needed so reactive game objects could be added or removed. The other problem that was identified was the lack of examples of FRP code for more complex games. This problem was also addressed in the paper by providing the code for a working Space Invaders-like game.

## 2.2.1 Yampa

In “the Yampa Arcade” the task of switching over dynamic collections of reactive entities was handled with Yampas delayed parallel switching functions[9].

```
dpSwitch :: Functor col =>
 (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)
```

Listing 2.4: The type signature for dPswitch, Yampas delayed parallel switcher

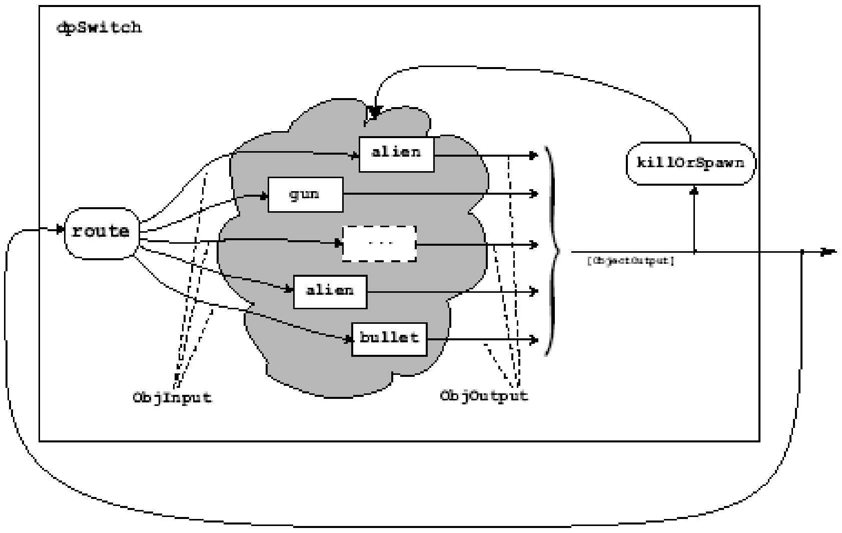

Figure 2.3: The structure of the Space Invaders game in the Yampa Arcade Paper.
Picture taken from [9]

`dpSwitch` is a function that is used to model the main loop found in games. The first and second arguments are the routing function and the initial collection of reactive objects respectively. The third argument is a function that updates the collection of objects in response to events generated from the object. The final argument is the continuation of the game after a parallel switch has been performed.

Every object in the collection is paired with their inputs by the routing function. The killandspawn function observes the output of the objects after input had been applied to them. When an object had to be added or removed it would produce a switching event that invokes the fourth argument, which is the continuation of the game. When the function is invoked, the state of running behaviors is preserved, the collection is updated and the game is resumed.

```
killOrSpawn :: (a, IL ObjOutput) -> (Event (IL Object -> IL Object))
```

Listing 2.5: The function killOrSpawn for updating the collection of game objects

The way events are distributed in this game is specified explicitly with a routing function. The advantage of this approach is that unexpected behaviors are prevented from occurring as game objects can only "see" values the programmer wants them to.

```
route :: (GameInput, IL ObjOutput) -> IL sf -> IL (ObjInput, sf)
```

Listing 2.6: The function route for distribution of input among objects

Also, in this implementation each game objects state is encapsulated in the objects code, instead of being a part of a monolithic game state. Each game object could be coded in separate modules and could be tested in isolation. Coding of the objects in the game could be performed incrementally. Also, encapsulation lessens the likelihood of errors such as reading another objects state by mistake.

### 2.2.2 FranTK

FranTk[11], a library for building GUIs provided functionality similar to Yampas dpSwitch.

FranTks Bvars, which are mutable variables that use IO, represent the state of separate GUI objects. Behaviors and events can be extracted from Bvars. Frantk allows for collections of behaviors with ListBvars

```
data BVar a

newListBVar :: [a] -> IO (ListVar a)

insertSetB :: SetBVar a -> Listener a
deleteSetB :: SetBVar a -> Listener a
resetSetB  :: SetBVar a -> Listener [a]
```

Listing 2.7: Bvars, ListVars and functions to update sets of bVars

Bvars are updated with listeners. A listener, is a function, that performs an IO action with the values passed to it. Each listener refers to a Bvar. The events that a listener responds to are defined explicitly by the programmer. Whenever an event occurs the Listeners associated with that event will update their Bvar.

insertSetB, deleteSetB allow behaviors to be removed from the collection and resetSetB allows for the collection of behaviors to be updated. Game objects can be modeled as behaviors and stored in a ListBVar which are updated with these functions.

This is an alternative method of handling dynamic collections of reactive objects, that uses Haskells IO monad.

## 2.3 Non FRP Implementations of Games

In Luths[12] implementation of an asteroids type game there was a monolithic data structure that contained the state of all objects within the game.

```
data State =
   State { bullets   :: [Bullet],
           asteroids :: [Asteroid],
           ship      :: Ship
         }
```

Listing 2.8:  The state datatype for Luth's Asteroids game

Clean, a language similar to Haskell, provided a library for programming platform games, called the Clean Game Library[13]. Every game object had access to a globalised mutable game state that was updated with I/O callbacks.

Having a large monolithic state is enough for simple games. But as the number of different objects increases, it becomes a chore to add more fields to the monolithic datatype so new types of game objects can be accommodated.

With a globalised mutable game state, it may be difficult eliminate bugs if they occur, as any object can access and change this state variable in unpredictable ways.

With Yampa each game object updates its own state information. There is no state data type that has to be changed to accommodate new objects. Also, separate objects can be implemented and tested in isolation before they are used in combination with other game objects. This method of incremental coding and testing will result in less time spent on debugging and more concrete code.

Also in the Space-Invaders example, operations such as collision detection, updating of the collection of game events and obtaining user input are abstracted from the objects. The game objects do not know how these operations are performed, there are no libraries of functions to call that would allow them to infer how they are performed, this allows code to be more maintainable.

## 2.4 Introduction to Yampa

Yampa will be used to program the game. There was enough documentation and examples of code in Yampa. Most importantly, a game – a clone of Space Invaders, had been successfully implemented in Yampa and was more sophisticated compared with previous games implemented in FRP.

This is a brief introduction to some of the Yampa syntax that is going to be used in the game.

### 2.4.1 Signal Functions

Yampa is based on continuous concepts such as signals and signal functions. A signal is a time-varying value. In other implementations of FRP these are called behaviors.

```
Signal a  ~ Time -> a
```

Listing 2.9: Signals are time varying values

A signal function is a function that transforms a signal into another signal.

```
SF a b ~ signal a -> signal b
arr :: (a -> b) -> SF a b
```

Listing 2.10: Signal functions and `arr`

The function `arr` lifts functions of type `(a -> b)` into stateless signal functions. These can be composed with other signal functions.

### 2.4.2 Signal function composition.

```
(<<<) :: SF b c -> SF a b -> SF a c
```

Listing 2.11: The signal function composition operator `(<<<)`

`(<<<)` is similar to Haskells composition operator `(.)`, it allows the pipelining of the output of one signal function, to the input of another signal function, resulting in a new function.

