Functional Reactive Programming

Studienarbeit
Abteilung Informatik
Hochschule für Technik Rapperswil

Herbstsemester 2015

Autor(en): Philipp Meier, Elias Geisseler
Betreuer: Josef Joller
ABSTRACT
Funktionale Programmiersprachen haben in der letzten Zeit an Interessenten gewonnen, nicht zuletzt weil sich Reactive Programming mit dem funktionalen Programmierparadigma sehr gut vereinen lässt (FRP).

Ziel dieser Arbeit war es, sich in FRP einzuarbeiten und ein einfaches Spiel zu erstellen welches unsere Fortschritte ansprechend aufzeigt. Das Lernen des für uns neuen FRP Paradigmas stand im Vordergrund. Im Gegensatz zur OO-Programmierung bei welcher bekannte Patterns und SW Engineering Techniken angewendet werden können, ist FRP noch nicht so weit entwickelt. Deswegen haben wir viele verschiedene Lösungsansätze verglichen und erarbeitet, um so unsere eigenen Best Practices zu erhalten.


Für die Schnittstelle vom Haskell-Teil zum Input (Tastatureingaben) und Output (Pixi.js-Rendering) haben wir einen JavaScript-Zwischenlayer implementiert.

Das Ergebnis der Arbeit ist ein funktionsfähiger Prototyp eines simplen 2D-Spieles welches in modernen Browsern läuft. Da wir keine guten Beispiele im Web für diesen FRP-Programmstack gefunden haben macht dies unsere Arbeit einzigartig.
EIGENSTÄNDIGKEITSERKLÄRUNG

Wir erklären hiermit,

- dass wir die vorliegende Arbeit selber und ohne fremde Hilfe durchgeführt haben, ausser derjenigen, welche explizit in der Aufgabenstellung erwähnt ist oder mit dem Betreuer schriftlich vereinbart wurde.

- dass wir sämtliche verwendeten Quellen erwähnt und gemäss gängigen wissenschaftlichen Zitierregeln korrekt angegeben haben.

- dass wir keine durch Copyright geschützten Materialien (z.B. Bilder) in dieser Arbeit in unerlaubter Weise genutzt haben.

Ort, Datum: Rapperswil, 17.12.2015

Name, Unterschrift:

Elias Geisseler         Philipp Meier
INHALTSVERZEICHNIS

Abstract ......................................................................................................................................................... 1
Eigenständigkeitserklärung ........................................................................................................................... 2
Danksagung ................................................................................................................................................... 4
Aufgabenstellung .......................................................................................................................................... 5
Management Summary ................................................................................................................................ 6
Technischer Bericht ....................................................................................................................................... 7
Einleitung / Übersicht ................................................................................................................................... 7
  Was ist Functional Reactive Programming? ............................................................................................. 7
  Vorgehen und Technologie-Entscheidungen .............................................................................................. 8
  Das Spiel .................................................................................................................................................... 11
Architektur ................................................................................................................................................... 12
  Grobübersicht .......................................................................................................................................... 12
  Modulübersicht ....................................................................................................................................... 13
  Wire Datenstrom .................................................................................................................................... 14
  Statemachine ......................................................................................................................................... 15
  Building mit GHCJS und FFI .................................................................................................................. 16
Fazit ............................................................................................................................................................. 20
Glossar .......................................................................................................................................................... 21
Literaturverzeichnis .................................................................................................................................... 22
Anhänge ....................................................................................................................................................... 23
  Installation .............................................................................................................................................. 23
  JAVA Map Compiler ............................................................................................................................... 23
DANKSAGUNG

Als erstes möchten wir uns bei Herrn Joller bedanken, der es uns ermöglichte, uns in ein neues Programmier-Paradigma einarbeiten zu können und uns dabei unterstützte. Er gab uns viele Freiheiten bei der Bewältigung der Aufgabenstellung, was dazu geführt hat dass wir bei dieser Arbeit sehr viel Neues dazulernen konnten.

