

# **Game-Engine-Architektur mit funktional-reaktiver Programmierung in Haskell/Yampa**

GEROLD MEISINGER

## **DIPLOMARBEIT**

eingereicht am  
Fachhochschul-Masterstudiengang

DIGITALE MEDIEN

in Hagenberg

im September 2010

© Copyright 2010 Gerold Meisinger

Alle Rechte vorbehalten

# **Erklärung**

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 22. September 2010

Gerold Meisinger

# Inhaltsverzeichnis

<b>Erklärung</b>	<b>iii</b>
<b>Vorwort</b>	<b>vii</b>
<b>Abstrakt</b>	<b>viii</b>
<b>Abstract</b>	<b>ix</b>
<b>1 Motivation</b>	<b>1</b>
1.1 Einleitung . . . . .	1
1.1.1 Struktur der Arbeit . . . . .	2
1.2 Computerspielentwicklung . . . . .	2
1.2.1 Programm-Performance und C++ . . . . .	3
1.2.2 Rapid Application Development . . . . .	4
1.2.3 Prototyping in der Computerspielentwicklung . . . . .	6
1.3 Game-Engine-Architektur . . . . .	9
1.3.1 Klassische objektorientierte Architektur . . . . .	9
1.3.2 Komponentenbasierte Architektur . . . . .	10
1.3.3 Anwendungsfall: Revive . . . . .	11
1.3.4 Mängel an der Architektur . . . . .	13
1.3.5 Lösungsansätze . . . . .	14
<b>2 Basisfunktionalitäten eines Computerspiels</b>	<b>15</b>
2.1 Einleitung . . . . .	15
2.1.1 Definition: Computerspiel . . . . .	15
2.1.2 Definition: Game-Engine . . . . .	16
2.2 Eingabe . . . . .	17
2.2.1 Zeit . . . . .	18
2.2.2 Human Interface Devices . . . . .	19
2.2.3 Externe Daten . . . . .	21
2.2.4 Sonstige . . . . .	21
2.3 Verarbeitung . . . . .	21
2.3.1 Physik . . . . .	22
2.3.2 Sonstige . . . . .	23

2.4	Ausgabe . . . . .	24
2.4.1	Grafik . . . . .	24
2.4.2	Debugging . . . . .	25
2.4.3	Sonstige . . . . .	26
<b>3</b>	<b>Funktional-reaktive Programmierung</b>	<b>27</b>
3.1	Einleitung . . . . .	27
3.2	Programmierparadigmen . . . . .	27
3.2.1	Imperative und deklarative Programmierung . . . . .	28
3.2.2	Objektorientierte Programmierung . . . . .	30
3.2.3	Funktionale Programmierung . . . . .	31
3.2.4	Reaktive Programmierung . . . . .	32
3.3	Haskell . . . . .	33
3.3.1	Notation . . . . .	34
3.3.2	Funktionen . . . . .	35
3.3.3	Typen . . . . .	36
3.3.4	Klassen . . . . .	37
3.4	Typklassen . . . . .	38
3.4.1	Funktoren . . . . .	39
3.4.2	Punktierte Funktoren . . . . .	39
3.4.3	Applikative Funktoren . . . . .	40
3.4.4	Monaden . . . . .	40
3.4.5	Arrows . . . . .	42
3.5	Yampa . . . . .	43
3.5.1	Signalfunktionen . . . . .	44
3.5.2	Reactimate . . . . .	47
3.5.3	Ereignisse . . . . .	47
3.5.4	Switches . . . . .	48
3.5.5	Implementierung . . . . .	49
<b>4</b>	<b>Implementierung der Game-Engine</b>	<b>51</b>
4.1	Einleitung . . . . .	51
4.1.1	Überlegungen . . . . .	51
4.2	Game-Objekte . . . . .	52
4.2.1	Eingabedaten . . . . .	52
4.2.2	Ausgabedaten . . . . .	53
4.2.3	Implementierung . . . . .	54
4.3	Game-Loop . . . . .	54
4.3.1	reactimate . . . . .	56
4.3.2	pSwitchB . . . . .	57
4.3.3	pSwitch . . . . .	58
4.3.4	Implementierung . . . . .	59
4.4	Game-Logic . . . . .	62
4.4.1	Identitäten . . . . .	62

4.4.2	Verwaltung . . . . .	63
4.4.3	Verteilung . . . . .	64
4.4.4	Beobachtungen . . . . .	64
4.5	Zusammenfassung . . . . .	65
<b>5</b>	<b>Implementierung der Basisfunktionalitäten</b>	<b>67</b>
5.1	Einleitung . . . . .	67
5.2	Eingabe . . . . .	67
5.2.1	Input Re-Mapping . . . . .	67
5.2.2	Ressourcen . . . . .	69
5.3	Verarbeitung . . . . .	71
5.3.1	Frameanimation . . . . .	71
5.3.2	Bewegung . . . . .	73
5.4	Ausgabe . . . . .	74
5.4.1	Rendering . . . . .	74
5.4.2	Debugging . . . . .	74
<b>6</b>	<b>Schlussbemerkungen</b>	<b>76</b>
6.1	Ergebnisse . . . . .	76
6.1.1	Wiederverwendbarkeit . . . . .	76
6.1.2	Dynamische Funktionalität . . . . .	78
6.1.3	Definition: Game-Engine . . . . .	80
6.2	Ausblick . . . . .	80
6.3	Persönliche Meinung . . . . .	81
6.3.1	Schwierigkeiten . . . . .	81
6.3.2	Schlusswort . . . . .	83
	<b>Literaturverzeichnis</b>	<b>84</b>

# Vorwort

Ich habe nach meiner höheren technischen Ausbildung für EDV ein weiteres Studium in Informatik gewählt, um fortgeschrittene Programmiertechniken zu lernen. Die vorliegende Arbeit entstand nach mehreren Semestern experimentieren mit wiederverwendbaren Game-Engine-Architekturen und es freut mich daher insbesonders, bei meiner Abschlussarbeit auf die funktional-reaktive Programmierung gestoßen zu sein und dadurch zumindest das Problem der Wiederverwendbarkeit in Game-Engines für mich als gelöst ansehen zu können.

Ich bedanke mich bei folgenden Personen:

- Meinen Eltern *Katharina* und *Wolfgang Meisinger* für die finanzielle Unterstützung, aber insbesondere dafür, dass sie mir beigebracht haben, der Bildung einen hohen Stellenwert zu geben und mich immer an besseren Personen zu orientieren,
- meinen Freunden und Kollegen *Roland Aigner*, *Alexander Dammerer*, *Christian Herzog* und *David Steiner* für die inhaltlichen Diskussionen und Kritiken,
- meinen Freunden *Thomas Kappelmüller*, *Dimitri Prandner* und *Johanna Secklehner* für die Rechtschreibkorrekturen,
- Professor *Roman Divotkey* für die Einführung in komponentenbasierte Game-Engine-Architekturen,
- Professor *Stephan Dreiseitl* und *Erik Pitzer* für die Einführung in alternative Programmierparadigmen,
- Professor *Jürgen Fuss* für die Einführung in formale Grundlagen der Mathematik,
- der *Yale Haskell Group* für die Implementierung der funktional-reaktiven Programmierung,
- und *Paul Graham* für die Essays über funktionale Programmierung.

# Abstrakt

Die Entwicklung von Computerspielen ist mit hohen Kosten verbunden, weshalb in der Softwareentwicklung wiederverwendbare und erweiterbare Funktionalität angestrebt wird. Ein aktueller Trend in der Computerspielentwicklung ist der Einsatz von Rapid Application Development und Prototyping-Software, wobei die schnelle Entwicklung spielbarer Prototypen und neuer Funktionalität in den Vordergrund gestellt wird. Dabei ist vor allem die dynamische Veränderung der Funktionalität zur Laufzeit wichtig, ohne dabei die Programmierumgebung gegenüber allgemeinen Programmiersprachen einzuschränken. Mit den bisher verwendeten objektorientierten Programmiersprachen und den komponentenbasierten Game-Engine-Architekturen konnten aber kaum Wiederverwendbarkeit, Erweiterbarkeit oder dynamische Funktionalität erreicht werden und schränkten die Programmierung durch die verwendeten Techniken zusätzlich stark ein.

In dieser Arbeit wird eine allgemeine Game-Engine-Architektur und verschiedene Basisfunktionalitäten von Computerspielen mittels funktionalreaktiver Programmierung in Haskell/Yampa behandelt. Durch die rein funktionale Programmierung kann die Funktionalität in vollständig unabhängige und wiederverwendbare Funktionen ausgelagert werden, während durch die reaktive Programmierung der Zeitverlauf völlig abstrahiert wird und die Funktionalität intuitiv aus reaktiven Elementen verknüpft werden kann. Weiters ermöglicht Yampa die Modellierung mit konzeptionell kontinuierlicher Zeit. Eine Game-Engine lässt sich dadurch insgesamt besser in deren elementare Bereiche aufteilen, wodurch sich der Datenfluss am Eingabe-Verarbeitung-Ausgabe-Modell veranschaulichen lässt und die grundlegende Funktionalität einer Game-Engine definiert werden kann. Abschließend werden die grundsätzlichen Grenzen von komponentenbasierten Game-Engine-Architekturen und dynamischer Funktionalität behandelt.

# Abstract

Development of computer games comes at a high cost, thus reusable and extensible functionality is very essential in software development. A recent trend in computer game development is establishing Rapid Application Development and use of prototyping software, emphasizing fast development of playable prototypes and new functionality. Especially dynamic modification of functionality during runtime without limiting the programming environment compared to general purpose programming languages is important. Reusability, extensibility and dynamic functionality could hardly be achieved with object-oriented programming and component-based game engine architectures and the techniques used actually constrained programming.

In this thesis, general game engine architecture and various basic features of computer games are examined using functional-reactive programming in Haskell/Yampa. Due to pure functional programming, functionality can be encapsulated into completely independent and reusable functions, while reactive programming abstracts the flow of time and functionality can be composed intuitively via reactive elements. Furthermore, continuous time semantics are possible with Yampa. Overall, a game engine can be more clearly defined and divided into its basic features, thus allowing the data flow to be comprehensibly illustrated on the input-processing-output model. Finally, the principle limits of component-based game engine architectures and dynamic functionality are discussed.

# Kapitel 1

## Motivation

### 1.1 Einleitung

Während in den frühen Jahren der Computerspiele nur ein bis zwei Personen an der Entwicklung beteiligt waren, ist dies heute nur mehr vereinzelt im *casual-* und *mobile gaming* Markt möglich. Große, kommerzielle Computerspielprojekte werden heutzutage von professionellen Studios durchgeführt, welche oft über 100 Mitarbeiter zählen und über Budgets im 2-stelligen Millionenbereich verfügen. Die Computerspielindustrie zählt mit insgesamt 11,7 Mrd. Dollar Umsatz im Jahr 2008 zu den größten Unterhaltungsindustrien in den USA [Ent09] und das bisher teuerste Computerspiel *Grand Theft Auto 4* hat geschätzte Kosten von 100 Mio. Dollar vorursacht<sup>1</sup>.

Computerspielprojekte besitzen einen hohen Kapitalaufwand und sind dadurch hohen Zeit- und Kostenrisiken ausgesetzt. Es wäre daher wünschenswert, diese Risiken zu verringern, indem Verbesserungen in den verschiedenen Projektbereichen durchgeführt werden. In der Softwareentwicklung wird dies unter anderem durch Wiederverwendung von bestehenden Programmcodes erreicht, wobei insbesondere in der Computerspielentwicklung *komponentenbasierte Game-Engine-Architekturen* eine hohe Wiederverwendbarkeit versprechen. Des Weiteren ist ein Trend in Richtung *Rapid Application Development* erkennbar, wobei die schnelle Entwicklung von spielbaren Prototypen im Vordergrund steht. Dadurch sollen bereits in frühen Projektstadien die Anforderungen besser spezifiziert und wiederum Kosten vermieden werden können. Eine *Game-Engine* mit hoher Wiederverwendbarkeit würde natürlich auch die Entwicklung von Prototypen zusätzlich beschleunigen.

Ziel dieser Arbeit war es, aufbauend auf den bisherigen Erfahrungen mit komponentenbasierten Game-Engine-Architekturen **ein neues Programmierparadigma**<sup>2</sup> zu finden und anzuwenden, welches einerseits die Wie-

---

<sup>1</sup><http://www.gamesindustry.biz/articles/gta-iv-most-expensive-game-ever-developed>

<sup>2</sup>Ein Programmierparadigma definiert welche Abstraktionen mit einer Programmiersprache unmittelbar ausgedrückt werden können.

derverwendbarkeit weiter erhöht, andererseits aber auch eine intuitive Programmierung und direkte Verknüpfung der Funktionalität ermöglicht. Bei der Recherche hat sich die **funktional-reaktive Programmierung** als, sehr geeignet für die Domäne der Computerspiele herausgestellt und wird am Beispiel der Programmiersprache *Haskell* und Programmbibliothek *Yampa* behandelt. Die *rein funktionale Programmierung* stellt dabei sicher, dass jede Funktionalität nur – und wirklich nur – von ihren Eingabeparametern abhängt, anstatt von externen Zuständen. Damit wird vor allem die Teilung in kleine, unabhängige Module ermöglicht und genau dadurch die Wiederverwendbarkeit erreicht. Die *reaktive Programmierung* ermöglicht eine einfache Modellierung von kontinuierlichen Eingabestromen, wodurch in Computerspielen insbesondere der Zeitverlauf abstrahiert werden kann. Mittels (imperativer) objektorientierter Programmierung (z. B. *Java*, *C++*, *C#*) konnten diese Ziele bisher nicht vollständig erreicht werden. Zusätzlich hat sich das reaktive Programmierparadigma sehr gut zur Beschreibung einer allgemeinen Game-Engine-Architektur herausgestellt.

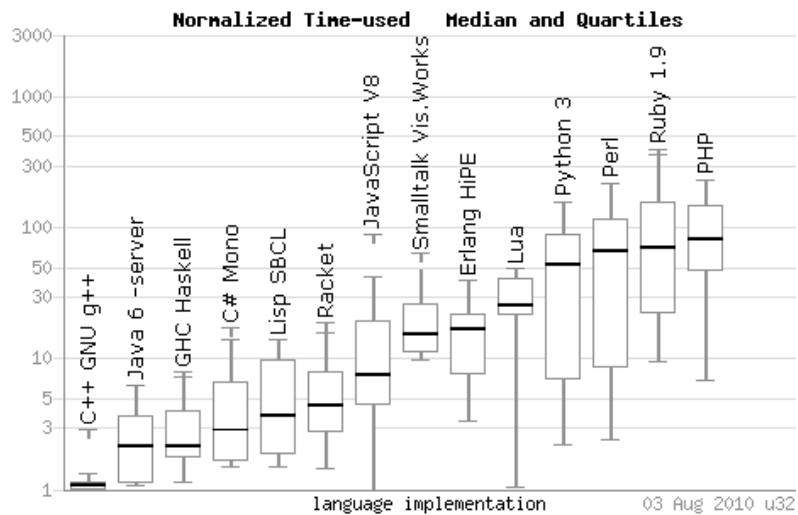
### 1.1.1 Struktur der Arbeit

In den folgenden Abschnitten werden als Motivation für ein neues Programmierparadigma zuerst die Softwareentwicklungsmethode Rapid Application Development und die aktuelle Vorgehensweise in der Computerspielentwicklung beschrieben, mitsamt den Schwierigkeiten die sich daraus ergeben. Abschließend wird die bisher empfohlene Lösung der komponentenbasierten Game-Engine-Architekturen beschrieben und die Probleme dargelegt, warum letztendlich die funktional-reaktive Programmierung gewählt wurde.

In Kapitel 2 werden die Eigenschaften von Computerspielen (z. B. Eingabe, Logik und Grafik) und häufig benötigte Basisfunktionalitäten (z. B. Bewegung und Kollisionserkennung) analysiert, während in Kapitel 3 näher auf die funktional-reaktiver Programmierung eingegangen wird. Abschließend wird in Kapitel 4 eine einfache Game-Engine implementiert und in Kapitel 5 die vorab beschriebenen Basisfunktionalitäten mittels funktional-reaktiver Programmierung in völlig unabhängige Module gekapselt.

## 1.2 Computerspielentwicklung

Computerspiele stellen interaktive *Virtual-Reality* dar und vereinen dabei viele Techniken und Inhalte aus verschiedenen Disziplinen, vor allem aus der Programmierung, aber auch aus Grafik, Sound, Narration usw. Diese Arbeit konzentriert sich aber vor allem auf die softwaretechnischen Aspekte eines Computerspiels. Die gesamten Aspekte und Bereiche der Computerspielentwicklung sind jedoch sehr gut beschrieben in [RM03] und [Gre09]. *Gregory* definiert Computer Spiele technisch als: *soft real-time interactive agent-based computer simulations* [Gre09, Abs. 1.2]. In der Computerspielentwicklung



**Abbildung 1.1:** Computer Language Benchmarks Game Ergebnisse vom 3. August 2010 (angepasst und entnommen aus <http://shootout.alioth.debian.org>).

wird allerdings statt *Agent* üblicherweise der Begriff *Game-Objekt* verwendet, wobei ein Game-Objekt die Objekte und deren Funktionalität im Computerspiel repräsentiert und sowohl sichtbare Objekte (z. B. den Spieler) als auch rein logische Objekte (z. B. eine Ziellinie) umfasst.

### 1.2.1 Programm-Performance und C++

Beginnend mit frühen Computerspielen wie z. B. *Space Travel*, treiben Computerspiele bis heute wesentlich die Entwicklung von Hardware – insbesondere von Grafikprozessoren – voran und werden zum Benchmarking von Hardware eingesetzt. Vor allem weil Performance ein sehr kritischer Faktor in Computerspielen ist, hat sich die Programmiersprache *C++* als Industriestandard in der Computerspielentwicklung durchgesetzt. Eine Gegenüberstellung der gängisten Programmiersprachenimplementierungen im Benchmark des *Computer Language Benchmark Game*<sup>3</sup> am 3. August 2010 ergab dabei, dass der *GNU C++ Compiler* mit Abstand die performantesten Programme erzeugt, vor dem *Java 6 -server* und dem *Glasgow Haskell Compiler*. Die Tests werden auf vier Computersystemen und zwölf verschiedenen Algorithmen durchgeführt, sind allerdings nach Aussage der Autoren des Benchmarks selbst, sehr vorsichtig zu interpretieren. Eine Übersicht der Ergebnisse vom August findet sich in Abbildung 1.1.

Es mehrt sich jedoch die Kritik am Einsatz von *C++* als alleinige Pro-

<sup>3</sup><http://shootout.alioth.debian.org>

grammiersprache in der Computerspielentwicklung. Vor allem nicht-performancekritische Programmteile werden immer öfters in sogenannte *Script-sprachen* ausgelagert. *Blow* kritisiert etwa [Blo04]:

Currently there are three levels of programming in games: script code, gameplay code, and engine code.

[...]

Visual Studio seems to be aimed heavily at developers of Visual Basic and C applications, and to the extent it caters to C++, it's meant for applications that make heavy use of COM objects and create many windows with variegated UI elements. We do very little of that stuff in modern games. We would much rather have that manpower spent to make the system compile programs quickly, or generate efficient code, or produce reasonable error messages for code that uses C++ templates. Even so, Visual C++ is the best compiler we have on PCs – with no competitive alternatives – so we're just sort of along for the ride.

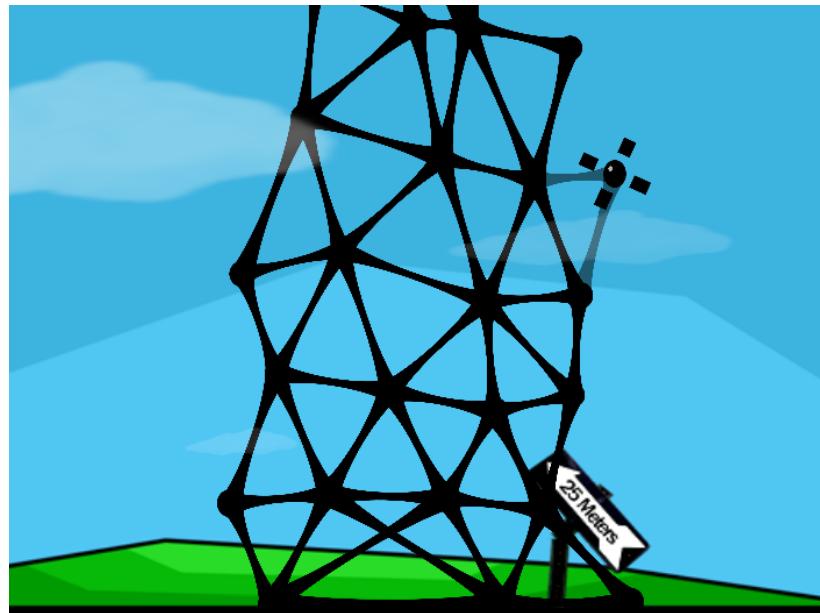
[...]

On the programming side, our compile/edit/debug cycles are usually far too long. Many games take half an hour or longer to compile when starting from scratch, or when a major C++ header file is changed. Even smaller changes, causing a minimal amount of recompilation and relinking, can take as long as two minutes. In general, C++ seems to encourage long build times. Once the build time has grown too long, a team may end up putting a significant amount of work into refactoring their source code to make it build more quickly.

Für einzelne Prototypen oder kleine Computerspiele wären heutige Computer leistungsstark genug, um höhere Programmiersprachen und spezielle Programme zur Entwicklung verwenden zu können. Besonders aber im Hinblick auf Rapid Application Development wäre die Programm-Performance jedoch eine vernachlässigbare Anforderung.

### 1.2.2 Rapid Application Development

*Rapid Application Development*, manchmal auch nur *Rapid Prototyping* genannt, ist eine Softwareentwicklungs methode, welche in der Computerspielentwicklung die schnelle Entwicklung von spielbaren Prototypen in den Vordergrund stellt. Computerspiele enthalten viele designhaltige und künstlerische Aspekte, welche sich jedoch nur sehr schwer planen lassen. Prototypen unterstützen dabei, diese Aspekte vorab gut abschätzen zu können. Das Design wird durch Rapid Prototyping ständig getestet, weiterentwickelt oder



**Abbildung 1.2:** *Tower of Goo*, der Prototyp zu *World of Goo*.

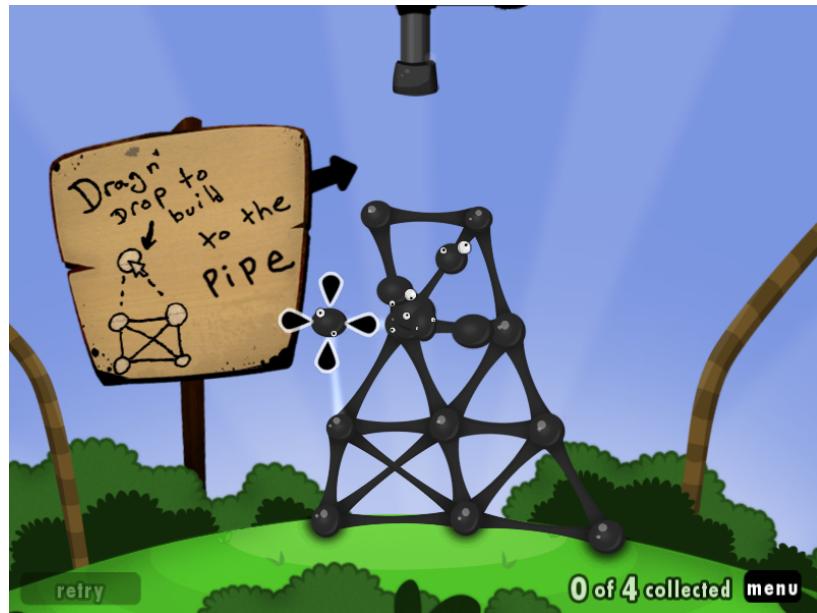
verworfen, wodurch vor allem unerwünschte Designentscheidungen, und damit auch sämtliche Kosten, für die Entwicklung vollständig ausgereifter Versionen davon, frühzeitig vermieden werden. *Deloura* schreibt dazu [Del09]:

[...] developing prototypes is an important process that helps establish the game's unique tone and direction before gearing up for full production. Finding the fun in a game concept in the early stages of a project makes it much easier to pitch and get funded, and highlights the game development challenges that will have to be solved during the length of the project.

Rapid Application Development wurde sehr erfolgreich im *Experimental Gameplay Project* eingesetzt, einem Studentenprojekt, welches mittlerweile als Webseite weitergeführt wird<sup>4</sup>. Ziel war es, innerhalb eines Semester jede Woche einen spielbaren Prototypen zu einem vorher festgelegten Thema zu erstellen [GGMS05]. Daraus entstand das Computerspiel *Tower of Goo* (Abbildung 1.2), welches später zum kommerziell erfolgreichen Titel *World of Goo* (Abbildung 1.3) erweitert wurde. Als Prototyping-Software wurde dabei ein selbstprogrammiertes Framework eingesetzt, welches ebenfalls in C++ geschrieben wurde.

---

<sup>4</sup><http://experimentalgameplay.com>



**Abbildung 1.3:** *World of Goo*, das kommerzielle Endprodukt zu *Tower of Goo*.

Bei Rapid Application Development in der Computerspielentwicklung wird dabei so vorgegangen, dass vor der Implementierung des Prototypen konkrete Entscheidungsfragen an das Game-Design formuliert werden, welche anschließend durch den Prototypen positiv oder negativ beantwortet werden [GH06]. Als Prototyp ist dabei alles erlaubt, was die gestellte Frage beantworten kann, weshalb die Funktionalität auch aus völlig fremden Bereichen ausgeborgt werden kann, solange diese dadurch ausreichend genug vorgetäuscht werden kann. Die Werkzeuge zur Erstellung von Prototypen umfassen z. B. Papier-und-Bleistift, Improvisation mittels bestehender Programme oder Modifikation bestehender Computer-Spiele. Erst in letzter Konsequenz sollte auf die Softwareentwicklung zurückgegriffen werden, um völlig neue Funktionalität zu implementieren, wenn dies auf andere Art nicht möglich ist. Insbesondere bei aufwendigen Algorithmen ist es aber selbst dann oft günstiger, einen leistungsstärkeren Computer zu kaufen, statt teure Entwicklungszeit in die Implementierung von schnellen Algorithmen zu stecken, welche mit hoher Wahrscheinlichkeit ohnehin nicht im Endprodukt verwendet werden.

### 1.2.3 Prototyping in der Computerspielentwicklung

Zum Prototyping ist wie in Abschnitt 1.2.2 beschrieben, grundsätzlich alles verwendbar, was eine Annäherung an das Endprodukt ermöglicht, während

nur mit klassischer Softwareentwicklung das eigentliche Endprodukt eines Computerspiels erstellt werden kann. Die technische Basis eines Computerspiels ist dabei eine Game-Engine, welche grundlegende Funktionalität zum Betrieb bereitstellt. Besonders bei kommerziellen Game-Engines bzw. Game-Frameworks wird aber oft auch zusätzliche Funktionalität mitgeliefert, wodurch die Entwicklung von vollständigen Computerspielen und auch Prototypen beschleunigt wird. Im Gegensatz dazu ist aber reine Prototyping-Software meist immer in irgendeiner Form auf einen bestimmten Anwendungszweck oder Genre beschränkt, indem die Funktionalität oder die Programmierumgebung für diese Bereiche vereinfacht wird. Dadurch wird die Entwicklung zwar oberflächlich erleichtert, insbesondere exotische Funktionalität kann dadurch aber wieder nur umständlich und unnötig komplex entwickelt werden. *Blow* schreibt dazu speziell im Zusammenhang von Kauf einer Game-Engine und Entwicklung neuer Technologie [Blo04]<sup>5</sup>:

If you're trying to make a game that is not doing anything new technologically, such a license [for a commercial game-engine] can be a safe decision. But if you're trying to be technologically expansive, you will probably run into the poor-fit problems mentioned earlier, but on a larger scale this time – you might find yourself spending \$500 thousand for code that you end up largely rewriting, disabling, or working around.

Die Grenze zwischen reiner Game-Engine und Prototyping-Software ist dabei jedoch unscharf, vor allem je allgemeiner anwendbar eine Prototyping-Software ist. Prototyping-Software muss jedoch, im Gegensatz zu einer Game-Engine, nur in der Entwicklung, nicht aber im Endprodukt verwendbar sein. Dadurch unterscheiden sich auch die Anwendungsbereiche und Anforderungen, welche an eine Prototyping-Software gestellt werden, wobei insbesondere die Performance nur einen Nebenfaktor darstellt:

**Prototyping:** In der Industrie liegt der Hauptanwendungsbereich von Prototyping-Software natürlich in der Entwicklung von Prototypen für Computerspiele, also darin neue Ideen für Design oder Funktionalität möglichst schnell auszutesten. Deshalb soll vor allem möglich sein die Funktionalität dynamisch zur Laufzeit verändern zu können, um dadurch die Entwicklung zu beschleunigen. Weiters soll die Entwicklungsumgebung eine **hohe Ausdrucksfähigkeit ermöglichen**, weil vor allem neue und exotische Funktionalität erforderlich ist. Idealerweise wird jedoch auch bereits **Grundfunktionalität** mitgeliefert, sodass der eigentliche Prototyp schnell in eine typische Computerspielumgebung eingefügt werden kann.

**Hobbyproduktion:** Im Hobbybereich der Computerspielentwicklung werden häufig *Game-Creator* eingesetzt (z.B. *The Games Factory* oder

---

<sup>5</sup>Die Einfügungen stammen vom Autor der vorliegenden Arbeit.

C/C++	73.2%
PENCIL AND PAPER	51.2%
FLASH	24.4%
LUA	22.0%
PRE-EXISTING ENGINE	7.3%
C#/XNA	4.9%

**Tabelle 1.1:** Welche Werkzeuge verwenden sie derzeit zur Erzeugung von Prototypen? (aus [Del09])

*Game Maker*), welche im Gegensatz zu Game-Engines meist einfacher sind, aber kaum zur Entwicklung von kommerziellen Produkten verwendet werden. Im künstlerischen Bereich wird oft Software zur Live-Virtualisierung verwendet (z. B. *VVVV* oder *Fluxus*), welche teilweise auch interaktiv gesteuert werden kann und sich dadurch ebenfalls kleine Computerspiele realisieren lassen.

**Ausbildung:** Die Entwicklung von eigenen Computerspielen ist vor allem bei Menschenkindern sehr beliebt, da es sich bei Computerspielen um ein stark visuelles Unterhaltungsmedium handelt. Dabei steht allerdings mehr der einfache und schnelle Einstieg im Vordergrund als der großer Funktionsumfang.