### 2.4.3 Arrow syntax

Arrow syntax is syntactic sugaring that allows signal functions to be programmed with less wiring combinators so readability is improved.

```
xSF :: SF SimbotInput Distance
xSF = let v = (vrSF &&& vlSF) >>> arr2 (+)
          t = thetaSF >>> arr cos
      in (v &&& t) >>> arr2 (*) >>> integral >>> arr (/2)
```

Listing 2.12: Signal functions written without arrow syntax. Code taken from [19]

```
proc pat -> do
      pat 1 <- sfexp 1 -< exp 1
      pat 2 <- sfexp 2 -< exp 2
      ...
      pat n <- sfexp n -< exp n
      returnA -< exp
```

Listing 2.13: Arrow Syntax. Code taken from [19]

In listing 2.13, `proc` is a keyword that is similar to "\" used in lambda expressions in Haskell. This is followed by

```
pat <- sfexp -< exp
```

Listing 2.14: A single line of arrow syntax. that binds a name to the output of a signal function

`sfexp` can only be a signal function while `exp` can be any Haskell expression. `pat` is used to name the output of the signal function. The code in listing 2.14 is similar to a Haskell let or where clause, where if `expr1` is of type `T1 -> T2`, then `expr2` must have type `T1` and `pat` will be of the type `T2`.

```
let pat = expr1 expr2
```

. Listing 2.15: The similarity between `pats` and `let` statements

Named output can only be used in an expression that follows it and cannot be used with `sfexpr`. Let statements such as `let a = b` can also be used with arrow syntax.

### 2.4.4 Discrete Events

Events are modeled in Yampa as discrete occurrences the time. Events can be "tagged" with information to associate a value with the occurrence.

```
data Event a = NoEvent | Event a

tag :: Event a -> b -> Event b

edge :: SF Bool (Event ())
```

Listing 2.16: discrete events in Yampa, `tag` and `edge`

`edge` is a rising edge detector, it is used as an event source, that produces an event the moment a condition is satisfied.

### 2.4.5 Switching Combinators

```
rSwitch :: SF a b -> SF (a,Event (SF a b)) b
```

Listing 2.17: `rSwitch`

Switching combinators are used to select between behaviors in response to events. The first argument to this function is the initial behavior of the switcher. Should an event occur the switcher assumes the behavior that is "tagged" to the event.

### 2.4.6 Integrals

```
integral :: VectorSpace a s => SF a a
```

Listing 2.18: `integral`

`integral` integrates with respect to time. The amount of time that has elapsed does not have to be provided, as the flow of time is abstracted.

# 3. Design and Implementation

This chapter, details the design and implementation of the game. The structure of the game is similar to that of the Space Invaders game detailed in the background section. Code for the game can be divided into 3 main sections:

- The main loop and the game facilities
- The game objects
- Graphics

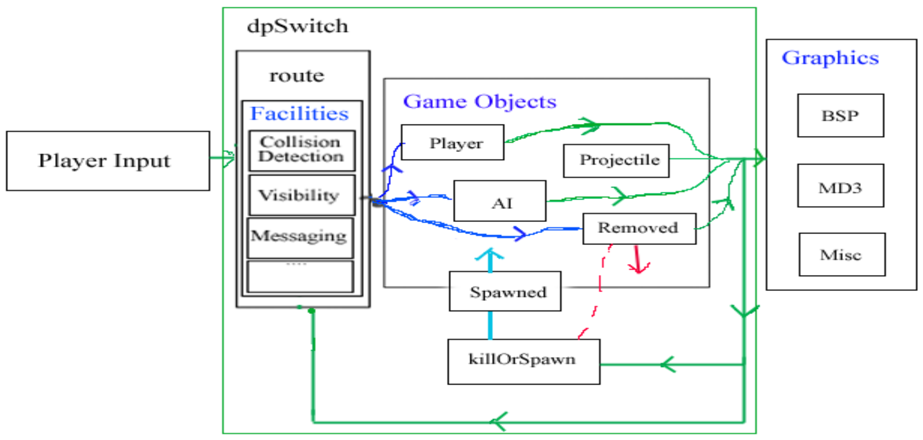

Figure 3.1: Structure of the game

## 3.1 Programming the main loop

In figure 3.1, the main loop obtains player input, then functions in the body of the loop use the previous game state and player input to obtain the next game state. After the game state is rendered, the player response is sampled, and the main loop begins its next iteration.

As detailed in the paper "the Yampa Arcade", Yampas delayed parallel switcher, `dpSwitch`, is used to program our main loop. The only difference is an extra argument which is the BSP map. `route` is used to distribute input among game objects and `killOrSpawn` updates the collection of game objects.

```
game :: IL Object -> BSPMap ->
   SF (GameInput, IL ObjOutput) (IL ObjOutput)
game objs bspmap = dpSwitch (route bspmap)
                            objs
                            (noEvent --> arr killOrSpawn)
                            (\sfs' f -> game (f sfs') bspmap)
```

Listing 3.1: The main loop implemented with `dpSwitch`

## 3.2 Game Facilities

This section describes the various facilities provided to our objects. As mentioned previously, the function `route`, which is a parameter of `dpswitch`, is used to pair each object with input. The `ObjInput` data type is composed of various events, player input and results from collision detection tests, that are returned from the games facilities.

```
route :: BSPMap->(GameInput,IL ObjOutput)->IL sf ->IL(ObjInput, sf)

data ObjInput = ObjInput {
       oiHit            :: !(Event [(ILKey, ObsObjState)]),
       oiMessage        :: !(Event [(ILKey, Message)]),
       oiCollision      :: !Camera,
       oiCollisionPos   :: !(Double,Double,Double),
       oiOnLand         :: !Bool,
       oiGameInput      :: !GameInput,
       oiVisibleObjs    :: !(Event [(ILKey,ObsObjState)])
     }
```

Listing 3.2: `route` and `ObjInput`

### 3.2.1 Messaging

Messaging allows objects to communicate with one another. These messages are collected and distributed among objects by `route`.

Game objects are stored in an identity list. The identity list associates each object with a unique key. An object can gain another objects key through collision events, visibility detection events or from a message.

```
type ILKey = Int
data IL a = IL { ilNextKey :: ILKey, ilAssocs :: [(ILKey, a)] }

data Message = Coord !(Double,Double,Double) |
                  PlayerLockedOn |
                  TargetPosition !(Double,Double,Double) |
                  EnemyDown
```

Listing 3.3: The identity list and the `Message` data type

### 3.2.2 Collision detection

Collision detection is important in any game. It is used determine the response of an object to its environment and other objects. Every object is tested for collisions with level geometry and other objects.

#### Collision detection with level geometry

The BSP data structure allows us to perform collision detection efficiently [15].

The position of an object, its bounding volume and its movement vector are parameters to functions that test for collisions with level geometry. The bounding volume may be a sphere or an axis-aligned bounding box.

