λ CS ： 基于 FRP.Yampa 的第一人称射击游戏

Haskell 期末项目报告

小组成员　孙锡波 17212020038
邱颖慧 17212020065
朱浩哲 17212020095

指导教师　　严昌浩

2017~2018 学年第一学期

目录

第一章　项目简介

1　项目内容

本项目为 Haskell 课程的期末项目作业，主要基于 Haskell + FRP.Yampa + OpenGL 实现的简单版的经典 CS 游戏。

2　项目目的

本项目最终构建了一个游戏项目，构建该游戏项目旨在熟悉 Haskell 基本的编程知识，学习利用 Haskell 编写游戏的解决方案及其中存在的问题、3D 游戏的渲染原理、网络传输的原理及 Haskell 的实现。

3　早期调研总结

早期调研主要参考了 Github 的一个总结性文档[2]，该文档中最新更新是在 2017 年 12 月 5 日，算是对 Haskell 整体应用环境比较全面的总结，包括 Compilers、GUI Programming、Machine Learning 等应用，并且作者给出了当前各方面应用发展的成熟程度。在作者的观点中 Haskell 在 Compilers、Server-side programming、Scripting / Command-line applications 等方面的应用发展较为成熟，原因在于 Haskell 语言的特性以及各种较为完整的库和工具。但是对于类似 GUI 和 Game Programming 这种需要响应式的应用也由于 Haskell 自身的特性导致维护状态十分地麻烦。

虽然在游戏编程领域状态维护比较麻烦，但是相关的研究和应用也非常多，Haskell Wiki 中详细描述了 Haskell 游戏当下的发展的状况[3]，包括各种文字资料、游戏引擎、显示库、demo 等等。

第二章　项目实现

1　游戏框架

游戏的整体框架如图 2.1 所示，游戏主要分为两个部分：客户端和服务器端。客户端和服务气端均有自己的循环和内核（Signal Function），这里的 SF（Signal Function）在 Yampa 框架定义。当把输入数据输入到 SF 中，SF 会执行一系列操作，包括 IO 操作，赋值操作等等，来达到更新状态、渲染、网络传输的目的。其中，内部会将会维护一个状态，也就是说，SF 的输入为外界输入（键盘事件、网络消息事件）和上一时刻的系统状态共同构成。