Im *Game Developer Magazine* wurde dabei zum Thema Game-Engines eine Umfrage mit etwa 100 leitenden Angestellten durchgeführt, um herauszufinden, welche Game-Engines verwendet werden und für welche Anwendungsbereiche diese eingesetzt werden [Del09]. Diese Befragung enthält einige interessante Ergebnisse:

1. Prototyping-Software ist für kleine Entwicklerstudios kaum verfügbar, da ausgereifte kommerzielle Game-Engines kaum erschwinglich sind und reine Prototyping-Software meist nur in großen Entwicklerstudios intern verfügbar ist.
2. Erweiterbarkeit eines Game-Frameworks wird einer großen Menge an vordefinierter Funktionalität vorgezogen.
3. Die Programmiersprache *C++* wird ebenfalls am häufigsten zur Erstellung von Prototypen eingesetzt, wie Tabelle 1.1 zu entnehmen ist.

Abschließend ist festzuhalten, dass der Einsatz von Rapid Application Development mithilfe von Prototyping-Software helfen kann, die hohen Kosten in der Computerspielentwicklung zu reduzieren, indem die Anforderungen besser abgeschätzt werden können. Bisherige Ansätze in Prototyping-Software und Game-Engine-Architektur sind jedoch meist auf einen bestimmten Anwendungszweck oder ein Genre beschränkt, indem die Funktionalität oder die Ausdrucksfähigkeit im Vergleich zu allgemeinen Program-

miersprachen eingeschränkt wird. In dieser Arbeit wird eine Game-Engine-Architektur mittels funktional-reaktiver Programmierung präsentiert, welche die volle Ausdrucksfähigkeit von vollwertigen Programmiersprachen besitzt und darüber hinaus eine intuitivere Modellierung mittels Arrow-Notation ermöglicht. Weiters wird durch die funktionale Programmierung eine hohe Wiederverwendbarkeit und Erweiterbarkeit erreicht.

### 1.3 Game-Engine-Architektur

Die Architektur einer Game-Engine beschreibt das Zusammenspiel der einzelnen Subsysteme (z. B. Physik und Grafik) und deren Kommunikation. Dabei sind vor allem wiederverwendbare und erweiterbare Architekturen sehr wichtig, damit sich die immer komplexeren Computerspielprojekte noch überblicken lassen. *Anderson et al.* kritisieren dabei vor allem in der Literatur über Computerspielentwicklung [AECM08]:

Given this, there exists a surprisingly small body of literature on game engine design. The available research has mainly focussed on game engine subsystem, such as rendering, AI (artificial intelligence) or networking. However, issues regarding the overall architecture of engines, which connects these subsystems, have merely been brushed over.

An der FH Hagenberg wird im Rahmen des Projekts *Cogaen*<sup>6</sup> mit sogenannten komponentenbasierten Game-Engine-Architekturen experimentiert. Ziel ist es einerseits, tiefe und komplexe Klassenhierarchien durch unabhängige und wiederverwendbare Komponenten für Game-Objekte bzw. Subsysteme abzulösen, andererseits die Funktionalität durch automatisch kommunizierende Komponenten dynamisch zur Laufzeit ändern zu können. Dadurch soll letztendlich eine Prototyping-Software zur Unterstützung von Rapid Application Development ermöglicht werden. Die Idee, komponentenbasierte Architekturen für Game-Engines zu verwenden, stammte dabei ursprünglich aus folgender Literatur: [Gol04], [Har04], [Ren05], [Buc05] und [Sto06].

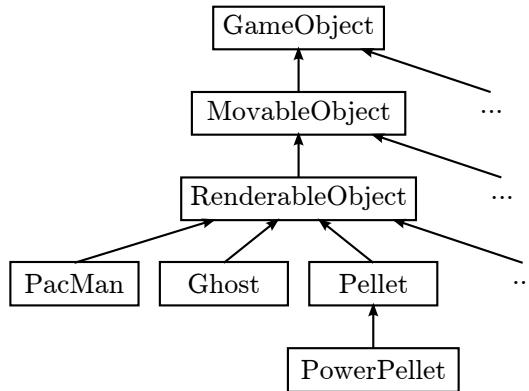
#### 1.3.1 Klassische objektorientierte Architektur

In klassischen, objektorientierten Game-Engine-Architekturen leiten sich Game-Objekte üblicherweise von einer Superklasse `GameObject` ab und jede abgeleitete Klasse implementiert je nach Bedarf mehr Funktionalität. Ein Beispiel ist in Abbildung 1.4 illustriert.

Als Problem dieser Architektur nennt *Gregory* dabei [Gre09, Abs. 14.2]:

---

<sup>6</sup><http://www.cogaen.com>



**Abbildung 1.4:** Eine mögliche Klassenhierarchie mit monolithischer Struktur für das Computerspiel *Pac-Man* (entnommen und angepasst aus [Gre09]).

A game object class hierarchy usually begins small and simple, and in that form, it can be a powerful and intuitive way to describe a collection of game object types. However, as class hierarchies grow, they have a tendency to deepen and widen simultaneously, leading to what I call a *monolithic class hierarchy*.

Als mögliche Lösung dafür wird oft eine komponentenbasierte Architektur vorgeschlagen, wobei die verschiedenen Funktionalitäten aus den Klassen getrennt werden und in unabhängige Komponenten mit wohldefinierten Schnittstellen gekapselt werden [Gre09, Abs. 14.2]. Dadurch kann ein Game-Objekt aus der jeweils benötigte Funktionalität zusammengesetzt werden.

### 1.3.2 Komponentenbasierte Architektur

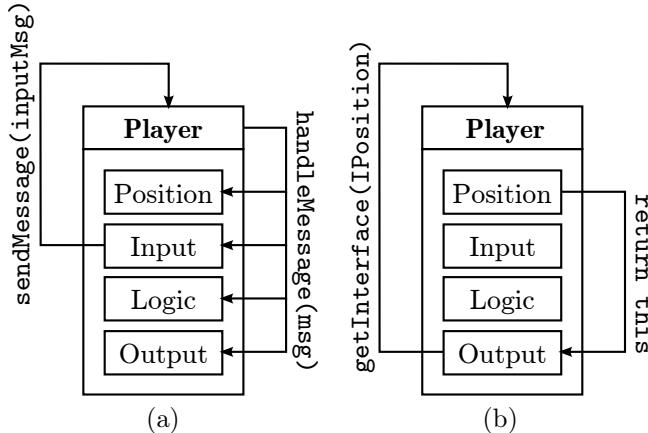
Softwarekomponenten sind folgendermaßen definiert [HC01]<sup>7</sup>:

A *software component* is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

A *component model* defines specific interaction and composition standards. A *component model implementation* is the dedicated set of executable software elements required to support the execution of components that conform to the model.

A *software component infrastructure* is a set of interacting software components designed to ensure that a software system or subsystem constructed using those components and interfaces will satisfy clearly defined performance specifications.

<sup>7</sup>Die Hervorhebungen stammen aus dem Originaltext.



**Abbildung 1.5:** Verschiedene Kommunikationstechniken für Game-Objekt-Komponenten. Explizit mittels *Messages* (a), Implizit und automatisch mittels *Klasseninterfaces* (b).

Bei Komponenten für Game-Objekte in *Cogaen* wurde vor allem die Idee verfolgt, dass ein Game-Objekt ein identifizierbarer Container mit voneinander unabhängigen Komponenten ist, welche jeweils nur für eine bestimmte Funktionalität zuständig sind. Die Kommunikation und Verknüpfung mit anderen Komponenten (*component model*) erfolgt dabei grundsätzlich über zwei verschiedene Arten: Entweder explizit über *Messages* oder *Events*, oder implizit mittels Referenzen, indem die Komponenten in einem Game-Objekt nach einem bestimmten *Klasseninterface* abgesucht werden. Diese Kommunikationsmechanismen sind in Abbildung 1.5 illustriert.

### 1.3.3 Anwendungsfall: Revive

*Revive* ist ein Computerspielprojekt welches über zwei Semester entwickelt wurde und als Basis die komponentenbasierte Game-Engine *ACES* verwendet, eine Weiterentwicklung des Vorgängerprojekts *Cogaen C#* und Übersetzung von *Cogaen C++*. Am Beispiel von *Revive* werden im nächsten Abschnitt einige Schwierigkeiten mit komponentenbasierten Game-Engine-Architekturen dargelegt. Eine Übersicht über die verwendete Technik und die Spielinhalte sind Tabelle 1.2 zu entnehmen. In *Revive* können mehrere Spieler in einer Arena auf Plattformen herumspringen und mittels Tastenkombos verschiedene Kampf- und Spezialangriffe ausführen, um damit die Gegenspieler zu besiegen. Ein Screenshot ist in Abbildung 1.6 dargestellt<sup>8</sup>.

<sup>8</sup>Ein Video dazu findet sich auf: <http://blog.scrambled-egg.net/revive>.

<i>Plattformen</i>	PC, XBox 360
<i>Programmiersprache</i>	C# 3.0
<i>Game-Engine</i>	ACES (komponentenbasiert)
<i>Grafik</i>	3D, mittels XNA 3
<i>Physik</i>	2D rigid body dynamics, mittels Farseer 2
<i>Genre</i>	Jump'n'Run, Beat'em'Up
<i>Spieleranzahl</i>	2-4

**Tabelle 1.2:** Technische Daten von *Revive*.**Abbildung 1.6:** Screenshot von Revive, einem Computerspiel mit komponentenbasierter Architektur.

Die gesamte Funktionalität eines Spielercharakters wurde mittels folgender Komponenten<sup>9</sup> abgebildet:

**GamepadInput:** Das Game-Objekt reagiert auf Eingaben des Gamepads.

**PhysicsBody:** Das Game-Objekt kann mittels physikalischen Kräften bewegt werden und besitzt eine Position im physikalischen Raum.

**Collision:** Das Game-Objekt reagiert auf physikalische Kollisionen mit anderen Game-Objekten.

**HealthPoints:** Das Game-Objekt verfügt über Lebensenergie und wird automatisch zerstört, wenn diese 0 erreicht.

**HitZones:** Das Game-Objekt kann an definierten geometrischen Zonen anderen Game-Objekte Schaden zufügen bzw. selbst getroffen werden.

**Model:** Das Game-Objekt wird durch ein 3D Modell an der physikalischen Position dargestellt.

---

<sup>9</sup> *Revive* verwendet noch mehr Komponenten, es sind aber nur die für diese Arbeit relevanten Komponenten aufgelistet.

**Logic:** Diese Komponente übernimmt die sonstige Logik des Game-Objekte und verwaltet zusätzliche Kommunikation zwischen den Komponenten. Sie nimmt eine Sonderstellung ein, da sie für jedes konkrete Computerspiel neu geschrieben werden muss. Letztendlich muss für ein Computerspiel immer eine spezifische Logik implementiert werden, beispielsweise um die Aktionen bei Kollisionen mit anderen Objekten zu definieren. Die Logic-Komponente wurde jedoch auch für minimale Funktionalität und Kommunikation missbraucht, welche ansonsten keinen Platz fand und ist ein gutes Anzeichen für Probleme in der Architektur.

#### 1.3.4 Mängel an der Architektur

Bei der konkreten Anwendung des komponentenbasierten Ansatzes in *Revive* stellten sich jedoch einige Probleme heraus, welche in folgender Liste aufgeführt sind:

- Die meisten Komponenten sind nur für sehr geringe Funktionalität zuständig, um z. B. die Kommunikation zwischen anderen Komponenten zu verwalten. In *Revive* besitzt der Lebensbalken als Vorgabe der GUI einen Wertebereich von 0.0 bis 1.0, die Lebensenergie des Spielercharakters hingegen einen Wertebereich von 0 bis 100, wodurch die Komponenten vorerst inkompatibel sind. Weitere typische Beispiele sind Orientierungsinformationen, welche sowohl als Vektoren, Matrizen oder auch Quaternionen dargestellt werden können. Dadurch ist eine zusätzliche Komponente für die Konvertierung nötig, wenn die ursprünglichen Komponenten gleich bleiben sollen, um das eigentliche Ziel von Wiederverwendbarkeit zu erreichen. Zumeist wird die Konvertierung daher kurzerhand in der Logik durchgeführt, um die Erstellung einer neuen Komponente und den damit verbundenen Aufwand zu umgehen.
- Das Ziel von komponentenbasierten Game-Engine-Architekturen ist möglichst unabhängige und wiederverwendbare Komponenten zu ermöglichen und zusätzlich zur Laufzeit miteinander kombinieren zu können. Dabei werden jedoch gegensätzliche Aspekte vermischt. Einerseits müssen Komponenten einem Protokoll folgen, um miteinander kommunizieren zu können, welches aber bis auf wenige mathematische Konzepte, für jedes konkrete Computerspiel spezifisch implementiert werden muss. Andererseits sollen Komponenten möglichst unabhängige Funktionalität implementieren, wodurch diese immer mehr auf ein einzelne Funktion reduziert werden, welche extra in eine Komponente gekapselt werden.
- Ein Game-Objekt kann grundsätzlich mehrere Komponenten mit dem selben Klasseninterface verwenden, dadurch entstehen aber Mehrdeutigkeiten bei der impliziten Suche nach einem bestimmten Klassen-

interface, wodurch die konkret verwendete Komponente undefiniert ist. In *Revive* verfügt z. B. ein Spielercharakter über mehrere *HitZone* Komponenten, um an unterschiedlichen Bereichen Schaden zuzufügen zu können. Die Mehrdeutigkeiten konnten jedoch durch eine hierarchische Verwaltung für Komponenten gelöst werden, wodurch sich die gewünschte Komponente automatisch aus der Position in der Hierarchie ergibt. Dabei wurden aber nur die ursprünglichen monolithischen Klassenhierarchien durch eine dynamische Variante ersetzt.

Diese Mängel lassen sich zwar leicht umgehen und sind für kleine Projekte wie *Revive* vernachlässigbar, bei großen Projekten würden diese jedoch zu grundlegenden Problemen führen.

### 1.3.5 Lösungsansätze

Nach einer Analyse dieser Mängel entstand zuerst die Überlegung, die Kommunikation der Komponenten über einen „kanalbasierten“ Ansatz zu lösen, ähnlich zu *Signals and Slots* im *Qt* Framework. Dieses Konzept wird *Signalthprogrammierung (signal programming)* genannt, wobei sich in *Qt* damit GUI-Elemente mit Aktionen verknüpfen lassen. Zum Beispiel wird das *Signal OnClick* eines *Buttons* mit einem entsprechenden *Slot* zum Beenden eines *Window* verknüpft. In der Game-Engine sollte sich dadurch, bei der Erzeugung eines konkreten Game-Objekts zur Laufzeit, die Position der Grafik direkt mit der physikalischen Position verknüpfen lassen. Zusätzlich sollten aber noch kleine Funktionen dazwischen geschaltet werden können, um z. B. die Position der Grafik etwas von der physikalischen Position zu versetzen, ohne dafür eine eigene Komponente nötig zu machen. Aus diesem Grund wurde die funktionale Programmierung in Betracht gezogen und zuerst erfolgreich in der Programmiersprache *Common Lisp* mit der Programmzbibliothek *Cells* implementiert. Dieses Konzept wird *Datenflußprogrammierung (dataflow programming)* genannt, wobei sich in *Cells* damit die einzelnen Variablen über Funktionen verknüpfen lassen und sich umgehend aktualisieren, wenn sich eine abhängige Variable ändert. Nach einiger Recherche stellte sich letztendlich die *funktional-reaktive Programmierung* als sehr fortgeschritten Variante davon heraus und schien vor allem sehr gut für die Problemdomäne der Computerspiele geeignet zu sein. Es existieren vor allem für die Programmiersprache *Haskell* einige Forschungsarbeiten über dieses Thema, darunter die Programmzbibliothek *Yampa*, weshalb diese für die vorliegende Arbeit herangezogen wurde und in Abschnitt 3.5 und Kapitel 4 und 5 behandelt wird.

## Kapitel 2

# Basisfunktionalitäten eines Computerspiels

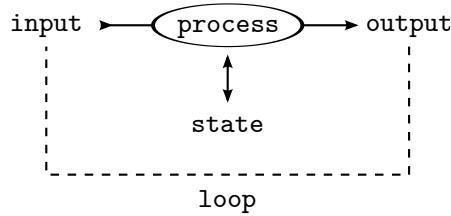
### 2.1 Einleitung

In diesem Kapitel wird eine Übersicht über den grundlegenden technischen Aufbau eines Computerspiels beschrieben. Eine Game-Engine stellt dabei die Basis jedes Computerspiels dar und es werden verschiedene *Subsysteme* verwendet, welche jeweils bestimmte Funktionalitäten bereitstellen (z. B. Benutzereingaben, Physik-Berechnung oder Grafikausgabe) und vom Anwendungsprogrammierer zu einem Computerspiel verknüpft werden. In dieser Arbeit werden die Subsysteme nach dem *Eingabe-Verarbeitung-Ausgabe Modell* (EVA) gegliedert, um damit später das Verständnis von funktional-reaktiver Programmierung zu erleichtern. Die Funktionalität wird nur auf Ebene des Anwendungsprogrammierers beschrieben, nicht deren technische Implementierung, da dieser grobe Überblick ausreichend sein sollte, um daraus die Basisfunktionalität und Komponenten für ein allgemeines Computerspiel zu definieren. Auf die Implementierung einiger dieser Komponenten wird später in Kapitel 5 näher eingegangen.

#### 2.1.1 Definition: Computerspiel

Gregory definiert ein Computerspiel folgendermaßen [Gre09, Abs. 1.2]:

Most two- and three-dimensional video games are examples of what computer scientists would call *soft real-time interactive agent-based computer simulations*. [...] In most video games, some subset of the real world – or imaginary world – is *modeled* mathematically so that it can be manipulated by a computer. [...] An *agent-based* simulation is one in which a number of distinct entities known as "agents" interact. [...] All interactive video games are *temporal simulations*, meaning that the virtual



**Abbildung 2.1:** Einfaches Schema einer *Game-Loop*. Durch die Wiederholung der Ausführung, kann ein interner Zustand aufgebaut werden.

game world model is *dynamic* – the state of the game world changes over time as the game's events and story unfold. [...] Finally, most video games present their stories and respond to player input in real-time, making them *interactive real-time simulations*.

Unter Betrachtung des EVA-Modells werden dabei nicht-deterministische Eingabedaten erzeugt (Zeit, Benutzereingaben und externe Daten) und in einem deterministischen Prozess verarbeitet, woraus nach jedem Verarbeitungsschritt ein neuer Ausgabezustand (*Game-State*) berechnet und letztendlich ausgegeben wird. Durch diesen Prozess kann dabei ein interner Zustand aufgebaut werden, welcher sich jedoch aus den bisherigen Eingaben völlig deterministisch herleiten lässt und dadurch von externen Zuständen unterscheidet. Ein Computerspiel wird nach jedem Schritt in einer *Game-Loop* genannten Schleife wiederholt, wobei aber nur die Ausführung wiederholt wird und nicht ein deterministischer Datenfluss von Ausgabe zur Eingabe stattfindet. Dieser Vorgang ist Abbildung 2.1 illustriert.

### 2.1.2 Definition: Game-Engine

Gold definiert eine *Game-Engine* wie folgt [Gol04]:

A series of modules and interfaces that allows a development team to focus on product game-play content rather than technical content.

Gregory wiederum beschreibt Game-Engines in seinem Buch *Game Engine Architecture* in seinen unterschiedlichen Formen und Ausprägungen. Er weist vor allem auf die unklare Grenze zwischen Computerspiele und Game-Engine hin und geht auf die Unterschiede in den verschiedenen Genres ein. Als Basis nennt er folgende häufig verwendete Subsysteme [Gre09, Kap. 14]:

- Game-Logic
- Levelverwaltung und -streaming

- Verwaltung der Game-Objekte
- Ereignis- und Nachrichtenbehandlung
- Scripting
- Spielregeln und Gameplay

Letztendlich fasst er jedoch wie folgt zusammen:

We should probably reserve the term "game engine" for software, that is extensible and can be used as the foundation for many different games without major modification.

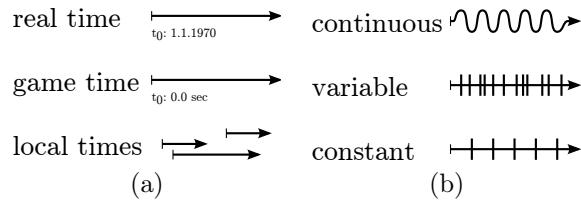
Besonders der Mangel an klaren Definitionen und Richtlinien zum Design von Game-Engines sind bei der Recherche zu dieser Arbeit aufgefallen. Zudem scheinen einige Bücher Game-Engines mit Grafik-Engines gleichzusetzen oder Einführungen in Programmiersprachen unter dem Titel „Computerspielentwicklung“ zu verkaufen. Diesen Zustand kritisieren *Anderson et al.* ebenfalls wie folgt [AECM08]:

This lack of literature and research regarding game engine architectures is perplexing. Books on the subject [...] tend to only briefly describe the high-level architecture before plunging straight down to the lowest level and describing how the individual components of the engine are implemented. Often authors present their own architectures as a de facto solution to their specific problem set, without necessarily justifying the decision-making process that led to their designs, and merely indicating the authors' personal preferences.

In den folgenden Abschnitten wird daher auf häufig benötigte Funktionalität in Computerspielen eingegangen und welche Aspekte in der Implementierung dabei besonders einen Einfluss auf die Game-Engine-Architektur haben und entsprechend berücksichtigt werden müssen.

## 2.2 Eingabe

Als Eingabe sind alle **für den Computer nicht-deterministischen Eingabedaten** zu verstehen. Ein Beispiel für Nicht-Determinismus ist die aktuelle Zeit, welche bei einer Abfrage mittels `getTime()` jedes Mal ein anderes Ergebnis liefert, obwohl die Prozedur immer mit dem selben (leeren) Parameter aufgerufen wird. Diese Feststellung und Abtrennung von den anderen Schritten (Verarbeitung und Ausgabe) sind sehr wichtig, da damit die Datenströme und die übergeordnete Architektur besser nachvollziehbar sind. Sobald alle Eingabedaten abgefragt wurden, werden diese an die Verarbeitung weitergereicht und können dort anschließend völlig deterministisch und **frei von Seiteneffekten**, also *rein funktional*, berechnet werden.



**Abbildung 2.2:** Unterscheidung der Zeit nach: Bezug (a), Auflösung (b).

### 2.2.1 Zeit

Die Natur der Zeit ein sehr komplexes Thema und Mittelpunkt vieler interessanter philosophischer und naturwissenschaftlicher Betrachtungen, auch in Computersimulationen (siehe [Blo09, Kap. 3] und [All91]). Da in Computerspielen die Zeit ein **ständiger und allgegenwärtiger Eingabeparameter** ist, von dem beinahe jede Funktionalität abhängig ist, sind Überlegungen über die Abbildung und Abstraktion der Zeit von grundlegender Bedeutung.

Es werden folgende grundlegende Begriffe im Zusammenhang mit Zeit unterschieden:

**Real time:** Die reale, physikalische Zeit des Spielers.

**Simulation time:** Die Zeit im Computerspiel selbst.

**Local time:** Die interne Zeit eines Objekts (z. B. einer Animation).

Bei der Implementierung der Zeit ist vor allem die Unterscheidung zwischen **diskreter bzw. kontinuierlicher Zeit** wichtig. Die aktuelle Zeit wird üblicherweise nur einmal pro Simulationsschritt abgefragt und es vergeht weiter Zeit, während der Computer Berechnungen durchführt. Dadurch tritt aber Zeit in der Simulation nur an vereinzelten (diskreten) Zeitpunkten auf und es kann keine genaue Aussage über die Zeitabschnitte „dazwischen“ (das Kontinuum) gemacht werden. Viele physikalische Formeln basieren jedoch auf kontinuierlicher Zeit, welche aber in einer Simulation nur angenähert werden kann und damit zwangsläufig ungenau ist. Aufgabe einer guten Implementierung ist es daher, diese Ungenauigkeit möglichst gering zu halten. Als Abhilfe werden einerseits basierend auf den bisherigen Verlauf, Abschätzungen über den weiteren Verlauf getroffen, andererseits nur Zeitabschnitte mit konstanten statt variablen Abständen verwendet, damit die Simulation und ihre Ungenauigkeiten zumindest vorhersagbar bleiben, auch wenn die Auflösung dadurch verringert wird. Die bisher genannten Punkte sind in Abbildung 2.2 illustriert.

Der eigentliche Zeitgeber in einem Computersystem ist normalerweise die System-Uhr. Diese zählt die Prozessortakte (auch *ticks* oder *cycles* genannt) seit einem bestimmten Startdatum, wobei die Takte zugleich das kleinste Zeitintervall darstellen. Das Betriebssystem stellt wiederum unterschiedlich-

che Prozeduren zum Auslesen der aktuellen Zeit zur Verfügung, wobei je nach Implementierung verschiedene Zeitaufösungen und Startdaten verwendet werden.

Der einfachste Ansatz in Computerspielen den Zeitverlauf auszudrücken ist mittels einer `update (elapsedTime)` Funktion, wobei üblicherweise diskrete Zeitverläufe verwendet werden. Der Zeitverlauf wird an alle Game-Objekte kommuniziert, welche wiederum entsprechend darauf reagieren. Einige Computerspiele erlauben es zusätzlich, den Verlauf der Zeit an sich zu beeinflussen (z. B. *Braid* [Blo07]), wobei ein typisches Beispiel die Pausierung der Simulation ist. *Blom* beschreibt verschiedene Manipulationstechniken, z. B. um Zeit zu beschleunigen, verlangsamen, anhalten oder sogar zurückzudrehen [Blo09, Kap. 4].

Die verschiedenen Subsysteme sollen wiederum mit unterschiedlichen Frequenzen betrieben werden können. Zum Beispiel sind für visuelle Darstellungen 60Hz ausreichend, während physikalische Berechnungen von höheren Frequenzen profitieren. In der Implementierung stellt sich zudem die grundlegende Frage, welches Subsystem für die Kommunikation mit der System-Uhr und Taktung der gesamten Simulation verantwortlich ist. Zusätzlich muss der Zeitverlauf auch an die anderen Subsysteme propagiert werden, wobei sich wiederum die erwartete Einheit (z. B. Sekunden, Mikrosekunden oder Ticks) unterscheiden kann.

In Netzwerken wird die Simulation der Zeit noch weiters dadurch erschwert, dass sämtliche Teilnehmer jeweils in eigenen kleinen Zeituniversen ablaufen und synchronisiert werden müssen.

### 2.2.2 Human Interface Devices

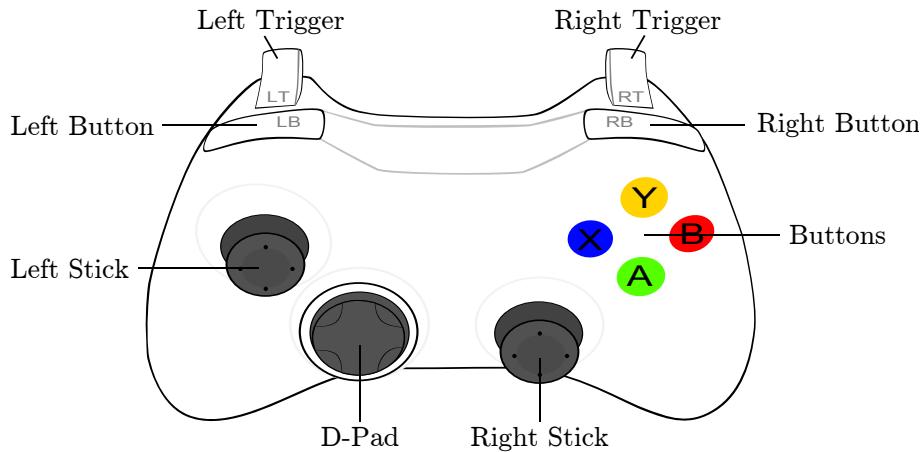
Als *Human Interface Devices* werden die gewohnten Eingabegeräte für den Benutzer bezeichnet (z. B. Tastatur, Maus und Joystick)<sup>1</sup>. Mit Benutzeraktionen können verschiedene Ereignisse ausgelöst werden, um die laufende Simulation zu steuern (z. B. Bewegung des Spielercharakters).

In Computerspielen ist dabei insbesondere die Zuordnung von Eingabe und Spieler zu beachten. Derselbe Spieler soll eine bestimmte Aktion (z. B. springen) auf verschiedenen Eingabegeräten ausführen können (z. B. Tastatur: [Ctrl], Maus: [LMB] oder Gamepad: [ButtonA]). In Mehrspieler-Modi wiederum sollen verschiedene Spieler mit demselben Eingabegerät verschiedene Spielercharaktere steuern können (z. B. Spieler 1 mit dem linken Bereich der Tastatur, Spieler 2 mit dem rechten Bereich).

Die einzelnen Eingabeelemente der Geräte können dabei verschiedene Formen annehmen, wie in Abbildung 2.3 illustriert. Letztendlich unterscheiden sich die Eingabeelemente aber durch folgende zwei Eigenschaften: **diskrete oder kontinuierliche Eingabe** (z. B. Button bzw. Stick) und **rela-**

---

<sup>1</sup>Die folgenden Überlegungen sollten auch auf komplexere Eingabegeräte anwendbar sein, wie Kameras, Motion-Controller, 3D Joysticks usw.



**Abbildung 2.3:** Illustration eines typischen Gamepads mit *Buttons* und *Triggers* bzw. *Sticks* als Beispiele für diskrete bzw. analoge Eingabeelemente mit absoluten Werten (angepasst und entnommen aus *Wikimedia Commons*).

**tive oder absolute Werte** (z. B. Mausrad bzw. Trigger). Diese Unterscheidung ist hilfreich, um beispielsweise höhere Elemente durch niedere Elemente ausdrücken zu können (sogenanntes *Input Re-Mapping*, siehe [Gre09, Abs. 8.5.7]). Zum Beispiel kann die Bewegung des Spielercharakters durch analoge, zweidimensionale Eingabedaten beschrieben werden, die wiederum entweder direkt von einem analogen *Stick* oder indirekt von vier digitalen *Buttons* kommen können, welche als Stick interpretiert werden. Ähnlich wie die Zeiteingabe liefern die kontinuierlichen Eingabeelemente ständig Daten, werden jedoch im Unterschied zur Zeit nur für wenige Game-Objekte verwendet, meist nur für die Spielercharaktere.

Die Benutzereingaben können anschließend entsprechend weiterverarbeitet werden, um daraus spezielle Muster zu erkennen oder Signalverarbeitungen durchzuführen [Gre09, Abs. 8.5.4]:

**Chords:** Mehrere Buttons wurden „gleichzeitig“ gedrückt.

**Multiple Button Sequence:** Mehrere Buttons wurden in einer bestimmten Kombination gedrückt.

**Thumb Stick Rotation:** Ein Stick wurde im Kreis bewegt.

**Dead Zone:** Ein Stick soll erst ab einem gewissen Schwellwert reagieren.

**Gesture Detection:** Ein bestimmtes Symbol wurde „nachgezeichnet“ (z. B. mit der Maus).

Viele Eingabegeräte werden heutzutage auch als zusätzliche Ausgabegeräte verwendet, z. B. Gamepads mit Force-Feedback oder LED-Anzeigen. Diese Funktionalität ist jedoch dem Ausgabebereich zuzuordnen.