Collision detection functions test for collisions and return the position where the collision occurred. This is used to correct the position of the object, so the objects movement is bounded by the geometry of the level.

Objects react differently in response to collisions. Separate functions take this into account when returning correcting the objects position. A position is that matches the collision response of the player is returned. For example, the player object uses a collision detection function that allows it to move over steps and slide against walls and floors. Whereas the collision detection function for a projectile just returns the point where it collides with the level.

#### Collision detection between objects

Collision detection can be performed on different combinations of bounding volumes, these include axis-aligned bounding boxes, rays and spheres.

Whenever a collision is detected, an Event containing the state of the object and its key, is sent to every other object that is involved in the collision. When a collision event has been received a response can be determined from the state information attached to the event. The key from an event allows messaging to be started with other objects involved in the collision

### 3.2.3 Visibility Testing

Visibility testing is used to determine what an AI object can see. If an object lies within the AIs field of view or is near the AI, a ray is “fired” from the AIs position to the object. If the ray collides with an obstacle then it is assumed the AI cannot see the object. If an object is visible its state information is revealed to the AI.

### 3.2.4 Updating the collection of GameObjects

The `killorspawn` function from the paper "The Yampa Arcade" is used to remove or insert objects in the game. The only difference is that it calls a modified version of `insertIL` when inserting objects into the identity list.

```
data ObjOutput = ObjOutput {
    ooObsObjState :: !ObsObjState,
    ooSendMessage :: !(Event [(ILKey,(ILKey,Message))]),
    ooKillReq     :: (Event ()),
    ooSpawnReq    :: (Event [ILKey->Object])
}
```

Listing 3.4: `ObjOutput`

Objects request their removal or the creation of another object with `ooKillReq` and `ooSpawnReq` respectively. These events are observed by `killorspawn` that updates the collection in response to them.

Objects inserted by `killorspawn` are of type `ILKey->Object`. The version of `insertIL` that is used, applies the `Ilkey` assigned to the object, which allows objects to use their key for messaging.

```
killOrSpawn :: (a, IL ObjOutput) -> (Event (IL Object -> IL Object))
killOrSpawn (_, oos) = (foldl (mergeBy (.)) noEvent es
      where
       es :: [Event (IL Object -> IL Object)]
       es = case ([ mergeBy (.)
                          (ooKillReq oo `tag` (deleteIL k))
                          (fmap (foldl (.) id . map insertILA_)
                                (ooSpawnReq oo))
               | (k,oo) <- assocsIL oos ]) of x -> x

insertILA_ :: (ILKey -> a) -> IL a -> IL a
insertILA_ f (IL {ilNextKey = k, ilAssocs = kas}) = il' where
    il' = IL {ilNextKey = k + 1, ilAssocs = (k, f k) : kas}

appFunc :: [(ILKey -> a)] -> [ILKey] -> [a]
appFunc [] _ = []
appFunc (f:fs) (k:ks) = (f k):(appFunc fs ks)
```

Listing 3.5: `killOrSpawn`, `insertILA` and `appFunc`

## 3.3 Implementation of Game Objects in Yampa

The entities in our game are modeled as separate objects that behave in parallel. Yampa allows us to simulate objects in parallel with its delayed parallel switcher, `dPswitch`. The various behaviors of our objects are modeled with Yampas signal functions, and state information is encapsulated within the functions.

```
type Object = SF ObjInput ObjOutput
```

Listing 3.6: The `Object` datatype

All game objects are of the type Object, the ObjInput and ObjOutput types mentioned previously, are the input and output types of the objects respectively.

### 3.3.1 Gravity

Gravity affects the position of an object along the y-axis. The function `falling'` is used to model the motion of an object that is falling. As dictated by the laws of physics, the velocity is the integral of the acceleration due to gravity and the displacement is the integral of the velocity. The displacement and initial position is summed to obtain the position of the object that is falling.

In imperative languages the amount of time that has elapsed is required to calculate integrals, but in Yampa the flow of time is abstracted. Mathematical formulas that use integration with respect to time can be elegantly expressed with Yampas `integral` function.

```
falling' :: Double -> Double -> SF () Double
falling' grav init = proc () -> do
      vel <- integral -< grav
      pos <- integral -< vel
      returnA -< (pos+init)
```

Listing 3.7: `falling'`

### 3.3.2 Jumping Falling and Landing

`falling` is used model the position of the player when jumping, falling and when it has landed.

```
1  falling :: SF (Bool,GameInput,Double) Double
2  falling = proc (land,gi,dt) -> do
3
4      --generate events that mark the beginning of a behavior.
5        key        <- keyStat      -< gi
6        landed     <- edge         -< land == True
7        isJumping <- edge          -<
8                   (fromEvent key) == ('e',True) && (land  == True)
9
10     --middle of jump is an local variable that stores
11     --whether we are in the middle of a jump. This
12       middleOfJump <- midJump -< (key, land)
13
14       notlanded <- edge -< land == False && middleOfJump == False
15
16
17       let grav = -200
18
19
20      --switch behaviors in reponse to events
21       pos    <- rSwitch (falling' grav 0) -<
22                  ((),    (isJumping `tag` jumping grav)     `lMerge`
23                          (landed     `tag` constant -0.05)   `lMerge`
24                          (notlanded `tag` falling' grav 0))
25       returnA -< pos
```
Listing 3.8: `falling`

Separate signal functions that model the jumping, falling and landing behaviors of the player object are written. Also, event sources are written for each behavior. In lines 6, 7 and 14, `edge` produces an event the moment a condition is satisfied. These events are used to signal the start of the falling, jumping and landing behaviors.

Events are associated with their behaviors with the `` `tag` `` infix and are merged with `lMerge`. The result of the merge is either an `Event()` tagged with a behavior or `noEvent`. Merging of events with `lMerge` is left biased, should two events occur simultaneously, only the leftmost event is returned.

When an event occurs, `rSwitch` assumes the behavior associated with the event. The events used come from user input and in-game events. Dynamic behaviors that respond to events can be programmed with these switching combinators.

### 3.3.3 Projectiles

The AI objects fires projectiles to damage the player. Projectiles move at a constant velocity and are not affected by gravity. The projectiles position is calculated by integration.

```
1 projectile :: (Vec3,Vec3) -> ILKey -> ILKey -> Object
2 projectile ((sx,sy,sz),(vx,vy,vz)) firedfrom id = proc oi -> do
3      let clippedPos = oiCollisionPos oi
4      let grounded = oiOnLand oi
5      let hits = oiHit oi
6
7
8     --the new position is the integral of the velocity along
9     --the vector (vx,vy,vz)
10    let vel = 500
11
12      x <- imIntegral sx-< vel*vx
13      y <- imIntegral sy-< vel*vy
14      z <- imIntegral sz-< vel*vz
15
16    -- a delay is used here to store the previous
17    --position of the projectile
18    oldpos <- iPre (sx,sy,sz) <<< identity -< (x,y,z)
19
20    --if the projectile has hit the player or the level
21    --generate an event
22      hitEv <- edge -< (isEvent clipEv || isEvent hits)
23
24      returnA -< ObjOutput {
25        --the state of the projectile is assigned to ooObsObjState
26         ooObsObjState = OOSProjectile {
27                                projectileOldPos = oldpos,
28                                projectileNewPos = (x,y,z),
29                                firedFrom = firedfrom},
30        --ooKillReq is used by the projectile to request its removal
31         ooKillReq     = hitEv,
32         ooSpawnReq    = noEvent,
33         ooSendMessage = noEvent
```

Listing 3.9: `projectile`

`imIntegral` used in lines 12 to 14, is similar to `integral`, its argument is used to specify the initial value of the integral.

