

B A C H E L O R A R B E I T

Vergleich und Bewertung von *Teaching Languages*
am Beispiel der Implementierung von
Computerspielen

Vorgelegt an der TH Köln
Campus Gummersbach
im Studiengang
Allgemeine Informatik

ausgearbeitet von:
LUKAS GOBELET
(Matrikelnummer: 11118011)

Erster Prüfer: Prof. Christian Kohls
Zweiter Prüfer: Alexander Dobrynin

Köln, im Juli 2021

Teaching Languages sind Programmiersprachen, welche explizit zum Erlernen von Programmierung gedacht sind. Ein anderes Ziel von *Teaching Languages* kann zudem sein, diese nach der Lerntheorie des Konstruktionismus zur kreativen Entfaltung und als Lernwerkzeug zu verwenden. Computerspiele sind fest in unserer Kultur verankert und können als großer Motivator dienen, Programmieren zu lernen.

In dieser Arbeit werden fünf wesentlich unterschiedliche *Teaching Languages* (Pyret, Quorum, Scratch, ToonTalk und Game-Changineer) in 12 unterschiedlichen Bewertungsbereichen bewertet. Zur Untersuchung der *Teaching Languages* wurden in jeder Sprache drei simple Computerspiele (Tic-Tac-Toe, Flappy-Bird und Tamagochi) implementiert. Die Implementierungen dienten zur Sammlung von praktischen Erfahrungen in den *Teaching Languages* und zudem wird in einem der Bewertungsbereiche die Einfachheit der Implementierungen der Computerspiele bewertet.

Es zeigte sich, dass die verschiedenen *Teaching Languages* verschiedene Stärken und Schwächen haben. Zudem stehen zwei der Bewertungsbereiche gewissermaßen in einem Zielkonflikt zueinander. Es konnten viele Erkenntnisse gewonnen werden, welche für das Design von zukünftigen *Teaching Languages* verwendet werden können.

Schlüsselwörter: Programmiersprachendesign, Bildung, Computerspiele, Konstruktivismus

Erklärung über die selbständige Abfassung der Arbeit

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer oder der Verfasserin/des Verfassers selbst entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort, Datum

Rechtsverbindliche Unterschrift

Inhaltsverzeichnis

| | |
|---|-----------|
| 1. Einleitung | 8 |
| 2. Grundlagen | 11 |
| 2.1. Psychologie & Lerntheorien | 11 |
| 2.1.1. Cognitive Load Theory | 11 |
| 2.1.2. Four Component Instructional Design | 12 |
| 2.1.3. Konstruktionismus | 14 |
| 2.2. Computerspiele | 15 |
| 2.2.1. Game Loop | 15 |
| 2.2.2. Koordinatensystem | 16 |
| 2.2.3. Sprite | 16 |
| 3. Methodisches Vorgehen | 17 |
| 3.1. Zielgruppen | 17 |
| 3.2. Auswahl der <i>Teaching Languages</i> | 18 |
| 3.2.1. Weitere <i>Teaching Languages</i> | 18 |
| 3.3. Auswahl der Spiele | 20 |
| 3.3.1. Tic-Tac-Toe | 20 |
| 3.3.2. Flappy Bird | 21 |
| 3.3.3. Tamagochi | 22 |
| 3.4. Bewertungsbereiche | 23 |
| 3.4.1. Einfachheit der Implementierung von Computerspielen | 24 |
| 3.4.2. Erlernbarkeit nach Konstruktionismus | 27 |
| 3.4.3. Inkrementelle Einführung | 28 |
| 3.4.4. Intuitive Syntax und Natürlichsprachlichkeit | 29 |
| 3.4.5. Feature Uniformity und Orthogonalität | 37 |
| 3.4.6. Unterstützung der wichtigsten Programmierkonzepte & Datenstrukturen | 38 |
| 3.4.7. Programmierumgebung (IDE) | 40 |
| 3.4.8. Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplin (<i>Clean Code</i>) | 46 |
| 3.4.9. Fehlermeldungen | 48 |
| 3.4.10. Ein hohes Abstraktionsniveau für Datentypen | 52 |
| 3.4.11. Anwendungsbereiche und Interoperabilität mit Technologien und anderen Programmiersprachen | 53 |
| 3.4.12. Dokumentation & Community Support | 54 |