### 2.2.3 Externe Daten

Bei externen Daten handelt es sich um Anhäufungen von Informationen, welche aus einem externen Speicher kommen (z. B. Bilder, Levedateien und Skripte) und werden in der Computerspielentwicklung üblicherweise als *assets*, *media* oder *ressources* bezeichnet. Die Daten selbst werden normalerweise mit spezieller Design-Software erstellt, da sie sich nur sehr schwierig programmietechnisch ausdrücken lassen. Ein Gegenbeispiel dazu wäre *Textursynthese*.

Die Daten sollten selbst dann möglichst nur bei Bedarf geladen werden, da sie im Vergleich zu reinem Programmcode sehr groß sind und dadurch nur langsam aus dem Festplattenspeicher geladen werden können. Daher werden oft sogenannte *Ressource-Manager* eingesetzt, welche die Verwaltung im Arbeitsspeicher übernehmen. Die geladenen Daten können anschließend sowohl in der Verarbeitung (z. B. einer Levedatei zufolge neue Objekte erzeugen) als auch in der Ausgabe (z. B. eine Bilddatei anzeigen) verwendet werden.

### 2.2.4 Sonstige

Eine weitere nicht-deterministische Eingabequelle stellt der Zufallsgenerator dar. Da Zufallsgeneratoren normalerweise aber nur spezielle Algorithmen sind und damit eigentlich immer deterministisch arbeiten müssen, wird als Startwert oft die aktuelle Zeit herangezogen, woraus anschließend die Zufallsreihen produziert werden können. Es gibt zwar Geräte, welche echten quantenphysikalischen Zufall erzeugen, diese sind aber im Heimbereich unüblich.

Eine weitere nicht-deterministische Eingabe (und Ausgabe) würde das Netzwerk darstellen, welches aber in dieser Arbeit nicht behandelt wird.

## 2.3 Verarbeitung

Sobald im Eingabeschritt sämtliche nicht-deterministischen Eingabedaten gesammelt wurden, werden diese an den Verarbeitungsschritt weitergereicht, woraus letztendlich der gesamte Ausgabezustand (*Game-State*) berechnet und angezeigt wird. Die eigentliche Funktionalität eines Computerspiels bezeichnet *Gregory* als *runtime object model*, welches folgende typische Anforderungen umfasst [Gre09, Abs. 14.2]:

- Game-Objekte dynamisch erzeugen oder entfernen
- Verbindung zu darunterliegenden Subsystemen
- Simulation der Funktionalität in Echtzeit
- Erzeugung neuer Typen von Game-Objekten
- Eindeutige Identitäten für Game-Objekte
- Suchanfragen von Game-Objekten

- Verweise auf andere Game-Objekte
- Unterstützung von Zustandsautomaten
- Replikation im Netzwerk
- Speichern und Laden von Spielzuständen

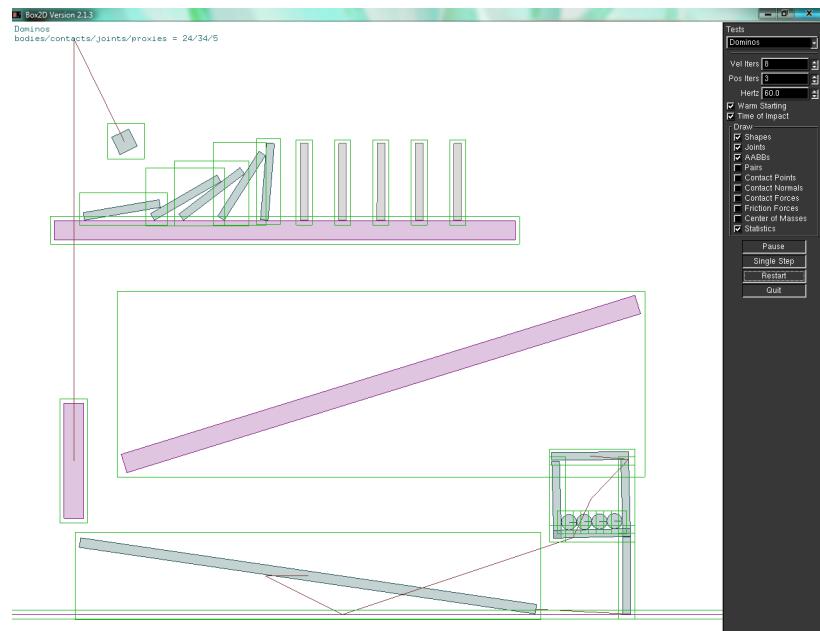
Da es sehr intuitiv ist, Begriffe aus der reale Welt in Computersimulationen zu übernehmen, vor allem um logisch zusammengehörende Funktionalität zu gruppieren, wird die Funktionalität eines Computerspiels oft in Objekten abgebildet, meist als *Game-Objekt* bezeichnet, gelegentlich aber auch *Actor*, *Agent* oder *Entity* genannt. Zum Beispiel entspricht der Spielercharakter genau einem Spieler-Objekt oder ein Gegner genau einem Gegner-Objekt, wobei dies auch für rein logische Objekte ohne räumliche Darstellung gilt, wie z. B. einer Ziellinie.

Unter Betrachtung des EVA-Modells müssen aber zuerst die zentral gesammelten Eingabedaten an die einzelnen Game-Objekte verteilt werden, damit diese auch entsprechend darauf reagieren können. Zusätzlich muss es jedoch auch die Möglichkeiten geben, aus einer globalen Sicht auf den Game-State Aussagen treffen zu können (z. B. Kollisionen zwischen Game-Objekten) und diese entsprechend darüber zu informieren, weshalb die nicht-deterministischen Eingabedaten um deterministische Ereignisse erweitert und erst anschließend entsprechend verteilt werden. Die Game-Objekte können dadurch **völlig unabhängig voneinander** deren Funktionalität ausführen und produzieren einen neuen Game-State (z. B. wird durch die Zeiteingabe eine Animation weitergespielt und ein Spielercharakter durch die Benutzereingabe an eine andere Position bewegt).

Der aktuelle Game-State ist dabei immer nur von allen bisherigen Eingabedaten abhängig (und der Funktionalität im Verarbeitungsschritt) und lässt sich daraus vollständig berechnen. Damit Game-Objekte hinzugefügt, entfernt und ein Ausgabezustand im nächsten Verarbeitungsschritt wieder dem entsprechenden Game-Objekt zugeordnet werden kann, müssen diese identifizierbar sein, weshalb die Game-Objekte einem zusätzlich einem Verwaltungssystem unterliegen müssen. Dabei ist darauf zu achten, dass Game-Objekte nicht während des Verarbeitungsschrittes entfernt werden dürfen, da ansonsten ungültige Verweise entstehen würden.

### 2.3.1 Physik

Damit sich Game-Objekte durch den Raum bewegen können ist meist eine direkte Veränderung der Position und einfache Kollisionserkennung ausreichend. Für komplexere Computer-Spiele werden Physik-Engines eingesetzt und sind zumeist durch die Realität inspiriert, arbeiten aber aus Gameplay-Gründen weniger strikt als die reale Physik (z. B. Bewegungsrichtung in der Luft umkehren) oder stellen aufgrund der Performance nur Annäherungen an die reale Physik dar (z. B. keine Kollision für Effekte).



**Abbildung 2.4:** Box2D Testbed - Domino demonstriert typische Funktionalität einer *rigid body dynamics* Physik-Engine wie Kollisionsgeometrie, Seile, Hebel usw.

Die gebräuchlichste Technik in Computerspielen ist *rigid body dynamics* und ermöglicht Kräfte auf Objekte mit fester Masse einwirken zu lassen und dadurch zu bewegen. Ein typisches Beispiel ist in Abbildung 2.4 dargestellt. Oft wird noch zusätzliche Funktionalität angeboten, wie z. B. Reibung, Kollisionserkennung oder Feder-Masse-Systeme [Gre09, Kap. 12]. Bei deren Implementierung muss vor allem unterschieden werden, ob die Funktionalität von einzelnen Game-Objekten berechnet werden kann (z. B. Kräfte) oder sich auf einen globalen Game-State bezieht und die Game-Objekte entsprechend benachrichtigt werden müssen (z. B. Kollisionen). Andere Techniken zur Physiksimulation werden meist nur als Effekt-Physik (z. B. *Ragdolls*) oder für spezielle Anwendungsbereiche (z. B. Flüssigkeiten) eingesetzt.

### 2.3.2 Sonstige

Ein weiterer wichtiger Bereich in Computerspielen ist künstliche Intelligenz. Diese umfasst verschiedene Funktionalität, wie z. B. automatische Wegfindung oder künstliche Computergegner. Wie auch in der Physik-Simulation muss dabei beachtet werden, ob die Funktionalität durch einzelne Game-Objekte und ihren Eingabedaten berechnet werden kann oder Aussagen über einen ganzen Game-State getroffen werden sollen.

Ein weiteres Beispiel für Funktionalität im Verarbeitungsbereich ist Animation. Die gängige Literatur ordnet Animation normalerweise unter Grafik ein, dem EVA-Modell zufolge lässt sich jedoch der aktuelle Animationsframe rein durch die Zeiteingaben vollständig berechnen, wobei die entsprechende Grafik erst im Ausgabeschritt angezeigt wird.

## 2.4 Ausgabe

Die Ausgabe bereitet den Game-State für den Benutzer in Form von Grafik und Ton auf oder speichert diesen auf einen externen Speicher. Programmtechnisch entspricht dieser einer Datensenke, da keine Rückgabedaten erwartet werden.

### 2.4.1 Grafik

Grafik ist vor allem in großen, kommerziellen Titeln besonders wichtig, damit realistische Spielwelten simuliert werden können und dadurch das Computerspiel für den Endkunden ansprechend wirkt. Die Berechnung der Grafik wird *Rendering* genannt und verbraucht normalerweise den größten Anteil der Rechenzeit, weshalb ständig an besserer Technik geforscht wird.

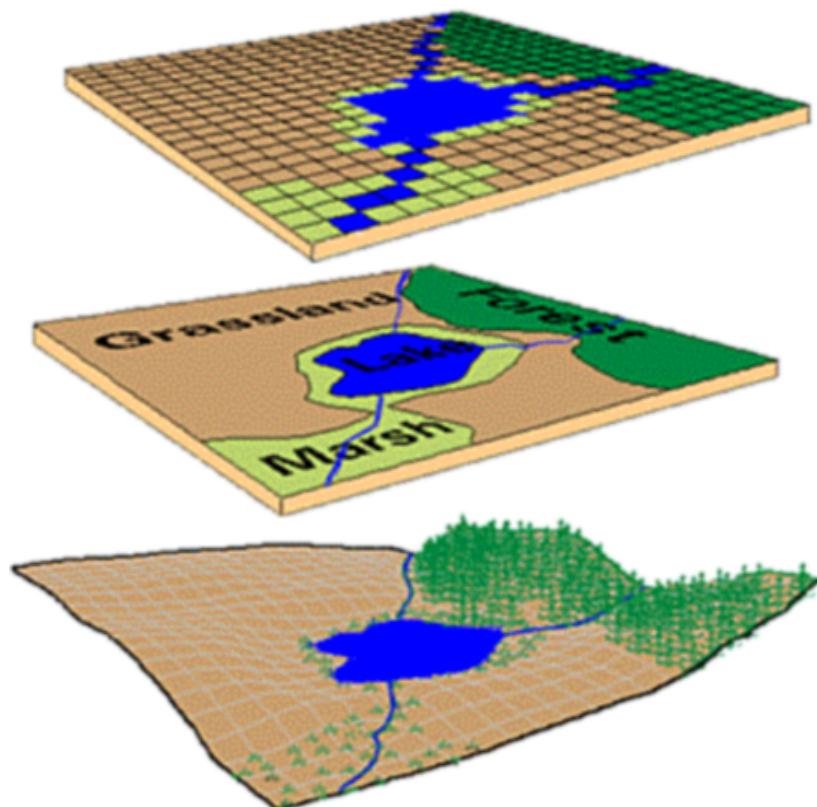
In zweidimensionalen Computerspielen wird hauptsächlich *Rastergrafik* eingesetzt, wobei vorab erstellte Bilddaten angezeigt werden. Zum Teil wird auch *Vektorgrafik*<sup>2</sup> verwendet, wobei anhand mathematischer Definitionen geometrische Formen, unabhängig von externen Bilddaten, gezeichnet werden. In dreidimensionalen Computerspielen wird hauptsächlich *Rasterization* von Polygonen eingesetzt, wobei 3D Modelle mitsamt Textur und Beleuchtung auf die 2D Ebene des Bildschirms projiziert und anschließend angezeigt werden. Die verschiedenen Techniken sind in Abbildung 2.5 dargestellt.

Für das Rendering werden heutzutage hauptsächlich *OpenGL* oder *DirectX* verwendet bzw. darauf aufbauende *Grafik-Engines*, welchen den Anwendungsprogrammierer mit zusätzlicher Funktionalität unterstützen. In Grafik-Engines wird häufig zusätzlich ein *Szenengraph* eingesetzt, welcher die räumliche Position der Objekte und ihre Beziehungen zu anderen Objekten beschreibt. Dieser dient vor allem der logischen Strukturierung der Szene und erlaubt es, weit entfernte Objekte vereinfacht darzustellen bzw. nicht sichtbare Objekte aus dem Rendering völlig auszuschließen, wodurch die Performance gesteigert werden kann.

Es existiert eine Reihe an weiteren Techniken zur Darstellung von Grafik, letztendlich wird jedoch immer der Ausgabezustand angezeigt, wodurch das Rendering kaum einen Einfluss auf die Architektur hat und sich diese Arbeit auf die einfachste Variante von zweidimensionaler Rastergrafik beschränkt.

---

<sup>2</sup>Vektorgrafik ist vor allem in webbasierten Computerspielen mit *Flash* sehr verbreitet.



**Abbildung 2.5:** (von oben nach unten) Rastergrafik, Vektorgrafik, Polygonmodell (angepasst und entnommen aus <http://www.lincoln.ac.nz>)

Weiters werden in Computerspielen auch häufig **grafische Benutzeroberflächen** (Englisch: *Graphical User Interfaces* (GUI)) eingesetzt. Da GUI-Systeme jedoch meist alle EVA-Bereiche übernehmen, sind sie keinem Bereich direkt zuzuordnen. GUIs stellen typische Elemente zur Verfügung, wie z. B. Buttons, Text, Eingabefelder oder Schieberegler. Damit können Menüs oder *Head-Up Displays* erstellt werden, z. B. zur Darstellung der Leistungenergie oder Inventare. Die GUI kann wiederum selbst Teil der Spielsszene sein, etwa zur Bedienung eines Computerterminals im Computerspiel, und umgekehrt, 3D Charaktere und Szenen der Grafik-Engine in der GUI darstellen, etwa zur Darstellung von 3D Modellen im Head-Up Display.

#### 2.4.2 Debugging

Eine wichtige Funktionalität für Entwickler ist die Ausgabe von zusätzlicher *Debug*-Information über interne Zustände, welche jedoch vor dem Endbenutzer verborgen bleiben sollen. Da Computerspiele Echtzeit-Simulationen durchführen und somit der gesamte Game-State mehrmals pro Sekunde be-



**Abbildung 2.6:** *Revive* mit aktivierten Debuggrafiken zeigt die Performance, Kollisionsgeometrien und Normalvektoren der Charaktere.

rechnet wird, sind klassische Konsolenausgaben aufgrund der anfallenden Datenmengen schnell überfüllt. Daher ist es wünschenswert, dass sich interne Information grafisch darstellen lässt, womit sie meist zusätzlich auch einfacher lesbar und intuitiv verständlich ist. Beispiele dafür sind die Anzeige von Kollisionsgeometrie, Bewegungsvektoren, Performancewerte, interne Werte bestimmter Objekte usw. Abbildung 2.6 zeigt die Debuggrafiken wie sie im Computerspiel *Revive* verwendet wurden.

#### 2.4.3 Sonstige

In Computerspielen ist neben der Grafik der Ton eine häufige Ausgabeform. Grundlegende Anwendungsarten sind z. B. ambient Sound, bereichsbegrenzter Sound oder komplexe physikalische Sounds. Ton ist jedoch, genau wie die Grafik, nur von dem vorab berechneten Ausgabezustand abhängig und wird daher in dieser Arbeit nicht weiter behandelt.

Weiters ermöglichen Speicherpunkte dem Spieler den Fortschritt im Spiel festzuhalten und später fortzusetzen. Der Zustand muss jedoch vorher mittels *Serialisierung* in eine lineare Struktur gebracht werden, um diesen in einer Datei speichern zu können. Oftmals werden dabei aufgrund des Speicherplatzes bestimmte Game-Objekte ausgelassen (z. B. Effekte) oder nur diejenigen internen Werte gespeichert, aus denen sich der ursprüngliche Zustand wieder herleiten lässt. Theoretisch würde sich der gesamte Game-State aus den bisherigen Eingabedaten neu berechnen lassen, wobei diese Tatsache meist nur zur Aufzeichnung von Videos in der Simulation (*Replays*) verwendet wird, weil die Wiederherstellungsdauer für einen Speicherpunkt vergleichsweise lang ist.

## Kapitel 3

# Funktional-reaktive Programmierung

### 3.1 Einleitung

In diesem Kapitel wird zuerst das *imperative Programmierparadigma* dem *deklarativen Programmierparadigma* gegenübergestellt, mit ihren jeweils größten Bereichen der *objektorientierten* bzw. *funktionalen Programmierung*. Diese Gegenüberstellung und Erklärung erscheint nötig, da Computerspielentwickler meist nur (imperative) objektorientierte Programmierung kennen und (deklarativer) funktionaler Programmierung oft mit großer Skepsis gegenüberstehen (ein Phänomen welches in [Gra04, Kap. 12] unter *The Blub Paradoxon* beschrieben ist). Anschließend werden die rein funktionale Programmiersprache *Haskell*, die mathematischen Konzepte der *Monaden* und *Arrows* und abschließend die Programmzbibliothek *Yampa* erklärt, welche zum weiteren Verständnis dieser Arbeit nötig sind.

### 3.2 Programmierparadigmen

Ein Computer ist eine Maschine die Berechnungen durchführt, wobei eine Programmiersprache die Anweisungen gibt, wie die Berechnungen durchgeführt werden. Mathematische Grundlage für Computer ist die *Turingmaschine*, welche mit wenigen elementaren Operationen jede berechenbare Funktion abbilden kann. Mittlerweile verwenden moderne Computer jedoch komplexere Maschinensprachen, wobei mit höheren Programmiersprachen dem Entwickler die Programmierung von Computern vereinfacht werden soll. Ein Programmierparadigma ist durch seine Kernkonzepte und -techniken definiert, welche eine Kernsprache bilden, in der alle möglichen Abstraktionen des Paradigmas ausgedrückt werden können. Je nach Problemdomäne sind bestimmte Programmierparadigmen besser geeignet und je nach Implementierung des Paradigmas unterscheidet sich auch der Programmierstil [RH04].

Eine *turing-vollständige* Programmiersprache kann dabei mathematisch beweisbar in jede andere Programmiersprache konvertiert werden. Diese Tatsache ist bei den folgenden Betrachtungen zu berücksichtigen, da sie sich nur auf die **unmittelbare Kernsprache** des zugrundeliegenden Programmierparadigmas beziehen kann. Ein funktionaler Programmierstil ist beispielsweise auch in einer prozeduralen Programmiersprache mit einer Überbeanspruchung von Konstanten und Macros möglich, jedoch dafür ursprünglich nicht vorgesehen und wird die Programmierung eher erschweren als unterstützten. Eine Übersicht aller Programmierparadigmen findet sich in Abbildung 3.1, wobei Überschneidungen in sogenannten *multiparadigmatischen Programmiersprachen* möglich sind.

Eine kurze Recherche zur Popularität von Programmiersprachen am 20. September 2010 ergab die Statistik in Tabelle 3.1. Die imperativen Programmiersprachen *Java*, *C* und *C++* nehmen erwartungsgemäß die ersten Plätze mit hohen zweistelligen Prozentbereichen ein und überwiegen somit vollständig. Die deklarativen Programmiersprachen liegen mit der *Lisp-Sprachfamilie* (z. B. *Common Lisp* oder *Scheme*) als größten Vertreter um einem Prozent und erst weit ab davon liegt *Haskell*. Interessanterweise nimmt jedoch *Haskell* in der Diskussionshäufigkeit den 5. Platz ein.

<i>TIOBE</i> <sup>1</sup>		<i>SourceForge</i> <sup>2</sup>		<i>LangPop Pop.</i> <sup>3</sup>		<i>LangPop Disc.</i> <sup>4</sup>	
1. Java	17.9%	1. C	16.2%	1. Java	676	1. C++	654
2. C	17.2%	2. Java	16.1%	2. C	549	2. C	609
3. C++	9.8%	3. PHP	10.6%	3. C++	509	3. Python	551
:	:	:	:	:		4. Java	479
13. Lisp	1.1%	18. Lisp	0.8%	17. Lisp	27	5. Haskell	376
:	:	:	:	:		:	:
42. Haskell	0.3%	31. Haskell	0.4%	23. Haskell	11	11. Lisp	236
insgesamt 100%		insgesamt 100%		max. 820 Punkte		max. 820 Punkte	

**Tabelle 3.1:** Popularität verschiedener Programmiersprachen.

### 3.2.1 Imperative und deklarative Programmierung

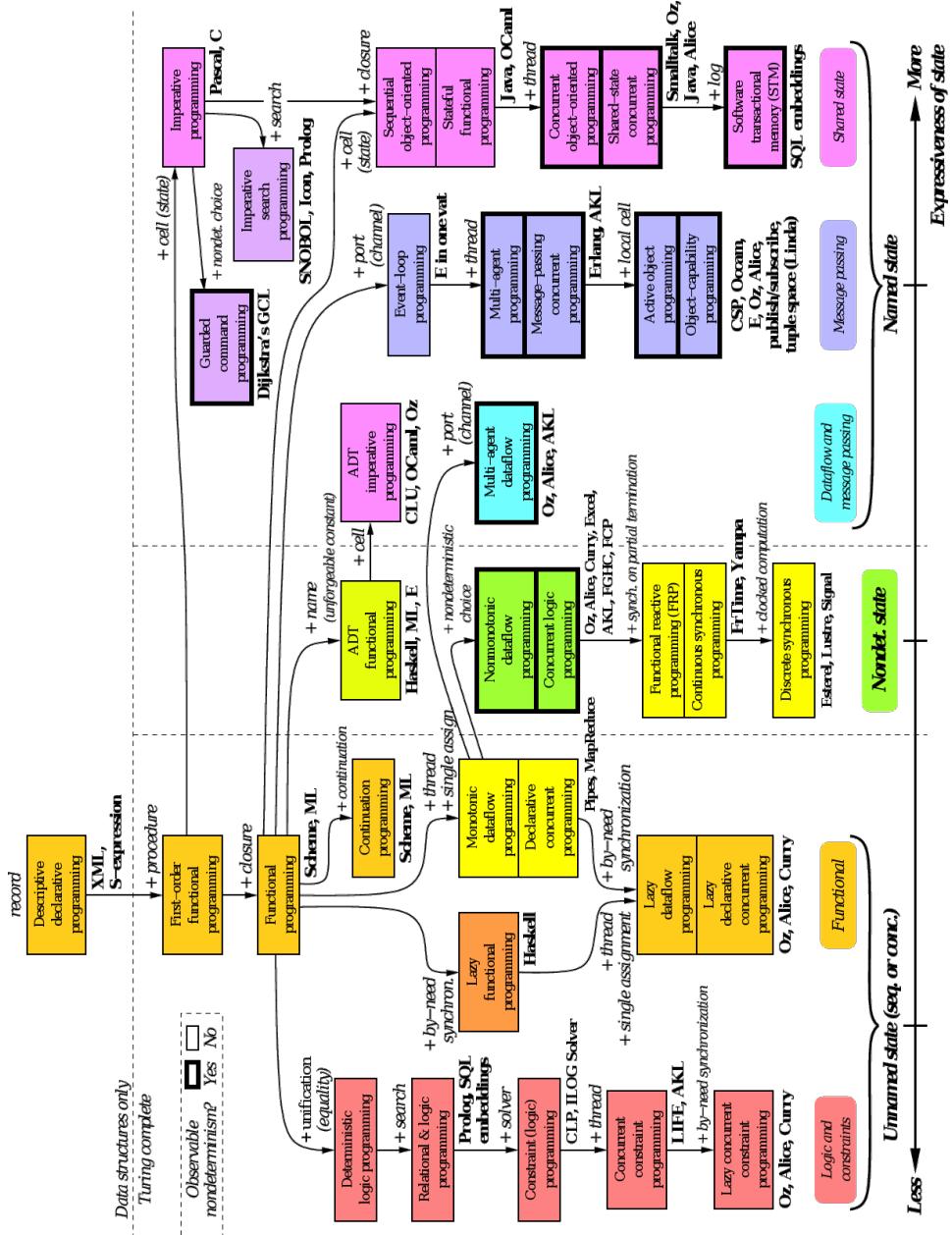
In der imperativen Programmierung besteht ein Programm aus einer Reihe von Anweisungen, die Zustände ändern, wobei dem Computer genau angewiesen wird, **wie** eine Berechnung durchzuführen ist.

<sup>1</sup><http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html>

<sup>2</sup><http://lang-index.sourceforge.net> (Language category: any)

<sup>3</sup><http://langpop.com> (Normalized Comparison)

<sup>4</sup><http://langpop.com> (Normalized Discussion)



**Abbildung 3.1:** Übersicht der Kernsprachen verschiedener Programmierparadigmen (entnommen aus [RH04]).

In der deklarativen Programmierung besteht ein Programm aus mathematischen Beschreibungen, wobei dem Computer angewiesen wird, **was** berechnet werden soll, unabhängig davon wie und in welcher Reihenfolge die Einzelschritte durchgeführt werden.

Die beiden Paradigmen werden folgend am werden in folgenden kurzen Codebeispielen anhand der funktionalen bzw. imperativen Programmierung am Beispiel der Summe von Quadratzahlen gegenübergestellt. Dabei stehen die Konzepte im Vordergrund und nicht die Länge des Programmcodes.

```
1 sum ∘ map (^ 2) $ [1..10]
```

Im funktionalen Programm wird zuerst eine Liste [1..10] mit den Zahlen von 1 bis 10 erzeugt und an eine Funktionskomposition `sum ∘ map (^ 2)` übergeben, welche zuerst eine neue Liste mit Quadratzahlen `map (^ 2)` erzeugt und abschließend die Summe `sum` davon zurückgibt.

```
1 total = 0
2 for i 1..10
3     total += i * i
4 return total
```

Im prozeduralen Programm wird eine Schleife `for i 1..10` von 1 bis 10 durchgelaufen, die aktuelle Laufvariable mittels `i * i` quadriert und einem Zustand `total` außerhalb der Schleife zugewiesen. Abschließend wird der Endzustand explizit mittels `return total` zurückgegeben.

Der Vorteil im funktionalen Beispiel ist, dass kleine unabhängige Programmteile zu komplexeren verknüpft werden können, einerseits da alle Eingabeparameter einer Berechnung direkt übergeben werden müssen und andererseits der Rücksprungpunkt genau definiert ist. Weiters lassen sich kleine Programmteile wesentlich einfacher isoliert testen.

### 3.2.2 Objektorientierte Programmierung

In der objektorientierten Programmierung wird durch die Syntax der Programmiersprache ein Programmierstil unterstützt, bei dem zusammengehörige Variablen und Methoden zu einem Objekt zusammengefasst werden. Methodenaufrufe beziehen sich dabei immer nur auf das zugehörige Objekt, wodurch die konkrete Implementierung nach außen hin gekapselt und versteckt wird. Je nach konkreter Programmiersprache werden weitere Prinzipien und Techniken eingesetzt, wie z. B. Vererbung und Polymorphismus.

Grundsätzlich wird dabei versucht, Objekte aus der realen Welt abzubilden, wodurch die Programmierung sehr intuitiv ist. *Gregory* schreibt im Zusammenhang mit Computerspielentwicklung etwa [Gre09, Abs. 14]:

It's natural to want to classify game object types taxonomically. This tends to lead game programmers toward an object-oriented language that supports inheritance. A class hierarchy is the most intuitive and straightforward way to represent a collection of interrelated game object types. So it is not surprising that the majority of commercial game engines employ a class hierarchy based technique.

### 3.2.3 Funktionale Programmierung

In der funktionalen Programmierung wird durch die Syntax der Programmiersprache ein Programmierstil unterstützt, bei dem Funktionen einen Basisdatentyp (*First-class value*) darstellen und nur, und wirklich nur, von ihren Eingabeparameter abhängen, womit sie völlig frei von Seiteneffekten (*Side effects*) arbeiten. Da Funktionen Basisdatentypen darstellen, sind auch Funktionen höherer Ordnung (*Higher-order functions*) möglich, welche neue Funktionen erzeugen können und/oder Funktionen als Eingabeparameter entgegennehmen, wodurch sogar der weitere Programmverlauf mittels Programmfortführungen (*Continuations*) gesteuert werden kann.

Zugrundeliegendes mathematisches Konzept ist der *Lambda-Kalkül*, welcher beispielsweise zur Herleitung von *Boolescher Algebra*, den *Natürlichen Zahlen* und sogar *Turingmaschinen* verwendet werden kann. Der Lambda-Kalkül kann somit die selben Funktionen wie eine *Turingmaschine* berechnen und besitzt die gleiche Mächtigkeit. Funktionen entsprechen dabei der eigentlichen mathematischen Definition, jedes Element einer Definitionsmenge genau auf ein Element einer Zielmenge abzubilden. Funktionen die keine Eingabeparameter übernehmen (`foo :: () → Int`) oder keine Ergebnisse zurückliefern (`foo :: Int → ()`) sind daher sinnlos, da Ersteres einem Wert gleichgesetzt werden kann bzw. Letzteres einer Datensenke entspricht. Damit zur Benutzerinteraktion trotzdem Ein- und Ausgabeoperationen durchgeführt werden können, muss jede funktionale Programmiersprache letztendlich dennoch Aufrufe mit sogenannten unreinen Seiteneffekten zulassen.

Der Mathematik kommt in der funktionalen Programmierung grundsätzlich eine höhere praktische Relevanz zu als in der imperativen Programmierung (siehe die Abschnitte 3.4.4, 3.4.5 oder [Yor09]). Es lassen sich Gleichungen und Formeln viel einfacher direkt übernehmen (z. B. *equational reasoning* oder *denotational semantics*) oder Beweismethoden zur Überprüfung der Korrektheit des Programms anwenden.

Die Vorteile funktionaler Programmierung sind vielfältig (siehe [Hug84]). Da Funktionen nur, und wirklich nur, von ihren Eingabeparametern abhängig sind, können sie völlig unabhängig existieren. Meist bieten sie nur wenig Funktionalität an, können aber sehr einfach zu komplexen Funktionen kombiniert werden. Aufgrund der fehlenden Seiteneffekte lässt sich Parallelisierung leichter einsetzen. Vor allem Letzteres ist aufgrund von Mehrkern-Prozessoren ein Grund, warum einige Artikel der funktionalen Programmierung wieder eine höhere Relevanz zusprechen [Swa09] [SK09] [Dor09].