In line 18, `iPre` is an initialised delay operator. It is used here to hold the position of the projectile that was calculated in the last iteration. The argument to `iPre`, is the initial value output at `t = 0`.

The projectile expires when it has collided. By setting `ooKillReq` to the value produced by the event source `hitEv`, a request to have the projectile removed from the collection of game objects is sent. The object is removed by the function `killOrSpawn`.

### 3.3.4 Player

The player object is the means by which a player can influence other objects in the game. Keyboard input is used to move the object through the level. Mouse input is used to change the view of the player. The player can fire a ray to cause damage.

In lines 35-39, the player's health points are stored in a local variable that is defined recursively. The keyword `rec,` when applied to a group of definitions, allows the input to a signal function to be declared, in lines that follow it instead of lines that precede it. It allows the output of a signal function to be used as its input.

In line 36, `iPre` is used to initialise the local variable. It is also used as a delay, which ensures the output of the function is used as input to the function only in the next iteration. It ensures that a feedback loop is well formed.

`ipre` is also used in signal functions that receive input from the games facilities, for example in lines 20, 27 and 31. Initially, input from the facilities arrives only after outputs from the objects have been received. The initial output from the object is produced with variables initialised with `iPre`. Once the initial output is received the facilities can produce input for the objects.

```
1 player :: Camera -> [(String,AnimState,AnimState)] ->
2              [(ILKey,Message)] -> ILKey -> Object
3 player cam modelAnims imsgs  id  = proc oi -> do
4
5      --extract data we need from the input
6      let gi          = oiGameInput oi
7      let clippedcam = oiCollision oi
8      let grounded   = oiOnLand    oi
9      let msgs       = oiMessage   oi
10     pPos        <- ptrPos            -< gi
11     forwardVel <- movementKS 400    -< gi
12     strafeVel  <- strafeKS   400    -< gi
13     trigger    <- lbp               -< gi
14     rtrigger   <- rbp               -< gi
15
16     dt <- getDt -< gi
17
18     --update the camera position and view vector based on -
19     --keyboard and mouse input
20     cam1 <- (iPre cam) <<< (arr setView) -< (pPos,clippedcam)
21     cam2 <- moves          -< (forwardVel, cam1)
22     cam3 <- strafes        -< (strafeVel, cam2)
```

Listing 3.10: `player`

```
     yVel <- falling       -< (grounded,gi)
     cam4 <- (arr dropCam) -< (cam3,yVel)
    --messages sent from enemy ai. our player object handles
    --player information requests and enemy killed messages
    msges <- iPre noEvent <<< identity -< msgs

    --collisions between the player and other objects
    --result in hit events
    hitEv <- iPre noEvent <<< identity -< oiHit oi

    --The player's health starts at 100 and decreases by 5 per
    --projectile hit
    rec
       currentHealth <- (iPre 100) <<< identity -<
          case (isEvent hitEv) of
            True -> currentHealth - (length (fromEvent hitEv)*5)
            False-> currentHealth

    --updates the player's score
    rec
      kills <- (iPre 0) <<< identity -<
           kills + (length (findKills  (event2List msges)))

    --ccam is the playerstate sent to enemy AI in reponse to
    --their request for player info.
    ccam <- (iPre cam) <<< identity -< clippedcam

    let msg4Enemy = map (toTargetPosition id (cpos ccam))
                              (findEnemies (event2List msges))

    returnA -< ObjOutput {
         --spawn a ray when the left trigger is pressed
         ooSpawnReq    = (trigger `tag` [(ray (cpos cam1)
                            (viewPos cam1) id)]),
         ooObsObjState = OOSCamera {
                            newCam =  cam4,
                            oldCam =  cam1,
                            health =  currentHealth,
                            ammo = 100,
                            score = kills},
         ooKillReq     = noEvent,
         ooSendMessage =
             case (event2List msges) of
               [] -> noEvent
               --sends the player state to enemy ai in response
               --to requests
               _ -> Event () `tag` msg4Enemy
```
Listing 3.10: `player`

### 3.3.5 Adversarial AI

AI objects are used to challenge the player. The AI in this game has 3 modes of behavior, patrol attack and death. Yampas switching function rSwitch was used as a high-level controller to switch between behaviors.

```
rec
  (newPos,oldPos, orientation,pitch,attackEv,(upperAnim,lowerAnim))
     <- drSwitch (patrol) -<(
           (objectInput, uEndEv,lEndEv),
                playerSighted      `tag` attack oldPos `lMerge`
                playerInfoReceived `tag` attack oldPos `lMerge`
                damageReceived `tag` death)

  (uEndEv ,upperState) <- updateAnimSF ua -< (gi, upperAnim)
  (lEndEv ,lowerState) <- updateAnimSF la -< (gi, lowerAnim)
```

Listing 3.11: switching used to model AI

Initially, the AI patrols a set of waypoints. When the player is sighted, the AI attacks by firing projectiles at the player. If the AI has lost sight of the player, it jumps to the last position of the player. The AI continues to attack until it is damaged and “dies”. As in `falling,` discrete events signal the start of behaviors, and the behavior tagged to the event is switched into.

Every behavior outputs information, such as the AIs position, the pitch of its torso or its orientation. `drSwitch` is used instead of `rSwitch` to allow the switch to be non-strict. The effect of the switch is observed an infinitely small time after the event occurred, instead of immediately This allows the output of `drswitch`, to be parameters of its behaviors, so the new behavior that is switched into can be a continuation of the previous behavior. For example, `oldpos` is a parameter of `attack`, so the AI can attack at the position it patrolled up to.

Animations are played to match the behavior of the Ai. The animations are updated with `updateAnimSF`. `uEndEv` and `lEndEv` are events generated at the end of each animation sequence. This is input to the behavior that `dPswitch` assumes. With these events, behaviors can be made to match the animations. For example, a projectile is spawned in response to the event produced at the end of the AIs attack animation. Also, animations can be composed with switchers, when the end of an animation is reached, an event is generated and a new animation is switched into.

When a player’s ray hits an AI, the state information of the ray is sent to the AI in a collision event. The state information contains the key of the player that fired the ray.

This key is used to message the player that it has killed the AI and allows the player to increase its score.