| | |
|--|------------|
| 3.5. Bewertungsbereichübergreifende Eigenschaften einer Programmiersprache und Reflexion | 55 |
| 3.5.1. Syntaktischer Zucker | 55 |
| 3.5.2. Statische vs. Dynamische Typisierung | 56 |
| 4. Implementierung und Bewertung der Teaching Languages | 58 |
| 4.1. Pyret | 58 |
| 4.1.1. Kurze Einführung in Pyret | 58 |
| 4.1.2. Beschreibung der Programmierumgebung | 58 |
| 4.1.3. Interaktive Anwendungen in Pyret | 58 |
| 4.1.4. Datendefinitionen in Pyret | 60 |
| 4.1.5. Bouncing Ball | 62 |
| 4.1.6. Tic-Tac-Toe | 63 |
| 4.1.7. Bewertung | 70 |
| 4.2. Quorum | 80 |
| 4.2.1. Kurze Einführung in Quorum | 80 |
| 4.2.2. Beschreibung der Programmierumgebung | 80 |
| 4.2.3. Interaktive Anwendungen in Quorum | 82 |
| 4.2.4. Bouncing Ball | 82 |
| 4.2.5. Flappy Bird | 84 |
| 4.2.6. Bewertung | 90 |
| 4.3. Scratch | 102 |
| 4.3.1. Kurze Einführung in Scratch | 102 |
| 4.3.2. Beschreibung der Programmierumgebung | 103 |
| 4.3.3. Bouncing Ball | 104 |
| 4.3.4. Tamagochi | 104 |
| 4.3.5. Bewertung | 109 |
| 4.4. ToonTalk | 120 |
| 4.4.1. Kurze Einführung in ToonTalk | 120 |
| 4.4.2. Beschreibung der Programmierumgebung | 121 |
| 4.4.3. Concurrent Constraint Logic Programming und Actor Model | 125 |
| 4.4.4. Bouncing Ball | 126 |
| 4.4.5. Tic-Tac-Toe | 127 |
| 4.4.6. Bewertung | 134 |
| 4.5. Game Changer | 142 |
| 4.5.1. Kurze Einführung in Game Changer | 142 |
| 4.5.2. Beschreibung der Programmierumgebung | 143 |
| 4.5.3. Bouncing Ball | 144 |
| 4.5.4. Flappy Bird | 145 |
| 4.5.5. Bewertung | 148 |
| 5. Zusammenfassung und Fazit | 155 |
| 5.1. Vergleich und Zusammenfassung der <i>Teaching Languages</i> | 155 |
| 5.2. Reflexion von GermanSkript | 157 |

| | |
|---|------------|
| 5.3. Diskussion & Fazit | 159 |
| Akronyme | 160 |
| Tabellenverzeichnis | 161 |
| Abbildungssverzeichnis | 163 |
| Quellenverzeichnis | 164 |
| A. Diverses | 172 |
| A.1. Bildschirmaufnahmen IDEs | 172 |
| A.2. Emails | 176 |
| A.2.1. Ken Kahn (ToonTalk) | 176 |
| A.2.2. Game-Changineer | 177 |
| A.3. Quorum: Auflistung der Packages der Standardbibliothek | 179 |
| B. Bewertungen | 182 |
| B.1. Pyret | 182 |
| B.2. Quorum | 185 |
| B.3. Scratch | 188 |
| B.4. ToonTalk | 190 |
| B.5. Game-Changineer | 193 |
| B.6. Rangliste pro Bewertungsbereich | 196 |
| C. Implementierungen | 198 |
| C.1. Implementierungen in Pyret | 198 |
| C.1.1. Fibonacci | 198 |
| C.1.2. Bouncing Ball | 198 |
| C.1.3. Tic-Tac-Toe | 199 |
| C.1.4. Flappy-Bird | 205 |
| C.1.5. Tamagochi | 210 |
| C.2. Implementierungen in Quorum | 218 |
| C.2.1. Fibonacci | 218 |
| C.2.2. Bouncing Ball | 218 |
| C.2.3. Tic-Tac-Toe | 219 |
| C.2.4. Flappy-Bird | 226 |
| C.2.5. Tamagochi | 232 |
| C.3. Implementierungen in Scratch | 240 |
| C.3.1. Fibonacci | 240 |
| C.3.2. Bouncing Ball | 241 |
| C.3.3. Tic-Tac-Toe | 242 |
| C.3.4. Flappy Bird | 243 |
| C.3.5. Tamagochi | 246 |
| C.4. Implementierungen in ToonTalk | 251 |
| C.4.1. Fibonacci | 251 |

| | |
|---|-----|
| C.4.2. Bouncing-Ball | 256 |
| C.4.3. Tic-Tac-Toe | 265 |
| C.5. Implementierungen in Game-Changineer | 266 |
| C.5.1. Fibonacci | 266 |
| C.5.2. Bouncing Ball | 266 |
| C.5.3. Tic-Tac-Toe | 266 |
| C.5.4. Flappy-Bird | 267 |
| C.5.5. Tamagochi | 268 |

1. Einleitung

Die Idee von *Teaching Languages* ist nicht neu und geht weit zurück in die Vergangenheit. Nur zehn Jahre später nach der Entwicklung der ersten kommerziellen höheren Programmiersprache Fortran im Jahre 1954 (vgl. [Bac78]), wurde 1964 die erste *Teaching Language* BASIC am *Dartmouth College* von den Professoren John Kemeny und Tom Kurtz entwickelt (vgl. [Col14]). Die Idee hinter BASIC war es, die Verwendung von Computern für alle Studenten – ebenso für Studenten der Sozial- und Humanwissenschaften – verfügbar zu machen. So sollte BASIC einfach zu lernen und zu benutzen sein, sodass die Hürde so gering ist, dass Anwender, ohne tiefere Sachkenntnisse, es verwenden können. Zusammen mit BASIC wurde gleichzeitig das *Dartmouth-Time-Sharing-System* entwickelt, das es erlaubte, mehrere Berechnungen gleichzeitig durchzuführen, sodass für kleinere Berechnungen nicht mehr so lange gewartet werden musste.