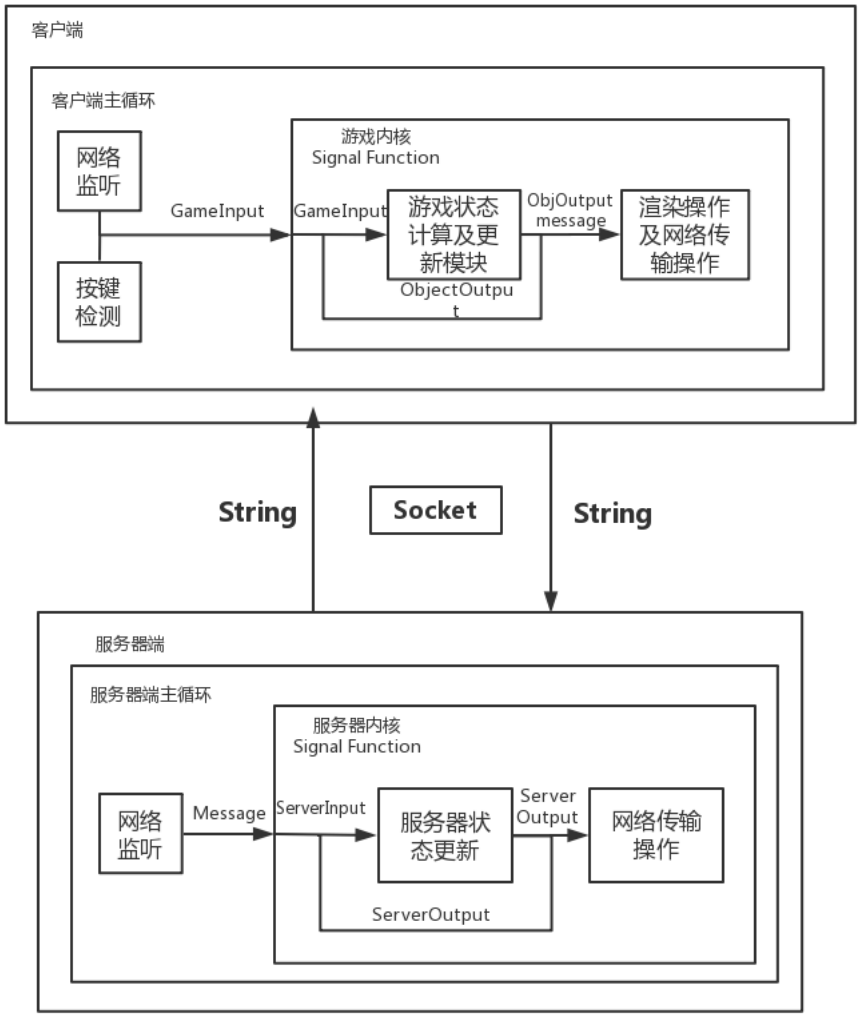

图 2.1: 游戏整体框架图

2　状态机实现

2.1　Yampa

Yampa （Yampa 为科罗拉多州的一条河流，中间有部分河段水流湍急，其余部分水流平缓）是函数响应式编程（Functional Reactive Programming，FRP）中的一种实现的方法[5]，可以广泛用于响应式系统中，类似 GUI 编程、游戏编程、Mobie Robot 等等领域，在 Yampa 的 Haskell Wiki[4] 中甚至提到过 Yampa 被用于 VR 领域的编程。

2.2　SF （Signal Function）

Yampa 中定义了一个 Signal Function，定义类型如下：

```
1 Signal a = Time -> a
2 SF a b = Signal a -> Signal b
```

其中，Signal 表示与时间有关系的值，对应到游戏中就是不同时间输入的响应事件。例如，当前需要计算一个又初始位置和初始速度的自由下落的球的位置。如果这里利用积分计算球的速度的话，就需要保存下来所有历史时刻球体的速度，计算位移的时候就需要对速度进行积分。因此，对于速度来说，它就是一个随时间而变化的变量。

Yampa 中提供的 Signal Function 可以很容易地进行游戏内逻辑的扩展，并且可以容易地维护游戏内部的状态，这两点决定了 Yampa 十分适合用来做游戏的开发。从实现方法上而言，Yampa 内部的实现方式可以避免多线程死锁的发生[5] 。

2.3　基于 dpSwitch 的状态机

在我们的游戏中，游戏的核心引擎由 Yampa 构成，最核心的状态转换机部分由定义于 Yampa 中的 dpSwitch 函数构成，dpSwitch 定义如下：

```
1 dpSwitch :: Functor col => (col is a collection)
2     (forall sf . (a -> col sf -> col (b, sf))) -> -- route : 预处理和分配函数
3     col (SF b c) -> -- SF（Signal Function）的集合
4     SF (a, col c) (Event d) -> -- 转换事件
5     (col (SF b c) -> d -> SF a (col c)) -> -- 执行转换事件并回调
6     SF a (col c)
```

这里游戏的核心设计主要参考了 Yampa 的一篇设计游戏的论文[1]，图 2.2 来自该论文中的游戏核心示意图，本项目的核心设计基本和该论文中的核心设计类似，对最后的转换事件和回调函数做了修改，适合游戏要求。

在图 2.2 中，Route 表示预处理函数和分配函数，其主要作用是将游戏输入信号（包括外部输入和内部上一时刻的状态）进行简单的处理并判断需要分配给哪一个 SF。中间部分为 SF 的集合，每一个 SF 表示一个游戏内部的物体。killOrSpawn 为检测事件函数，检测游戏内物体是否需要新增或者被消灭，然后对 SF 的集合进行添加或者删除的操作。最后生成一个新的输出，内部的状态进行了更新。