Messaging is used here to request the state of the player object when the AI has lost sight of the player. A message is sent to the AI containing the position of the player, which is then used by the AI to jump to the player's location. Another way that state information could be requested would be to include the handling of state information requests as part of the games facilities. This would remove the need for a request and response. Instead, state information could be received immediately after a request.

## 3.4 Animating signal functions

Yampas signal functions are executed with functions that connect Haskell with Yampa. These functions apply the input and elapsed time sample obtained from Haskell to the signal functions, the output from the signal function is then used to perform an IO action such as rendering.

For our implementation we use the functions `react`, `actuate`, `reactinit` and the data type `ReactHandle` instead of `reactimate`. `React,` because this allows control to be surrendered to HOpenGL so input samples from HOpenGLs callbacks can be obtained.

```
reactInit :: IO a
    ->(ReactHandle a b -> Bool -> b -> IO Bool)
    -> SF a b
    -> IO (ReactHandle a b)

type ReactHandle a b = IORef (ReactState a b)

react :: ReactHandle a b
      -> (DTime,Maybe a)
      -> IO Bool
```

Listing 3.12: `reactInit`, `ReactHandle` and `react`

`react` is the function responsible for animating signal functions. `Reactinit` is called once to perform initialisation and returns the `ReactHandle` which is used by `react`. The `ReactHandle` datatype preserves the state of execution of `react` across callbacks.

Keyboard and mouse input obtained from HOpenGL callbacks are stored in IORefs. These are retrieved and sent with the amount time that has elapsed to `react`. `react` is placed within HOpenGLs `idle` callback, which is called every time HOpenGL has completed a display operation to draw a single frame of the games graphics.

```
repeatedly (1/60) ()
```

Listing 3.13: `repeatedly` used to generate an event at 60Hz

The rate at which the screen is redrawn is synchronised to an event that is repeatedly produced at 60hz by a signal function. Frames are skipped to maintain the rate of execution of the program. Also, this limits the framerate of the game to 60 FPS.

## 3.5 Graphics and Animations

Graphics are used to render the world in which the player sees. The BSP[17] file format stores information used for rendering levels, while the MD3 file format is used for character animations. Also, the Targa image file format was used for our textures.

Various editors that support these file formats could be used to create content for the game. Also, the level editor that supports BSP generates the information used in algorithms for efficient rendering and collision detection with BSP.

### 3.5.1 BSP

The BSP file format stores data structures in contiguous sections of the file. The vertex and texture coordinates used by HOpenGL are read into buffers, the rest of the data structures are stored in Haskell lists. The lists are sequentially accessed, and not randomly indexed.

```
data Tree  = Leaf BSPLeaf | Branch BSPNode Tree Tree

data BSPLeaf = BSPLeaf {
    cluster           :: Int,
    area              :: Int,
    leafMin           :: (Double,Double,Double),
    leafMax           :: (Double,Double,Double),
    leafface          :: Int,
    numOfLeafFaces    :: Int,
    leafBrush         :: Int,
    numOfLeafBrushes  :: Int,
    leafFaces         :: [BSPFace],
    leafBrushes       :: [BSPBrush]
}
data BSPNode = BSPNode {
    planeNormal :: (Double,Double,Double),
    dist        :: Double,
    front       :: Int,
    back        :: Int,
    nodeMin     :: (Int,Int,Int),
    nodeMax     :: (Int,Int,Int)
}
```

Listing 3.14: `Tree`, `BSPLeaf` and `BSPNode`

The data structures are assembled into nodes and leaves. These are then assembled into a binary tree.

Visibility tests are performed to reduce the amount of geometry that has to be rendered. Precalculated visibility information is stored within the BSP file as bitfields. Bit masking is performed to see if a leaf is potentially visible from the player's location. If a leaf is potentially visible, the bounding box of the leaf is tested to see if it lies within the view frustum.

If a leaf has passed the visibility tests, the `leafFaces` are drawn. Each polygon face has pointers that refer to the vertex and texture information stored in the buffer. These pointers are passed to HOpenGL, and the vertices the pointer refers to are rendered with OpenGL vertex arrays. Vertex arrays allow for efficient rendering, as they eliminate repeated calls to functions in OpenGL..

```
drawElements :: PrimitiveMode ->
     NumArrayIndices -> DataType -> Ptr a -> IO ()
```

Listing 3.15: `drawElements` renders vertex arrays

Once a face is drawn, it is flagged to prevent redrawing of the face. This has to be done to prevent repeated rendering, as faces are shared between leaves.

### 3.5.2 MD3 Animation Format

MD3 is an animation format for Quake 3[16]. In this game, the MD3 animation format is used for character animations. Playback of animations is performed with vertex interpolation. Animations are stored as a sequence of keyframes. Each keyframe is composed of vertices. By interpolating between keyframe vertices, the pose of the model between frames can be approximated. The frame approximated by interpolation is rendered between keyframes so animations appear to be smoother.

MD3 files store vertex information and transforms. There are separate MD3 files for the legs, body, weapon and head. Shader files specify which textures are applied to the model. An animation,cfg file specifies the number of frames and the rate at which each animation is played.

Sections of the model stored in separate md3 files are assembled into a hierarchy. The legs form the root of the hierarchy. This is followed by the torso. Lastly the weapon and the head are placed at the bottom of the hierarchy.

Tags are the joints of the model. Each tag has a translation and rotation that has to be applied to join separate sections of model. Each rotation is stored as a 3x3 matrix, these are converted into quaternions so the rotations can be interpolated. There are separate tags for every frame of animation. The translation and rotations of a tag affects the position and orientation of sections lower in the hierarchy.

There are separate sets of animations for the torso and legs of a model. For example, an attack animation can be played for the torso and running or walking animations can be played for the legs. State information stores what animation is being played, which frames are used for interpolation and the last time the animation was played.

```
data Model = Model {
   modelRef    :: !MD3Model,
   weapFire    :: IORef (Maybe (IO())),
   pitch       :: IORef (Maybe (IO())),
   upperState :: IORef AnimState,
   lowerState :: IORef AnimState
}
data AnimState = AnimState {
                     anims          :: ![MD3Animation],
                     currentAnim    :: !MD3Animation,
                     currentFrame   :: !Int,
                     nextFrame      :: !Int,
                     currentTime    :: !Float,
                     lastTime       :: !Float
                 }
```

Listing 3.16: The `Model` and `AnimStat` data types.

Before rendering, the state information for the legs and torso animations is updated. State information for the torso and leg animations of the model are set by writing to `IORefs`.

To render the model, the section at the top of the hierarchy is drawn. Sections lower in the hierarchy are transformed by applying the translations and rotations stored in tags. A push matrix operation is performed so the current matrix stack can be preserved, and this process is repeated for sections lower in the hierarchy. As mentioned previously, the vertices and transforms of the two keyframes referred to in the animation state, are interpolated, and the vertices and transforms from the interpolation are used when rendering the model

Visibility tests are performed before a character model is rendered, to determine if it can be seen. If a model is visible, vertex arrays are used to render the model.