Bei neueren *Teaching Languages* spielt ebenso die Interaktivität eine zentrale Rolle.

Der nächste Meilenstein in der Geschichte der *Teaching Languages* war dann die Entwicklung der Programmiersprache Logo im Jahre 1967, welche für ihre leichte Erlernbarkeit und die *Turtle-Grafik-Programmierung* bekannt ist.

Seymour Papert – der Mitentwickler von Logo – begründete die Lerntheorie des Konstruktionismus, welche besagt, dass Wissen durch aktives Handeln aufgebaut wird. Papert's Vision war es, Computer in der Bildung zentral einzusetzen, um die Schüler zu ermächtigen, motivierende computerbasierte Projekte umzusetzen. Computer sollten die Bildung aus seiner Sicht revolutionieren, indem sie als mächtiges Werkzeug verwendet werden, um etwas über die Welt zu erfahren und um spannende Projekte zu ermöglichen (vgl. [Pap99]). Auf die konstruktionistische Lerntheorie wird im zweiten Kapitel (Grundlagen) eingegangen.

Papert's Vision ist noch nicht vollumfänglich umgesetzt worden. Vielleicht liegt es mitunter daran, dass Programmieren Lernen eine inhärent komplexe Lernaufgabe darstellt. *Teaching Languages* versuchen genau diese Hürde herabzusetzen. Um zu verstehen, was die Schwierigkeiten beim Lernen sind und wie diese verringert werden können, kann die *Cognitive Load Theory* (CLT) verwendet werden, die ebenfalls im zweiten Kapitel näher erläutert wird.

Mit der fortschreitenden Digitalisierung und der Allgegenwärtigkeit von Computern, ist das Thema *Teaching Languages* nach wie vor sehr relevant. Es ist sehr wichtig, dass ein verantwortungsbewusster Umgang mit Computern gelehrt wird und zudem jeder Mensch dazu ermächtigt wird, Computer als mächtiges Werkzeug zu benutzen, um kreative Projekte umzusetzen. Obendrein gibt es den Vorschlag von Jeanette M. Wing *Computational Thinking* neben Lesen, Schreiben und Rechnen als weitere fundamentale Fähigkeit im 21. Jahrhundert hinzuzufügen (vgl. [Win06]). *Computational Thinking* meint, so zu denken, wie ein Computerwissenschaftler – also in mehreren Abstraktionsleveln und ein Problem

und dessen Lösung so zu formulieren, dass die Lösung von einem informationsverarbeitenden Agenten ausgeführt werden kann (vgl. [Win11]).

Im Jahre 2020 belegte die *Teaching Language* Scratch den 20. Platz im TIOBE-Index (s. [TIO20]), der die Popularität von Programmiersprachen misst. Dies zeigt die Relevanz von *Teaching Languages*.

Diese Arbeit befasst sich mit der Untersuchung, Bewertung und dem Vergleich grundsätzlich verschiedener *Teaching Languages* (Scratch, Pyret, Quorum, ToonTalk und GameChangineer) am Beispiel der Implementierung von Computerspielen. Diese werden anhand einer Reihe von Bewertungskriterien in verschiedenen Bewertungsbereichen, welche im dritten Kapitel aufgestellt werden, bewertet.

Computerspiele wurden gewählt, da Computerspiel-Kultur zum einem in unserer Kultur unabhängig von sozialen Schichten und besonders bei Jugendlichen besonders stark vertreten sind und somit als großer Motivator dient, Computerprogrammierung zu lernen. Es gibt außerdem Evidenz dafür, dass das Programmieren von Computerspielen mehr als irgendeine andere Projektart, die meisten Programmierkonzepte wie Bedingungen und Variablen umfasst (vgl. [AW12]). Außerdem baut das Programmieren von Computerspielen auf der konstruktionistischen Idee auf, dass ein großer Lernerfolg stattfinden kann, wenn persönliche motivierende Projekte umgesetzt werden.