本项目中主要修改了 route 函数，route 函数检测当前物体是否是玩家物体，如果是玩家物体，则需要将检测玩家与剩下的物体是否发生碰撞。

2.4　游戏内的 SF

本游戏内客户端定义的 SF 主要在文件 Object.hs 中，分别有 observer、player、bullet、bulletHole、terrain0。observer 中包括己方视角转动角度计算、开火事件判断、生命值系统、位置计算、碰撞检测、死亡判定。player 为本地的敌人，其中包括了位置和视角方向更新、死亡事件检测、生成事件检测、退出事

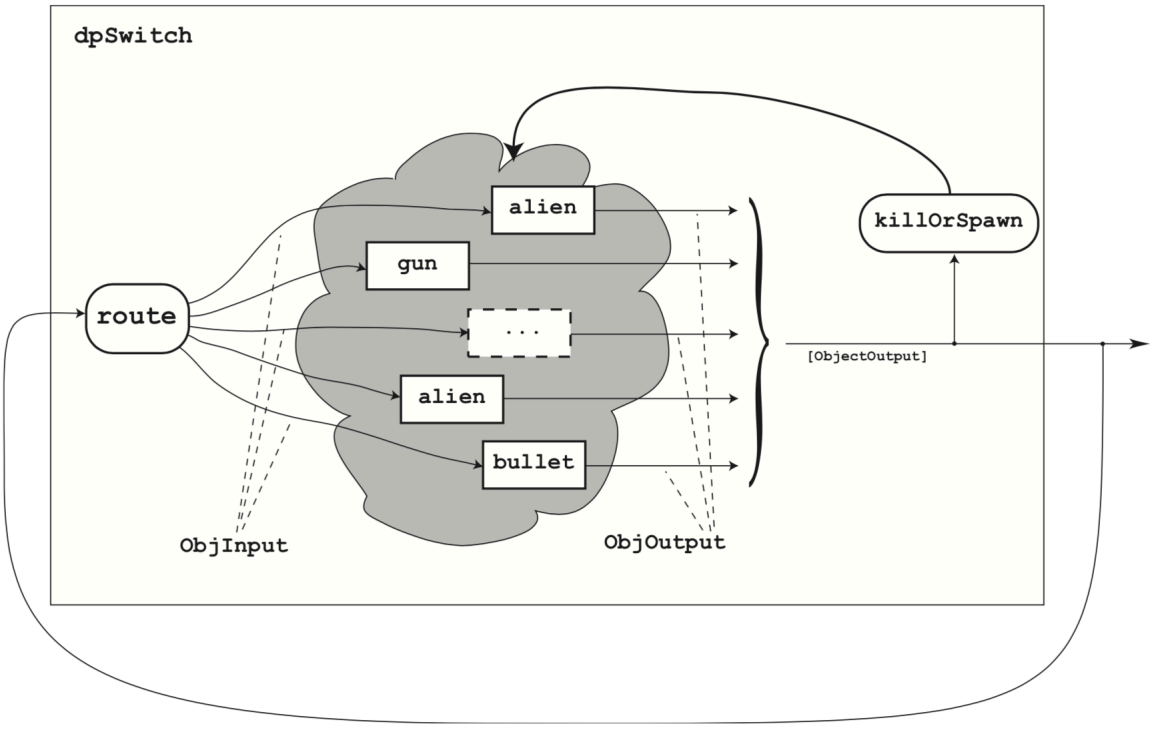

图 2.2: 游戏核心示意图

件检测。bullet 为 observer 开火后需要生成的 SF，包括了消失事件检测。bulletHole 为弹孔，其中不包含任何操作，仅仅告知之后的渲染操作需要渲染弹孔。terrian0 为地形的 SF，同 bulletHole 相同，不包括任何操作，仅仅告知渲染操作需要渲染弹孔。

上述 SF 都为游戏中定义的 ObjectSF，如下所示。

```
type ObjectSF = SF ObjInput ObjOutput

data ObjInput = ObjInput {
   oiGameInput :: !GameInput, -- 外届输入
   oiColliding :: !(Maybe ObjOutput) -- 碰撞标志

}

data ObjOutput = ObjOutput {
   ooObsObjState :: !ObsObjState, -- Object相关状态
   ooKillReq :: !(Event ()), -- 消灭标志
   ooNetworkMsgs :: ![CSMsg], -- 需要发送至服务的消息列表
   ooSpawnReq :: ![ObjectSF], -- 需要生成的SF列表
   ooBounds :: !BoundingVolume -- Bounding Box定义
}
```