### 3.5.3 Textures

Textures are images used to decorate the surface of 3D geometry. Haskell doesn't have libraries for loading images, as it is relatively new language.

The Targa image format is a format for describing bitmap images. It supports 24-bit color and transparencies. It is relatively easier to load compared with JPEG images and many image editors support it.

To load the image, the header of the image file is read to determine the dimensions of the image and the number of bits per pixel. The image is loaded into a temporary buffer. An OpenGL texture object is created and the contents of the buffer are copied into OpenGLs memory.

# 4. Benchmarks

In this section the performance of the game is assessed. We measure the framerate of the game and profile the game to analyse the run-time contribution from rendering, executing object code and game facilities.

The test platform used in this benchmark is equipped with an Athlon XP 1900+ CPU, 512MB of DDR266 Ram and a Nvidia GeForce 4 MX 64MB. The operating system is Mandrake 10. The resolution of the game is 640x480 with 32 bit color. For framerate measurements the game was compiled in GHC with the –O2 option. For run-time analysis the program was compiled with the +RTS –p –O2 –auto-all profiling options.

In general, 60 FPS is a desirable framerate, while 30 FPS is a minimum for smooth graphics.

## 4.1 Benchmark 1

The purpose of this benchmark is to measure the performance of the program in an optimum situation where less performance is spent on rendering graphics. On average a low amount of polygons have to rendered and most AI objects are obscured from the players view and are not rendered.

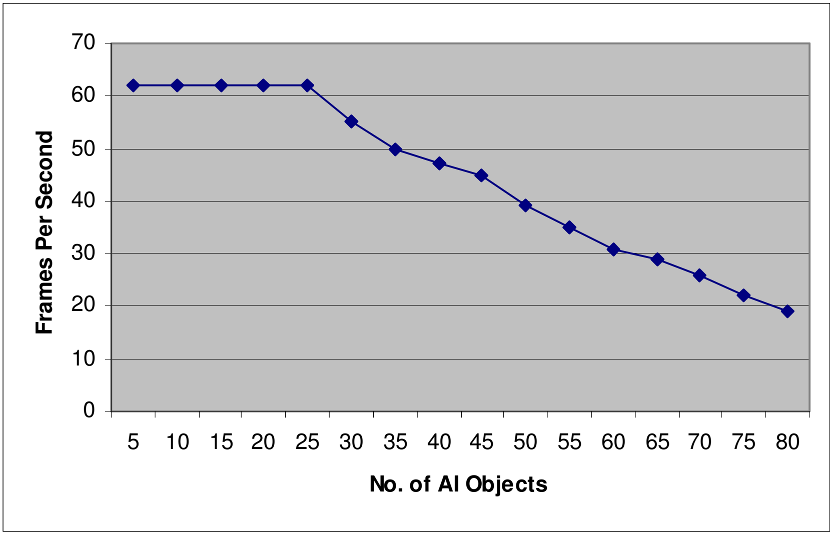

Figure 4.1: Benchmark 1 FPS vs No. of AI Objects

As mentioned previously the framerate is limited to 60 FPS, so no change in framerate is observed at 5 to 25 objects. The framerate drops to 30 FPS at 62 objects. Even though 60 objects is a reasonable number for this genre, given the specifications of this computer, there are many commercial games that outperform our game in terms of the number of AI objects that can be handled and in terms of rendering performance.

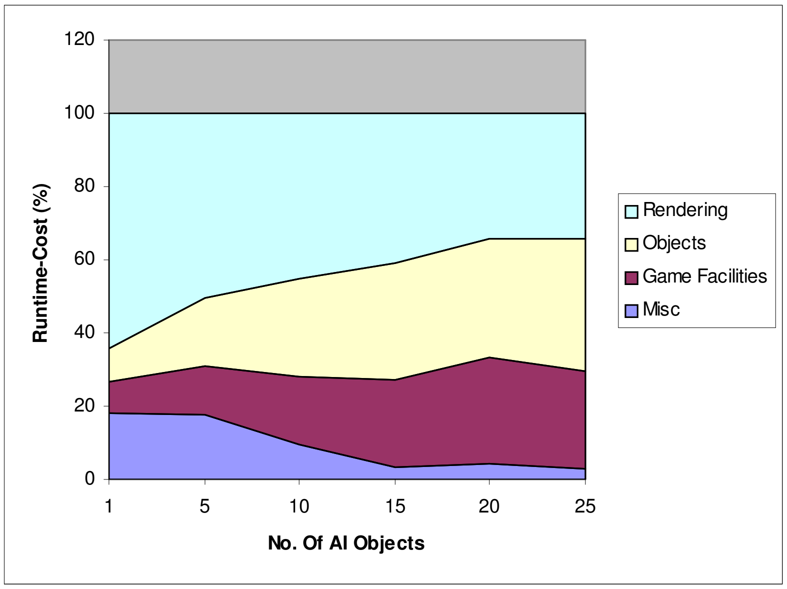

Figure 4.2: Benchmark 1 runtime cost vs No. of AI objects (stacked area graph)

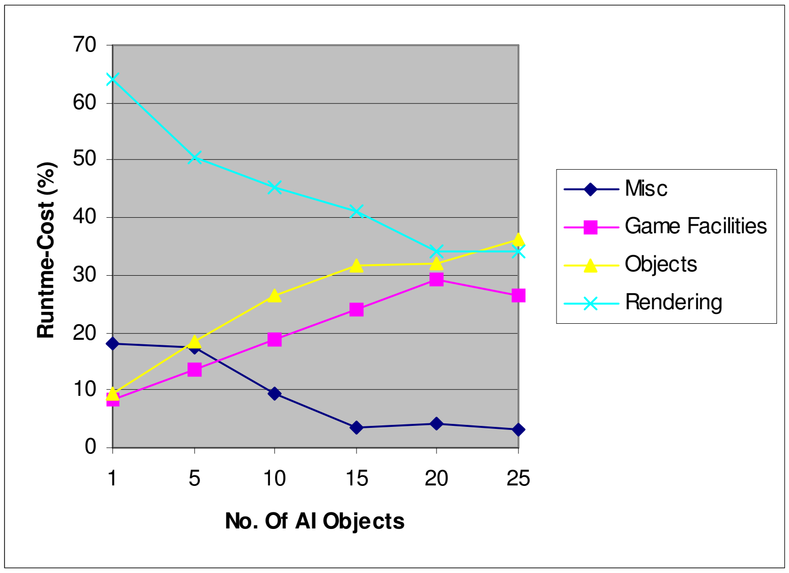

Figure 4.3 Benchmark 1 runtime cost vs No. of AI objects

Runtime costs are divided into rendering, objects, game and miscellaneous. Miscellaneous accounts for time spent idling and obtaining input for animating signal functions. As the number of objects increases, runtime contribution from rendering decreases while runtime contribution from the execution of object code and game facilities increases. Also, the amount of time spent idling decreases.