In jeder der untersuchten Sprachen wurden drei Computerspiele programmiert. Der Quellcode der Implementierungen ist im Anhang enthalten (s. Anhang C). Außerdem ist der Quellcode auch auf *Github* veröffentlicht (s. [Gob21b]). Die Programmierung der Computerspiele diente zum Sammeln praktischer Erfahrung in den *Teaching Languages*, welche in die Bewertung eingeflossen sind und zum anderen werden die Programmiersprachen im vierten Kapitel anhand der Implementierungen der Computerspiele eingeführt. Außerdem geht es in einem der Bewertungsbereiche darum, wie einfach die Implementierung der Computerspiele in der jeweiligen *Teaching Language* ist.

Neben dedizierten *Teaching Languages* gibt es auch Programmierlernumgebungen wie *Processing* (s. [Pro21]) oder *Kojo* (s. [fou20]), welche industrielle Programmiersprachen in einer Lernumgebung verwenden, welche die Hürde des Erlernens der Programmiersprache und das Erstellen von Projekten verringert. So verwendet *Processing* die Programmiersprache Java und *Kojo* die Sprache Scala. Der Vorteil dieser Lernumgebungen ist, dass dabei eine industrielle Programmiersprache gelernt wird, welche über die Lernumgebung hinaus verwendet werden kann und einen leichteren Übergang in die professionelle Programmierung ermöglicht.

Der Nachteil gegenüber *Teaching Languages* ist, dass industrielle Programmiersprachen nicht für Beginner, sondern für Experten designt wurden und die Sprachen oft komplex für Beginner sind.

Teaching Languages sind ebenfalls in einer Lernumgebung eingebettet, welche ein sehr wichtiger Bestandteil dieser ist und deshalb stark in die Bewertung miteinbezogen wird.

Meine persönliche Motivation *Teaching Languages* zu untersuchen, resultiert aus meiner Praxisprojekt-Arbeit eine „deutsche“ Programmiersprache mit dem Namen GermanSkript (s. [Gob21a]) zu entwickeln. Das primäre Ziel hinter GermanSkript war es, eine Teilmenge der deutschen Sprache in eine Programmiersprache einzubetten. Dieses Ziel umzusetzen ist

auch ganz gut gelungen – nur machte der vollkommene Fokus auf dieses Ziel, GermanSkript nicht unbedingt zu einer lernenswerten Programmiersprache. Die Natürlichsprachlichkeit von GermanSkript ist allerdings ein Vorteil für eine *Teaching Language*. In den folgenden Bewertungen kommt GermanSkript nicht weiter vor, doch erlaube ich mir eine Reflexion von GermanSkript im Bezug auf die Bewertungskriterien einer *Teaching Language* im fünften Kapitel, nachdem die Ergebnisse der Bewertungen zusammengefasst wurden sind.

2. Grundlagen

2.1. Psychologie & Lerntheorien

Da es in dieser Arbeit um *Teaching Languages* geht, muss verstanden werden, wie das Lernen erleichtert werden kann und die *Cognitive Load Theory* (CLT) liefert ein gutes Modell dafür. Des Weiteren sollte sich angeguckt werden, wie etwas gut auf Seiten des Lehrenden vermittelt werden kann. Hierfür wird das *Four Component Instructional Design* erklärt, das auf die CLT basiert. Schließlich sollte sich noch angeschaut werden, was Lernende selbst motiviert und hierfür wird ein kurzer Einblick in die Lerntheorie des *Konstruktionismus* gegeben.

2.1.1. Cognitive Load Theory

Die *Cognitive Load Theory* (CLT, zu dt. Theorie der Kognitiven Belastung) ist eine psychologische Theorie, die beschreibt wie leicht oder schwer es fällt, etwas zu lernen. Die Theorie wurde 1988 von John Sweller aufgestellt und wurde in Laufe der Zeit immer weiter entwickelt (vgl. [SMP19]). Sie basiert auf das Mehrspeichermodell des Gedächtnisses. Das Arbeitsgedächtnis ist begrenzt in der Anzahl der Informationen, die es gleichzeitig handhaben kann. Nach [Cow01] können 4 ± 1 Informationen im Arbeitsgedächtnis gehalten werden.

Die Theorie definiert drei verschiedene Arten von kognitiver Belastung:

- **intrinsische Belastung:** Dies ist die Belastung, die sich aus der Komplexität des zu Lernenden ergibt. Sie ist abhängig von dem Vorwissen und Erfahrungen des Lernenden, die sogenannten Schemata – mentale Modelle, die im Langzeitgedächtnis gespeichert sind. Informationen, die im Langzeitgedächtnis gespeichert sind und in das Arbeitsgedächtnis geladen werden, nehmen keine kognitiven Ressourcen im Arbeitsgedächtnis ein. Und dies unterscheidet einen Beginner von einem Experten. So wurde herausgefunden, dass bei professionellen Schach-Spielern der Hauptgrund für ihre Überlegenheit in Problemlösungsstrategien erinnerte Schachkonfigurationen sind, welche im Langzeitgedächtnis gespeichert sind (vgl. [SMP19]).
- **extrinsische Belastung:** Die extrinsische Belastung ergibt sich aus der Klarheit und Darbietung des Lernmaterials und der Lerninstruktionen. Diese Belastung kann der Lehrende gezielt durch die Vermittlung des Lernmaterials reduzieren.
- **lernbezogene Belastung:** Die lernbezogene Belastung, ist die Belastung des Lernens an sich, also das zu Lernende mit dem Vorwissen zu verknüpfen, schon gelernte Schemata zu erweitern oder neue Schemata aufzubauen. Dabei ist die lernbezogene Belastung keine Belastung an sich. So wurde herausgefunden, dass wenn die extrinsische Belastung reduziert wird, die totale Belastung ebenso reduziert wird und die