3　OpenGL 渲染实现

3.1　向量类定义

3D 游戏内由于涉及了大量的三维坐标的计算，会包括叉乘、点乘、求模、向量相加、向量相减、三维坐标点等基本操作和信息。因此，本项目的 Vec3d.hs 文件中定义了 Vec3d 的类型类，提供这些基本操作的接口和函数定义。

3.2　地形渲染

其中，游戏中大体上复现了 CS 游戏中 iceworld 的经典地图，主要分为几个部分：天空、地面、墙面、障碍物。其中，天空使用的是半径较大的球体，内部帖纹理图。地面和四个墙面总共使用了五个矩形面。障碍物使用简单的立方体。

其中，人物是不能进行穿墙的，因此需要进行碰撞检测，这里的碰撞检测使用简单的 Bouding Box，测量两者的最大坐标值和最小坐标值这个范围是否有交集，如果有交集表示发生了碰撞。

本项目使用的是 GLFW（基于 OpenGL 的渲染框架）来进行的渲染操作。读入的方式使用了 GLFW 内置的读入函数，将 tga 格式（存储信息：RGBA）的图像读入。

3.3　玩家视角渲染

玩家视角的渲染主要是对相机的控制，FPS 的游戏原理就是首先在三维空间中建模，然后在三维空间中放置相机，并调整角度，最后通过计算来获得如何将三维空间投影到相机视角中。如图 2.3 所示，相机有一个自己的坐标，对相机的操作相当于在 OpenGL 对视角矩阵进行操作。投影则是对投影矩阵进行操作，用户可以自己设定投影矩阵的投影方式[6]。

游戏中需要根据鼠标位置对相机进行实时的位置和视角更新，更新方式就是对视角矩阵进行转换。视角矩阵的转换公式为 $view = [T \times R_{roll} \times R_{yaw} \times Rpitchl]^{-1}$ 其中 T 为视角矩阵，R_{roll} 表示 x 轴旋转矩阵，R_{yaw} 表示 z 轴旋转矩阵，R_{pitch} 表示 y 轴旋转矩阵。

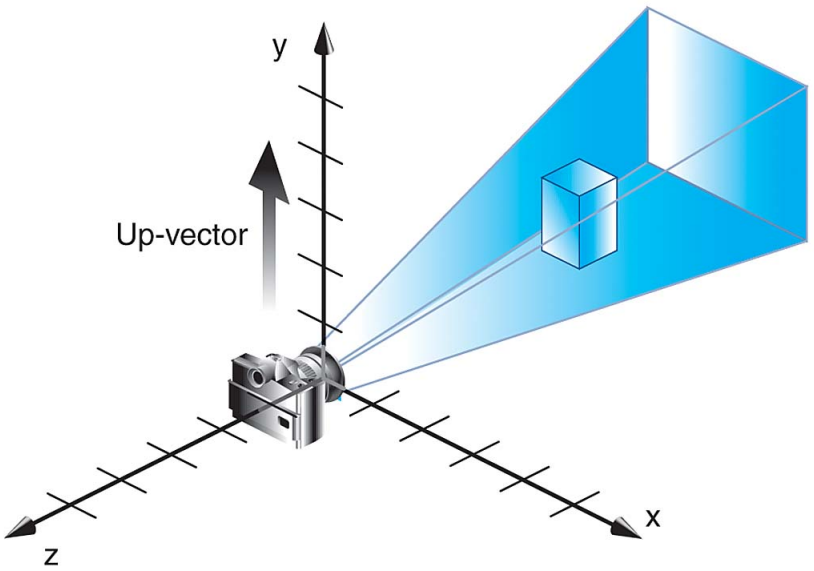

图 2.3: 相机坐标和投影

3.4　敌方玩家渲染

由于部分原因，本项目仅仅实现了很简单的人物模型，将人物抽象为一个球体和一个圆柱体的结合，如图 2.4 所示。渲染同之前的地形的渲染原理类似。

图 2.4: 敌方玩家人物模型

3.5　子弹渲染

本项目中子弹原型为圆柱体，朝向同相机视角相同，因此渲染的函数与之前的渲染共用相同的函数，只是提供给函数的高度和底面半径数据不同。

本项目内部定义子弹在 3 秒之后如果不发生碰撞时间则消失，由于子弹速度较快，因此利用简单的 Bounding Box 的检测方式可能无法检测到子弹的碰撞事件。因此，游戏中子弹是通过检测射线和物体的碰撞来检测碰撞事件的。