Danken möchten wir auch der GHCJS Community, welche uns Tipps und Tricks jenseits der Dokumentation gab.
AUFGABENSTELLUNG

AUFGABE
- Einarbeiten in die funktionale Programmierung
- Erarbeitung einer Übersicht über existierende Functional Reactive Programming Frameworks (Reactive Banana, Sodium, Reactive, Yampa, Netwire)
- Einarbeiten in FRP mithilfe eines FRP Packages (Tutorial)
- Umsetzung der Erkenntnisse in einem einfachen Spiel

ERWARTETE RESULTATE


Das Ergebnis ist ein Showcase für FRP.

TERMINE
Start: 2015.09.14
Ende: 2015.12.18

BETREUUNG
Betreuer: Joller, Josef M.

Wöchentliches Treffen mit Betreuer jeweils am Dienstag.
MANAGEMENT SUMMARY

Unser Ziel war es uns in FRP einzuarbeiten und gelerntes in einem einfachen Spiel umzusetzen. Das Spiel zeigt unseren Fortschritt in einem einfach verständlichen Umfeld auf.


Das Spiel ist aber auch Technisch interessant, denn Haskell mit FRP auf einem Browser laufen zu lassen ist eine neue Entwicklung die offiziell noch nicht zuvor gemacht wurde.
TECHNISCHER BERICHT

EINLEITUNG / ÜBERSICHT

WAS IST FUNCTIONAL REACTIVE PROGRAMMING?


Im Grunde funktioniert es ähnlich wie in einem Excel-Sheet. Wird zum Beispiel der Wert der Zelle A1 aus der Summe der Zellen B1 und C1 berechnet (A1=B1+C1), dann wird A1 automatisch aktualisiert sobald sich der Wert der Zellen B1 oder C1 ändert. [1]

Reactive Programming eignet sich vor allem für die Realisierung von interaktiven Programmen, deren Daten/State sich laufend ändern. Man kann sich zum Beispiel einen Internet-Chat vorstellen bei dem andauernd neue Messages hereinkommen. Der Stream von Messages wäre dann eine Reactive Value (Je nach Framework auch Observable, Behavior, Wire, etc.).


Abbildung 1: Beispiel für die Anwendung der funktionalen Methoden Map und Scan auf eine Reactive Value
Vorgehen und Technologie-Entscheidungen

Im Folgenden erläutern wir unser Vorgehen bei der Einarbeitung in *Haskell* und FRP und begründen die Entscheidungen die wir dabei treffen mussten.

Wahl der Programmiersprache und Einarbeitung
Die Wahl der Programmiersprache war wohl die schwerwiegendste Entscheidung.


Da wir bei der Studienarbeit aber vor allem etwas Neues lernen wollten, entschieden wir uns stattdessen in das uns noch relativ unbekannte Gebiet der Funktionalen Programmiersprachen einzutauchen. Im Modul *Compilerbau* hatten wir bereits einen kleinen Einblick in die Funktionale Programmierung und in *Haskell* erhalten. *Haskell* ist eine pure funktionale Programmiersprache, d.H. man ist gezwungen die funktionale Denkweise zu verinnerlichen und kann nicht auf imperative oder objektorientierte Lösungsansätze zurückgreifen, wie man es z.B. in *Scala* könnte.

Da *Haskell* die mitunter am meisten verbreitete funktionale Programmiersprache ist, und Herr Joller auch bereits einiges an Erfahrung mit ihr hat, war sie die offensichtliche Wahl.


Grundsätzlich können wir das Buch und die Videos als einen ersten Einstieg in *Haskell* empfehlen. Es ist aber bei *Haskell* wie in anderen Programmiersprachen auch, man kann zwar die Syntax und die Eigenheiten der Sprache lernen, für den praktischen Gebrauch ist es aber fast genauso wichtig die Funktionalitäten und Eigenheiten der Standard-Libraries zu erlernen. Dazu können wir die *WikiBooks*\(^3\) Seite zu *Haskell* zum Nachschlagen empfehlen.