freigegebene extrinsische Belastung nicht einfach durch die lernbezogene Belastung ersetzt wird. Vielmehr verteilt die lernbezogene Belastung extrinsische zu intrinsische Aspekte der Lernaufgabe (vgl. [SMP19]).

Die CLT ist besonders relevant, wenn es darum geht Programmieren zu lernen, da Programmieren sehr komplex ist und eine hohe intrinsische Belastung für Beginner aufweist. Die Belastung ist so hoch, weil Syntax, Semantik und das Problem, das gelöst werden soll, gleichzeitig im Arbeitsgedächtnis gehalten werden muss.

Bei den *Teaching Languages* geht es darum, diese Komplexität zu reduzieren, wie z.B. bei der blockbasierten visuellen Programmiersprache *Scratch*, die die kognitive Belastung reduziert, indem durch die Blöcke, die Komplexität der Syntax komplett wegfällt.

2.1.2. Four Component Instructional Design

Four Component Instructional Design (4C-ID) ist ein Design-Modell, das auf die CLT basiert und beschreibt, wie Kurse entworfen sein müssen, damit sie das Lernen von komplexen, technischen Fähigkeiten (wie z.B. der Programmierung) ermöglichen. Das erste mal wurde 4C-ID 1992 von Merriënboer beschrieben und gleichzeitig mit der CLT weiterentwickelt (vgl. [SMP19]).

Dieses Design-Modell wird nun anhand des Beispiels der Programmierung erklärt: Zunächst basiert das Modell auf der Annahme, dass komplexe Fähigkeiten einzelne wiederkehrende Aufgabe beinhalten, welche konsistent über Aufgabenstellungen und Routinen sind und einzelne nicht wiederkehrende Aufgaben, welche auf Probleme, Schlussfolgerungen und Entscheidungen treffen basieren.

Wiederkehrende Aufgaben sind in der Programmierung z.B.

- Das Erstellen einer neuen Funktion, Klasse, Methode, ...
- Das Iterieren über ein Array
- Das Schreiben einer Bedingung oder Schleife
- Debugging
- Das Schreiben von Tests

Nicht wiederkehrende Aufgaben in der Programmierung basieren auf Überlegungen, die die Programmieraufgabe an sich betreffen wie z.B.

- Wie strukturiere ich mein Programm, um das Problem zu lösen?
- Wie sind die Beziehungen zwischen den Objekten bei der objektorientierten Programmierung?
- Welche Funktionen kann ich miteinander verbinden?
- Welche Felder brauche ich für meine Datentypen?
- Welchen Algorithmus verwende ich?

4C-ID gibt vier Komponenten an, in die ein Kurs unterteilt wird und beschreibt wie diese Komponenten organisiert und miteinander verbunden werden sollten, damit die kognitive Belastung nicht zu hoch wird. Diese Komponenten sind:

1. Lernaufgabe (*learning tasks*)
2. Unterstützende Informationen (*supportive information*)
3. Prozedurale Informationen (*procedural information*)

4. Teilaufgaben-Wiederholungen (*part-task practice*)

Lernaufgaben sind Aufgaben, die vorzugsweise aus dem *realen* Leben gegriffen sind und beinhalten wiederkehrende sowie nicht wiederkehrende Fähigkeiten.

Um die intrinsische kognitive Belastung zu reduzieren, sollten diese innerhalb eines Kurses der Komplexität aufsteigend nach organisiert werden. Lernende starten mit simplen Aufgaben, doch umso mehr Expertise die Lernenden erhalten, desto komplexer werden die Aufgaben.

Um die extrinsische Belastung zu bewältigen, sollte die Unterstützung durch den Lehrer für den Lernenden bei der Lernaufgabe zuerst sehr hoch sein. Dann mit den weiteren Lernaufgaben und erhöhter Komplexität graduell fallen, bis der Lernende die Lernaufgaben der Einheit selbstständig lösen kann. Mit der Lernaufgabe der nächsten Einheit ist die Unterstützung durch den Lehrer dann wieder zunächst groß. Hierfür eignet es sich dem Lernendem am Anfang die Lernaufgabe durch fertige Beispiele (*worked examples*) zu zeigen, dann bei der nächsten Aufgabe ein Problem anzugeben, das schon halb gelöst ist (*completion problems*), bis der Lernende dann am Ende der Einheit ein Problem selbstständig lösen kann (*conventional problems*).