Die Nachteile funktionaler Programmierung sind hauptsächlich praktischer Natur. *Wadler* nennt z. B. fehlende Programmbibliotheken und Ausbildung [Wad98]. *Hague* kritisiert vor allem im Zusammenhang mit Computerspielentwicklung die Schwierigkeit, einen kompletten *Game-State* funktional abzubilden und die generell weniger intuitive Programmierung [Hag07]. Auf dieses Problem wird in Kapitel 4 eingegangen.

### 3.2.4 Reaktive Programmierung

Programme lassen sich hinsichtlich der Benutzerinteraktion in drei Kategorien aufteilen [Ber00]:

**Transformierende Systeme** bilden Eingabedaten auf Ausgabedaten ab und stoppen anschließend (z. B. Funktionen oder Compiler).

**Interaktive Systeme** interagieren nur gelegentlich mit dem Benutzer, wodurch die weitere Berechnung gesteuert wird (z. B. Batchprozesse oder Webseiten).

**Reaktive Systeme** erhalten einen ständigen Strom an **kontinuierlichen** Eingabedaten, auf die sie entsprechend reagieren und sofort wieder ausgeben (z. B. Signalprozessoren oder Computerspiele).

Es kann sich dabei grundsätzlich um kontinuierliche Eingaben jeglicher Art handeln (z. B. Temperatursensoren, Kamerasignale, Audioströme), wobei aber für Computerspiele hauptsächlich die Zeit abstrahiert wird, wodurch die restliche Funktionalität anschließend in Abhängigkeit der kontinuierlichen Zeiteingaben ausgedrückt werden kann.

Die Forschung von reaktiver Programmierung fand bisher hauptsächlich in der Programmiersprache *Haskell* statt, weshalb meist nur der Begriff *funktional-reaktive Programmierung* (*functional reactive programming*) verwendet und als domänenspezifische Programmiersprache implementiert wird. Die erste Implementierung wurde 1997 von *Elliott* und *Hudak* mit *Fran* veröffentlicht und ermöglicht es, Animationen in Abhängigkeit der Zeit auszudrücken [EH97]. Der Namensgeber *Elliott* würde jedoch mittlerweile den Begriff *functional temporal programming* wählen [Ell09a]. Vorher existierte nur das sogenannte *synchronous dataflow programming*, z. B. in den Programmiersprachen *Esterél*, *Lustre* oder *Signal*. Bei diesen wird jedoch die Zeit diskret abgebildet, wodurch die Funktionalität nur sehr unnatürlich ausgedrückt werden kann. In der reaktiven Programmierung mit Zeiteingaben hingegen erscheinen die reaktiven Elemente kontinuierlichen Zeitströmen zu unterliegen, auch wenn tatsächlich nur die diskreten Zeiteingaben versteckt werden. Die weitere Forschung beschränkte sich jeweils auf unterschiedliche Problembereiche:

**Domäne:** Roboter mittels *Frob* [HCNP03], GUIs mittels *FranTK* und *Fruit*, Computervision mittels *FVision*.

**Ausdrucksstärke:** Frühe Implementierungen beschränkten die Anzahl reaktiver Elemente, wodurch die Anzahl statistisch zur Programmübersetzung festgelegt war und keine neuen Elemente dynamisch hinzugefügt werden konnten [NCP02].

**Zeitlöcher:** Durch gewollte Verzögerungen in der Berechnung können reaktive Elemente den Anschluss zur aktuellen Simulationszeit verlieren [Scu09] [LH07].

**Speicherlöcher:** Aufgrund nicht-strikter Evaluierung entstehen oft große Datenstrukturen, welche nicht mehr abgearbeitet werden [LH07].

**Hybride Zeitmodelle:** Vereinigung von kontinuierlicher Zeit mit diskreten Ereignissen [Wan02].

**Performance** [WTH02] [Nil05] [Ell09b]

Die aktuellen Implementierungen sind *Yampa* (letzte Aktualisierung: November 2008) und *Reactive* (letzte Aktualisierung: Juli 2010), welche alle oben genannten Bereiche abdecken und dadurch erstmals auf allgemeine Problemdomänen anwendbar sind. Die Besonderheiten von *Yampa* sind, dass die Funktionalität intuitiv mittels Arrows verknüpft werden kann, wodurch vor allem Zeit- und Speicherlöcher garnicht erst ausgedrückt werden können, ohne dadurch aber sinnvolle Ausdrücke einzuschränken. *Reactive* ist die jüngste Implementierung und es wurde hauptsächlich versucht, die Performance zu verbessern, indem alle Eingabedaten nur bei Bedarf von den reaktiven Elementen selbst angefordert werden (*pull*), statt diese allen direkt zu übergeben (*push*), wie dies in *Yampa* der Fall ist [Ell09b] [Ell09a]. Laut Angaben der Entwickler ist jedoch die derzeitige Version von *Reactive* noch etwas fehleranfällig. Für diese Arbeit wurde *Yampa* gewählt, da mehr theoretische und praktische Arbeiten vorliegen und die Programmbibliothek ausgereift und stabil ist. Es existiert eine bekannte Erweiterung von *Yampa* namens *FRVR*<sup>5</sup>, wobei einige Umstrukturierungen im Code durchgeführt wurden, um die Funktionalität von *Yampa* selbst erweitern zu können [Blo09].

### 3.3 Haskell

*Haskell* ist eine **rein funktionale Programmiersprache** (*pure functional*) und nach dem Mathematiker *Haskell Brooks Curry* benannt. Sie entstand aus dem Bedarf, eine einheitliche Programmiersprache für die Erforschung funktionaler Programmierung zu entwickeln und ist stark von der Programmiersprache *Miranda* beeinflusst. Die erste Version wurde 1990 mit *Haskell 1.0* veröffentlicht, 1997 wurde sie mit *Haskell 98* erstmals standardisiert und befindet sich derzeit im *Haskell 2010* Standard. Die Programmiersprache vereint dabei einige sehr einzigartige Eigenschaften:

**Rein funktional:** In *Haskell* wird durch die Syntax grundlegend das funktionale Programmierparadigma unterstützt, allerdings wird ein Programm klar getrennt in einen rein funktionalen Teil und einen nicht-deterministischen Ein-/Ausgabeteil (*IO Monade*). Ein Vorteil davon ist, dass Programme sehr klar strukturiert und nachvollziehbar sind. Ein Nachteil ist aber, dass dadurch teilweise sehr abstrakte Konstrukte entstehen. Im Vergleich dazu sind in *Lisp* jederzeit Funktionsaufrufe mit Seiteneffekten möglich (*impure functional*).

---

<sup>5</sup><https://frvr.svn.sourceforge.net/svnroot/frvr/AFRP/trunk>

**Nicht-strikte Evaluierung:** In *Haskell* werden Berechnungen nur bei Bedarf durchgeführt (*non-strict evaluation*). Wenn nötig, kann jedoch trotzdem explizit strikte Evaluierung (*strict evaluation*) erzwungen werden. Ein Vorteil davon ist, dass beispielsweise bei komplexen Berechnungen auf Listen die einzelnen Listenelemente nur nach und nach zurückgegeben und berechnet werden, wodurch nicht ständig neue Zwischenliste erzeugt werden. Ein Nachteil ist aber, dass die genaue Ausführungsreihenfolge undefiniert und schwer nachvollziehbar ist. Im Vergleich dazu wird in *Lisp* nach jedem Funktionsaufruf eine komplett neue Liste erzeugt und die Evaluierung kann nur mittels *Macro SPELs*<sup>6</sup> verzögert werden.

**Starke statische Typisierung:** In *Haskell* müssen Datentypen bereits vor der Programmübersetzung genau definiert sein. Im Gegensatz zu anderen stark typisierten Programmiersprachen leiten sich die konkreten Typen meist durch Typinferenz (*type inferencing*) automatisch ab und müssen selten explizit angegeben werden. Ein Vorteil davon ist, dass noch bereits während der Programmübersetzung auf Gültigkeit überprüft wird und dadurch sehr, sehr viele fehlerhafte Ausdrücke garnicht erst möglich sind. Ein Nachteil ist aber, dass die Datentypen teilweise sehr generisch und schwer nachvollziehbar sind. Im Vergleich dazu verwendet *Lisp* schwache, dynamische Typisierung (*weak dynamic typing*), wodurch ein Wert jeden beliebigen Typ annehmen und zur Laufzeit wechseln kann, dadurch aber möglicherweise auch erst zur Laufzeit inkompatible Typen entstehen.

### 3.3.1 Notation

Für Funktionsaufrufe kann grundsätzlich die in funktionaler Programmierung übliche *Prefix-Notation* verwendet werden, wodurch allerdings auch viele Klammerungen nötig sind. Zusätzlich ist jedoch auch *Infix-Notation* möglich, um vor allem mathematischen Berechnungen mit Operatoren ein typisches Aussehen zu verleihen.

```

1 (+) :: Int → Int → Int -- Funktionssignatur
2 (+) 1 2                  -- Aufruf in Prefix-Notation
3 1 + 2                  -- Aufruf in Infix-Notation
4
5 add :: Int → Int → Int -- Funktionssignatur
6 (add 1 2)                -- Aufruf in Prefix-Notation
7 1 `add` 2                -- Aufruf in Infix-Notation

```

In der funktionalen Programmierung sind Listen ein grundlegendes und wichtiges Konzept, weshalb in *Haskell* eine Reihe an Notationen zur Verfügung stehen. Eine Liste besteht aus *Cons-Zellen*, um die Werte einer Liste zu

---

<sup>6</sup>SPELs sind ein von Barski eingeführter Begriff, um den grundlegenden technischen Unterschied zu prozeduralen Macros hervorzuheben.

speichern und mit dem Operator `(:)` erzeugt werden, und einem leeren Wert `[]`, welcher das Ende der Liste (*Nil*) angezeigt. Einfache Listen können durch die Ausdrücke `1 : 2 : 3 : []` bzw. `[1, 2, 3]` erstellt werden, während komplexere Listen mittels **Listennotation** (*list comprehensions*) produziert werden können. Der folgende Codeabschnitt zeigt einige Beispiele für Listennotation:

```
1 [x * x | x ← [1..3]]           -- [1, 4, 9]
2 [x      | x ← [1..9], even x]   -- [2, 4, 6, 8]
3 [x + y | x ← [1..2], y ← [3..4]] -- [4, 5, 5, 6]
```

*Haskell* kann zusätzlich durch Spracherweiterungen (*language extensions*) die Syntax an sich erweitern (z. B. die *Arrow-Notation* in Abschnitt 3.4.5). Domänen spezifische Programmiersprachen (*domain-specific languages*) können dadurch relativ einfach implementiert werden, vor allem in Verbindung mit Infix-Notation und Operatoren. *Haskell* stellt somit eine Basis für Programmiersprachen (*host language*) dar und kann für einen speziellen Anwendungsfall angepasst werden, ohne jedesmal eine neue Programmiersprache von Grund auf neu entwickeln zu müssen. In dieser Hinsicht ist *Haskell* allerdings weniger mächtig als *Lisp*, in der die Programmiersprache allein durch *Macro SPELs* erweitert werden kann und überhaupt keine Unterscheidung zwischen Daten, Funktionen und Syntax getroffen wird (*Metaprogrammierung*).

### 3.3.2 Funktionen

Neue Funktionen können in *Haskell* grundsätzlich mit klassischen Funktionsdefinitionen erzeugt werden. Für Funktionen höherer Ordnung können, wie in den meisten funktionalen Programmiersprachen üblich, direkt beim Funktionsaufruf zusätzliche anonyme Funktionen (*lambda functions*) definiert werden oder bestehende Funktionen mittels Funktionskomposition zu komplexen Funktionen verknüpft werden. Mit dem Funktionsapplikations-Operator `(\$)` können zudem Funktionen rechts-assoziativ aufgerufen werden, wodurch vor allem die in funktionaler Programmierung typischen Klammierungen wegfallen.

Die Funktionssignaturen in *Haskell* stellen grundsätzlich Abbildungen von Typen dar. Zum Beispiel ist `Int → Int → Int` eine Funktion die zwei `Int` entgegennimmt und daraus ein neuen `Int` berechnet. Alternativ kann dies jedoch auch betrachtet werden als eine Funktion, welche nur einen `Int` entgegennimmt und eine neue Funktion (`Int → Int`) zurückgibt. Der Vorteil davon ist, dass sich **Funktionssignaturen von bestehenden Funktionen beliebig erweitern lassen** und zusammen mit den Techniken *Currying* und *Sections* mächtige Konstrukte ermöglichen lassen. *Currying* erlaubt es bei Funktionsdefinitionen die Parameterbezeichnungen wegzulassen (*point-free style*), wenn sie sich automatisch aus den verwendeten Funktionen

ergeben<sup>7</sup>. Mit *Sections* werden die Parameter einer Funktion nur teilweise angewendet, wobei die offenen Parameter erst direkt beim Funktionsaufruf übergeben werden müssen.

```

1 inc :: Int → Int           -- Funktionssignatur
2 inc x = (+) 1 x           -- mit point-wise style
3 inc  = (+) 1               -- und point-free style (Currying).
4
5 sqrt  (1 + 2 + 3)         -- Vorrang mittels Klammern
6 sqrt $ 1 + 2 + 3          -- und Funktionsapplikation.
7
8 map inc      [1..10]       -- Funktionsanwendung auf Liste,
9 map (inc ∘ inc) [1..10]     -- mittels Funktionskomposition,
10 map (λ x → 1 + x) [1..10] -- anonyme Funktion,
11 map (1 +)   [1..10]       -- und einer Section.

```

### 3.3.3 Typen

Eines der zentralen Konzepte in *Haskell* sind Typen. Es stehen einfache Datentypen (z. B. `Int`, `Fractional`, `Float`) und komplexe Datentypen (z. B. Funktionen, Tupel, Listen) zur Verfügung. Zusätzlich können eigene Datentypen (*algebraic data types*) aus den bestehenden Datentypen oder symbolischen Werten (ähnlich zu *enums*) zusammengesetzt werden. Die einzelnen Felder können optional auch benannt werden (*records*), wobei ein Zugriff darauf mit gleichlautenden Funktionsaufrufen möglich ist. Bei einem Datentyp wird zwischen dem eigentlichen Typ und seinen verschiedenen Wertausprägungen unterschieden, welche mithilfe von Konstruktoren erzeugt werden können (z. B. Datentyp `Shape` mit den Wertausprägungen `Circle` oder `Polygon`, wobei diese völlig unterschiedlich aufgebaut sein können). Mittels *pattern matching* und *deconstruction* ist es anschließend möglich, für jede konkrete Wertausprägung jeweils eine eigene Funktion zu definieren bzw. sogar Bedingungen an konkrete Werte oder sogar die Struktur der Werte zu stellen. Dadurch werden viele Verzweigungen im Funktionsrumpf vermieden und das Programm kann sauber strukturiert werden. Zusätzlich können bei der Typdefinition noch zusätzliche **Typparameter** angegeben werden, wodurch ein Typkonstruktor definiert wird und polymorphe Datentypen (*polymorphic data types*) möglich sind (z. B. von `Tree t` zu `Tree Int` oder `Tree String`). Der Unterschied von Eingabeparametern, Typen und Typkonstruktoren ist für das weitere Verständnis sehr entscheidend, da in den folgenden Kapiteln oft sehr generische Funktionssignaturen verwendet und erweitert werden. Gegebenenfalls sollte daher auf entsprechende *Haskell* Literatur zurückgegriffen werden! Das folgende Codebeispiel zeigt einfache Beispiele für die genannten Techniken:

---

<sup>7</sup>Der *point-free style* wird nur bei kurzen Funktionsdefinitionen empfohlen, um die Übersicht und Klarheit zu bewahren.

```

1 data Shape           -- Typdefinition
2     = None          -- und Konstruktoren.
3     | Circle Int
4     | Polygon [(Int, Int)]
5
6 data Tree t = Branch Tree Tree -- Polymorphe Typdefinition
7           | Leaf t      -- mit Typparameter t.
8
9 drawShape :: Shape → IO ()    -- Funktionssignatur
10
11 -- Pattern matching auf symbolischen Wert None
12 drawShape None      = putStrLn "Empty shape"
13
14 -- Pattern matching auf Circle mit Radius 0 und Nicht-0
15 drawShape (Circle 0) = drawPoint
16 drawShape (Circle x) = drawCircle x
17
18 -- Deconstruction einer Liste in Gesamtliste, Kopf und Rest
19 drawShape (Polygon l@[x:xs]) = drawPolygon l >> drawLine x

```

### 3.3.4 Klassen

In *Haskell* sind Klassen eher im mathematischen Sinn zu verstehen und definieren abstrakte Eigenschaften und Bedeutungen, welche von einem konkreten Typ durch Klasseninstanzen<sup>8</sup> festgelegt werden müssen. Dies entspricht in etwa einem *Interface* oder einer *abstrakten Klasse* in der objektorientierten Programmierung. Zusätzlich können Gleichungen definiert werden, wodurch oft nur wenige der geforderten Funktionen tatsächlich implementiert werden müssen (*Minimal complete definition*). Das folgende Codebeispiel zeigt diese Techniken anhand der `Eq` Klasse, welche zwei Werte miteinander vergleichen kann, wobei nur entweder `(==)` oder `(/=)` implementiert werden muss:

```

1 class Eq t where
2     (==), (/=) :: t → t → Bool
3
4     x /= y  =  ¬(x == y)
5     x == y  =  ¬(x /= y)
6
7 data MyData = Foo Int String | Bar Int String
8
9 instance Eq MyData where
10    Foo i1 str1 == Bar i2 str2  =  i1 == i2 && str1 == str2

```

---

<sup>8</sup>Instanzen werden vor der Programmübersetzung definiert und sind nicht mit Laufzeitinstanzen in der objektorientierten Programmierung zu verwechseln.

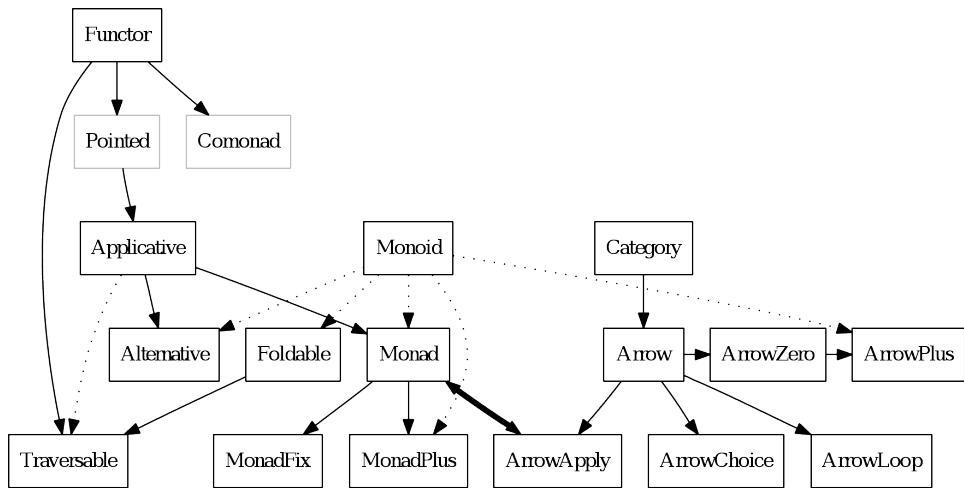


Abbildung 3.2: Standard-Typklassen in Haskell (entnommen aus [Yor09]).

### 3.4 Typklassen

In *Haskell* werden einige mathematische Typklassen definiert, welche für einen Typkonstruktor (z. B. Tree t) zusätzliche Bedeutungen erfordern und dadurch für vielerlei Funktionalität auf einer sehr abstrakten Ebene verwendet werden können, wobei die Bedeutungen durch eine Klasseninstanz des Typkonstruktors implementiert werden. Mit Typklassen kann in einer Funktionssignatur der Unterschied zu reinen Funktionen ausgedrückt werden, sodass nicht nur eine Transformation von Eingabeparametern zu einem Ausgabewert stattfindet, sondern die Berechnung innerhalb des Kontextes eine veränderte Bedeutung annimmt. Eine Übersicht aller Typklassen ist in Abbildung 3.2 illustriert.

Die folgenden Beschreibungen basieren auf der *Typeclassopedia* [Yor09], wobei nur die für diese Arbeit relevanten Typklassen **Functor**, **Pointed**, **Applicative** beschrieben werden, welche anschließend für das Verständnis der Typklassen **Monad** und **Arrows** nötig sind. Alle Typklassen müssen bei der Instanzierung zusätzlich gewisse Gesetze einhalten, allerdings wird in dieser Arbeit auf deren Auflistung verzichtet. Folgend werden metaphorische Interpretation bewusst vorsichtig eingesetzt, da Typklassen sehr abstrakte Konzepte sind und durch Metaphern sehr leicht falsche Vorstellungen vermittelt werden würden! Die Anwendungsmöglichkeiten sind dadurch vorerst schwer ersichtlich, allerdings ermöglichen Typklassen sehr mächtige Funktionalität auf einer sehr abstrakten Ebene, weshalb nur sehr wenig konkrete Implementierung nötig ist.

### 3.4.1 Funktoren

Ein Funktor (*functor*) ist die grundlegendste Typklasse und stellt den eigentlichen Berechnungskontext (*computational context*) dar, welche eine normale Funktion in einen Kontext hebt (*lifting*) und eine neue Bedeutung für die Transformation der Werte innerhalb des Kontextes erfordert. Der umgebende Kontext an sich darf dabei nicht verändert werden.

Eine einfache Interpretation eines Funktors ist ein Container, wobei der Container den Kontext darstellt und verschiedene Elemente eines bestimmten Typs beinhaltet und die Bedeutung im Kontext lautet, die Funktion einfach auf alle Elemente anzuwenden. Ein Container ist nur eine mögliche Interpretation, da die Anwendung der Funktion innerhalb des Kontextes immer durch die konkrete Instanzierung definiert wird. Wie der Kontext ursprünglich erzeugt wird oder woher dieser kommt, ist dabei für einen Funktor irrelevant. In *Haskell* ist die Klasse **Functor** folgendermaßen definiert:

```
1 class Functor ctx where
2     fmap :: (t1 → t2) → ctx t1 → ctx t2
```

Die Klasse **Functor** nimmt bei der Instanzierung einen **Typkonstruktör** **ctx** entgegen (z.B. **Tree**, nicht jedoch **Tree Int** oder nur **Int**) und erfordert die Implementierung einer Funktion **fmap**, welche der Transformation innerhalb des Kontextes eine Bedeutung gibt. Bei Anwendung von **fmap** werden möglicherweise unterschiedliche innere Typen **t1** und **t2** verwendet und die Werte entsprechend geändert, allerdings bleibt der Kontext **ctx** immer gleich (z.B. bleibt ein **Tree** immer ein **Tree**). Mittels *Currying* lässt sich die Funktionssignatur etwas anders darstellen, wodurch die Funktionsanwendung innerhalb des Kontextes klarer ersichtlich wird:

```
1 fmap :: (    t1 →      t2)
2          → (ctx t1 → ctx t2)
```

### 3.4.2 Punktierter Funktoren

Ein punktierter Funktor (*pointed functors*) ist eine Typklasse, welche für einen Berechnungskontext (Funktör) nur eine weitere Bedeutung für einen Standardkontext erfordert, wobei ein einfacher Wert in den Kontext gehoben wird. Für die Interpretation von Containern ist der Standardkontext ein Container mit nur einem Element (z.B. **[x]**). In *Haskell* kann die Klasse **Pointed** folgendermaßen definiert werden<sup>9</sup>:

```
1 class Functor ctx ⇒ Pointed ctx where
2     pure :: t → ctx t      -- auch unit oder return genannt
```

---

<sup>9</sup>**Pointed** ist derzeit nur in der Programmzbibliothek **category-extras** enthalten, nicht in den Standardbibliotheken.

### 3.4.3 Applikative Funktoren

Ein applikativer Funktor (*applicative functor*) ist eine Typklasse, welche für einen punktierten Funktor (Berechnungskontext mit Standardkontext) eine zusätzliche Bedeutung für eine kontextbezogene Funktionsapplikation erfordert. In *Haskell* kann die Klasse **Applicative** folgendermaßen definiert werden:

```
1 class Pointed ctx => Applicative ctx where
2     (<*>) :: ctx (t1 -> t2) -> ctx t1 -> ctx t2
```

Im Unterschied zu einem normalen Funktor ist die übergebene Funktion daher im Kontext eingeschlossen. Ähnlich zur Funktionsapplikation (\$), kann dadurch eine Applikation von Funktionskontexten auf andere Kontexte durchgeführt werden. Zum Beispiel wird die Instanzierung von **Applicative** für den Typkonstruktor [] in *Haskell* standardmäßig als nicht-deterministische Berechnung interpretiert, wobei die Liste alle möglichen nicht-deterministischen Werte enthält und eine kontextbezogene Funktion, d. h. die nicht-deterministische Berechnung, entsprechend auf jeden einzelnen Wert angewendet wird, wie in folgendem Codebeispiel:

```
1 pure (+) <*> [1, 2] <*> [3, 4]
2 -- [4, 5, 5, 6]
```

Die normale Funktion (+) wird mit **pure** in den Kontext von nicht-deterministischen Berechnungen gehoben [(+)], bei der ersten Applikation auf alle Werte in der Liste angewendet, wodurch sich [(+ 1), (+ 2)] ergibt. Bei der nächsten Applikation werden die (teilweise angewendeten) Funktionen wiederum einzeln mit allen Werten der nächste Liste aufgerufen. Für die Interpretation von normalen Listen wird hingegen der Datentyp **ZipList** bereitgestellt und wendet in der Instanz **Applicative** alle Funktionen in der Liste als Funktionskomposition auf jeden Wert an.

### 3.4.4 Monaden

Eine Monade (*monad*) ist eine Typklasse, welche für einen applikativen Funktor (Berechnungskontext mit Standardkontext und kontextbezogener Funktionsapplikation) eine zusätzliche Bedeutung für den Kollaps zweier Monaden in eine Monade mit der Funktion **join** erfordert. Historisch entstanden Monaden um 1960 in der *Kategorientheorie* der Mathematik und wurden 1988 erstmals von *Moggi* für die funktionale Programmierung übernommen. Die bisher nötigen Funktionen **fmap**, **pure** und **join** stellen die elementaren Funktionen von Monaden dar und werden *Kleisli Triple* genannt. Die Klasse **Monad** lässt sich daher folgendermaßen definieren:

```
1 class Applicative m => Monad m where
2     join :: m (m t) -> m t
```

Erst 1990 wurden Monaden durch *Wadler* am Beispiel von Listenausdrücken einer breiteren Masse von Programmierern verständlich zugänglich gemacht [Wad92]. Wird eine Liste als Monade interpretiert, wendet `fmap` die übergebene Funktion auf alle Elemente an (entspricht `map`), erzeugt mit `pure` eine Liste mit nur einem Element (entspricht `[x]`) und kollabiert mit `join` eine Liste aus Listen in eine neue Liste des selben Typs (entspricht `concat`). Monaden können allerdings für verschiedene Funktionalität verwendet werden, Listen sind nur eine mögliche Interpretation. Weitere Beispiele für Monaden sind Zustandsänderungen (`IO`, `State`), Fehlerbehandlungen (`Maybe`, `Either`) usw.

In *Haskell* wird allerdings eine alternative Definition von Monaden verwendet, welche die elementaren Funktion vereint und dabei vor allem die Möglichkeit betont, dass sich Monaden hintereinander kombinieren lassen und eine Berechnung in einer Monade eine anschließende Berechnung beeinflussen kann:

```
1 class Applicative m ⇒ Monad m where
2     (≫) :: m t1 → (t1 → m t2) → m t2    -- bind
```

Mit Monaden kann in einer Funktionssignatur somit ebenfalls der Unterschied zu reinen Funktionen ausgedrückt werden, sodass nicht nur eine Transformation von Eingabeparametern zu einem Ausgabewert stattfindet, sondern eine Berechnung innerhalb eines Berechnungskontext durchgeführt wird, welcher sich hintereinander kombinieren lässt und sich (potenziell) darin enthaltenen Werten bedient. Die Bedeutung der Kombination muss allerdings wiederum durch die konkrete Instanzierung definiert werden.

Es lässt sich eine Monade instanzieren, welche bei der Kombination eine Reihenfolge in der Berechnung erzwingt und zusätzlich eine Zustandsänderung durchführt, sodass über den Zwischenzustand bei der Kombination keine Aussage getroffen werden kann. Diese Tatsache wird in der Hauptprozedur `main` durch die *IO Monade* genutzt und nimmt in *Haskell* eine Sonderstellung ein, da es der einzige Bereich ist, indem Seiteneffekte zugelassen sind<sup>10</sup>. In reinen Funktionen können somit keine *IO Monaden* verwendet werden, da sie ansonsten Seiteneffekte aufweisen würden, allerdings können reine Funktionen in *IO Monaden* verwendet werden, wodurch große Bereiche eines Programms rein funktional strukturiert werden können.

In *Haskell* wird zusätzlich eine eigene Syntax für Monaden bereitgestellt, die sogenannte *do-notation* oder *Monad comprehensions*, wodurch die Zwischenergebnisse von `(≫)` mittels `←` auf Bezeichner gebunden werden können und einen imperativen Programmierstil ermöglichen. Das folgende Codebeispiel zeigt wie die Klasse `Monad` (aus historischen Gründen) tatsächlich in *Haskell* definiert ist und eine Anwendung der *IO Monade* in *Haskell* mittels Standardsyntax und entsprechender *do-Notation*:

---

<sup>10</sup>ausgenommen `System.IO.Unsafe`

```

1 class Monad m where
2     return :: t → m t                      -- pure
3     (≫=)  :: m t1 → (t1 → m t2) → m t2    -- bind
4
5 main :: IO ()
6 main = getLine ≫= (λ x → putStrLn $ "Hello " ++ x)
7
8 mainDo :: IO ()
9 mainDo = do
10     name ← getLine           -- :: IO String
11     putStrLn $ "Hello " ++ name -- :: String → IO ()

```

Durch Monaden können Seiteneffekte im Code klar deklariert werden und das Programm behält seinen funktionalen Stil. Dies wird jedoch teilweise als inkonsistent kritisiert [Hoy08], tatsächlich verdankt *Haskell* jedoch einen Großteil seiner Popularität den Monaden, da bisher ähnliche Strukturen nur mit komplexen und fehleranfälligen Gebilden aus anonymen Funktionen möglich waren.

### 3.4.5 Arrows

Ein *Arrow* ist eine Typklasse und stellt ebenfalls einen Berechnungskontext dar, wobei die Klasse eine Bedeutung für die Kombination von Arrows erfordert, aber die Berechnung zusätzlich von einem bestimmten Eingabetyp abhängig ist, im Unterschied zu Monaden. Weiters kann bei der Kombination von Arrows ausgedrückt werden, dass nur einer von zwei Eingabeparametern berechnet wird, während der zweite unverändert durchgereicht wird.