---

\(^1\) [http://reactivex.io/](http://reactivex.io/)


\(^3\) [http://en.wikibooks.org/wiki/Haskell](http://en.wikibooks.org/wiki/Haskell)
WAHL DER PLATFORM
Herr Joller hat uns darauf aufmerksam gemacht dass es auch möglich ist Haskell in JavaScript zu kompilieren. Das war für uns sehr verlockend, da JavaScript unabhängig vom Betriebssystem in jedem Browser läuft. Auch für die Umsetzung des User Interfaces haben wir mit JavaScript im Browser viele Möglichkeiten und unser Team hatte damit bereits ein wenig Erfahrung.


HASTE
Der Haste-Compiler hat bei uns anfangs den besten Eindruck gemacht. Er lässt sich leicht via Installer auf Windows installieren, die Dokumentation und die Beispiele machen einen guten Eindruck, und der vom Compiler generierte JS-Code ist schnell und kompakt.

Unsere anfänglichen Tests mit Haste verliefen gut. Auch die Schnittstelle von Haskell-Code zu JS-Native-Code via FFI (Foreign Function Interface) hat funktioniert.


GHCJS
Der GHCJS-Compiler rühmt sich darauf, praktisch alle puren Haskell-Libraries in JavaScript übersetzen zu können. Deshalb war er nach unseren Problemen mit Haste die nächste offensichtliche Option.

Die Installation von GHCJS auf Windows ist eher mühsam. Es gibt keinen Installer, sondern man muss man sich den Quellcode von GHCJS selber herunterladen und kompilieren. Das kann auch gerne mal länger als 30 Minuten gehen und benötigt zudem eine Unix-Umgebung (MSYS2).

Der Aufwand zahlte sich aber aus: Tatsächlich hatte GHCJS keinerlei Probleme Netwire nach JavaScript zu übersetzen.

Leider ist die Dokumentation für GHCJS noch nicht so gut, was zu ein paar Problemen mit dem FFI führte. Glücklicherweise konnte uns damit aber jemand aus dem IRC-Channel von GHCJS weiterhelfen.

Ein weiterer Nachteil von GHCJS ist die gewaltige Menge von JS-Code der generiert wird. Unser Spiel war am Ende ca. 30 KB Haskell-Code, aus dem GHCJS satte 4 MB JavaScript-Code erzeugte!

Diese Nachteile waren für uns aber nicht so gravierend, weshalb wir uns schlussendlich für GHCJS entschieden haben. Am Ende unseres Projektes hatten wir zwar Probleme mit der Performance, wir können aber nicht mit Sicherheit sagen ob diese mit GHCJS zusammenhängen.

4 https://wiki.haskell.org/The_JavaScript_Problem
WAHL EINES FRP-FRAMEWORKS
Da wir uns im Gebiet von FRP noch überhaupt nicht auskannten, war es schwierig sich im Voraus für ein FRP-Framework zu entscheiden. Nachdem wir ein bisschen recherchiert haben, entschieden wir uns auf Empfehlung der Haskell-Community und unseres Betreuers für *Netwire*.


Wires lassen sich in *Netwire* beliebig kombinieren, verschachteln und aneinander hängen. Dadurch lässt sich der Datenfluss innerhalb der Applikation auf deklarative Weise abbilden. (Siehe auch Kapitel „Was ist Functional Reactive Programming?“)


Das führte dazu dass wir sehr viel herum probieren und “basteln” mussten: Wir haben verschiedene Lösungsansätze ausprobiert bis wir eine für uns funktionierende Methode gefunden haben.

GRAFISCHE DARSTELLUNG (USER INTERFACE / RENDERING)

Da die API von *WebGL/Canvas* eher low-level und damit mühsam zu verwenden ist, haben wir uns entschieden eine bestehende JS Render Engine zu verwenden.