## 4.2 Benchmark 2

This benchmark is used to measure the performance of the program when there is a high load on all areas of the program. In addition to the AI objects, projectiles have to be handled as all AI objects will be firing at the player. Furthermore, increased stress will be put on the graphical areas of the program as all objects are visible to the player and will have to be rendered. The level used in this benchmark has a higher number of polygons compared with the level used in benchmark 1.

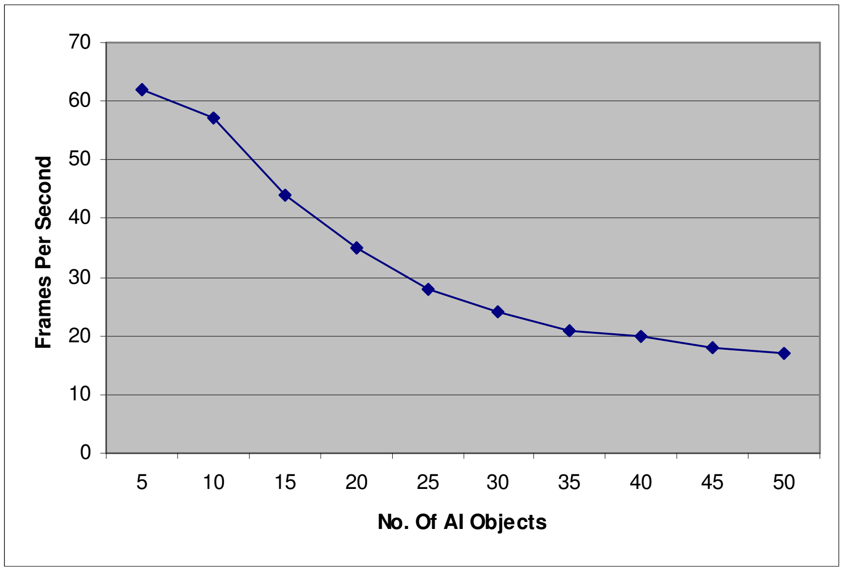

Figure 4.4: Benchmark 2 FPS vs No. of AI Objects

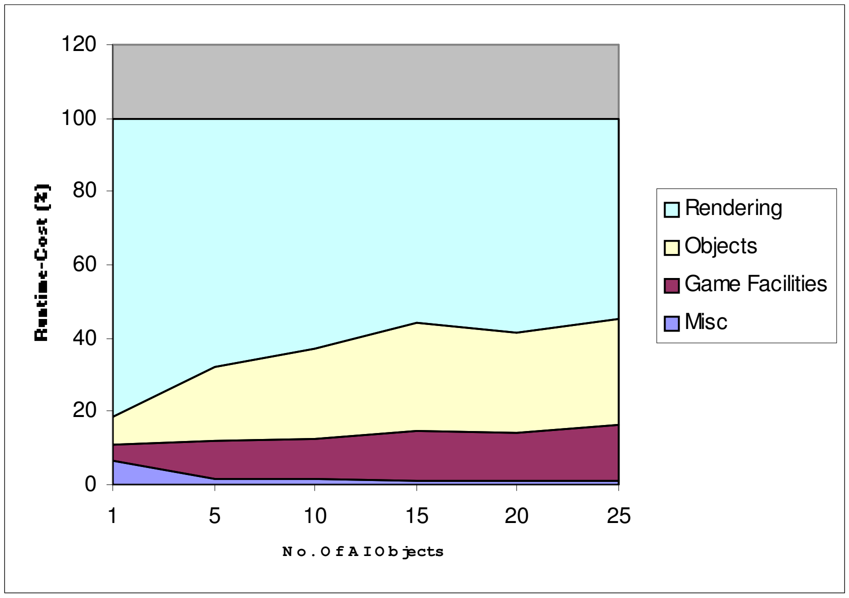

Figure 4.5: Benchmark 2 runtime cost vs No. of AI objects (stacked area graph)

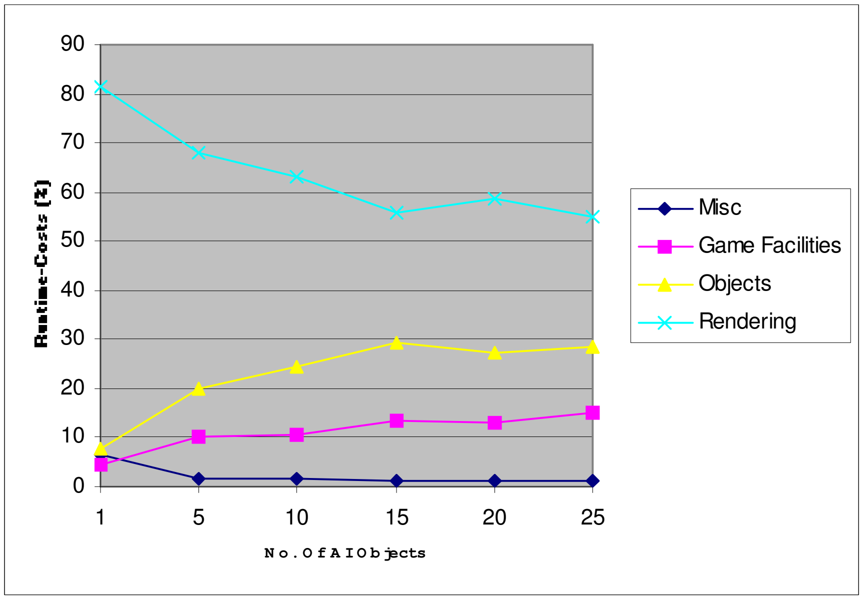

Figure 4.6: Benchmark 2 runtime cost vs No. of AI objects

In this benchmark, rendering is the dominant runtime cost. The framerate drops to 30 FPS at 23 AI objects compared with 62 AI objects in the previous benchmark. The cumulative cost of rendering all AI objects and the level is greater than the costs from executing object code and game facilities.

## 4.3 Comments

As an initial prototype the performance of the game is adequate, but the performance of the game will have to be improved greatly before it can match the performance of commercial games. There is room for improvement, as the implementations of algorithms for rendering and collision detection in our game are not optimised.

# 5. Discussion

This section discusses the results of this thesis project. The use Yampa and Haskell for programming this game is critiqued.

## 5.1 Yampa

In the game, entities are modeled as separate objects and are updated simultaneously. There are games that model game objects as separate threads to achieve this[18]. Yampas parallel switchers and signal functions can be used to simulate objects and update them in parallel. This can be achieved with less code compared with multithreading. Synchronisation and concurrency issues associated with the use of virtual machines and multithreading can be avoided.

Yampas events and switching combinators allow us to program event-driven state machines, which are commonly used in game programming. Different behaviors of the AI are modeled as separate continuous functions. The switcher is the high-level controller, and different behaviors are switched according to event occurrence.

In Yampa, each objects state information is encapsulated within signal functions. Information about the state of the object can only be obtained from its output. This allows the internal representation of the object to be changed without changes to other areas of the program.

As mentioned previously, game objects can be developed and tested in isolation. In this implementation, once the game facilities were completed, game objects were developed incrementally. The player object was implemented first. Later, other objects such as the AI objects and projectiles were programmed one after the other, and added to the game. Only minor changes had to be performed to accommodate the new objects.