Historisch wurden Arrows 1998 von *Hughes* [Hug00] eingeführt und stellen Verallgemeinerungen von Monaden dar<sup>11</sup>. Seine ursprüngliche Motivation war es, eine allgemeine Kombinationsmöglichkeit für Parser zu finden, welche zuvor nur dadurch erreicht werden konnte, indem die Verwendung von Monaden aufgegeben wurde und dadurch ein Großteil an bereits implementierten Kombinationsmöglichkeiten weggefallen wäre. Die Klasse **Arrow** wird in *Haskell* folgendermaßen definiert:

```

1 class Arrow arr where
2     arr   :: (in → out) → arr in out      -- pure
3     (≫;) :: arr in1 out1 → arr in2 out2 → arr in1 out2
4     first :: arr in1 out1 → arr (in1, in2) (out1, in2)

```

**arr** hebt eine Funktion ( $\text{in} \rightarrow \text{out}$ ) in den Kontext von Arrow **arr** (*lifting*).  
**≫;** ermöglicht zwei Arrows zu kombinieren (*composition*).  
**first** ermöglicht es ein Tupel ( $\text{in}_1, \text{in}_2$ ) als Eingabeparameter zu verwenden (*widening*), sodass der erste Eingabeparameter  $\text{in}_1$  berechnet und der zweite Eingabeparameter  $\text{in}_2$  nur weitergereicht wird.

---

<sup>11</sup>Theoretisch lassen sich alle Monaden durch einen Arrow abbilden.

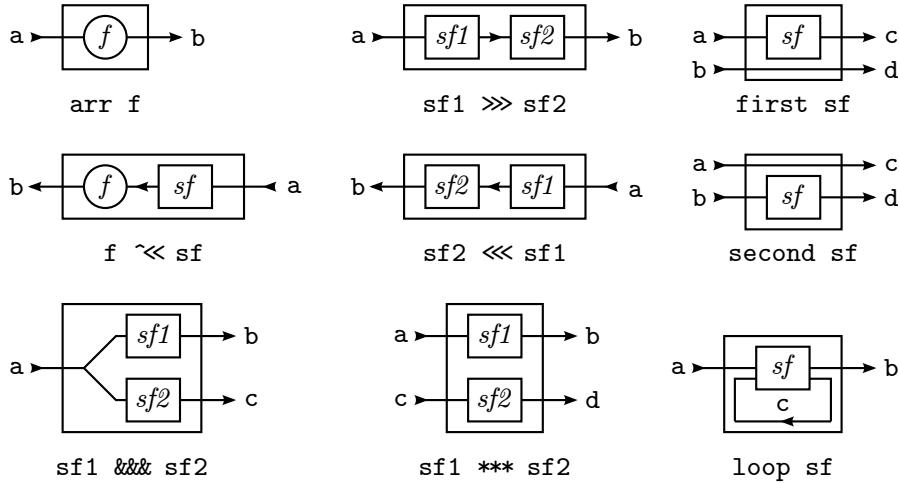


Abbildung 3.3: Illustration grundlegender Arrow Funktionen.

Diese drei Funktionen stellen eine minimale Definition dar, woraus sich alle weiteren grundlegenden Kombinatoren ableiten lassen, welche in Abbildung 3.3 illustriert sind. Es existieren zusätzliche Spezialisierungen der Klasse `Arrow`, vor allem die Klasse `ArrowLoop`, mit der rekursive Berechnungen deklariert werden.

Die direkte Verwendung der Arrow-Funktionen würde jedoch bei größeren Konstrukten unübersichtlich werden, da keine Bezeichnungen für dazwischenliegende Berechnungen bei der Kombination verwendet werden können (*point-free style*). Deshalb wurde 2001 von *Paterson* eine neue Notation für Arrows eingeführt [Pat01], ähnlich zur *do-Notation* von Monaden. Diese Notation ermöglicht es im Programmcode eine ähnliche Struktur wie in Abbildung 3.4 zu erreichen, wobei das entsprechende Codebeispiel folgendermaßen aussieht:

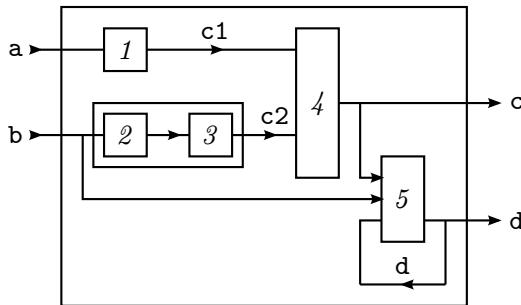
```

1 arrow = proc (a, b) → do
2   c1 ← arrow1 ⤵ a
3   c2 ← arrow3 <<< arrow2 ⤵ b
4   c  ← arrow4 ⤵ (c1, c2)
5   rec
6     d ← arrow5 ⤵ (c, b, d)
7   returnA ⤵ (c, d)

```

### 3.5 Yampa

*Yampa* wurde von der *Yale Haskell Group* entwickelt und ist eine Programmiersbibliothek, mithilfe derer funktional-reaktive Programmierung mit kontinu-



**Abbildung 3.4:** Ein Diagramm eines komplexeren Arrow kombiniert aus einfacheren Arrows.

ierlichem Zeitverlauf als eingebettete Programmiersprache in *Haskell* möglich ist. Ein Programm wird dabei aus einem System von reaktiven Elementen aufgebaut, welche mit kontinuierlichen Signalen verknüpft werden und auf einige davon Zeit einwirken scheint. Die reaktiven Elemente sind nur von allen bisherigen Eingabedaten seit deren Erzeugung abhängig und von keinem Zustand außerhalb, wodurch die Vorteile funktionaler Programmierung bestehen bleiben. Die reaktiven Elemente können somit, ähnlich wie Funktionen, vollkommen unabhängig existieren und lassen sich zu komplexen Systemen kombinieren.

Laut *Hackage Haskell Repository* wurden bisher insgesamt vier Computerspiele mit *Yampa* entwickelt und sind in Tabelle 3.5 abgebildet:

**Space Invaders:** Eine Beispielimplementierung des gleichnamiges Klassikers von den *Yampa*-Entwicklern. Die vorliegende Arbeit basiert auf dem dazugehörigen Paper *The Yampa Arcade* [CNP03].

**Frag:** Ein Klon des Egoschooter *Quake 3* mit Unterstützung für BSP Maps, MD3 Modelle und einem Computergegner [Che05].

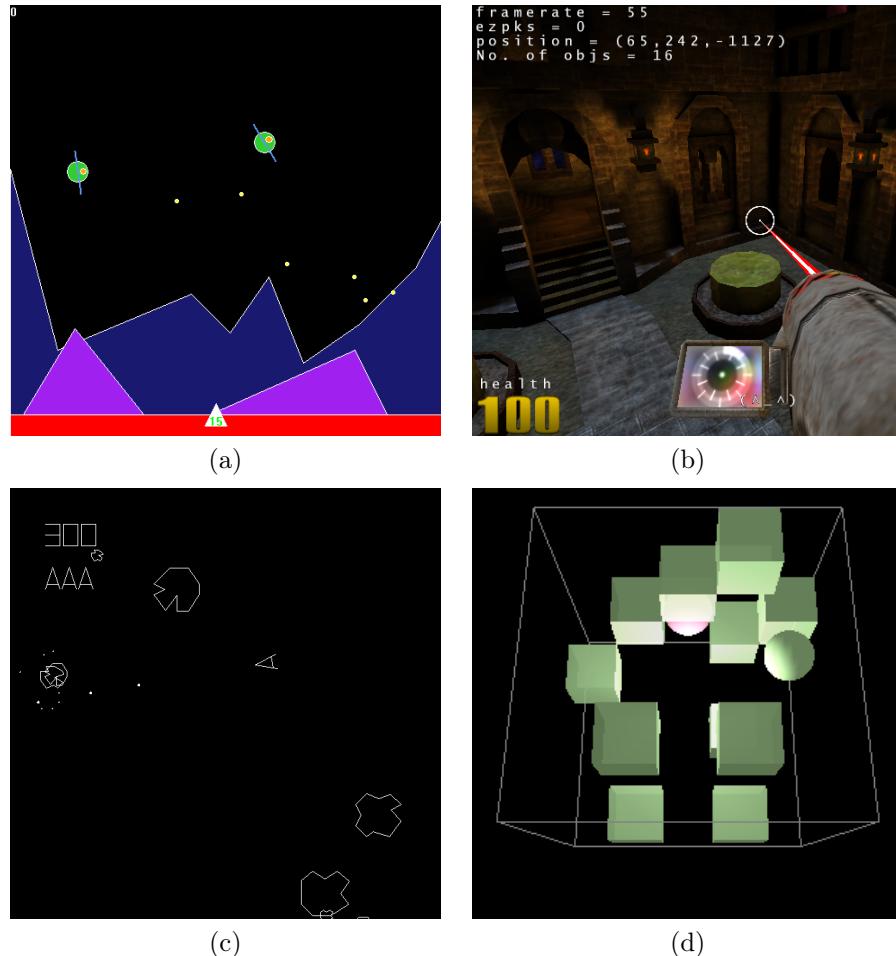
**Haskelloids:** Ein Klon des Klassikers *Asteroids*.

**Cuboid:** Ein einfaches 3D Puzzle.

### 3.5.1 Signalfunktionen

Die Basis von *Yampa* sind zeitbeeinflusste Berechnungen namens *Signalfunktionen* (*signal functions*) und sind als Datentyp **SF** implementiert. Ein einzelnes Signal lässt sich dabei **interpretieren** als eine Funktion, welche kontinuierliche, zeitabhängige Werte erzeugt und eine Signalfunktion als eine Transformation von einem Signal zu einem neuen Signal. Auf die tatsächliche Implementierung wird allerdings erst in Abschnitt 3.5.5 eingegangen.

<pre>1 Signal t ~:: Time → t 2 SF in out ~:: Signal in → Signal out</pre>	-- Definition ungültig, -- nur Interpretation!
---------------------------------------------------------------------------	---------------------------------------------------



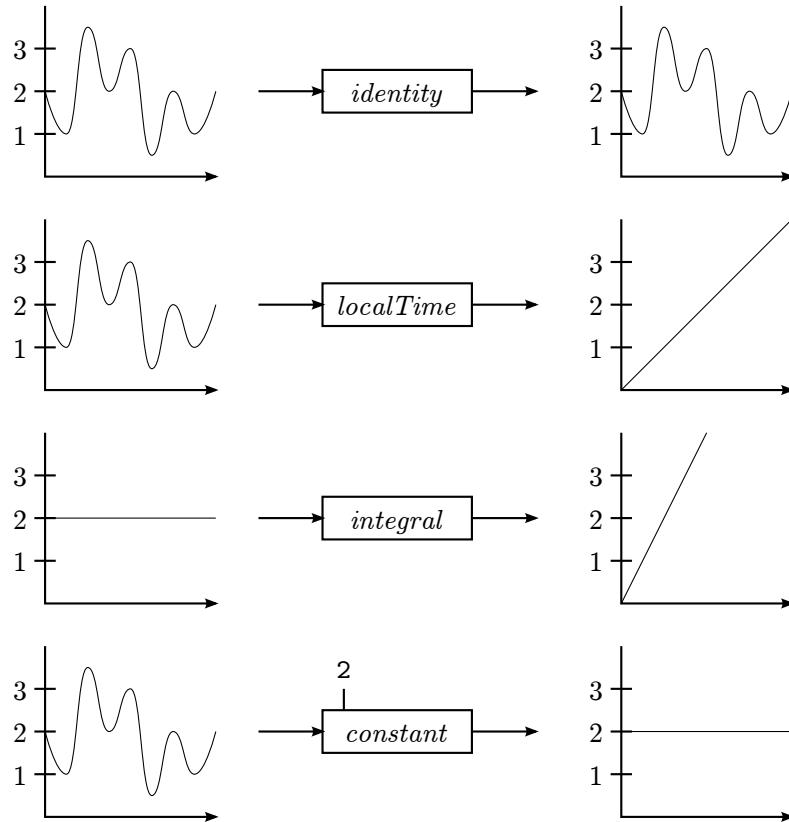
**Abbildung 3.5:** Screenshots aller Computerspiele, welche mit *Yampa* entwickelt wurden. Space Invaders (a), Frag (b), Haskelloids (c), Cuboid (d).

Signalfunktionen instanzieren die Klasse `ArrowLoop`, wodurch die *Arrow-Notation* verwendet werden kann. Die Funktionalität lässt sich gut am Integral veranschaulichen, dessen Werte über den Verlauf der Zeit aufsummiert werden. Normale Funktionen sind im Gegensatz dazu zustandslos, da sie nur von der aktuellen Eingabe abhängig sind. Diese können jedoch mit der Funktion `arr` in den Kontext von Arrows gehoben werden und lassen sich mit anderen Signalfunktionen zu komplexerer Funktionalität kombinieren. Das folgende Codebeispiel zeigt die Verwendung von Integralen:

```

1 exampleSF = proc in → do
2     a ← integral ↵ 2
3     b ← integral ↵ in
4     c ← arr (1 +) ≪< integral ↵ 2
5     returnA ↵ (a, b, c)

```



**Abbildung 3.6:** Die Signale auf der linken Seite werden an die Signalfunktionen angelegt und erzeugen die Signale auf der rechten Seite. Für die Signalfunktion `constant` muss zusätzlich ein Parameter bei der Erzeugung angegeben werden.

Dieses Beispiel lässt sich lesen als: Definiere ein neues reaktives Element `exampleSF`, mit dem kontinuierlichen Eingabestrom `in` (z. B. Joystickachse) und den kontinuierlichen Ausgabeströmen `(a, b, c)`. Signal `a` integriert die Zeit um den Faktor 2. Signal `b` schwankt über die Zeit in Abhängigkeit des Eingabestroms `in` (z. B. vertikale Position). Signal `c` hebt zuerst die (unvollständige) Funktion `(1 +)` zu einer Signalfunktion und addiert anschließend die doppelte Zeit. Die Verknüpfung findet an dieser Stelle im *point-free style* mit `<<<` statt, wodurch die Zwischenwerte nicht benannt werden müssen.

In *Yampa* werden insgesamt vier grundlegende Signalfunktionen definiert, mit denen die meisten kontinuierlichen Simulationen modelliert werden können, welche in Abbildung 3.6 illustriert sind.

### 3.5.2 Reactimate

Eine Simulation wird in *Yampa* mit der Funktion `reactimate` durchgeführt und entspricht im Grunde einer *Game-Loop*. Zuerst werden in einer *IO Monade* `init` die Eingabedaten `in` gesammelt und damit die Signalfunktion `SF in out` initialisiert. `SF` kann dabei eine einfache Signalfunktion wie `integral` sein oder ein komplexes System aus vielen dynamischen Signalfunktionen (z. B. *Game-Objekte*). `SF` produziert eine Ausgabe `out`, welche anschließend in der *IO Monade* `output` ausgegeben werden kann. In der *IO Monade* `input` werden anschließend die Eingabedaten – diesmal zusätzlich mit dem Zeitverlauf – gesammelt und oben beschriebener Prozess in einer Schleife wiederholt. Diese Schleife läuft solange, bis `output` den Wert `True` zurückliefert und dadurch den Prozess terminiert.

```

1 reactimate :: IO in           -- init
2      → IO (DTime, Maybe in)   -- input
3      → out → IO Bool         -- output
4      → SF in out            -- process
5      → IO ()
```

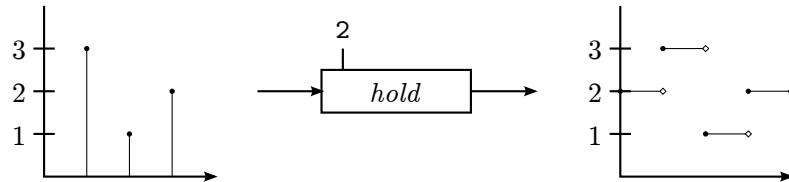
Dieser Prozess kann alternativ mit der Funktion `react` in Einzelschritten ausgeführt werden. Zusätzlich lässt sich mit `embed` eine eingebettete Simulation mit höherer Frequenz betreiben (z. B. für die Physik-Engine) oder automatisierte Testläufe durchführen. Folgendes Codebeispiel zeigt einen einfachen Testlauf mit `embed`:

```

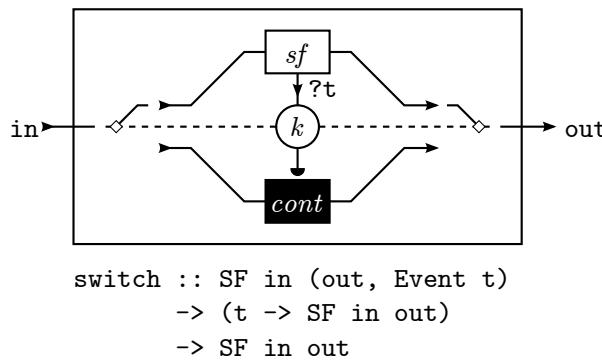
1 embed :: SF in out → (in, [(DTime, Maybe in)]) → [out]
2
3 main = do
4     putStrLn ∘ show ∘ embed $
5         integral (1, [(1.0, Just 2), (1.0, Just 3)])
6 -- [0.0, 1.0, 3.0] 6.0 fehlt aufgrund des Erstverzugs
```

### 3.5.3 Ereignisse

Neben kontinuierlichen Eingabeströmen ist es auch möglich, dass Ereignisse an diskreten Zeitpunkten auftreten (z. B. Mausklicks), welche jedoch keine Zeitdauer besitzen. Diese werden durch den Typ `Event t` abgebildet, wodurch der Eingabestrom entweder `NoEvent` oder `Event` mit einem konkreten Wert vom Typ `t` sein kann (und entspricht somit genau der Monade `Maybe`). Diskrete Ereignisse können mit der Funktion `hold` wiederum in kontinuierliche Ströme umgewandelt werden, wie in Abbildung 3.7 illustriert. Damit gleichzeitige Ereignisse in der richtigen Reihenfolge bearbeitet werden, können sie mit verschiedenen `merge` Funktionen zu einem neuen Gesamtereignis mit definierter Reihenfolge vereint werden. Zusätzlich können verschiedene zeitbasierte Ereignisquellen erzeugt werden, welche z. B. Ereignisse zu bestimmten oder periodischen Zeitpunkten auslösen.



**Abbildung 3.7:** Auf der linken Seite werden diskrete Ereignisse ausgelöst und als kontinuierliches Signal festgehalten.



**Abbildung 3.8:** Diagramm der Signalfunktion `switch`. Das Fragezeichen steht für ein mögliches Ereignis, welches den *Switch* auslöst. Das *switching* kann wiederum als Umleitung der Signale interpretiert werden. Die vorhergehende Funktionalität kann anschließend nie mehr erreicht werden und kann daher entfernt werden. Der C-förmige Pfeil verwendet dabei eine *Continuation*, wobei Cont bewusst als Black-Box dargestellt ist, da nur die Eingabe- und Ausgabetypen bekannt sind.

### 3.5.4 Switches

Ein *Switch* ermöglicht es, das Verhalten einer Signalfunktion zur Laufzeit zu ändern, indem auf ein Ereignis reagiert wird und anschließend eine neue Signalfunktion die weitere Verarbeitung übernimmt. Die einfachste Variante ist die Funktion `switch`, welche nur einmalig in eine neue Signalfunktion wechselt. Die Signalfunktion `switch` ist in Abbildung 3.8 illustriert und besitzt folgende Funktionssignatur:

```

1 switch :: SF in (out, Event t)
2           -> (t -> SF in out)
3           -> SF in out
  
```

Die Definition von `switch` lässt sich lesen als: Erzeuge eine Signalfunktion, die insgesamt den Typ `SF in out` besitzt, sich aber vorerst wie eine Signalfunktion `SF in (out, Event t)` verhält, welche wiederum potenziell ein `Event t` erzeugt. Wird das Ereignis ausgelöst, wird der darin enthaltene

	<i>immediate</i>	<i>delayed</i>
<i>once</i>	switch	dSwitch
<i>recurring</i>	rSwitch	drSwitch
<i>parallel using broadcasting</i>	pSwitchB rpSwitchB	dpSwitchB drpSwitchB
<i>parallel using routing</i>	pSwitch rpSwitch	dpSwitch drpSwitch
<i>continuation</i>	kSwitch	dkSwitch

**Tabelle 3.2:** Alle 14 Switchfunktionen in *Yampa*.

Wert vom Typ  $t$  an eine Funktion ( $t \rightarrow SF$  in  $out$ ) übergeben, welche daraus eine neue Signalfunktion, wiederum vom Typ  $SF$  in  $out$ , erzeugt. Die erzeugte Signalfunktion kann ebenfalls wieder ein *Switch* sein. Das Verhalten wird dabei sofort geändert (*immediate*) und anschließend nochmal berechnet. Es gibt jedoch zusätzlich eine verzögerte Variante (*delayed*), welche zum Zeitpunkt des Ereignisses noch den alten Wert ausgibt und erst danach das neue Verhalten annimmt.

Neben dem *einmaligen Switch* gibt es noch einen *externen Switch* (*recurring switch*), dem die gewünschte Signalfunktion direkt übergeben werden kann. Diese zwei *Switches* gibt es wiederum in einer Variante für Listen von Signalfunktionen und werden *parallele Switches* (*parallel switches*) genannt. Diese sind vor allem für Computerspiele interessant, da dadurch dynamische Strukturen abgebildet werden können (z. B. ein System von Game-Objekten, siehe Abschnitt 4.2). Die Eingabewerte können dabei entweder an alle enthaltenen Signalfunktionen verteilt werden (*broadcasting*) oder gezielt nur an bestimmte Signalfunktionen weitergereicht werden (*routing*). Zusätzlich gibt es noch einen *continuation switch* (*kSwitch* [ $!$ ]), der basierend auf den Ein- und Ausgabewerten des alten Verhaltens entscheidet, ein neues Verhalten anzunehmen oder nicht. *Yampa* verfügt somit insgesamt über 14 verschiedene *Switches*, welche in Tabelle 3.2 aufgelistet sind.

### 3.5.5 Implementierung

Das Ziel der Entwickler von *Yampa* war es eine Umgebung für **kontinuierliche Zeit** zu schaffen, wobei die konkrete Implementierung versteckt werden soll (siehe [Nil05], [NCP02] und [Blo09, Kap. A]). Dadurch soll einerseits die tatsächlich diskrete Zeit vor dem Anwendungsprogrammierer wegabstrahiert werden, z. B. sollen Integrale basierend auf den diskreten Zeiteingaben für verschiedene Anwendungsfälle angepasst werden können (z. B. *Rechteckverfahren* oder *Runge-Kutta-4*), konzeptionell nach außen hin aber wieder kontinuierlich wirken. Andererseits soll der Anwendungsprogrammierer vor

fehlerhaften Ausdrücken bewahrt werden, z. B. würden sich Signale der Zeiteingabe entziehen, wenn diese direkt verwendet werden könnten und in Listen verpackt werden. Zusätzlich muss der erste Ausgabestrom einer Signalfunktion unabhängig von der ersten Zeiteingabe definiert sein und nimmt anschließend immer nur Zeitverläufe entgegen. Die Signalfunktionen in *Yampa* sind daher folgendermaßen aufgeteilt:

```
1 data SF  in out = SF  (in → Transition in out)
2 data SF' in out = SF' (DTime → in → Transition in out)
3 type Transition in out = (SF' in out, out)
```

Dem Anwendungsprogrammierer stehen nur uninitialisierte Signalfunktionen vom Typ **SF** zur Verfügung, welche die Eingabeströme ohne Zeiteingabe entgegennehmen, um daraus einen Signalübergang **Transition** zu erzeugen. Ein Signalübergang besteht aus einer aktvierten Signalfunktion **SF'** **in** **out** und dem aktuellen Signalwert **out**. Der genaue Zeitpunkt, wann die Signalfunktion aktiviert wird, ist vor allem bei der Verwendung von verzögerten Switches wichtig. Das eigentliche Konzept von zeitbeeinflussten Signalen wird in aktvierten Signalfunktionen hinter sogenannten Signalübergangsfunktionen (*signalfunction transition functions*) (**DTime** → **in** → **Transition** **in** **out**) versteckt und nehmen – zusätzlich zu den Eingabeströmen – die diskreten Zeitverläufe entgegen, um daraus anschließend wieder einen neuen Signalübergang zu erzeugen usw. In *Yampa* werden intern verschiedene Typen für Signalübergangsfunktionen verwendet, einerseits für die unterschiedlichen Übergänge von einem uninitialisierten Zustand in einen aktvierten bzw. von einem aktvierten Zustand in einen weiteren aktvierten, und andererseits um für die unterschiedlichen Übergänge jeweils eigene Optimierungen anbieten zu können (z. B. konstante oder tatsächlich zeitbeeinflusste Funktionen) [Nil05].

Die Signalfunktion **integral** steht ebenfalls nur als **SF** zur Verfügung, wird aber intern mittels einer Signalübergangsfunktion zu einem **SF'** transformiert, welche die eigentlichen diskrete Zeiteingaben entgegennimmt. Es müssen nur selten eigene Signalfunktionen mittels internen Signalübergangsfunktionen definiert werden<sup>12</sup>, da die meisten reaktiven Elemente mithilfe von **integral** implementiert werden können. Es können aber grundsätzlich verschiedene Integrale oder andere Funktionen definiert werden, welche aus den diskreten Zeiteingaben entsprechend (sinnvolle) kontinuierliche Ausgabeströme erzeugen.

---

<sup>12</sup>Tatsächlich ist dies derzeit auch nur mit der *FRVR* Version von *Yampa* möglich, wie in Abschnitt 3.2.4 beschrieben

## Kapitel 4

# Implementierung der Game-Engine

### 4.1 Einleitung

In diesem Kapitel wird die Implementierung einer einfachen funktional-reaktiven Game-Engine beschrieben. Die Game-Engine basiert auf dem *Space Invaders* Beispiel der *Yampa*-Entwickler [CNP03], in dieser Arbeit werden jedoch speziell die Anforderungen an eine allgemeine Game-Engine-Architektur behandelt. Zuerst werden anhand eines einzelnen interaktiven Prozesses die Motivation und Überlegungen für die Implementierung einer vollständigen Game-Engine beschrieben. Anschließend erfolgt die Definition der Game-Objekte und die Implementierung der darüberliegenden Game-Loop und Game-Logic. Zur Abfrage der Zeit, Benutzereingaben und Ausgabe wurde als Subsystem *Simple DirectMedia Layer* (SDL) mit Haskell-Anbindung *hssl* gewählt. Die folgenden Beschreibungen werden bewusst teilweise wiederholt, um das Verständnis zu erleichtern.

#### 4.1.1 Überlegungen

Eine Game-Engine entsprechend dem EVA-Modell ist in seiner einfachsten Form nur ein einzelner interaktiver Prozess ohne Zeitverlauf und ermöglicht einfache Computerspiele wie z.B. Textadventures. Dieser einzelne Prozess kann in mehrere Teilprozesse aufgeteilt werden, um den Gesamtprozess besser überblicken zu können, wobei die Struktur der Teilprozesse vorerst als statisch angenommen wird.

Ein Teilprozess sollte logisch zusammengehörige Funktionalität kapseln, um möglichst unabhängig arbeiten zu können und wird folgend als *Game-Objekt* bezeichnet. Dadurch müssen jedoch die Benutzereingaben, statt an einen Gesamtprozess, an alle Game-Objekte einzeln weitergegeben werden (mittels *broadcasting* oder *routing*). Damit die Teilprozesse garantiert voll-

ständig unabhängig voneinander sind, dürfen keine Kommunikation oder sonstige Abhängigkeiten zwischen den Game-Objekten existieren. Sollen Game-Objekte auf Ereignisse von außerhalb reagieren können, muss die Benachrichtigung darüber noch vor deren Verarbeitung mitübergeben werden, wobei sich die Ereignisquelle somit immer nur auf den vorherigen Game-State (und die aktuellen Benutzereingaben) beziehen kann. Diese Funktionalität wird folgend als *Game-Logic* bezeichnet.

Eine Game-Engine mit Game-Logic und interaktiven Game-Objekten ohne Zeitverlauf ermöglicht rundenbasierte Computerspiele wie z. B. Brettspielsimulationen. Als Einschränkung sind dabei die Anzahl und Struktur der Game-Objekte immer noch gleichbleibend. Wird zusätzlich noch der Zeitverlauf in den Teilprozessen berücksichtigt, lassen sich bereits einfache animierte Computerspiele mit gleichbleibender Anzahl an Game-Objekten entwickeln, wie z. B. Pong.

Sollen zusätzlich neue Game-Objekte hinzugefügt oder bestehende entfernt werden können, müssen diese einem übergeordneten Verwaltungssystem unterliegen. Eine Game-Loop mit Game-Logic und interaktiven, zeitbeeinflussten Game-Objekten mit dynamischer Struktur ermöglicht grundsätzlich alle denkbaren Computerspiele.

## 4.2 Game-Objekte

Game-Objekte sind voneinander unabhängige Teilprozesse mit logisch zusammengehöriger Funktionalität, welche Eingabedaten entgegennehmen und Ausgabedaten produzieren, um dadurch insgesamt den neuen Game-State abzubilden. Folgend werden die verschiedenen Datentypen für die Eingabe- und Ausgabedaten definiert, wobei diese je nach verwendetem Subsystem und konkreten Anforderungen angepasst werden müssen.

### 4.2.1 Eingabedaten

Die nicht-deterministischen Eingabedaten des verwendeten Subsystem *SDL* sind einerseits vom Typ *SDL.Event* und umfassen sowohl Tastatur-, Maus- und Joystickeingaben als auch Änderungen in Fensterfokus und -größe. Andererseits stehen Ressourcen als Typ *SDL.Surface* zur Verfügung, diese werden zum leichteren Verständnis erst in Abschnitt 5.2.2 behandelt. Folgendes Codebeispiel zeigt eine mögliche Definition der **nicht-deterministischen Eingaben**, wobei nur die Benutzereingaben verwendet werden:

```
1 data Input = [SDL.Event]
```

Damit die Game-Objekte zusätzlich auf Ereignisse der Game-Logic reagieren können, müssen die Eingabedaten um zusätzliche Ereignisse erweitert werden. Die erweiterten Eingabedaten basieren nur auf dem vorherigen

Game-State (und den bereits gesammelten Benutzereingaben), wodurch diese völlig deterministisch sind und somit in einer reinen Funktion, also außerhalb der *IO Monade*, berechnet werden können. In *Yampa* ist für deterministische Ereignisse der Typkonstruktor `Yampa.Event` vorgesehen, wodurch die konkreten Typen in einem Ereignistyp gekapselt werden und dadurch die verschiedenen Ereignisfunktionen von *Yampa* verwendet werden können. Ein Ereignis besitzt entweder den Wert `NoEvent` oder `Event t`, wobei `t` den Datentyp des Ereignisses definiert und die konkreten Ereignisdaten enthält.

Eine zusätzliche Erweiterung wären Nachrichten [Che05]. Im Unterschied zu Ereignissen werden diese durch andere Game-Objekte erzeugt, nicht durch die Game-Logic, und müssen somit den ursprünglichen Sender beinhalten, womit eine Identifikation der verschiedenen Game-Objekte nötig ist.