Damit wir nicht die komplette API von *Pixi* in Haskell wrappern mussten, haben wir einen Zwischenlayer in *JavaScript* implementiert. Der Zwischenlayer erhält laufend den aktuellen Gamestate (= GameBoard) von der *Haskell*-Logik und erzeugt entsprechend die benötigten *Pixi*-Objekte (Sprites, Texturen, Animationen, Filter, etc.).

Die verschiedenen *Pixi*-Objekte bilden somit einen eigenen State für die Grafik (= ViewModel), und müssen immer mit dem vom *Haskell*-Teil übergebenen GameBoard synchronisiert werden. Für diesen Zweck haben wir den Elementen des GameBoard (Pickups, Enemies und Bushes) eine eindeutige ID vergeben.
DAS SPIEL

Um gelerntes in praktische Anwendung umzumünzen haben wir ein einfaches Spiel erstellt. Dabei steht weniger der Spielspass im Vordergrund. Das Spiel sollte eher als eine kontinuierlich wachsende Spielwiese dienen, welche uns ermöglichte, gelernte Konzepte in einem Projekt auszutesten, anstelle von vielen kleinen „Hello World“-Programmen die wenig Zusammenhang haben. Die höhere Komplexität zwang uns aber auch dazu verschiedene Lösungen genauer unter die Lupe zu nehmen und die Konsequenzen genauer abzuschätzen.

Das Spiel wurde beinahe jede Woche (implizite Iteration) um ein Feature erweitert, z.B. Pickups oder Wegfindung/Ai der Gegner. Durch diese Methode wurde also immer ein spielbarer Prototyp angestrebt, daher eher ein Bottom-up-Approach. Um dennoch ein gewisses Ziel vor Augen zu haben, entschieden wir uns eine Vision zu erarbeiten:

Es soll ein Top-Down Spiel werden vergleichbar mit „The Legend of Zelda: A Link to the Past“. Dabei sollen nur Basisfeatures implementiert werden: Bewegung, Animation, Gegner, Game Over State, ein Punktesystem ...

Abbildung 2: Eine Szene aus unserem Spiel
ARCHITEKTUR
GROBÜBERSICHT
Die Architektur des Spiels lässt sich grob in 3 Layer unterteilen (siehe Bild). Der Top-Layer in Haskell für die Spiellogik, der Mid-Layer in JavaScript für das Adaptieren von HS ↔ JS und der Bottom-Layer für die Darstellung der Grafik.

Abbildung 3: Übersicht über die Architektur unseres Spiels
MODULÜBERSICHT

HASSELS TEIL

● Game Loop: Main.hs  
Enthält rekursiven Loop für das Steppen der Wires und das Starten des Renderings.

● Game Logic: Logic.hs  
Enthält den Grossteil der Spiellogik in Form von Wires die miteinander kombiniert werden.

● Input Wire: Input.hs  
Stellt eine Wire zur Verfügung um die Anbindung an „Input Polling“ reactive zu machen.

● Ai: Ai.hs  
Enthält verschiedene Jagd-Strategien der Enemies und wählt eine passende aus.  

● Map / Pathing: Map.hs, Pathing.hs  
Die Map enthält Informationen über die Position der Bäume (= Bushes) und der Wege. Pathing nutzt eine A*-Haskell.Library um einen idealen Pfad zu einem Ort zu finden.

● Random: Random.hs  
Random Utilities für das Erzeugen von zufälligen Int und Float Werten.

● Domain: Domain.hs  
Enthält die Datentypen für die Logik welche per FFI an den Mid-Layer gesendet werden können.

● Collision: Collision.hs  
Funktionen und Interfaces für die Erkennung von Kreiskollisionen und Positions korrektur.