Die einzelnen Lernaufgaben in einer Einheit sollten außerdem eine hohe Variabilität mit sich bringen, sodass Lernaufgaben miteinander verglichen werden können. So kann ein Programmierkurs vereinfacht so aufgebaut werden, dass zunächst ganze kleine Programme gezeigt werden, schließlich Programme, bei denen bestimmte Funktionen fehlen, die selbst implementiert werden müssen bis zur schlussendlichen selbstständigen Implementierung eines kleinen Programms. Um die Lernaufgaben etwas zu variieren, könnte man dann auch noch den Prozess des Debuggings lehren, da visuelles Debugging ein gutes Verständnis für die Ausführung eines Programmes aufbauen kann.

Unterstützende Informationen unterstützen die sich nicht wiederholenden Aspekte einer Lernaufgabe. Sie erklären die Theorie und wie Aufgaben in der Domain systematisch angegangen werden können. Dabei setzen sie dabei an, was die Lernenden schon wissen und bringen das Wissen bei, das benötigt wird, um die Lernaufgabe zu meistern.

Unterstützende Informationen sollten dabei am besten vor der Lernaufgabe gezeigt werden, da die kognitive Belastung beim gleichzeitigen Arbeiten an der Lernaufgabe sonst zu hoch ist. So kann das Wissen vorher im Langzeitgedächtnis abgelegt werden und während dem Arbeiten an der Lernaufgabe aktiviert und verfeinert werden.

Auf das Programmieren bezogen wären unterstützende Informationen bspw. solche, die folgende Fragen beantworten:

- Was ist ein Programm?
- Was ist eine Variable?
- Was ist eine Funktion?
- Welche Datentypen gibt es?

Bei **prozeduralen Informationen** handelt es sich um Anweisungen, wie etwas gemacht werden soll, um die wiederkehrenden Handlungen einer Lernaufgabe zu lernen.

Prozedurale Informationen haben eine geringe Komplexität und sollten während der Ausführung der Lernaufgabe verarbeitet werden. Dabei handelt es sich bspw. um Schritt-für-Schritt-Anleitungen oder der Lehrer tritt als Assistent auf. Auf das Programmieren

bezogen wären es bspw. Anleitungen wie die Programmierumgebung funktioniert, wie z.B. ein Programm ausgeführt wird, wie debuggt werden kann oder wie eine neue Quellcode-Datei hinzufügt wird.

Teilaufgaben-Wiederholungen sind Übungen einer einzelnen Teilaufgabe zum Automatisieren grundlegender oder kritischer wiederholenden Fähigkeiten. Das Automatisieren reduziert dann die kognitive Belastung, wenn an einer ganzen Lernaufgabe gearbeitet wird.

Hierfür könnten sich in der Programmierung auf einen bestimmten Teilespekt der Programmierung konzentriert werden, wie z.B. das Erstellen einer Klasse mit Feldern und dem Konstruktor.

2.1.3. Konstruktionismus

Konstruktionismus ist eine Lerntheorie, die zwischen 1980 und 1990 von der Psychologin Idit Harel und dem Mathematiker Seymour Papert begründet wurde.

Die Idee hinter dem Konstruktionismus lässt sich durch die allgemeinbekannte Idee *Learning by doing* grob und kurz zusammenfassen. Papert war der Ansicht das Lernen (im Sinne von Aufbau von Wissensstrukturen) besonders gut gelingen kann, indem der Lernende bewusst etwas kreiert, wie z.B. eine Sandburg am Strand oder eine Theorie des Universums (vgl. [HP91]).

Papert's Idee des Konstruktionismus wurde maßgeblich von seiner Zusammenarbeit mit dem Biologen und Psychologen Jean Piaget an der Genfer Universität von 1958 - 1963 beeinflusst. Dabei ist Jean Piaget ein Mitbegründer der philosophischen Theorie des Konstruktivismus (Achtung diesmal mit 'v'!). Dabei geht es darum, dass Menschen beim Lernen mit ihrer Wahrnehmung die Welt nicht abbilden, sondern inkrementell konstruieren, was mit der Schemabildung in der CLT übereinstimmt.

In den 1960er Jahren kam Papert dann in die USA, wo er das *MIT Artificial Intelligence Laboratory* zusammen mit Marvin Minsky gründete. Das Team kreierte 1967 dann die Programmiersprache Logo. Aus Logo sind dann auch die sogenannten *Turtle-Grafiken* entstanden. Dabei geht es darum, 2D-Grafiken durch Programmieranweisungen, die an eine virtuelle Schildkröte gesendet werden, zu erstellen. Es werden etwa Anweisungen wie '*bewege die Schildkröte um 20 nach vorn*', '*rotiere um 90° nach rechts*' oder '*wechsel die Farbe nach "rot"*' an die Schildkröte gesendet (vgl. [Fou15]).