Die deterministischen und nicht-deterministischen Eingabedaten werden letztendlich zu einem gemeinsamen Typ zusammengefasst und an die Game-Objekte verteilt. Folgendes Codebeispiel zeigt eine mögliche Definition der **erweiterten Eingabe**, wobei als deterministisches Ereignis nur eine Kollisionsnachricht, ohne zusätzliche Informationen, verwendet wird.

```
1 data ObjInput = ObjInput { ndInput :: Input
2                           , collisionEvt :: Yampa.Event () }
```

Dabei ist zu beachten, dass *Haskell* grundsätzlich statisch typisiert ist, weshalb der gesamte Eingabetyp vor der Programmübersetzung definiert sein muss und nicht plötzlich zur Laufzeit zusätzliche unbekannte Eigenschaften annehmen kann. Zum Beispiel ist für Kollisionsereignisse ein eigenes Feld vorgesehen und müsste für zusätzliche Benachrichtigungen ebenfalls um ein zusätzliches Feld erweitert werden.

#### 4.2.2 Ausgabedaten

Die Ausgabedaten eines Game-Objekts sind die einzige Möglichkeit nach außen zu kommunizieren und umfassen den beobachtbaren Ausgabezustand und die Anträge an das Verwaltungssystem und mögliche Nachrichten an andere Game-Objekte. Der beobachtbare Ausgabezustand kann je nach Flexibilität oder Komplexität des Computerspiels nur eine einfache Position sein, wobei dann die konkrete Darstellung im Rendering bestimmt wird, oder es müssen zusätzliche Darstellungsinformationen mitübergeben werden, wodurch die geometrische Form oder komplexere Darstellungen mittels externer Ressourcen vom Game-Objekt selbst bestimmt werden können. Folgendes Codebeispiel zeigt eine mögliche Definition des **beobachtbaren Ausgabezustands**:

```
1 data ObservableState
2   = Geometry { position :: Position2
3               , shape    :: Shape }
4   | Image { position :: Position2
5             , resourceId :: FilePath }
```

Game-Objekte können durch Anträge an das Verwaltungssystem hinzugefügt oder entfernt werden. Die Löschanträge können sich grundsätzlich auf das Game-Objekt selbst beziehen oder die Identitäten von anderen Game-Objekten beinhalten. Letztendlich entscheidet jedoch das übergeordnete Verwaltungssystem selbst darüber, ob die Anträge ausgeführt werden oder nicht. Folgendes Codebeispiel zeigt eine mögliche Definition für den **gesamten Ausgabetyp**:

```
1 data ObjOutput = ObjOutput
2   { state          :: ObservableState
3   , removeRequest :: Yampa.Event ()
4   , createRequests :: Yampa.Event [GameObject] }
```

Wenn ebenfalls eine Kommunikation zwischen den Game-Objekten möglich sein soll, müssen die Ausgabedaten um Nachrichten erweitert werden und den gewünschten Empfänger angeben. Dabei ist zu beachten, dass einerseits dem Game-Objekt die Identität des Empfängers schon vorab bekannt sein muss oder entsprechende Suchabfragen für Game-Objekte möglich sein müssen. Andererseits dürfen die gesendeten Nachrichten erst im nächsten Verarbeitungsschritt zur Verfügung stehen, da ansonsten die Game-Objekte nicht rein von ihren Eingabedaten abhängig sein würden.

#### 4.2.3 Implementierung

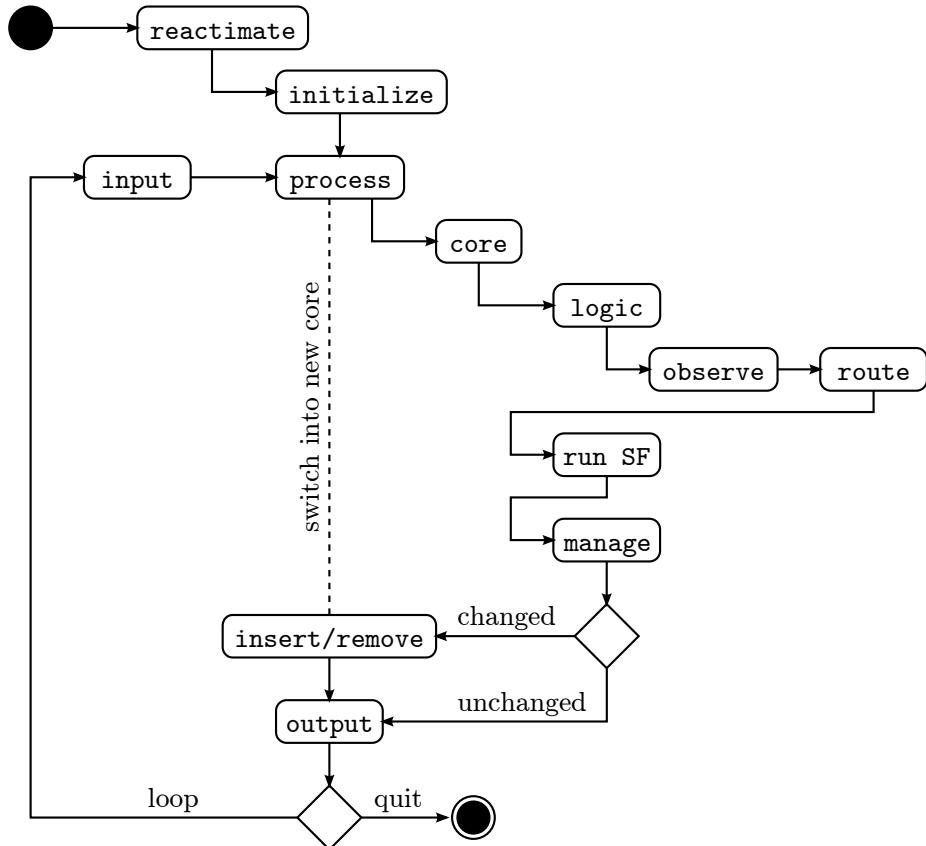
Letztendlich ist ein Game-Objekt eine Signalfunktion und führt eine Verarbeitung von Eingabe- zu Ausgabedaten durch. Dadurch stellt ein Game-Objekt wiederum ein kleines reaktives System dar und unterliegt ebenfalls dem Zeitverlauf durch *Yampa*. Die konkrete Verarbeitung stellt die eigentliche Funktionalität eines Computerspiels dar und wird in Abschnitt 5.2 näher behandelt. Der endgültige Game-Objekt-Typ wird daher folgendermaßen definiert:

```
1 type GameObject = SF ObjInput ObjOutput
```

Dynamische Datentypen konnten aus Zeitgründen nicht mehr implementiert werden, allerdings wird auf die grundsätzlichen Überlegungen in Abschnitt 6.1.2 eingegangen.

### 4.3 Game-Loop

Die Game-Loop stellt den übergeordneten Prozess eines Computerspiels dar und sammelt die nicht-deterministischen Eingabedaten (z. B. Zeit, Benutzereingaben, Ressourcen), übergibt diese der Verarbeitung, woraus der beobachtbare Game-State produziert wird, welcher letztendlich dargestellt wird. Die Verarbeitung beinhaltet die Game-Logic und das Verwaltungssystem der



**Abbildung 4.1:** Aktivitätsdiagramm einer Game-Loop mittels *Yampa reactivate*. Bei einer Änderung der Struktur von Signalfunktionen wird mittels *Continuations* in einen neuen `core` gewechselt, hier durch eine strichlierte Linie angedeutet.

Game-Objekte. Damit die Game-Logic Aussagen über den letzten Game-State treffen kann, muss dieser entsprechend rückgeführt werden und anschließend die erweiterten Eingabedaten an die Game-Objekte weitergeleitet werden. Nach der Verarbeitung muss die Struktur der Game-Objekte, basierend auf deren Anträgen, angepasst werden. Die Rückführung des Game-State muss dabei von einem übergeordneten Prozess durchgeführt werden, weshalb die eigentliche Verarbeitung nochmals in einen eigenen Kern gekapselt wird. Die konkrete Funktionalität der Verarbeitung ist dabei vor der Game-Loop verborgen. Abbildung 4.1 zeigt ein Aktivitätendiagramm der nachfolgenden Implementierung einer Game-Loop.

### 4.3.1 reactimate

Die Basis einer Simulation in *Yampa* bildet die Funktion `reactimate` und entspricht genau einem Eingabe-Verarbeitung-Ausgabe-Prozess. Das folgende Codebeispiel zeigt den grundsätzlichen Aufbau einer Game-Loop mit `reactimate`<sup>1</sup> und dem Subsystem *SDL*:

```

1 main :: IO ()
2 main = do
3     reactimate initialize input output process
4
5 initialize :: IO Input
6 initialize = do
7     SDL.init
8     ...
9     events ← getEvents
10    return events
11
12 input :: IO (DTime, Maybe Input)
13 input = do
14     ...
15     events ← getEvents
16     dtime ← getElapsedSeconds
17     return (dtime, Just events)
18
19 output :: [ObjOutput] → IO Bool
20 output states = do
21     screen ← SDL.getVideoSurface
22     SDL.fillRect screen Nothing Color.black
23     render states screen
24     SDL.flip screen
25     return False
26
27 output [] = SDL.quit >> return True
28
29 process :: SF Input [ObjOutput]
30 process = ...

```

In der Prozedur `initialize` wird zuerst das externe Subsystem *SDL* initialisiert, wodurch ein grafisches Fenster angezeigt wird und die ersten nicht-deterministischen Eingabedaten gesammelt werden. Anschließend wird durch `reactimate` die Signalfunktion `process` vorerst nur initialisiert, weshalb noch kein Zeitverlauf übergeben werden muss (siehe Abschnitt 3.5.5). Die Signalfunktion `process` wird später alle Game-Objekte beinhalten, welche den Game-State produzieren. Die Prozedur `output` führt ein Rendering des aktuellen Game-State durch, ist aber mit einem *Guard* überladen, welcher überprüft, ob noch ein Ausgabezustand erzeugt wurde. Wenn es sich um

---

<sup>1</sup>Die Funktionssignaturen wurden zum besseren Verständnis etwas vereinfacht.

eine leere Liste handelt, wird der Prozess beendet, da keine Funktionalität mehr existiert und dadurch auch keine neuen Game-Objekte mehr erzeugt werden können. Im nächsten Schleifendurchlauf wird in `input`, zusätzlich zu den normalen Eingaben, auch der Zeitverlauf an `reactimate` übergeben, wodurch anschließend der kontinuierliche Zeitverlauf simuliert werden kann. Dieser Prozess wird anschließend durch `reactimate` in einer Schleife wiederholt.

Die Funktionssignaturen der übergebenen Funktionen an `reactimate` können mittels *Sections* beliebig erweitert werden, ohne `reactimate` anpassen zu müssen. Zum Beispiel kann `process` von `SF Input [ObjOutput]` zu `[GameObject] → SF Input [ObjOutput]` erweitert werden und die Liste an Game-Objekten beim Aufruf von `reactimate` mittels (`process objs`) übergeben werden. Mit *Sections* wird dadurch die Funktion bereits teilweise angewendet und für `reactimate` ist wieder nur die ursprüngliche Funktionssignatur `SF Input [ObjOutput]` sichtbar, wodurch die – von *Yampa* vorgegebene – Schnittstelle erfüllt bleibt. Weiters ist zu beachten, dass kein deterministischer Datenfluss von der Ausgabe zur Eingabe stattfindet. Da sich diese Bereiche aber in einer *IO Monade* befinden, könnte diesen ebenfalls eine gemeinsame Referenz (`IORRef`) übergeben werden, wodurch eine (unreine) Kommunikation möglich wäre. Dies wird für den Ressource-Manager in Abschnitt 5.2.2 genutzt.

Soweit ist der Aufbau sehr eingängig, wobei die eigentliche Funktionalität in der Signalfunktion `process` stattfindet und eine Liste von parallelen und unabhängigen Teilprozessen (Game-Objekte) übernimmt.

### 4.3.2 pSwitchB

Bei `process` handelt es sich nur um eine einzelne Signalfunktion, allerdings stellt *Yampa* die Signalfunktionen `parB` bzw. `par` für parallele Signalfunktionen mit gleichbleibender Struktur zur Verfügung. In Computerspielen sollen jedoch neue Game-Objekte erzeugt und bestehende entfernt werden können, weshalb *Yampa* zusätzlich die Signalfunktionen `pSwitchB`, `pSwitch` und `dpSwitch`<sup>2</sup> für **parallele Signalfunktionen mit veränderlicher Struktur** zur Verfügung. Bei einer Strukturänderung wird dabei jedes Mal in eine neue Signalfunktion mit entsprechend veränderter Struktur gewechselt. Die Signalfunktion `pSwitchB` ist in Abbildung 4.2 illustriert und besitzt folgende Funktionssignatur:

```

1 pSwitchB :: Functor col
2      ⇒ col (SF in out)
3      ⇒ SF (in, col out) (Event mng)
4      ⇒ (col (SF in out) → mng → SF in (col out))
5      ⇒ SF in (col out)

```

---

<sup>2</sup>Für weitere parallele Switches siehe Abschnitt 3.5.4

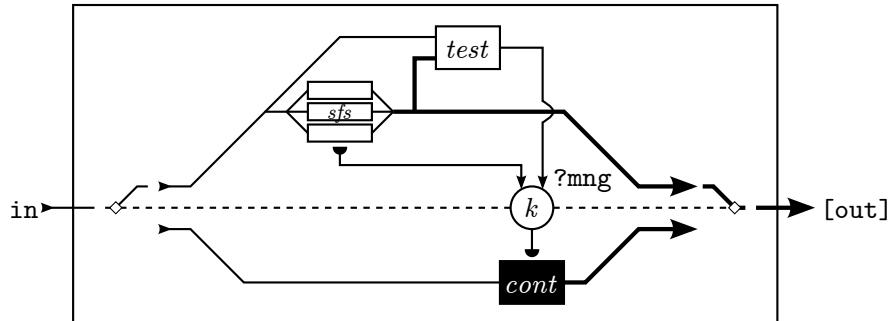
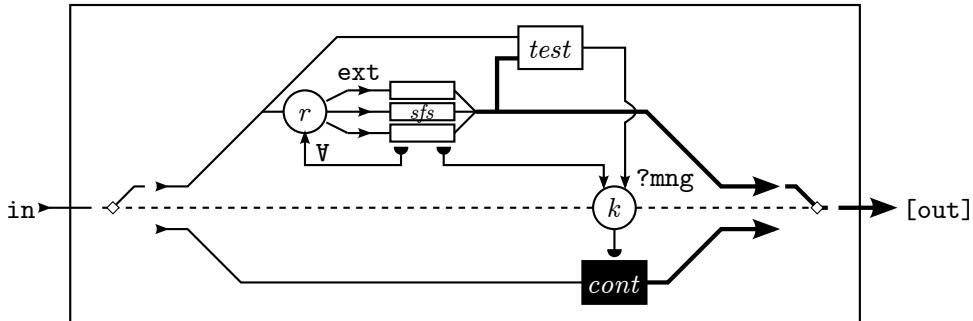


Abbildung 4.2: Diagramm der Signalfunktion pSwitchB.

Die Definition von `pSwitchB` lässt sich lesen als: Erzeuge eine Signalfunktion vom Typ `SF in (col out)`, welche immer, auch nach einem Wechsel, einen Eingabestrom `in` entgegennimmt und eine Liste (*Funktör*) an Ausgabeströmen `col out` erzeugt. Die Liste an Signalfunktionen wird durch `col (SF in out)` bestimmt. Nach der Berechnung der Signalfunktionen nimmt eine Test-Signalfunktion `SF (in, col out)` (`Event mng`) den ursprünglichen Eingabestrom `in` und die Ausgabeströme `col out` entgegen und entscheidet basierend darauf, ob die Struktur an Signalfunktionen geändert werden muss. Wenn keine Änderungsanträge (*manage requests*) gestellt werden, wird das leere Ereignis `NoEvent` erzeugt, ansonsten wird ein Ereignis `Event mng` erzeugt. Wenn ein Ereignis auftritt, reagiert der Switch darauf, indem die im Ereignis enthaltenen Änderungsanträge `mng`, zusammen mit der ursprünglichen Struktur an Signalfunktionen `col (SF in out)` an eine Verwaltungsfunktion übergeben werden. Diese verarbeitet die Liste den Änderungsanträgen entsprechend und wechselt in eine neue Signalfunktion, welche wiederum den ursprünglichen Gesamttyp `SF in (col out)` besitzen muss. Die neue Signalfunktion kann grundsätzlich jeglicher Art sein, oft wird aber wieder ein paralleler Switch verwendet, um danach wieder ein veränderliche Struktur von Signalfunktionen mit ähnlichem Verhalten annehmen zu können.

### 4.3.3 pSwitch

Ein Problem bei `pSwitchB` ist aber, dass sämtliche Eingabedaten mittels *broadcasting* an alle Signalfunktionen weitergeleitet werden. Dadurch wird einerseits mehr Arbeit als nötig verrichtet, andererseits stehen die Daten anschließend dort zur Verfügung, wo diese nicht benötigt werden und könnten versehentlich in fehlerhaften Ausdrücken verwendet werden (z.B. die Verarbeitung einer Nachricht, welche an ein anderes Game-Objekt gerichtet war). Deshalb stellt *Yampa* zusätzlich die parallelen Switches `pSwitch` und `dpSwitch` bereit, bei denen die Datenweiterleitung durch eine eigene



**Abbildung 4.3:** Diagramm der Signalfunktion pSwitch.

Verteilungsfunktion (*routing*) definiert werden kann. Die Funktionssignatur ist ähnlich zu `pSwitchB`, sieht jedoch einen zusätzlichen Parameter für die Verteilungsfunktion vor. Die Signalfunktion `pSwitch` ist in Abbildung 4.3 illustriert und besitzt folgende Funktionssignatur:

```

1 pSwitch :: Functor col
2      ⇒ (forall sf. (in → col sf → col (ext, sf)))
3      → col (SF ext out)
4      → SF (in, col out) (Event mng)
5      → (col (SF ext out) → mng → SF in (col out))
6      → SF in (col out)

```

Der Eingabestrom `in` kann dabei um zusätzliche Eingabedaten `ext` erweitert werden und wird anschließend gezielt an die Liste von Signalfunktionen `col` (`SF ext out`) verteilt, welche den erweiterten Eingabetyp entgegennehmen. In der Verteilerfunktion werden die konkreten Typen der Signalfunktionen mittels  $\forall$  `sf` (*universelle Quantifikation*) versteckt, wodurch die einzige mögliche Operation ist, diese mit den erweiterten Eingabedaten `ext` zu einem Tupel (`ext, sf`) zu paaren. Dadurch soll verhindert werden, dass die konkreten Signalfunktionen versehentlich vorab verwendet werden. `pSwitch` löst anschließend diese Paare wieder auf, indem die erweiterten Eingabedaten auf die gepaarten Signalfunktionen angewendet werden.

#### 4.3.4 Implementierung

Die Definition der parallelen Switches sieht dabei allerdings keine Rückführung der Ausgabedaten vor. Dies ist jedoch nötig, damit in der Game-Logic Aussagen über den Game-State aus dem vorherigen Verarbeitungsschritt getroffen werden können und die nicht-deterministischen Eingabedaten um die deterministischen erweitert werden. In folgendem Codebeispiel wird deshalb die parallele Verarbeitung der Game-Objekte nochmals in eine eigene Funktion `core` gepackt und der vorherige Game-State mittels `rec` rückgeführt:

```

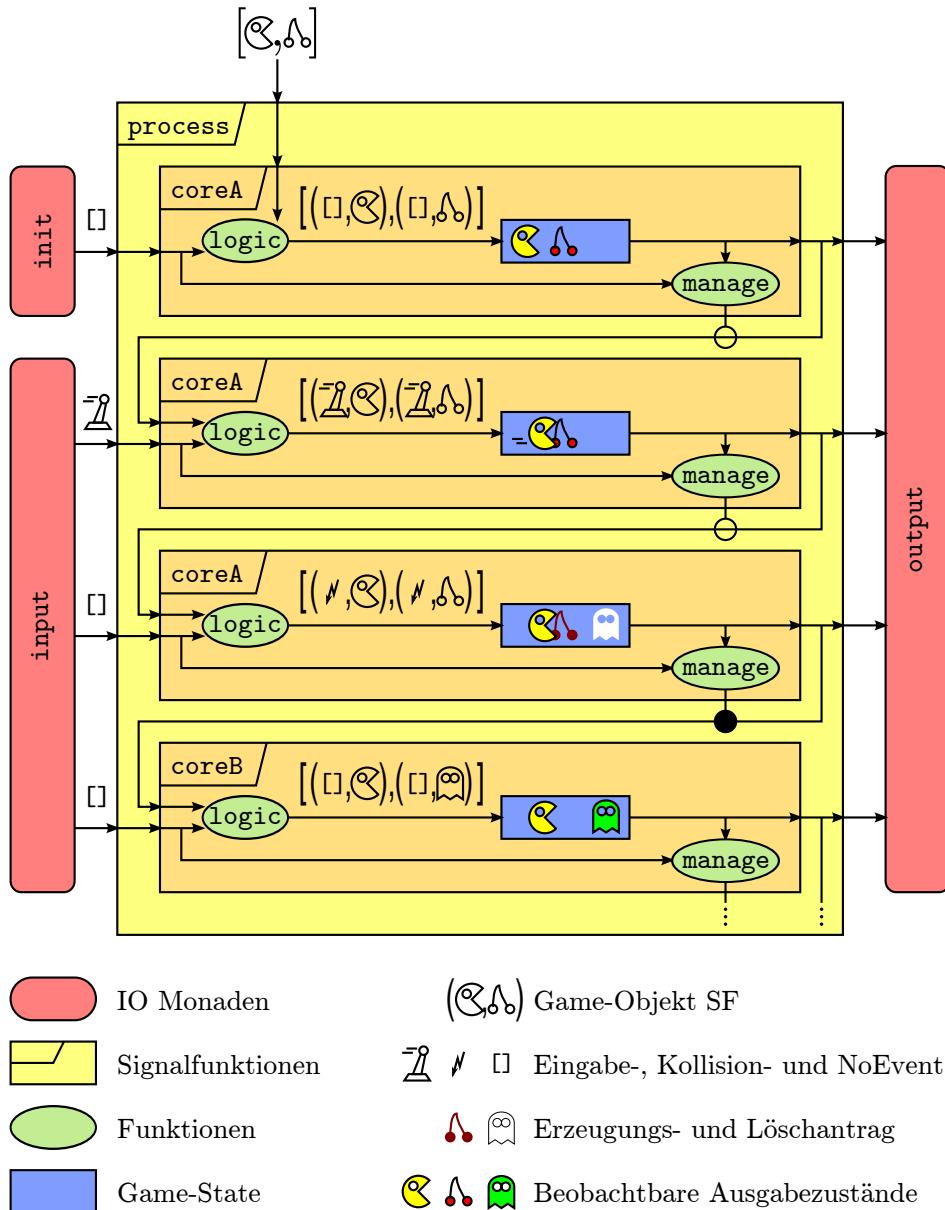
1 process :: [GameObject] → SF Input [ObjOutput]
2 process objs0 = proc input → do
3     rec
4         states ← core objs0 ← (input, states)
5     returnA ← states
6
7 core :: [GameObject] → SF (Input, [ObjOutput]) [ObjOutput]
8 core objs = dpSwitch logic
9             objs
10            (arr manage >>> notYet)
11            (λ objs' mng → core (mng objs'))

```

Die Funktionssignatur von `core` nimmt dabei sowohl die nicht-deterministischen Eingabedaten `Input`, als auch den Game-State `[ObjOutput]` entgegen. Daraus werden später in der Funktion `logic` die erweiterten Eingabedaten `ObjInput` erzeugt. Weiters ist zu beachten, dass `states` vor dem ersten Verarbeitungsschritt undefiniert ist und in der Funktion `logic` darauf noch kein Zugriff erfolgen darf.

Aufgrund der hohen Komplexität und den vielen Konzepten wird folgend der Ablauf nochmals mithilfe eines Datenflussdiagramms in Abbildung 4.4 am Beispiel eines einfachen Computerspiels erläutert. Die virtuelle Welt besteht dabei aus dem Spielercharakter Pac-Man und einer Kirsche, wobei die Kirsche bei Berührung zerstört wird und ein Geist erscheint.

1. In der Prozedur `init` werden die ersten nicht-deterministischen Eingaben gesammelt und in der Funktion `logic` zusammen mit den ersten Game-Objekten (Pac-Man und Kirsche) verknüpft. Da keine Ereignisse [] auftraten, führen die Game-Objekte einfach deren Funktionalität aus und produzieren den beobachtbaren Ausgabezustand, welcher in der Prozedur `output` angezeigt wird.
2. Der Ausgabezustand wird ebenfalls in den `core` rückgeführt. Da keine Änderungsanträge an die Funktion `manage` gestellt wurden, bleibt auch die Struktur der Game-Objekte gleich.
3. In `input` werden wieder die nicht-deterministischen Eingaben gesammelt, diesmal wird jedoch eine Benutzereingabe getätigt. Diese wird an alle Game-Objekte verteilt, wobei Pac-Man darauf mit einer Bewegung reagiert. Da die Game-Objekte nichts voneinander wissen, ist in diesem Schritt auch noch nichts über die bestehende Kollision bekannt und die Struktur der Game-Objekte bleibt weiterhin gleich.
4. Der Ausgabezustand wird wieder an `core` rückgeführt, welche diesen an `logic` weiterleitet, wo die bestehende Kollision entdeckt und die Game-Objekte entsprechend mit einem Ereignis benachrichtigt werden.
5. Die Kirsche verknüpft dieses Ereignis mit einem Löschantrag für sich selbst und einen Erzeugungsantrag für einen Geist. Da dadurch die Struktur an Game-Objekten geändert werden muss, erzeugt die Funk-



**Abbildung 4.4:** Datenflussdiagramm einer Game-Loop am Beispiel eines einfachen Computerspiels. Pac-Man wird durch eine Benutzereingabe auf die Kirsche bewegt, wodurch ein Geist erscheint.

tion `manage` ein Ereignis und der `dpSwitch` in `core` reagiert darauf. Daraus wird ein neuer `core` mit geänderter Struktur erzeugt.

6. Im nächsten Schritt treten wiederum keine Ereignisse auf, wodurch die Game-Objekte nur angezeigt werden.

## 4.4 Game-Logic

Die Game-Logic beobachtet aus einem globalen Standpunkt den Game-State und kann dadurch Aussagen über Game-Objekte und deren Beziehungen untereinander treffen. Daraus werden deterministische Ereignisse erzeugt und an die entsprechenden Game-Objekte verteilt. Bisher können aber die Game-Objekte weder voneinander unterschieden noch ein Ausgabezustand wieder einem bestimmten Game-Objekt zugeordnet werden. Daher werden zusätzlich Identitäten für Game-Objekte eingeführt und diese einer übergeordnete Verwaltung unterstellt. Die Definition der Game-Objekte bleibt dadurch unberührt. Anschließend wird die Verteilung durchgeführt, wobei darin die Beobachtung des Game-States und die Erweiterung der Eingabedaten stattfindet.

### 4.4.1 Identitäten

Bisher wurden für die Verwaltung von Game-Objekten nur einfache Listen verwendet, wodurch die Game-Objekte grundsätzlich anonym sind. Damit jedoch die Ausgabezustände in der Game-Logic wieder den einzelnen Game-Objekten zugeordnet und die erweiterten Eingabedaten entsprechend weitergeleitet werden können, müssen diese über Identitäten verfügen. Zusätzlich können dadurch Game-Objekte miteinander über Nachrichten kommunizieren.

Die einfachste Form einer Identität ist eine fortlaufende Nummer, alternativ können aber auch vordefinierte Namen oder kollisionsfreie Zufallszahlen (GUID) verwendet werden. Bei der Implementierung von Identitäten ist zu beachten, dass diese über eine Änderung der Struktur hinweg erhalten bleiben müssen. In dieser Implementierung wurde die `IdentityList` vom *Space Invaders* Beispiel verwendet, welche verschiedene Listenoperationen zur Verfügung stellt und beim Einfügen von neuen Elementen automatisch fortlaufende Nummern erzeugt [CNP03]. Dadurch bleibt der Datentyp der Game-Objekte gleich und diese werden nur in eine übergeordnete `IdentityList` eingefügt. Allerdings müssen die Funktionssignaturen der Game-Loop entsprechend angepasst werden:

```
1 output :: IL ObjOutput → IO Bool
2 process :: IL GameObject → SF Input (IL ObjOutput)
3 core     :: IL GameObject → SF (Input, IL ObjOutput) (IL ObjOutput)
```

Fortgeschrittene Formen von Identitäten könnten die Game-Objekte zusätzlich mit Informationen über verschiedene Eigenschaften ausstatten (z. B. Name, Gruppe, Typ), wodurch die Game-Objekte mittels Suchanfragen ausgewählt werden können (z. B. mittels LINQ). Dadurch kann die Game-Logic beispielsweise alle Game-Objekte des Typ „Spielercharakter“ auswählen und nur diesen die Benutzereingaben weiterleiten und ein Game-Objekt kann an alle der Gruppe „Gegner“ eine Nachricht schicken.

#### 4.4.2 Verwaltung

In der Verwaltungsfunktion `manage`<sup>3</sup> sollen die Änderungsanträge der Game-Objekte, welche in `ObjOutput` als `removeRequest` und `createRequests` gespeichert sind, auf die aktuelle Struktur von Game-Objekten angewendet werden. Als Vorgabe von `pSwitch` muss die Funktion `manage` die Funktionssignatur  $SF (in, col out) \rightarrow Event mng$  besitzen, wodurch die Eingabedaten von `process`, d. h. die nicht-deterministischen Eingaben `Input` und der vorheriger Game-State `IL ObjOutput`, und zusätzlich der aktuelle Game-State zur Verfügung stehen. Daraus soll entschieden werden, ob die Struktur der Game-Objekte geändert werden soll und ein entsprechendes Ereignis ausgelöst werden muss. Wenn ein Ereignis ausgelöst wird, wird im Switch die Änderungsfunktion  $(col (SF ext out) \rightarrow mng \rightarrow SF in (col out))$  ausgeführt, welche im `core` als  $(\lambda objs' mng \rightarrow core (mng objs'))$  implementiert ist. Der enthaltene Wert im Ereignis Event `mng` muss daher eine Funktion sein, welche die aktuelle Liste von Game-Objekten `objs'` entgegennimmt und daraus, bei Anwendung der Funktion `mng`, eine veränderte Liste für den neuen `core` erzeugt:

```

1 manage :: ((Input, IL ObjOutput), IL ObjOutput)
2           → Yampa.Event (IL GameObject → IL GameObject)
3 manage ((input, _), states) =
4   if pressed Quit input
5     then Event (λ _ → emptyIL)
6     else foldl (mergeBy (∘)) noEvent events
7   where
8     events :: [Yampa.Event (IL GameObject → IL GameObject)]
9     events = [ mergeBy (∘)
10               (removeRequest state `tag` (deleteIL objId))
11               (fmap (foldl (∘) id ∘ map insertIL_)
12                (createRequests state))
13               | (objId, state) ← assocsIL states ]

```

Der Code sieht etwas komplex aus, erzeugt aber im Grunde nur eine lange Funktionskomposition von teilweise angewendeten `deleteIL` und `insertIL_` Funktionen und fügt diese zu einem Gesamtereignis zusammen. Der einfachste Fall ist dabei, dass der Quit Button gedrückt wurde und die Game-Loop durch eine leere Liste beendet wird. Ansonsten wird der gesamte Game-State durchlaufen und von jedem Game-Objekt die Änderungsanträge zu einem einzelnen Ereignis vereint und diese einzelnen Ereignisse wiederum zu einem großen Gesamtereignis vereint. Dadurch wird ein Ereignis nur dann ausgelöst, wenn auch ein Änderungsantrag gestellt wurde, sei es aber auch nur von einem Game-Objekt. Die Änderungsanträge werden dabei in Einfüge- oder Löschoperationen mit entsprechenden Identitäten umgewandelt und zu einer großen Funktionskomposition zusammengefügt, welche letztendlich auf die Liste von Game-Objekten angewendet wird.