JAVASCRIPT TEIL

● Input Polling: getKey.js  
Simples mapping von JS-Key-Events in ein Array. Die Funktion getKey() wird von der Haskell-Logik aus aufgerufen um abzufragen ob ein bestimmter Key gedrückt ist. (Polling)

● Render Logic: renderGame.js  
Ansteuerung der Pixi.js Library. Um performant zu sein werden sichtbare Elemente zwischen zwei renderGame() aufrufen zwischen gespeichert, renderGame übernimmt diese Aufgabe.
**WIRE DATENSTROM**


Hier ein Kurzes Beispiel: Der Player und seine Position sind am Anfang mit dem Input und der statischen Map berechenbar. Der Punktestand ist aber abhängig von den Pickups welche der Player berührt hatte. Zudem kann der Player unverwundbar sein, wenn er gerade eben ein Leben verloren hat weil er von einem Enemy berührt wurde. Somit kann die Player-Struktur erst dann final erstellt werden, wenn alle diese Datenflüsse eintreffen. Wir können diese Daten erst im wGameBoard bündeln (siehe Bild), darum ist der Player überall verteilt. Der Datenfluss ist jedoch klar ersichtlich, man weiß exakt wo welche Daten in welcher Folge kombiniert werden. Bei OO ist dieser Datenfluss nicht so klar ersichtlich.

![Abbildung 4: Übersicht über den wGameBoard Wire. Ein Wire wird hier als Hexagon mit Inputs und Outputs dargestellt.](image)
Hier eine kurze Erklärung zu jedem Wire im Bild:

- **wPos (Position):** Mit gegebener Richtung (= Input) kann die neue Position des Spielers berechnet werden.
- **wDir (Direction):** Bleibt der Spieler stehen merkt sich diese Wire die Gehrichtung der letzten Eingabe. Wird benötigt wenn der Spieler stehen bleibt
- **wPick (Pickups):** Erstellt neue Pickups und bestimmt mit welchen der Spieler kollidiert ist (aufheben).
- **wEnemy: Erstellt neue Enemies und berechnet deren Position. Benutzt Ai um den Spieler zu jagen.**
- **wEnemyColl: Prüft ob der Spieler von einem Enemy berührt wird. Es werden evtl. Lebenspunkte abgezogen und die Unverwundbarkeits-Zeit neu berechnet.**


**STATEMACHINE**

Wires können während der Laufzeit einfach durch andere ersetzt werden, siehe Wire Kapitel. Wir haben diese Tatsache ausgenutzt um eine Statemachine zu erstellen „Wire for State“-Pattern sozusagen. Der Input kann so auf wNewBoard, wGameBoard oder wOverBoard umgeleitet werden. Diese Wires repräsentieren folgende States:

- **wNewBoard:** Info-Screen vor dem Start des Spieles
- **wGameBoard:** Aktive Spielsimulation
- **wOverBoard:** Letzter Spielstatus mit „Game Over“-Overlay

![Abbildung 5: Statemachine mit den 3 Wires](image-url)

NETWIRE WIRE


In der Dokumentation werden viele verschiedene Arten von Wires beschrieben, diese lassen sich auf häufig untereinander kombinieren:

- Konstanten
- Stateless-/Stateful-Wires
- Switches die auf andere Wires umleiten können oder diese verbinden
- One-Shot Funktionen die nach Ablauf auf andere Wires wechseln
- Wire mit/ohne Session-Step

Wir haben für dieses Projekt hauptsächlich Wires in folgender Form verwendet:

```haskell
positionWire startx starty inertia = go startx starty inertia
where
  go xOld yOld inertiaOld = mkGen (\dt (inputDirectionX, inputDirectionY) -> do
      ...
      return (Right (xNew, yNew), go xNew yNew inertiaNew)
```

Im Haskell-Code lässt sich diese Form beinahe in jedem File finden. Diese Form des Wires ist sehr flexibel (Session-Step, Stateful/Stateless) und hat einen hohen Wiedererkennungswert. Wir möchten hier auf die einzelnen Teile dieser Wire eingehen:

- go xOld yOld inertiaOld, go ist die Recursive-Wire-Construction-Function, hier mit 3 Parametern (State)
- go startx starty inertia, ist der initiale Aufruf der Wire-Construction-Function, hier können Anfangswerte übergeben werden
- go xNew yNew inertiaNew, ist der rekursive Aufruf der Wire-Construction-Function, hier wird der neue State übergeben.
- mkGen generiert eine Stateful-Session-Stepped-Wire mit der übergebenen Verarbeitungsfunktion. (ist Teil der Netwire Library)
- \( dt \) ist der Session-Step-Input der Wire (Delta-Time seit dem letzten Step)
- \((inputDirectionX, inputDirectionY)\) ist der Haupt-Input der Wire
- \(\text{Right} (xNew, yNew)\) ist der Haupt-Output der Wire, in diesem Fall blockiert die Wire nie.

(Right (xNew,yNew), go xNew yNew inertiaNew) ist der eigentliche Output der Wire. Er ist aufgeteilt in Haupt-Output und in eine neue Wire, welche den nächsten verarbeitenden Schritt macht. Mit Hilfe dieser Wire kann der State „verändert“ werden.
STATEFUL WIRE VS. STATELESS GAMESTATE OBJECT

Während der Implementierung haben wir uns an Beispielen im Internet orientiert wie Netwire genutzt werden kann. Zwei mögliche Grundideen lassen sich daraus klar erschliessen:

1. Der State wird an Ort und Stelle durch Recursive-Wire-Construction gehalten
2. Der State ergibt sich aus altem State und dem Session-Step

Variante 2 zeichnet sich dadurch aus als dass alle Wires stateless sind. So wird der alte Gamestate und der Input (z.B. Gehrichtung) dem Wire übergeben- Der Wire erstellt einen neuen Gamestate aus dem alten Gamestate und dem Input, dieser neue Gamestate wird dann der nächsten Wire weiter gegeben und so weiter modifiziert, bis der Gamestate für das aktuelle Frame bereit ist. Jede Wire kann alles im State verändern, was zu Indecent Exposure führt und Inappropriate Intimacy begrüsst.

In Variante 1 wird der State durch das übergeben des neuen States an die Recursive-Wire-Construction-Function gehalten:

\[
\text{positionWire startx starty inertia} = \text{go startx starty inertia}
\]

where

\[
\text{go } x\text{Old } y\text{Old } \text{inertia} = \text{mkGen } (\Delta t \ (\text{inputDirectionX}, \\ \text{inputDirectionY}) \rightarrow \text{do } \ldots)
\]

Hier ist \textit{go} die \textit{Wire-Construction-Function}, \textit{xOld}, \textit{yOld} und \textit{inertia} bilden den State der alten Wire.


RENDER GAME STATEFUL

PERFORMANCEPROBLEME
Auf allen Browser lassen sich Performanceprobleme feststellen. Die Framerate bricht plötzlich und ohne klaren Grund ins Bodenlose, Ruckler können bis zu einer Sekunde lang andauern (Frame zu Frame). Es ist aber meist so, dass sich die Lager nach einer anfänglichen Ruckelperiode verbessert. Da der GHCJS Output-Code abstrakt und unleserlich ist, kann man nur schwer herausfinden wo das Problem liegt. Ebenfalls lässt sich die Garbage-Collection nicht gezielt auslösen. Garbage-Collection erfolgt zufällig, was ebenfalls zu Framerate-Einbrüchen im Sekundenbereich führt.