The semantics of Yampa may take a while for a programmer to understand, it may not be obvious where to insert a delay or where to use a delayed version of the switchers. Yampa only minimises space leaks. The programmer has to know how to prevent them and remove them if they occur. It may be time consuming to find these space leaks and remove them should they occur.

The facilities of the game must be implemented with pure functions in Haskell. There is a possibility, that it may be impossible to implement certain facilities without the mutable variables that use the IO monad. UnsafePerformIO transforms functions that use the IO Monad into pure functions, but it is possible to break referential transparency this way. Safety must be guaranteed by the programmer.

## 5.2 Haskell

With Haskell, some of the datatypes used in the game were expressed elegantly. Matrices, quaternions, coordinates and vectors are expressed elegantly with tuples. Recursive datatypes were used to define character models and binary trees used in our game. Lists can be pattern matched, and higher order functions such as `map` were frequently used in the game to perform operations across list elements.

Another advantage of programming in Haskell was that programming errors were reduced. The programmer is informed of errors at compilation time instead of at run time. This leads to development time savings as the amount of time that would have been spent debugging is reduced.

The Haskell OpenGL library, HOpenGL, has many differences compared with other OpenGL libraries, a notable difference is that GLx prefixes have been removed from the names of all functions and constants, which is elegant. Syntactic sugaring was provided in the form of the ($=) operator. This allows us to set up OpenGL state variables in a manner similar to C which uses an equals sign which is convenient.

For non-trivial rendering tasks, HOpenGL offers high performance options such as vertex arrays, vertex buffer objects and display lists. Vertex arrays and display lists were used in our game for improved rendering performance.

The only flaw of HopenGL, was documentation is not as good as it could have been. The HOpenGL equivalents of constants and functions found in OpenGL libraries for other languages were not listed, so time had to be spent searching for them.

# 6. Conclusion

In conclusion, a first person shooter was successfully implemented with Haskell and Yampa.

Datatypes used in game development can be expressed elegantly in Haskell. HOpenGL provides the necessary support for accelerated 3D graphics needed in games.

Yampa allows an event-driven game engine to be programmed with less code. Objects can be programmed with signal functions that encapsulate state and are updated simultaneously. The use of Yampa for the prototyping of games should not be overlooked. Test versions of games can be written in less time and a viable game idea can be found sooner.

## Future work

From the results of this thesis the following work is suggested:

The performance of functions in the game can be improved. A comparison could be made of the style and performance of an optimised version of this game with another version that uses FFI and c to program performance critical sections of the game.

The genre of the game we are programming is a first person shooter. Another genre that could be implemented is Real time strategy, where a significantly greater number of objects have to be simulated compared to first person shooters.

# Bibliography

[1]Computer and video game industry
From Wikipedia http://en.wikipedia.org/wiki/Video_game_industry

[2]Kevin Flood, Game Unified Process (GUP)
http://www.gamedev.net/reference/articles/article1940.asp

[3]Simon Peyton Jones. http://www.haskell.org/aboutHaskell.html

[4]Conal Elliott, An Embedded Modeling Language Approach to Interactive 3D and Multimedia Animation (© 1999 IEEE), IEEE Transactions on Software Engineering, 25(3), May/June 1999, pp 291-308.

[5]Paul Hudak. The Haskell School of Expression { Learning Functional Programming through Multimedia. Cambridge University Press, New York, 2000.

[6]John Peterson, Gregory Hager, and Paul Hudak. A language for declarative robotic programming. In International Conference on Robotics and Automation, 1999.

[7]Alastair Reid, John Peterson, Greg Hager, and Paul Hudak. Prototyping real- time vision systems: An experiment in DSL design. In Proc. Int'l Conference on Software Engineering, May 1999.

[8]Antony Courtney and Conal Elliott. Genuinely functional user interfaces. In Proc. of the 2001 Haskell Workshop, September 2001.

[9]Antony Courtney and Henrik Nilsson and John Peterson. The Yampa Arcade. In Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03), pages 7 - 18, Uppsala, Sweden, August 2003. ACM Press.

[10]George Russell. Email message, subject: Fruit & co, February 2003. Message posted on the Haskell GUI mailing list, available at http://www.haskell.org/- pipermail/gui/2003-February/000140.html

[11]Meurig Sage. Frantk: A declarative gui system for haskell. In Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP 2000), September 2000.

[12]Christoph L¨uth. Haskell in space. In Simon Thompson Michael Hanus, Shriram Krishnamurthi, editor, Functional and Declarative Programming in Education (FDPE 2002), pages 67– 74. Technischer Bereicht 0210, Institut f¨ur Informatik und Praktische Mathematik, Christian-Albrechts-Universit¨at Kiel, September 2002.

[13]Peter Achten and Rinus Plasmejer. Interactive functional objects in clean. In Proc. of 9th International Workshop on Implementation of Functional Languages, IFL'97, volume 1467 of LNCS, pages 304–321, September 1997. Clean game library available at http://cleangl.sourceforge.net/

[14]Joe van den Heuvel and Miles Jackson, Pool Hall Lessons: Fast, Accurate Collision Detection between Circles or Spheres http://www.gamasutra.com/features/20020118/vandenhuevel_02.htm January 18, 2002

[15] Nathan Ostgard, Quake 3 BSP Collision Detection http://www.devmaster.net/articles/quake3collision/ 16/08/2003

[16] phaethon@linux.ucla.edu Description of MD3 Format http://linux.ucla.edu/~phaethon/q3a/formats/md3format.html 2004 Apr 30.

[17] Morgan McGuire Rendering Quake 3 Maps http://graphics.cs.brown.edu/games/quake/quake3.html July 11, 2003

[18]Harry Kalogirou Multithreaded Game Scripting with Stackless Python http://harkal.sylphis3d.com/2005/08/10/multithreaded-game-scripting-with-stackless-python/ August 10 2005

[19] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson Arrows, Robots, and Functional Reactive Programming http://www.haskell.org/yampa/ AFPLectureNotes.pdf August 2002

# Glossary

| | |
|---|---|
| BSP - | BSP is Binary Space Partitioning, a algorithm for drawing scenes and performing collision detection efficiently. |
| First person Shooter - | A first-person shooter is a shooting game where the player's view of the game world is exactly that of the character the player assumes. |
| Frustum - | It is the area of the world visible to the current camera. The frustum is bounded by 6 planes. These planes are named the near, far, left, right, top and bottom planes. |
| Higher-order functions - | Higher-order functions are functions that take functions as an input or output a function. |
| Quaternions - | Quaternions represent rotations in three dimensions. Spherical Linear Interpolation (SLERP) is used to smoothly interpolate between two quaternions |
| Real Time Strategy - | Real time strategy is a strategy game that emphasises unit management. The game progresses in real time. |
| Referential Transparency – | In computing, a referentially transparent function is one that, given the same parameter(s), always returns the same result. |

# Listings