---

<sup>3</sup>Entspricht der Funktion `killAndSpawn` des *Space Invaders* Beispiel [CNP03].

#### 4.4.3 Verteilung

Die Verteilungsfunktionalität ist Teil der Funktion `logic`<sup>4</sup> und ruft die verschiedenen Beobachtungsfunktionen auf, wodurch die erweiterten Eingabedaten erzeugt werden und anschließend verteilt werden. Als Vorgabe von `pSwitch` muss die Funktion `logic` die Funktionssignatur (`in → col sf → col (ext, sf)`) besitzen, wodurch die Eingabedaten von `process`, d. h. die nicht-deterministischen Eingaben `Input` und der vorheriger Game-State `IL ObjOutput`, zur Verfügung stehen. Diese sollen zu erweiterten Eingaben umgewandelt werden und mit den Game-Objekten gepaart werden:

```

1 logic :: (Input, IL ObjOutput) → IL sf → IL (ObjInput, sf)
2 logic (input, states) objs = mapIL route objs
3 where
4   route (objId, obj) =
5     (ObjInput { ndInput      = input
6                , collisionEvt = if objId ∈ collidingObjs
7                               then Yampa.Event ()
8                               else Yampa.NoEvent
9                }
10               , obj)
11   collidingObjs =
12     observeCollisions ∘ assocsIL $ fmap state states

```

Es wird die gesamte Liste an Signalfunktionen durchlaufen und zusammen mit den erweiterten Eingabedaten gepaart, wobei die Benutzereingaben an alle Game-Objekte mitübergeben werden. Diese können jedoch mittels der Identitäten entsprechend unterschieden werden, um sie nur an Spieler-Objekte zu übertragen oder sogar eine Unterscheidung zwischen den Spielern zu treffen. Das Kollisionsereignis wird ebenfalls für jedes Game-Objekt einzeln bestimmt. Es wird zuerst eine Liste von Kollisionen erzeugt und anschließend überprüft, ob das aktuelle Game-Objekt darin enthalten ist und dementsprechend ein Ereignis erzeugt oder nicht. Zum leichteren Verständnis sind jedoch in `collisionEvt` keine weiteren Informationen enthalten, wie z. B. beteiligte Objekte oder genaue Kollisionspositionen.

Eine sehr wichtige Konsequenz aus der funktionalen Programmierung ist dabei, dass in den Beobachtungsfunktionen und den Game-Objekten nur jene Daten zur Verfügung stehen, welche vorher übergeben wurden! Die Verteilungsfunktion kann dadurch sehr genau steuern, welche Daten weitergereicht werden und auch entsprechend Daten vorenthalten.

#### 4.4.4 Beobachtungen

Die Beobachtungsfunktionen sind ebenfalls Teil der Funktion `logic` aus dem vorherigen Abschnitt und beobachten die übergebenen Eingabedaten aus der Verteilungsfunktion, um daraus Aussagen über die Beziehungen zwischen

---

<sup>4</sup>Entspricht der Funktion `route` im *Space Invaders* Beispiel [CNP03].

Game-Objekten zu treffen. In der Funktion `logic` wurden nur die beobachtbaren Ausgabezustände übergeben, wodurch z. B. Kollisionen aufgrund der aktuellen Position und geometrischen Form beobachtet werden können. Zum leichten Verständnis überprüft das folgende Codebeispiel jedoch nur auf gleiche Position, nicht auf Kollisionen. Es können jedoch grundsätzlich alle Arten von Beobachtungen implementiert werden, wenn die entsprechenden Daten übergeben werden. Im Computerspiel *Frag* wurden weitere Beobachtungen hinzugefügt, wie z. B. Berührungen mit Boden, genaue Kollisionspositionen oder sichtbare Objekte für Computergegner [Che05].

```

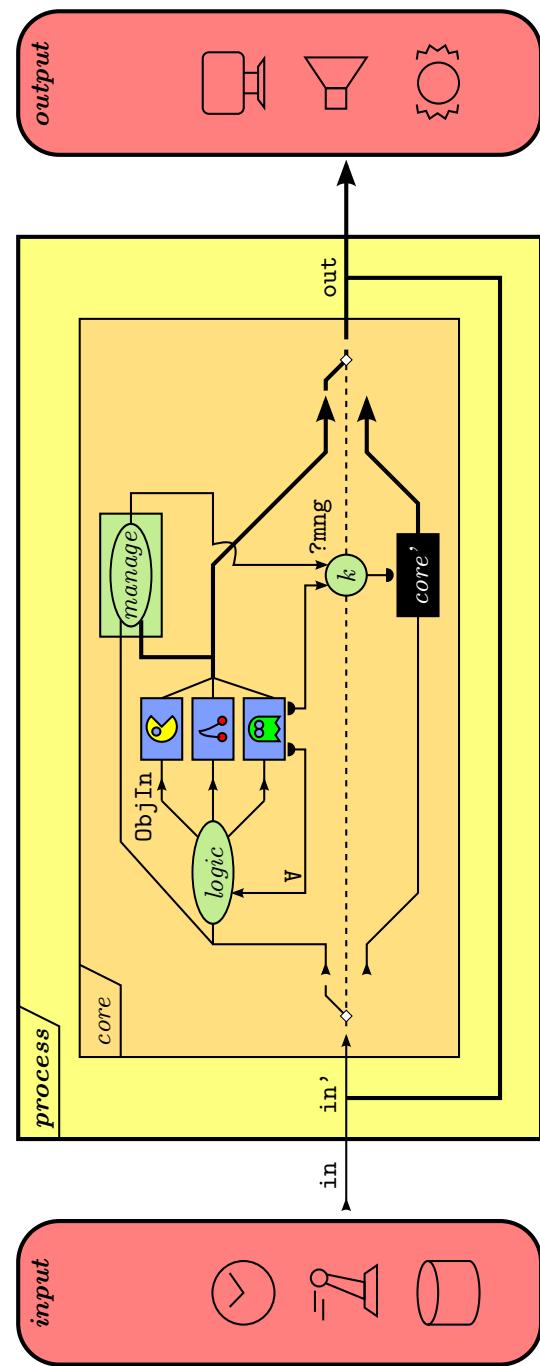
1 observeCollisions :: [(ILKey, ObservableState)] → [ILKey]
2 observeCollisions idStates = concat (collisions idStates)
3   where
4     collisions [] = []
5     collisions ((objId, state):idStates') =
6       [ [objId, objId']
7         | (objId', state') ← idStates', state `collide` state' ]
8       + collisions idStates'
9
10    collide :: ObservableState → ObservableState → Bool
11    (Geometry p1 _) `collide` (Geometry p2 _) = p1 == p2

```

In `observeCollisions` werden alle beobachtbaren Ausgabezustände rekursiv durchlaufen und auf gleiche Position überprüft. Kollidieren zwei Game-Objekte miteinander, werden beide in eine Kollisionsliste – als Liste mit zwei Elementen – hinzugefügt. Bei jedem rekursiven Durchlauf wird die Liste mit den noch-zu-überprüfenden Game-Objekten um ein Element verkürzt, wodurch alle Paare von Game-Objekten nur einmal überprüft werden. Abschließend wird die Kollisionsliste ausgeflacht, wodurch nur mehr eine einfache Liste aller kollidierender Game-Objekte übrig bleibt. Bei der Verteilung der Beobachtungen muss anschließend nur mehr überprüft werden, ob das entsprechende Game-Objekt in der Liste enthalten ist.

## 4.5 Zusammenfassung

In Abbildung 4.5 ist abschließend der gesamte Aufbau einer allgemeinen Game-Engine mit funktional-reaktiver Programmierung in *Yampa* zusammengefasst.



**Abbildung 4.5:** Diagramm an einer allgemeinen Game-Engine mittels funktional-reaktiver Programmierung in *Yampa*.

# Kapitel 5

## Implementierung der Basisfunktionalitäten

### 5.1 Einleitung

In diesem Kapitel wird die Implementierung der Basisfunktionalitäten eines Computerspiels aus dem Kapitel 2 beschrieben. Bei den Basisfunktionalität handelt es sich nur mehr oder weniger komplexe Signalfunktionen, welche anschließend in Game-Objekten verwendet werden können.

### 5.2 Eingabe

#### 5.2.1 Input Re-Mapping

Das Subsystem *SDL* ermöglicht die Zustände der üblichen Eingabegeräte über verschiedene Ereignisse abzufragen, welche folgend aufgelistet sind:

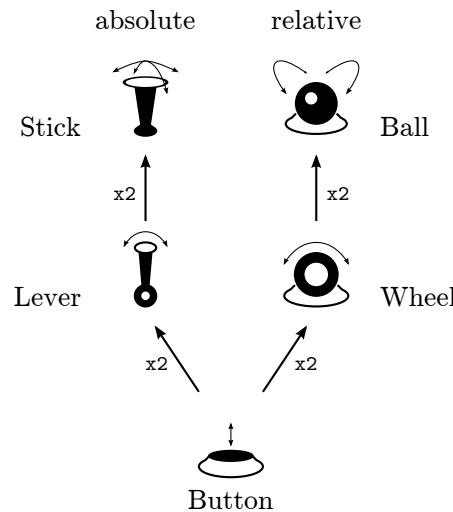
**Tastatur:** KeyDown und KeyUp.

**Maus:** MouseMotion, MouseButtonDown und MouseButtonUp.

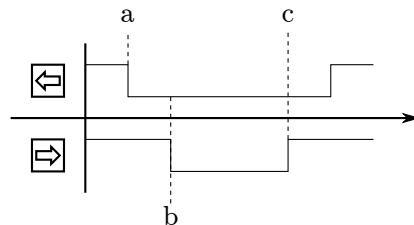
**Joystick:** JoyAxisMotion, JoyBallMotion, JoyHatMotion, JoyButtonDown und JoyButtonUp.

Diese sind jedoch vorerst miteinander inkompatibel, es wäre aber wünschenswert, sogenanntes *Input Re-Mapping* durchzuführen, um einerseits verschiedene Eingabegeräte zu einem logischen Eingabegerät zu vereinen, andererseits sollen höhere Eingabeelemente durch niedere ausgedrückt werden können, wie in Abbildung 5.1 dargestellt.

Die Schwierigkeit beim Input Re-Mapping ist allerdings, dass dadurch viele mehrdeutige Ereignisse auftreten können, welche entsprechend gefiltert oder aufbereitet werden müssen. Einerseits können zwei Buttons gleichzeitig gedrückt werden, wobei entweder bei unabhängigen Aktionen beide ausgeführt werden sollen (z. B. springen und schießen) oder bei ausschließenden



**Abbildung 5.1:** Input Re-Mapping ermöglicht höhere Eingabeelemente durch niedere auszudrücken.



**Abbildung 5.2:** Wenn der Button [Links] gedrückt bleibt (a) und dann [Rechts] gedrückt wird (b), kann der Spielercharakter entweder weiter nach links laufen oder plötzlich nach rechts. Wenn anschließend der Button [Rechts] wieder losgelassen wird (c), kann der Spielercharakter entweder stehen bleiben oder wieder nach links laufen.

Aktionen nur eine ausgeführt werden soll (z. B. beschleunigen und bremsen). Andererseits kann ein Lever aus zwei Buttons gebildet werden, wodurch die Buttons gegenläufige Bedeutungen annehmen (links gegen rechts). Diese Mehrdeutigkeiten sind in Abbildung 5.2 illustriert.

```

1 buttonsSDL :: [SDL.Events.Event] → [(SDL.SDLKey, Bool)]
2 buttonsSDL = ...
3
4 buttonLever :: Eq id ⇒ id → id → SF [(id, Bool)] Lever
5 buttonLever negBtn posBtn = proc inputs → do
6     let btn = getLatest [negBtn, posBtn] ∘ inputs
7     lever ← trackAndHold 0.0 ⤵ dir negBtn posBtn btn
8     returnA ⤵ lever

```

```

9   where
10    dir :: Eq id => id -> id -> (id, Bool) -> Maybe Lever
11    dir negBtn posBtn (btn, state) = ... -- -1.0 0.0 0.0 +1.0
12
13 object :: Position2 -> Velocity2 -> SF ObjInput ObjOutput
14 object p0 v0 = proc events -> do
15   let buttons = buttonsSDL events
16   horz -> buttonLever SDL.SDLK_LEFT SDL.SDLK_RIGHT -< buttons
17   vert -> buttonLever SDL.SDLK_DOWN SDL.SDLK_UP -< buttons
18   position -> mover p0 v0 -< vector2 horz vert
19   ...

```

In diesem Codebeispiel wird davon ausgegangen, dass das letzte KeyDown- und KeyUp-Ereignis die höchste Priorität hat und wird entsprechend für den Button-Lever herangezogen. Zuerst werden die Benutzereingaben aus *SDL* mittels `buttonsSDL` gefiltert und in eine einheitliche assoziative Liste [(*id*, *Bool*)] konvertiert. Die Signalfunktion `buttonLever` ist dadurch vom konkreten Subsystem unabhängig und benötigt als Eingabeparameter nur eine Identität des negativen und positiven Button, welche in den laufenden Benutzereingaben gesucht werden sollen. Mit der Funktion `dir` wird daraus die aktuelle Richtung bestimmt oder wenn kein Button gefunden wurde, mittels `trackAndHold` der alte Wert gewählt. In einem Game-Objekt kann daraus entsprechend der Button-Lever gebildet werden und mit einem weiteren Lever, zu einem Stick kombiniert werden, welcher wiederum als Beschleunigungsvektor für die Bewegungsfunktion verwendet werden kann.

### 5.2.2 Ressourcen

Sollen Game-Objekte zusätzlich auch externe Ressourcen entgegennehmen können, müssen diese entweder direkt in den nicht-deterministischen Eingabedaten, als bereits geladene Ressourcen, mitübergeben werden, wenn die Verarbeitung in der Game-Logic oder den Game-Objekten davon abhängig sind (z. B. externe Skripten), oder nur Verweise auf externe Ressourcen, wenn nur die Ausgabe davon abhängig ist (z. B. Bilddateien). Dabei ist zu beachten, dass Ressourcen nur in einer *IO Monade* geladen werden können, weshalb die Ressourcen bereits geladen sein müssen, wenn diese im Verarbeitungsschritt verwendet werden sollen.

Die einfachste Variante, Ressourcen einzusetzen, wäre, einmalig alle benötigten Ressourcen zu laden. Dies ist zwar für kleine Computerspiele eine ausreichende Lösung, bei größeren würden jedoch die Ladezeit und der Speicherverbrauch problematisch werden. Eine alternative Variante wäre, das Computerspiel in mehrere Abschnitte zu unterteilen (z. B. Levels) und nach jedem Abschnitt die Game-Loop `reactimate` zu beenden, die neuen Ressourcen zu laden und anschließend wieder neu zu starten. Dies hat jedoch den Nachteil, dass dadurch die Game-Engine auf entsprechende Computer Spiele eingeschränkt wird. Eine bessere und allgemein verwendbare Variante

ist, einen Ressourcen-Manager einzusetzen, welcher einerseits die Ressourcen lädt und verwaltet, andererseits den Game-Objekten erlaubt neue Ladenanträge zu stellen und diesen die Ressourcen entsprechend im nächsten Eingabeschritt mitübergibt. Da jedoch kein deterministischer Datenfluss von Ausgabe zur Eingabe möglich ist, müssen die Ladenanträge mit einer nicht-deterministischen Kommunikation übertragen werden.

```

1 data Resource = FileLn String | Image SDL.Surface
2 data Input    = Input     { ..., requested :: IL Resource }
3 data ObjInput = ObjInput { ..., resource :: Maybe Resource }
4 data ObjOutput = ObjOutput { ..., request   :: Maybe FilePath }
5
6 type ResourceManager = Map FilePath Resource
7 type RequestManager = IL FilePath

```

Zuerst werden die Eingabe- und Ausgabedaten für die Ressourcen entsprechend erweitert. Die nicht-deterministischen Eingaben `Input` werden um die angeforderten Ressourcen erweitert, wobei zwischen antragstellendem Game-Objekt und Ressource eine Identifikation `IL Resource` stattfinden muss, damit diese später in der Verteilerfunktion entsprechend weitergeleitet werden kann. Die erweiterten Eingabedaten der Game-Objekte `ObjInput` müssen entsprechend um die (möglicherweise) angeforderte Ressource und die Ausgabedaten `ObjOutput` um einen (möglichen) Ressourcenantrag erweitert werden. Die Anzahl der geladenen Ressourcen und Ressourcenanträge pro Game-Objekt wurde zum besseren Verständnis auf jeweils eins beschränkt. Das Subsystem *SDL* verwendet für Ressourcen den Typ `SDL.Surface` und speichert darin Rastergrafiken, während als Verweise einfache Dateipfade vom Standardtyp `FilePath` verwendet werden können.

Die Funktionalität des Ressourcen-Manager selbst wird auf zwei Mechanismen aufgeteilt. Einerseits die Verwaltung der Ressourcen selbst, welche die Ressourcen über einen `FilePath` identifiziert und die geladenen Ressourcen entsprechend im Hauptspeicher verwaltet, und andererseits eine Verwaltung für neue Ressourcenanträge, welche in der Prozedur `output` die Anträge im Game-State sammelt und über eine unreine Kommunikation mittels `IORef` an die Prozedur `input` übermittelt.

```

1 main = do
2   resMgr <- newIORef (empty::ResourceManager)
3   reqMgr <- newIORef (empty::RequestManager)
4   reactimate (initialize resMgr reqMgr)
5           (input      resMgr reqMgr)
6           (output      resMgr reqMgr)
7           (process objs0)
8
9 input :: IORef ResourceManager → IORef RequestManager → ...
10 input resMgrRef reqMgrRef _ = do
11   requested ← getRequested resMgrRef reqMgrRef
12   ...

```

```

13
14 output :: IORef ResourceManager → IORef RequestManager → ...
15 output resMgrRef reqMgrRef _ objsIL = do
16     let requests = fmap request $ objsIL
17     states      = elemsIL ∘ fmap state $ objsIL
18     resMgr ← loadRessources (elemsIL requests) resMgrRef
19     render screen resMgr states
20
21     writeIORef reqMgrRef requests
22     ...

```

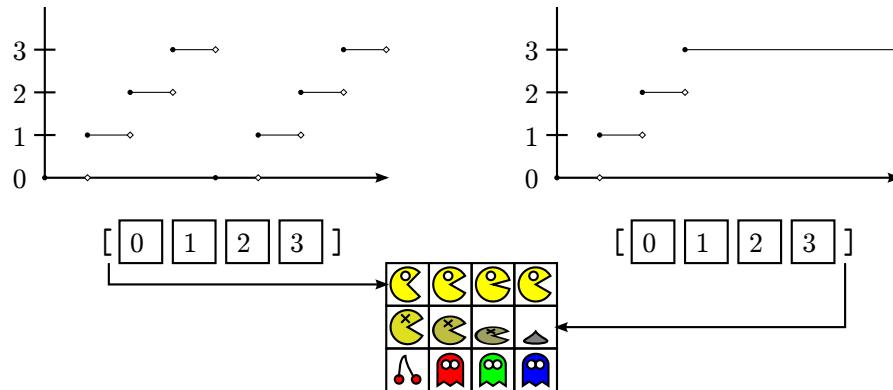
Die IO-Referenzen des Ressourcen-Manager müssen vor `reactimate` erstellt werden und an die einzelnen Eingabe-/Ausgabe-Bereiche mitübergeben werden und auch dementsprechend die Funktionssignaturen erweitert werden. In der Prozedur `input` werden die Anträge gelesen, die Ressourcen entsprechend aus der Ressourcen-Verwaltung geholt und den nicht-deterministischen Eingabedaten hinzugefügt, wo sie anschließend durch die Verteilerfunktion an die Game-Objekte weitergeleitet werden können. Die Game-Objekte können dadurch anschließend die Ressourcen in deren Funktionalität verwenden (z. B. Skripte), neue Anträge stellen oder einfach nur auf externe Ressourcen verweisen. Bei einem Verweis werden die Ressourcen erst im Rendering aus der Ressourcen-Verwaltung geladen (z. B. Bilder), wodurch nicht ständig die Ressourcen von Eingabe zur Ausgabe transportiert werden müssen. Neue Ressourcen werden in der Prozedur `output` geladen, wenn diese nicht bereits in der Ressourcen-Verwaltung enthalten sind und können anschließend sofort im Rendering verwendet werden.

Bei einer früheren Implementierung wurde ursprünglich `reactimate` erweitert und eine zusätzliche *IO Monade* `inpull` zwischen `input` und `process` eingeführt. Dadurch sollten die Anträge vor `input` verborgen werden und die Ressourcenanträge nur an `inpull` rückgeführt werden, um einen funktionalen Stil zu bewahren, aber die Ressourcen dennoch vor dem nächsten Verarbeitungsschritt laden zu können. Da aber in der Prozedur `output` ohnehin eine unreine Kommunikation möglich ist und in den Ausgabezuständen die Ressourcenanträge enthalten sind, wurde diese Implementierung später wieder verworfen.

## 5.3 Verarbeitung

### 5.3.1 Frameanimation

Eine Animation ist grundsätzlich eine Änderung der visuellen Darstellung über die Zeit, wobei die Änderung visuelle Eigenschaften umfassen kann wie Geometrie, Farbe, Position usw. In zweidimensionalen Computerspielen mit Rastergrafik sind die einfachste Form Frameanimationen, wobei der aktuelle Frame in Abhängigkeit der Zeit aus einer Reihe von möglichen Frames ausge-



**Abbildung 5.3:** Beide Frameanimationen verweisen nur auf eine Liste von Frames, welche erst in der Ausgabe mithilfe des entsprechenden Spritesheets angezeigt wird.

wählt wird. Dabei ist zwischen der kontinuierlichen Funktionalität und den eigentlichen Bilddaten im sogenannten *Spritesheet* zu unterscheiden. Eine Frameanimation stellt eine zeitabhängige Funktion dar, berechnet im Verarbeitungsschritt aber nur den Index des aktuellen Frames, während die eigentlichen Bilddaten erst in der Ausgabe ausgewählt und angezeigt werden. Eine Frameanimation spielt dabei entweder eine Reihe von Frames in einer Schleife ab (z. B. Laufanimation) oder stoppt am Ende (z. B. Sterbeanimation), wie in Abbildung 5.3 illustriert.

```

1 type Frame = SDL.Rect
2 data ObjOutput = ...
3   | FrameAnimation { position  :: Position2
4           , frame      :: Frame
5           , resourceId :: FilePath }
6
7 animationObj :: Time → [Frame] → ResourceId → GameObject
8 animationObj fps frames resId = proc input → do
9   frame ← animateLoop (length frames) `mod` fps
10 ...
11   returnA ← FrameAnimation position (frames!!frame) resId
12
13 framesByRow :: Int → Int → Int → Int → [Frame]
14 framesByRow frames offsetX offsetY width height =
15   [SDL.Rect (offsetX + frame * width) offsetY width height
16   | frame ← [0..frames]]
17
18 animateLoop :: Int → SF Time Int
19 animateLoop n = proc fps → do
20   t ← integral `mod` fps
21   returnA ← floor t `mod` n

```

In diesem Codebeispiel wird zuerst ein Game-Objekt mit Animationsfunktionalität definiert, bei dessen Erzeugung eine Liste von Frames `frames` übergeben werden muss, welche sich auf den entsprechenden Spritesheet `resId` beziehen. Die Frames können entweder händisch oder durch verschiedene Hilfsfunktionen erstellt werden. Beispielsweise erzeugt die Funktion `framesByRow` eine Liste von Frames, welche die Animation durch eine einfache Zeile im Spritesheet beschreiben. Eine erweiterte Funktion könnte eine Animation über mehrere Zeilen beschreiben. Die eigentliche Animation wird mit der Signalfunktion `animateLoop` berechnet und wählt das aktuelle Frame aus, welches aber erst im Ausgabezustand verwendet wird.

Es sind dabei verschiedene Mischformen (z. B. Laufanimation mit Anlauf) oder Erweiterungen möglich (z. B. Frameanimationen mit unterschiedlicher Framedauer). Im Projekt *DIVE* wurden beispielsweise dreidimensionale Modelle mit Skelettanimation implementiert, wobei zwischen zwei Animationen feine Übergänge mittels *Interpolation* möglich sind [Blo09].

### 5.3.2 Bewegung

Die einfachste Form einer Bewegung in der Physik ist die *gleichförmige Bewegung*, wobei ein Objekt mit gleichbleibender Geschwindigkeit durch einen reibungslosen Raum bewegt wird (z. B. ein Asteroid im Weltall). Eine allgemeinere Form stellt die *gleichförmig beschleunigte Bewegung* dar, wobei die Geschwindigkeit durch eine gleichbleibende Beschleunigung immer weiter erhöht wird (z. B. fallendes Objekt unter Schwerkraft). Die allgemeinste Form der Bewegung wird durch die *Newtonischen Bewegungsgesetze* beschrieben:

$$v(t) = v_0 + \int a \cdot dt \quad (5.1)$$

$$p(t) = p_0 + \int v \cdot dt \quad (5.2)$$

Mit der Signalfunktion `integral` lassen sich die Bewegungsgleichungen beinahe direkt in den Code übernehmen, wobei die Zeit durch *Yampa* versteckt wird. Mathematisch wird dabei zwischen absoluten Positionen (*Affiner Raum*) und Richtungen bzw. relativen Verschiebungen (*Vektorraum*) unterschieden. *Yampa* unterscheidet diese ebenfalls als `Point` und `Vector` und stellt entsprechend unterschiedliche Additionsoperatoren `(.+^)` und `(^+^)` zur Verfügung.

```

1 mover :: Point → Vector → SF Vector Point
2 mover p0 v0 = proc a → do
3     v ← (v0 ^+^) ≪ integral ↵ a
4     p ← (p0 .+^) ≪ integral ↵ v
5     returnA ↵ p

```

Die Definition von `mover` lässt sich lesen als: Erzeuge eine neue Signalfunktion `mover` mit einer fixen Startposition `p0` und einen fixen Startgeschwindigkeit `v0`, welche einen Beschleunigungsvektor `a` entgegennimmt und

daraus die aktuelle Position  $p$  berechnet. Die Position stellt dabei einen kontinuierlichen Wert dar, wodurch z. B. für ein Spieler-Objekt der Beschleunigungssvektor direkt mit einem kontinuierlichen Eingabeelement (z. B. Stick) verknüpft werden kann. Allerdings sind dadurch Sprünge in der Position nur über diskrete Ereignisse möglich und der Geschwindigkeitsvektor sollte eher als aufsummierter Verschiebungsvorschlag von der Ursprungsposition interpretiert werden. Im Gegensatz dazu wird in imperativen Programmiersprachen normalerweise direkt an der Position operiert, wodurch diese jederzeit unabhängig von der Geschwindigkeit gesetzt werden kann.

Die Geschwindigkeit ist in dieser Implementierung allerdings unbegrenzt. Im *Space Invaders* Beispiel wird diese beispielsweise durch vorzeitiges Deaktivieren der Beschleunigung begrenzt [CNP03]. Wird eine Physik-Engine mit Kräften und Massen verwendet, ergibt sich die Maximalgeschwindigkeit meist automatisch durch Reibung und Luftwiderstand, wobei diese aber nur sehr schwer genau eingestellt werden kann.

## 5.4 Ausgabe

### 5.4.1 Rendering

Das Rendering ist überraschenderweise relativ unspektakulär und hat kaum Einfluss auf die Architektur. Es muss nur die Funktion `render` auf jeden beobachtbaren Ausgabezustand eines Game-Objekts aufgerufen werden, welche entsprechende Funktionsüberladungen implementiert. Aufwendigere Rendering-Techniken können in entsprechende Grafik-Engines ausgelagert werden.

```

1 output states = do
2   screen ← SDL.getVideoSurface
3   SDL.fillRect screen Nothing Color.black
4   mapM_ (render screen) ∘ elemsIL ∘ fmap state $ states
5   SDL.flip screen
6   return False
7
8 render :: ObservableState → SDL.Surface → IO ()
9 render (Geometry position shape) screen = do
10   drawShape ...

```

Zuerst wird der alte Bildinhalt gelöscht, die Funktion `render` auf alle beobachtbaren Ausgabezustände angewendet, welche jeweils in den neuen Bildinhalt zeichnen. Abschließend werden die Grafikpuffer getauscht, wodurch der neue Bildinhalt angezeigt wird.

### 5.4.2 Debugging

Debugging zeigt, neben den eigentlichen Benutzerausgaben, zusätzliche Informationen für den Entwickler an. Die einfachste Möglichkeit ist, den beobachtbaren Ausgabezustand anders darzustellen (z. B. die Ausrichtung eines

Game-Objekts mittels Hilfslinien). Oft sollen aber interne Werte angezeigt werden, welche jedoch nicht im beobachtbaren Ausgabezustand enthalten sind (z. B. die Bewegungsvektoren, woraus nur eine endgültige Position erzeugt wird). Es muss daher der Ausgabezustand mit Debug-Informationen erweitert werden, jedoch sollen der ursprüngliche Code und die Schnittstellen möglichst unverändert bleiben. Es ist zwar grundsätzlich möglich, von jeder Funktion eine zusätzliche Debug-Variante zu erstellen und diese bei Bedarf zu verwenden, jedoch würde dies zu doppelten Implementierungen führen. Die Debug-Informationen können auch nicht an der Funktionssignatur „vorbeigeschmuggelt“ werden, da in rein funktionalen Programmiersprachen der Ausgabetyp immer genau deklariert ist. Eine Möglichkeit wäre daher, aufgrund der letzten Ausgabezustände auf die internen Werte zu schließen (z. B. die Geschwindigkeit aus alter und neuer Position berechnen), dies würde aber einen großen Aufwand und teilweise genaue Informationen über die interne Funktionalität erfordern. Die einzige Möglichkeit sind daher nur unreine Funktionen mittels `unsafePerformIO`, welche allerdings die eigentliche Funktionalität nicht beeinflussen dürfen. Da durch Debugging nur die Anzeige in einer *IO Monade* modifiziert wird, ist diese Bedingung grundsätzlich gegeben.