In einer Rede an eine Konferenz von Pädagogen in Japan stellt Papert in den 1980er Jahren dem Konstruktionismus den Instruktionismus gegenüber (s. [Pap80]) und stellt seine Vision für die Zukunft des Lernens vor.

Der Instruktionismus besagt, dass, um eine bessere Lehre zu ermöglichen, die Unterrichtsanweisungen verbessert werden müssen. Der Konstruktionismus hingegen besagt, dass die Motivation an Projekten zu arbeiten und etwas zu erschaffen ein Lernen ermöglicht, das komplexe Fähigkeiten, wie das Erlernen von Mathematik mit einbeziehen kann. Der Instruktionismus stellt sich die Frage zur Verbesserung der Bildung „Wie kann das Lehren verbessert werden?“, wohingegen sich der Konstruktionismus die Frage stellt „Wie kann das Lernen verbessert werden?“. Technologie spielt im Konstruktionismus eine wichtige Rolle. Hier dient sie als „Baumaterial“, um interessante Projekte zu ermöglichen. Dabei ist es auch besonders wichtig das Physische mit dem Virtuellen zu verbinden, so

z.B. das Steuern von Robotern, oder nachdem eine Grafik in Logo erstellt wurden ist, diese auszudrucken und auszumalen. Ebenso nimmt die Technologie auch im Instruktionismus im computergestützten Unterricht ein wichtige Rolle ein.

Im Kontext von Computerspielen spiegelt sich die instruktionelle Sichtweise im Konzept von *Serious Games* wieder, bei welchen Bildung und nicht Unterhaltung das primäre Ziel ist (vgl. [Mic06]). Beim Konstruktionismus ist es *Constructionist Gaming*, welches es dem Spieler erlaubt etwas zu kreieren und eigene Projekte umzusetzen. Eines der bekanntesten Computerspiel in diesem Genre ist *Minecraft*: ein virtuelles Sandbox-Spiel, in dem der Spieler Ressourcen in einer prozedural generierten Welt abbaut und diese kombiniert, um etwas zu erschaffen. Zudem hat *Minecraft* eine große *Modding-Community*, indem die Programmiersprache Java verwendet wird, um das Spiel zu erweitern.

Ein weiterer wichtiger Begriff in dem Zusammenhang ist das Konzept eines Mikrokosmos. Damit ist eine Umgebung gemeint, die Lernende selbstständig entdecken können. Anstatt den Lernenden genaue Anweisungen zu geben, haben sie die Möglichkeit selbst Entdeckungen zu machen, etwas zu erfinden und zu lernen. In diesem Zusammenhang hat Papert die Programmierlernumgebung *MicroWorlds* erschaffen. *MicroWorlds* verwendet die Logo-Programmiersprache und ermächtigt Lernende interaktive Spiele, mathematische und wissenschaftliche Experimente und Simulationen und Geschichten zu kreieren (vgl. [Inc19]).

2.2. Computerspiele

2.2.1. Game Loop

Ein sehr wichtiges Konzept für die Implementierung von Computerspielen ist das Konzept des *Game Loop*. Fast jedes Spiel hat einen *Game Loop*. Ein *Game Loop* wiederholt folgende drei Aktionen fortgehend (vgl. [Nys14]):

1. Benutzereingabe verarbeiten
2. Spielzustand aktualisieren
3. Spielgrafik rendern

Wie schnell ein Game Loop läuft ist entscheidend dafür, wie viele Bilder pro Sekunde gerendert werden. Dies wird als *Frames pro Sekunde* (FPS) bezeichnet.

Ein *Game Loop* kann unterschiedlich implementiert werden. Beispielsweise können die Updates in immer gleichen Zeitintervallen und so häufig wie möglich gerendert werden, um die CPU zu entlasten (vgl. [Nys14]). Eine andere Möglichkeit ist es der Update-Funktion, genau mitzuteilen wie viel Zeit vergangen ist, sodass die Updates die Zeit berücksichtigen können, die vergangen ist, was folgendermaßen implementiert werden kann:

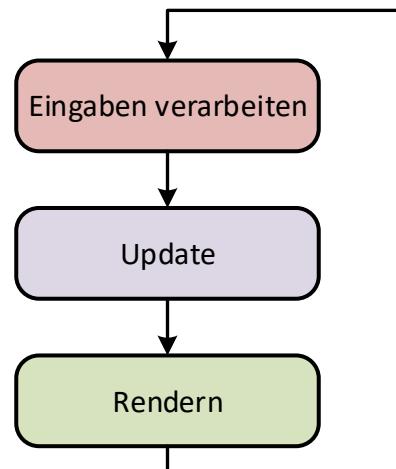


Abb. 2.1. *Game Loop*

```

double lastTime = 0;
while (true)
{
    double current = getCurrentTime();
    double elapsed = current - lastTime;
    lastTime = current;
    processInput();
    update(elapsed);
    render();
}

```

Listing 2.1. Event Loop mit zeitabhängigen Updates

Es wird hier nicht tiefer in unterschiedliche Implementierungen von *Game Loops* eingegangen. Das Konzept ist nur wichtig, um die Implementierungen der Spiele besser zu verstehen.