游戏中定义打到玩家头部球体会扣减敌方生命值，因此需要检测射线与球体的碰撞，检测方式就是计算球体中心到射线的距离，如果小于球体半径，则表示发生了碰撞。

图 2.5 所示为子弹与墙面的碰撞检测，检测的方法就是检测子弹射线和墙面的交点，通过解方程组得到交点，然后判断交点是否在墙内部以及子弹是否穿过墙来决定是否发生了碰撞。

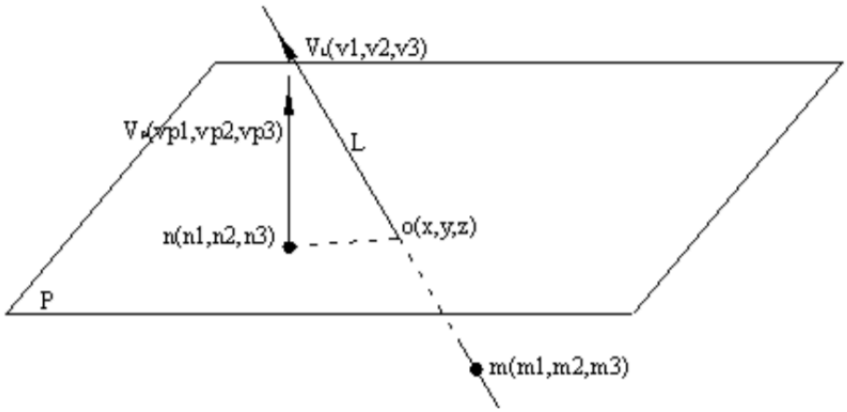

图 2.5: 线和面的交点

3.6　弹孔渲染

弹孔是子弹和墙面发生碰撞留下的弹孔，主要是通过碰撞检测得到碰撞点位置以及碰撞点平面的法向量，有了这两个信息就在该位置生成一个矩形的面，贴上弹孔的图，结果如图 2.6 所示。

图 2.6: 弹孔示意

4　网络实现

服务器端的实现如图 2.7 所示，利用两个线程执行两个循环。其中一个为监听程序，监听客户端发来的消息，如果接受到消息，则将消息写入信道。另一个线程则是主业务循环，不停地从信道中读取消息，发送给服务器端的 SF。