Eine entsprechende Funktionalität würde der Debugger *Hood* bieten, welcher dazwischenliegende Werte in einer Berechnung mittels `observe` beobachten kann und mittels Aufrufen von `unsafePerformIO` an einer externen Stelle sammelt. Allerdings werden die Informationen **erst nach der Programmausführung (post-mortem debugger)** in der Konsole ausgegeben. Eine grafische Anzeige würde die Erweiterung *GHood* bieten, jedoch ebenfalls erst nach der Programmausführung [Rei01]. Das folgende Codebeispiel zeigt eine Debug-Ausführung mittels *GHood*:

```

1 main = do
2     run0 $ reactivate initialize input output process
3
4 mover p0 v0 = proc a → do
5     v ← (v0 ^+^) ∘ observe "Velocity: " ≪ integral ↵ a
6     ...

```

Die Ausgabe der Debug-Informationen innerhalb der Game-Loop konnten jedoch aus zeitlichen Gründen nicht mehr implementiert werden. Grundsätzlich könnten aber die Debug-Informationen in jedem Ausgabeschritt von der externen Sammelstelle geholt werden und entweder einfach alle Debug-Informationen auf einmal oder nach jedem Game-Objekt entsprechend gerendert werden.

# Kapitel 6

## Schlussbemerkungen

### 6.1 Ergebnisse

Ziel dieser Arbeit war es ursprünglich, eine bessere Lösung für die Verknüpfung von Game-Objekt-Komponenten zu entwickeln, indem Membervariablen von Objektinstanzen mittels kleiner Funktionen verbunden werden können. Aufgrund einiger Essays von *Paul Graham* über *Lisp* und der Ähnlichkeit der Problemstellung mit dem funktionalen Programmierparadigma wurde zuerst die Programmiersprache *Common Lisp* dafür eingesetzt. Die Funktionalität konnte erfolgreich mit der Programmbibliothek *Cells* implementiert werden, allerdings musste der Zeitverlauf aufwendig an alle Objekte verteilt werden und sollte daraufhin, durch die Entwicklung einer kompletten Game-Engine, abstrahiert werden. Erst bei der weiteren Recherche wurde die funktional-reaktive Programmierung entdeckt und löste nicht nur die ursprünglichen Probleme, sondern führte zu weiteren Verbesserungen:

1. Vollständig abstrahierte und konzeptionell kontinuierlich erscheinende Zeit mittels Signalfunktionen.
2. Einfache Verknüpfung der Funktionalität und vollständige Wiederverwendbarkeit mittels unabhängigen Funktionen und Signalfunktionen.
3. Unterteilung einer Game-Engine in dessen elementare Bereiche und besser nachvollziehbarer Datenfluss (EVA-Modell) mittels rein funktionaler Programmierung.
4. Definition des Begriffs „Game-Engine“.

#### 6.1.1 Wiederverwendbarkeit

Das eigentliche Ziel komponentenbasierter Game-Engine-Architekturen war einerseits unabhängige Funktionalität, um dadurch eine hohe Wiederverwendbarkeit zu erreichen, andererseits eine automatisierte Kommunikation, um die Funktionalität dynamisch zur Laufzeit verändern zu können. Laut der Definition von Komponenten werden dabei, durch das Komponentenmo-

dell, **spezifische Interaktions- und Verknüpfungsstandards** definiert [HC01], also eine Art Kommunikationsprotokoll für Komponenten. Allerdings handelt es sich dabei um völlig gegensätzliche Ziele!

Das Komponentenmodell der Game-Objekt-Komponenten stellt deren kleinsten gemeinsamen Nenner der Kommunikation dar und schränkt somit deren Unabhängigkeit genau auf alle, statisch zur Programmübersetzung definierten, spezifischen Interaktions- und Verknüpfungsmechanismen (z. B. Nachrichten und Ereignisse) ein. Bei der Implementierung einer Komponente muss das Komponentenmodell entsprechend berücksichtigt werden, weshalb es auch bei der Programmübersetzung festgelegt wird und die Komponente nicht plötzlich automatisch auf neue Interaktionsmechanismen reagieren kann. Alle Komponenten, welche einem bestimmten Komponentenmodell unterliegen, können möglicherweise untereinander automatisch kommunizieren, sobald diese jedoch mit anderen Komponenten außerhalb des bekannten Komponentenmodell interagieren sollen, muss wieder eine spezifische Verknüpfung durch den Anwendungsprogrammierer stattfinden. Selbst wenn sich Komponenten anderer Komponentenmodelle oder Reflexionsmechanismen bedienen, wird dadurch nur das Komponentenmodell auf diese Bereiche ausgeweitet, niemals aber darüber hinaus.

In den bisher verwendeten komponentenbasierten Game-Engine-Architekturen konnten daher auch nur einfache Komponenten wiederverwendet werden, bei Game-Objekt-Komponenten vor allem jene, welche nur mathematische Prinzipien implementieren. Zum Beispiel benötigen zwar viele Komponenten eine physikalische Position, weshalb eine entsprechende Definition im Komponentenmodell der Game-Engine sinnvoll erscheint. Auch darauf aufbauende Funktionalität, wie z. B. eine Bild- oder Bewegungskomponente, ist für sehr einfache Computerspiele noch ausreichend. Allerdings müssen bereits bei etwas komplexeren Computerspielen die Komponenten ohnehin immer für jedes konkrete Game-Objekt spezifisch verknüpft werden. Durch die objektorientierte Programmierung wird dies aber noch zusätzlich dadurch erschwert, dass in Klassen und Komponenten relativ komplexe Funktionalität implementiert wird, wodurch, im Gegensatz zu Funktionen, die Funktionalität nicht aus elementaren Funktionen verknüpft werden kann. Es ist daher auch sehr bezeichnend, wenn eine Komponente immer mehr auf grundlegende Funktionalität reduziert wird und dadurch immer mehr die Form einer Funktion annimmt, um in möglichst vielen Anwendungsfällen benutzt werden zu können. Die ursprünglichen Überlegungen, um die Mängel von Game-Objekt-Komponenten zu lösen (z. B. hierarchische Komponenten und Verknüpfung von Komponenten mittels kleiner Funktionen), sind ein Musterbeispiel für ein sehr bekanntes Phänomen in der funktionalen Programmierung, welches als *Philip Greenspun's Tenth Rule*<sup>1</sup> bekannt geworden ist:

---

<sup>1</sup>[http://en.wikipedia.org/wiki/Greenspun's\\_Tenth\\_Rule](http://en.wikipedia.org/wiki/Greenspun's_Tenth_Rule)

Any sufficiently complicated platform contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of a functional programming language.

Die einzige Möglichkeit, vollständig unabhängige Funktionalität zu erreichen, sind nur: Funktionen! Eine Funktion ist nur, und wirklich nur, von seinen Eingabeparametern abhängig, nicht von externen Zuständen. Die nötigen Eingabeparameter und konkreten Typen einer Funktion werden dabei statisch zur Programmübersetzung definiert. Das „Komponentenmodell der Funktionen“ schränkt deren Verwendung sozusagen auf alle Typen und Verknüpfungsmechanismen der jeweiligen Programmiersprache ein (praktisch findet also keine Einschränkung statt). In funktionalen Programmiersprachen sind aber, im Gegensatz zu objektorientierten Programmiersprachen, ausgereifte Verknüpfungsmechanismen vorhanden (z. B. Funktionskomposition, Funktionen höherer Ordnung und Sections).

Die vorliegende Arbeit argumentiert daher: Die Funktionalität eines bestimmten Game-Objekt-Typ muss spezifisch für jedes konkrete Computerspiel implementiert werden, weshalb auch die Kommunikation innerhalb eines Game-Objekt spezifisch ist und daher nicht allgemein wiederverwendbar sein kann. Allerdings kann die Funktionalität eines Game-Objekts aus unabhängigen Funktionen oder Signalfunktionen verknüpft werden, welche wiederum vollständig wiederverwendbar sind. Folgend werden die verschiedenen Typen von Funktionalität gegenübergestellt:

**Funktionen:** Reine mathematische Funktionen, deren Ausgabe nur vom aktuellen Eingabeparameter abhängig ist. Dadurch sind sie von äußeren Zuständen unabhängig und vollständig wiederverwendbar.

**Signalfunktionen:** Zeitbeeinflusste Funktionen, deren Ausgabe nur von allen bisherigen Eingabeparameter seit dem Zeitpunkt der Erzeugung (`localTime`) abhängig ist. Dadurch sind sie ebenfalls rein funktional und mit *Yampa* wiederverwendbar.

**Game-Objekte:** Zeitbeeinflusste Funktionen, deren Ausgabe von einem Eingabeparameter aus einer ursprünglich nicht-deterministischen Eingabequelle eines spezifischen Subsystems abhängig ist. Dadurch sind sie zwar ebenfalls rein funktional, aber der Ein- und Ausgabetyp muss für jedes Subsystem spezifisch angepasst werden, wodurch sie nur zusammen mit dem konkreten Subsystem wiederverwendbar sind. Die Funktionalität muss aber meistens ohnehin für jedes konkrete Computerspiel neu implementiert werden.

### 6.1.2 Dynamische Funktionalität

Das zweite Ziel von komponentenbasierten Game-Engine-Architekturen ist, die Funktionalität während einer laufenden Simulation dynamisch verändern oder erweitern zu können, indem neue Subsysteme, Game-Logic oder

Game-Objekte von außen hinzugefügt oder bereits laufende angepasst werden. Dadurch kann die Entwicklung eines Computerspiels stark beschleunigt werden, da die Simulation nicht ständig neu gestartet und das Programm neu übersetzt werden muss, und ist daher besonders für Rapid Application Development interessant. Dynamische Funktionalität ist grundsätzlich ein gutes Anwendungsbereich für komponentenbasierte Architekturen, allerdings wurde durch das bisher verwendete Komponentenmodell mittels automatisierter Kommunikation die Erweiterbarkeit auf die Funktionalität der Game-Objekte eingeschränkt, wobei es sich dabei um ohnehin spezifische Bereiche handeln würde.

Die Aufgabe einer Game-Engine ist eigentlich nur, und dies ist durch das funktionale Modell klar ersichtlich, Eingabedaten zu sammeln, an Game-Objekte zu verteilen, die Game-Objekte zu verwalten und einen Ausgabezustand zu produzieren. Die nicht-deterministischen Bereiche müssen aber ohnehin von Subsystemen übernommen werden, wodurch die Game-Engine nur eine entsprechende Verwaltung (*hooks*) für verschiedene Subsysteme (*plugins*) bereitstellen müsste. Es muss dabei jedoch sichergestellt werden, dass die verschiedenen Datentypen eines Subsystems (z. B. `SDL.Surface`) nur an jene Game-Objekte weitergeleitet werden, welche die entsprechenden Datentypen bereits statisch bei der Programmübersetzung eingebunden haben und damit umgehen können. Auch die Beobachterfunktionalität der Ausgabezustände (z. B. Kollisionen) muss vom Subsystem für dessen spezifische Datentypen bereitgestellt werden (z. B. Kollisionsgeometrien), wobei die Game-Engine nur die Verwaltung der verschiedenen Beobachter übernimmt.

Grundsätzlich muss aber der Eingabe- und Ausgabetyp der Game-Objekte dynamisch erweiterbar sein, damit ein Game-Objekt Datentypen von verschiedenen Subsystemen entgegennehmen und entsprechende Ausgabezustände produzieren kann. Allerdings verwendet Haskell statische Typisierung, weshalb dynamische Typen nur mit aufwendigen Techniken erreicht werden können. In dieser Arbeit konnte eine entsprechende Implementierung aus Zeitgründen nicht mehr umgesetzt werden, allerdings wurde die grundsätzliche Möglichkeit in anderen Arbeiten bewiesen [SC05] [PSSC04].

Durch entsprechende Mechanismen könnten anschließend im Eingabeschritt weitere Subsysteme und Game-Objekte zur Laufzeit hinzugefügt werden oder auch Veränderungsoperationen eingeschleust werden. Ein Subsystem könnte dadurch von anderen Subsystemen unabhängig sein und diese würden erst in der Funktionalität eines Game-Objekts durch den Anwendungsprogrammierer verknüpft werden. Deshalb sollten konkrete Subsysteme und Game-Objekte auch nicht Teil der Game-Engine sein, sondern nur von einem Game-Framework für eine konkrete Game-Engine angebunden werden. Es ist weiters vorstellbar, dass Subsysteme mit ähnlicher Funktionalität einer Abstraktionsschicht unterstellt werden (z. B. Grafikfunktionalität unabhängig von *OpenGL* oder *DirectX*), wobei die Abstraktionsschicht ebenfalls Teil des Game-Frameworks wäre. Die Basisfunktionalität für Compu-

terspiele und Funktionalität eines Game-Objekts könnte dadurch zumindest unabhängig vom konkreten Subsystem, allerdings spezifisch für die Abstraktionsschicht, implementiert werden.

### 6.1.3 Definition: Game-Engine

Eine Game-Engine ermöglicht verschiedene konkrete Subsysteme zu verwalten, um mit einer Game-Loop eine interaktive, temporale Simulation zu betreiben, in der mit einer Game-Logic eine dynamische Struktur von Game-Objekten als unabhängige, parallele und identifizierbare Prozesse verwaltet und gesteuert werden, woraus ein Game-State als beobachtbarer Ausgabezustand produziert und verzögerungsfrei angezeigt wird.

Im Gegensatz dazu bedient sich ein Game-Framework nur einer Game-Engine und stellt verschiedene wiederverwendbare Programme und Funktionalität zur Verfügung, um grundsätzlich die Entwicklung von Computerspielen zu erleichtern und verschiedene Subsysteme an die Game-Engine anzubinden.

## 6.2 Ausblick

Die Basisfunktionalität für Computerspiele ist teilweise jetzt bereits völlig unabhängig von konkreten Subsystemen und müsste nur um zusätzliche Funktionalität erweitert werden. Zum Beispiel ist die Signalfunktion `animateLoop` nur vom *Haskell* Standarddatentyp `Int` und vom *Yampa* Signalfunktionstyp `SF` abhängig. Es könnte weitere Animationsfunktionalität hinzugefügt werden (z. B. `animateSkelet`) und wäre, zumindest zusammen mit *Yampa*, völlig wiederverwendbar.

Einige Funktionen sind allerdings von konkreten Datentypen eines Subsystems abhängig (z. B. `framesByRow` von `SDL.Rect`). Langfristig sollte daher im Game-Framework eine Abstraktionsschicht für Subsysteme mit ähnlicher Funktionalität eingeführt werden und entsprechende Übersetzungen von konkreten auf abstrakte Datentypen implementiert werden (z. B. von `SDL.Rect` auf ein abstraktes *Frame* und umgekehrt). Dadurch könnte auch Funktionalität, welche sich bisher auf spezifische Subsysteme bezog, zumindest zusammen mit der Abstraktionsschicht, ebenfalls wiederverwendet werden.

Die vorgestellte Implementierung der Game-Engine ist derzeit noch völlig statisch, weshalb die Subsysteme und Eingabe-/Ausgabetypen für jedes konkrete Computerspielprojekt angepasst werden müssen. Die Game-Engine könnte um eine Verwaltung für dynamisch geladene Subsysteme erweitert werden und entsprechend eine dynamische Anpassung der Eingabe-/Ausgabetypen vornehmen. Dadurch könnte die Funktionalität während einer laufenden Simulation verändert werden und würde die Entwicklung eines Computerspiels weiter beschleunigen.

Die offizielle Implementierung von *Yampa* ermöglicht derzeit nur die bestehende interne Funktionalität zu verwenden und zu erweitern, aber keine Änderungen der internen Struktur. *Blom* implementierte mit *FRVR*<sup>2</sup> eine Version von *Yampa*, welche auch diese Erweiterungen ermöglicht [Blo09]. Im Quellcode von *Yampa* steht zudem vermerkt, dass eine Umstellung des Zeitdatentyps auf Integer eine höhere Genauigkeit bei fortschreitender Simulation bieten würde, da Fließkommazahlen bei größeren Werten ungenauer werden. Dadurch müsste *Yampa* zusätzlich auf Millisekunden umgestellt werden, um eine ausreichende Zeitauflösung zu erhalten.

Die funktionale Programmierung und die Signalfunktionen von *Yampa* eignen sich hervorragend für visuelle Entwicklungsumgebungen und würden die Entwicklung für Nicht-Programmierer entsprechend erleichtern. Einige Ansätze wurden von *Blom* beschrieben und auch eine „Visuelle Entwicklungsumgebung zur Erzeugung von Haskell AFRP Code“ vorgestellt [Blo09]. Leider wurde vom Autor der vorliegenden Arbeit keine öffentlich zugängliche Version dieses Programms gefunden.

Aus einer Diskussion im Web entstand auch die Idee, ein funktional-reaktives Computerspiel für einen Webplayer zu implementieren, da dadurch der Zugang und die Demonstration des Konzepts erleichtert würde. Eine kurze Recherche zur grundsätzlichen Umsetzbarkeit ergab:

**Silverlight:** *Reactive Extensions for .NET (Rx)*<sup>3</sup> sind für *Silverlight* ab Version 3 verfügbar.

**Java Applets:** Mit *Frappe*<sup>4</sup>, einer reaktiven Programmmbibliothek der *Yampa*-Entwickler.

**Unity:** *Rx* wäre ebenfalls für *C#* ab *.NET 3.0* und *JavaScript* verfügbar, allerdings wird nur *C# .NET 2.0* und eine Eigenimplementierung von *JavaScript* verwendet, welche aber keine – für *Rx* notwendigen – anonymen Funktionen unterstützt.

**Flash:** Es wurde keine reaktive Programmmbibliothek gefunden.

## 6.3 Persönliche Meinung

### 6.3.1 Schwierigkeiten

Die Entwicklung an der vorliegenden Arbeit begann im Wintersemester 2009 nach drei Semestern Erfahrung mit komponentenbasierten Game-Engine-Architekturen in den objektorientierten Programmiersprachen *C++* und *C#*. Das darauffolgende Semesterprojekt *Common Lisp/Cells* und der Kurs „Alternative Programmierparadigmen“ waren, neben den Essays von *Paul Graham*, der erste Kontakt mit funktionaler Programmierung überhaupt. Die

<sup>2</sup><https://frvr.svn.sourceforge.net/svnroot/frvr/AFRP/trunk>

<sup>3</sup><http://msdn.microsoft.com/en-us/devlabs/ee794896.aspx>

<sup>4</sup><http://www.haskell.org/frappe>

Motivation und Vorteile des funktionalen Programmierparadigma leuchten grundsätzlich ein, allerdings erscheinen manche Programmkonstrukte teilweise seltsam und erfordern ungewohnte Denkweisen und Programmiertechniken (z. B. Rekursionen statt Schleifen und Funktionen statt Daten). Insbesondere die Tatsache, dass in der funktional-reaktiven Programmierung alle Eingabedaten an alle Game-Objekte einzeln verteilt werden müssen, statt einen Verweis oder ein Datenobjekt herumzureichen, waren zuerst etwas verwunderlich.

*Common Lisp* existiert seit 1984, seltsamerweise wurden aber bisher nur sehr wenig einführende Literatur und Entwicklungsumgebungen veröffentlicht. Auch über fortgeschrittene Themen existiert nur wenig Literatur, z. B. wurden für *Macro SPELs* nur *On Lisp* und *Let Over Lambda* gefunden, und für *CLOS* überhaupt nur *The Meta-Object Protocol*. Die Syntax und vielen Klammerungen von Lisp waren weniger problematisch als zuerst vermutet, allerdings werden für viele Funktionen sehr unsprechende Bezeichnungen verwendet (z. B. `rplacd` oder `eq/eql/equal`). Am meistens verwirrte jedoch die Inkonsistenz des funktionalen Programmierparadigmas, da letztendlich doch wieder unreine Funktionsaufrufe und globale Objekte möglich sind. Die strikte Trennung in einen deterministischen und nicht-deterministischen Teil wurde erst durch *Haskell* klar.

*Haskell* besitzt, trotz seiner kleineren Gemeinschaft gegenüber *Common Lisp*, mehr für Anfänger zugängliche Einführungen. Ältere Literatur verwendet noch eine etwas andere Syntax, da sich der *Haskell*-Standard in einem laufenden Überarbeitungsprozess befindet, wodurch allerdings kleine Widersprüche entstehen und den Lernprozess erschweren. Etwas ungewohnt ist zudem der häufige Einsatz von Mathematik in der Programmierung und entsprechend schwierig zu lernen sind abstrakte mathematische Konzepte wie Monaden und Arrows. Die statische Typisierung findet zwar durchgehend jeden Typfehler, im Gegensatz zu anderen statischen Programmiersprachen, allerdings sind Fehlermeldungen aufgrund der generischen Konstrukte oft unverständlich und bremsen die Entwicklung von Prototypen. Weiters fehlen derzeit noch ausgereifte Debugger, welche helfen würden, den Programmverlauf besser nachzuvollziehen, insbesondere in nicht-strikt evaluierten Programmiersprachen wie *Haskell*.

Nach der Einarbeitung in funktionale Programmierung und *Haskell* folgte die Einarbeitung in funktional-reaktive Programmierung, *Yampa*, Arrows und Arrow-Notation. Es gibt zwar einige Dokumente über *Yampa*, allerdings behandeln diese hauptsächlich die mathematische Herleitung und die kontinuierlichen Zeitverläufe. Im Nachhinein betrachtet werden allerdings alle wesentlichen Punkte ausgezeichnet erklärt und behandelt, sofern die grundlegende Materie erstmal verinnerlicht ist.

### 6.3.2 Schlusswort

Es gibt von der Volksschule bis zu höheren technischen Lehranstalten einen natürlichen Verlauf im Mathematikunterricht von den Grundrechnungsarten bis zu komplexer Mathematik. Seltsamerweise scheint aber bisher genau in der angewandten Programmierung die Mathematik kaum Relevanz zu finden<sup>5</sup>, obwohl es sich um eine grundlegend mathematische Disziplin handeln würde. Es freut mich daher, dass ich in meiner Abschlussarbeit auf die funktional-reaktive Programmierung gestoßen bin. Erst durch ein deklaratives Programmierparadigma ergibt auch die Mathematik in der Programmierung wieder Sinn. Neben der funktionalen Programmierung existieren aber noch viele weitere Programmierparadigmen, welche alle für verschiedene Problemdomänen bessere Abstraktionen bieten und eigentlich mehr Aufmerksamkeit verdienen, als dies derzeit der Fall ist. Ich hoffe daher mit der funktional-reaktiven Programmierung zumindest für die Computerspielentwicklung eine Alternative zu den bestehenden Ansätzen gezeigt zu haben.

---

<sup>5</sup> Abgesehen von der Algorithmik.

# Literaturverzeichnis

- [AECM08] Anderson, Eike F., Steffen Engel, Peter Comninos und Leigh McIoughlin: *The case for research in game engine architecture*. In: *Future Play '08: Proceedings of the 2008 Conference on Future Play*, Seiten 228–231, 2008.
- [All91] Allen, James F.: *Time and time again: the many ways to represent time*. International Journal of Intelligent Systems, 6:341–355, 1991.
- [Ber00] Berry, Gérard: *The esterel v5 language primer*. Technischer Bericht Version v5.91, Centre de Mathématiques Appliquées, Juni 2000.
- [Blo04] Blow, Jonathan: *Game development: Harder than you think*. Queue, 10(1):28–37, 2004.
- [Blo07] Blow, Jonathan: *Indie prototyping, braid, & making innovative games*. Webvideo, März 2007. <http://video.google.com/videoplay?docid=3727092195323329487>.
- [Blo09] Blom, Kristopher James: *Dynamic Interactive Virtual Environments*. Dissertation, University of Hamburg, Department of Informatics, September 2009.
- [Buc05] Buchanan, Warrick: *A generic component library*. In: *Game Programming Gems 5*, Seiten 177–187. Charles River Media, 2005.
- [Che05] Cheong, Mun Hon: *Functional programming and 3d games*. Diplomarbeit, University of New South Wales, November 2005.
- [CNP03] Courtney, Antony, Henrik Nilsson und John Peterson: *The yampa arcade*. In: *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, Seiten 7–18, September 2003.
- [Del09] Deloura, Mark: *Game engine showdown*. Game Developer Magazine, 16(5):7–12, Mai 2009.

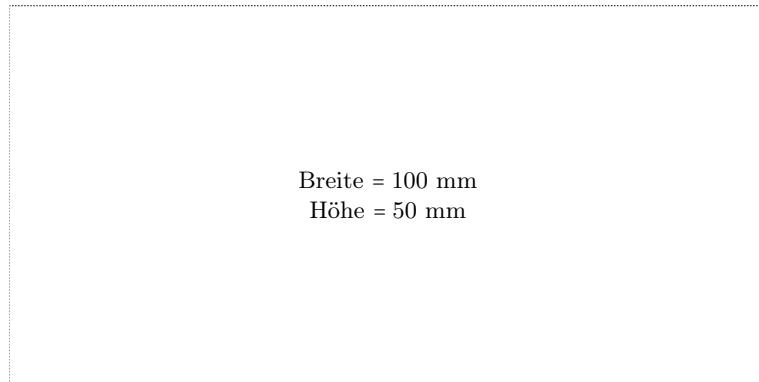
- [Dor09] Dornheim, Christoph: *Funktionsprinzip*. C't, (19):180–185, August 2009.
- [EH97] Elliott, Conal und Paul Hudak: *Functional reactive animation*. In: *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '97)*, Seiten 263–273, 1997.
- [Ell09a] Elliott, Conal: *Push-pull functional reactive programming*. Webvideo, 2009. <http://vimeo.com/6686570>.
- [Ell09b] Elliott, Conal: *Push-pull functional reactive programming*. In: *Haskell Symposium*, Seiten 25–36, 2009.
- [Ent09] Entertainment Software Association: *Essential Facts*, Mai 2009.
- [GGMS05] Gabler, Kyle, Kyle Gray, Kucic Matt und Shalin Shodhan: *How to prototype a game in under 7 days*. Webartikel, Oktober 2005. [http://www.gamasutra.com/features/20051026/gabler\\_01.shtml](http://www.gamasutra.com/features/20051026/gabler_01.shtml).
- [GH06] Gingold, Chaim und Chris Hecker: *Advanced prototyping*. Webaudio, 2006. <http://chrishecker.com/images/2/20/Gdc2006-AdvancedPrototyping.mp3>.
- [Gol04] Gold, Julian: *Object-Oriented Game Development*. Pearson Addison-Wesley, 2004.
- [Gra04] Graham, Paul: *Hackers & Painters*. O'Reilly & Associates, Inc., Mai 2004.
- [Gre09] Gregory, Jason: *Game Engine Architecture*. Transatlantic Publishers, April 2009.
- [Hag07] Hague, James: *Follow-up to: Admitting that functional programming can be awkward*. Webblog, November 2007. <http://prog21.dadgum.com/4.html>.
- [Har04] Harmon, Matthew: *A system for managin game entities*. In: *Game Programming Gems 4*, Seiten 69–83. Charles River Media, 2004.
- [HC01] Heineman, George T. und William T. Councill: *Component-Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, Juni 2001.
- [HCNP03] Hudak, Paul, Antony Courtney, Henrik Nilsson und John Peterson: *Arrows, robots, and functional reactive programming*. In: Jeuring, Johan und Simon Peyton Jones (Herausgeber): *Advanced Functional Programming*, Seiten 159–187. Springer, 2003.

- [Hoy08] Hoyte, Doug: *Let Over Lambda – 50 Years of Lisp*. HCSW and Hoytech, 2008.
- [Hug84] Hughes, John: *Why functional programming matters*. Technischer Bericht, Programming Methodology Group, University of Goteborg, November 1984.
- [Hug00] Hughes, John: *Generalising monads to arrows*. Science of Computer Programming, 37(1-3):67–111, Mai 2000.
- [LH07] Liu, Hai und Paul Hudak: *Plugging a space leak with an arrow*. Electronic Notes in Theoretical Computer Science, 193:29–45, 2007.
- [NCP02] Nilsson, Henrik, Antony Courtney und John Peterson: *Functional reactive programming, continued*. In: *Haskell '02: Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*, Seiten 51–64, 2002.
- [Nil05] Nilsson, Henrik: *Dynamic optimization for functional reactive programming using generalized algebraic data types*. In: *Proceedings of the Tenth ACM SIGPLAN International Conference on Functional Programming (ICFP'05)*, Seiten 54–65, September 2005.
- [Pat01] Paterson, Ross: *A new notation for arrows*. ACM SIG-PLAN Notices, 36(10):229–240, Oktober 2001.
- [PSSC04] Pang, André, Don Stewart, Sean Seefried und Manuel M. T. Chakravarty: *Plugging haskell in*. In: *Proceedings of the ACM SIGPLAN workshop on Haskell*, Seiten 10–21, 2004.
- [Rei01] Reinke, Claus: *GHood – graphical visualisation and animation of haskell object observations*. In: *ACM SIGPLAN Haskell Workshop*, September 2001.
- [Ren05] Rene, Bjarne: *Component based object management*. In: *Game Programming Gems 5*, Seiten 25–37. Charles River Media, 2005.
- [RH04] Roy, Peter Van und Seif Haridi: *Concepts, Techniques, and Models of Computer Programming*. MIT Press, 2004.
- [RM03] Rollings, Andrew und Dave Morris: *Game Architecture and Design: A New Edition*. New Riders Games, Oktober 2003.
- [SC05] Stewart, Don und Manuel M. T. Chakravarty: *Dynamic applications from the ground up*. In: *Proceedings of the ACM SIGPLAN Workshop on Haskell*, Seiten 27–38, September 2005.

- [Scu09] Sculthorpe, Neil: *Safe functional reactive programming through dependent types*. Webvideo, 2009. <http://vimeo.com/6632457>.
- [SK09] Schirmer, Uwe und Christian Kücherer: *Renaissance des F. iX*, (12):54–57, Dezember 2009.
- [Sto06] Stoy, Chris: *Game object component system*. In: *Game Programming Gems 6*, Seiten 393–404. Charles River Media, 2006.
- [Swa09] Swaine, Michael: *It's time to get good at functional programming*. Dr. Dobbs, Seiten 14–16, Januar 2009.
- [Wad92] Wadler, Philip: *Comprehending monads*. Mathematical Structures in Computer Science, 4:461–493, Dezember 1992.
- [Wad98] Wadler, Philip: *Why no one uses functional languages*. ACM SIG-PLAN Notices, 33(8):23–27, August 1998.
- [Wan02] Wan, Zhanyong: *Functional Reactive Programming for Real-Time Reactive Systems*. Dissertation, Computer Science Department, Yale University, Oktober 2002.
- [WTH02] Wan, Zhanyong, Walid Taha und Paul Hudak: *Real-time FRP*. In: *International Conference on Functional Programming (ICFP'01)*, Seiten 146–156, 2002.
- [Yor09] Yorgey, Brent: *The typeclassopedia*. The Monad.Reader, 13:17–68, März 2009.

# Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



Breite = 100 mm  
Höhe = 50 mm

— Diese Seite nach dem Druck entfernen! —