Testmaschine/ Umgebung: ThinkPad W541, i7-4810MQ, Win 10, Balanced Performance Setting, Firefox 42, 2015.12.17
FAZIT
Der von uns verwendete Stack mit Haskell, Netwire, GHCJS und Pixi.js war gut um sich mit den verschiedenen Technologien und FRP vertraut zu machen. Für ein reales, größeres Projekt ist er unserer Meinung nach aber nicht oder nur bedingt geeignet. Der Grund dafür ist vor allem, dass der von GHCJS erzeugte Code praktisch unlesbar und nur schwer mit dem Haskell-Quellcode assoziierbar ist. Das hat zur Folge, dass sich Probleme nur schwer orten lassen (Man kann keinen Breakpoint im Haskell-Code setzen) und Performance-Profiling praktisch unmöglich ist.

Wir haben das Spiel hauptsächlich Bottom-Up implementiert, sprich Schritt für Schritt Features hinzugefügt. Dieses Vorgehen ist sicher empfehlenswert für Neueinsteiger, die noch experimentieren wollen. Nun jedoch würden wir das System wohl Top-Down konzipieren, also zuerst den Datenfluss definieren und dann implementieren.

Haskell ist eine sehr flexible Sprache, diese Flexibilität hat aber auch Ihre Kehrseiten. Wir hatten zu Beginn sehr viele Mühen, unsere Programmfehler zu erkennen und die Compiler-Errors richtig zu interpretieren. Die Schwierigkeit der Compiler-Errors ist häufig die Type-Inference, da Typen „so spät wie möglich“ gebunden werden, können Fehler mit ganz anderen Codestellen assoziiert werden. Ein Ausweg bietet das Freiwillige Typisieren der Haskell-Funktionen, damit werden Fehler nicht durch das ganze System getragen. Wir empfehlen allen Anfängern die Typen zu deklarieren.
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>FRP</strong></td>
<td><em>Functional Reactive Programming</em></td>
</tr>
<tr>
<td><strong>HS</strong></td>
<td><em>Haskell</em></td>
</tr>
<tr>
<td><strong>JS</strong></td>
<td><em>JavaScript</em></td>
</tr>
<tr>
<td><strong>FFI</strong></td>
<td>Foreign Function Interface. Die GHCJS Schnittstelle mit <em>JavaScript</em>.</td>
</tr>
<tr>
<td><strong>GHCJS</strong></td>
<td>Cross-Compiler von HS zu JS</td>
</tr>
<tr>
<td><strong>Map</strong></td>
<td>Karte/Landkarte/Umgebung in Spielen</td>
</tr>
</tbody>
</table>
LITERATURVERZEICHNIS


ANHÄNGE

INSTALLATION

Das Spiel wurde auf einem Windows System in einer MSYS2 64bit Umgebung entwickelt und sollte daher mit allen gängigen Unix Systemen kompatibel sein. Für die einfache Kompilation wurde ein Build-Script erstellt, „build.sh“ im Stammverzeichnis.

Folgende Komponenten müssen installiert oder konfiguriert werden um das Script erfolgreich auszuführen:

- GHCI Version 7.10.2
- GHCJS Version 0.2.0
    Im Verzeichnis „lib“ ist eine kompatible Version mit unserem Projekt abgelegt.
    Bitte GHCJS Github-Repo für die Installation beachten: https://github.com/ghcjs/ghcjs

    Snapshot installation:
    cabal install ./lib/ghcjs.tar.gz
    ghcjs-boot

Hinweis: Um Ihre Umgebung zu schützen empfiehlt es sich eine Cabal-Sandbox in Stammverzeichnis des Projekts zu erstellen.

JAVA MAP COMPILER

Um die Map zu ändern muss die Initialisierungsliste in „src/Map.hs“ angepasst werden. Wir haben ein kurzes Java Programm geschrieben welches ein PNG Bild in ein Initialisierungsstring umwandelt. Dieses Programm befindet sich in „jsrc/Bushing“. Die „map.png“ befindet sich ebenfalls an diesem Ort.

Am einfachsten lässt sich das Programm benutzen wenn man es durch die Eclipse IDE managen lässt.