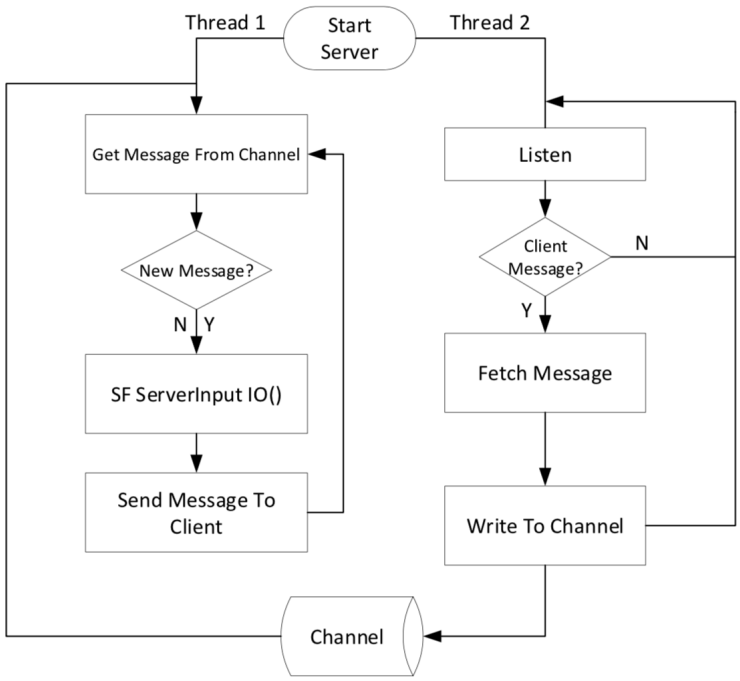

图 2.7: 服务器端流程图

服务器端的 SF 定义如下。SF 中维护了一个 ServerState 的服务器状态，根据输入信息来实时更新状态。

```
objSF :: SF (ServerInput, (ServerState, ServerState)) (IO(), (ServerState, ServerState))
objSF = proc (si, (sprev, s0)) -> do
   inputChange <- edgeBy (\old new -> if old/=new then Just new else Nothing) dummyServerInput -< si -- 检测输入变化
   let s1 = updateServerState (s0,inputChange) -- 更新服务器状态

   lps <- moveObjs allBullets bulletPos bulletVel -< s1 -- 更新子弹位置
   let s2 = s1 {allBullets = zipWithIL (\l p -> l {bulletPos = p})
            (const Nothing) (const Nothing)
            (allBullets s1) lps}
      hits = checkHits (sprev, s2) -- 检测子弹与玩家的碰撞事件
      scMsgs = toMessages (s2, inputChange, hits) -- 发送消息

   returnA -< (sendAll s2 scMsgs, (s2, s1))
```

服务器端与客户端发送是通过字符串来发送，因此需要定义一个类型类，将游戏中表示各种 Object 的数据结构信息转换到 String，还要能够从 String 转换为 Object 的数据结构信息。因此，定义了一个类型类，如下所示。这里仅利用 Bullet 做例子，Bullet 需要将其中的信息（位置信息，发射的玩家信息，ID，速度信息）打包为 String，然后发送，然后另一端进行解包更新相应的信息。

```
class Stringifiable a where
    stringify :: a -> String -- Object 转换为 String
    destringify :: String -> a -- String 转换为 Object

instance Stringifiable Bullet where
```

```
6
7   stringify l = (show $ bulletID l) ++ delim ++
8               (show $ bulletpID l) ++ delim ++
9               (showVec3d $ bulletPos l) ++ delim ++
10              (showVec3d $ bulletVel l)
11     where delim = ";"
12
13  destringify s = let untildelim = span (/= ';')
14                      (p1,s1) = untildelim s
15                      (p2,s2) = untildelim $ drop 1 s1
16                      (p3,s3) = untildelim $ drop 1 s2
17                      (p4,_) = untildelim $ drop 1 s3
18                  in Bullet {bulletID = read p1,
19                             bulletpID = read p2,
20                             bulletPos = readVec3d p3,
21                             bulletVel = readVec3d p4}
```

第三章　总结与展望

通过这次 Haskell 的期末 Project，我们首先是对 Haskell 的编程熟悉了很多。其次，在开发游戏的时候学到了很多额外的知识例如 OpenGL、Yampa 等等知识。对于本项目来说，考虑到运用了 SF，因此扩展起来很方便，可改进的地方有如下：加入声音（考虑 OpenGL）、从外部读入人物模型（考虑 objHackage）、物理引擎等等。

参考文献

[1] Courtney, Antony, Nilsson, Henrik, Peterson, and John. The yampa arcade. *In Proceedings of the 2003 ACM SIGPLAN Haskell Workshop (Haskell'03*, pages 7 – 18, 2003.

[2] Gabriel. State of the haskell ecosystem. https://github.com/Gabriel439/post-rfc/blob/master/sotu.md. Dec 5, 2017.

[3] HaskellWiki. Game development. https://wiki.haskell.org/Game_Development. 23 May 2017.

[4] HaskellWiki. Yampa. https://wiki.haskell.org/Yampa. 20 June 2017.

[5] Paul Hudak, Antony Courtney, Henrik Nilsson, and John Peterson. *Arrows, Robots, and Functional Reactive Programming*. Springer Berlin Heidelberg, 2003.

[6] 向世明. *OpenGL 编程与实例*. 电子工业出版社, 1999.