2.2.2. Koordinatensystem

Bei der 2D-Grafik-Programmierung werden Zeichenobjekte über ein kartesisches Koordinatensystem auf den Zeichenbereich platziert. Dabei gibt es je nach Grafik-Bibliothek verschiedene Konventionen, auf die später verwiesen werden und die hier an dieser Stelle kurz beschrieben werden.

linkshändig oder rechtshändig

In einem traditionellen rechtshändigen kartesischen Koordinatensystem zeigt die x-Achse nach rechts und die y-Achse nach oben. In einem linkshändigen Koordinatensystem hingegen zeigt die x-Achse nach rechts und die y-Achse jedoch nach unten.

Ursprung des Koordinatensystems

Außerdem ist es wichtig zu wissen, wo sich der Ursprung des Koordinatensystem auf der Zeichenoberfläche befindet. Bei rechtshändigen Koordinatensystem befindet sich der Ursprung häufig an der unteren linken Ecke der Zeichenoberfläche. Bei linkshändigen Koordinatensystem befindet sich der Ursprung häufig an der oberen linken Ecke der Zeichenoberfläche. Der Ursprung des Koordinatensystems kann sich allerdings auch in der Mitte befinden.

2.2.3. Sprite

Ein Sprite ist in der 2D-Grafik-Programmierung ein Grafikobjekt, welches von der Grafikkarte über den Hintergrund eingefügt wird (vgl. [Wik21d]).

3. Methodisches Vorgehen

Es soll ein Überblick geschaffen werden, welche Stärken und Schwächen jede der untersuchten *Teaching Languages* aufweist. Deswegen wird eine Punktzahl von -2 bis $+2$ für jeden Bewertungsbereich angegeben. Eine Punktzahl von -2 gibt dabei an, dass die Sprache gegen die Empfehlungen der Bewertungskriterien in dem Bereich designt wurde ist und $+2$ gibt an, dass die Sprache alle Bewertungskriterien des Bereichs besonders gut erfüllt.

Jedes Bewertungskriterium für einen Bereich wird über diese Skala bewertet. Dabei werden die Bewertungskriterien innerhalb eines Bereichs unterschiedlich gewichtet. Aus der Gewichtung und der Bewertung wird dann die Punktzahl für den ganzen Bewertungsbereich berechnen:

- **-2:** Bewertungskriterium wird missachtet.
- **-1:** Bewertungskriterium wird unzureichend erfüllt.
- **0:** Bewertungskriterium wird teilweise erfüllt.
- **1:** Bewertungskriterium wird erfüllt.
- **2:** Bewertungskriterium wird besonders gut erfüllt.

Die Bewertung der *Teaching Languages* nach den Bewertungskriterien erfolgt detailliert im vierten Kapitel. Die Bewertungen werden in einem Evaluationsbogen eingetragen, welche im Anhang zu finden sind (s. Anhang B).

Im Weiteren wird die Auswahl der untersuchten *Teaching Languages*, der implementierten Computerspiele und die Bewertungsbereiche beschrieben und begründet.

3.1. Zielgruppen

Da es um *Teaching Languages* geht, die in der Regel innerhalb des Bildungsweges beigebracht werden, ist eine Unterteilung der Zielgruppen in die Bildungsbereiche sinnvoll. Weil die Bildungssysteme international ziemlich unterschiedlich sind, ist die folgende Unterteilung auf das deutsche Bildungssystem ausgerichtet.

| Zielgruppe | Abkürzung | Altersklasse |
|------------------|-----------|--------------|
| Kindergarten | <i>K</i> | 4 – 6 |
| Primarstufe | <i>P</i> | 5 – 11 |
| Sekundarstufe I | <i>S1</i> | 9 – 17 |
| Sekundarstufe II | <i>S2</i> | 14 – 20 |
| Hochschule | <i>H</i> | > 17 |

Tabelle 3.1. Zielgruppen

In der USA, Australien, Kanada, Türkei und in den Philippinen werden die Bildungsbereiche Kindergarten bis Sekundarstufe II häufig als *K-12* („Kay through 12“) bezeichnet.

net. Dabei bezieht sich das „K“ auf Kindergarten und die zwölf auf die 12. Klasse (vgl. [Wik21c]). Dies nur als Referenz, weil die zitierten Studien diese Bezeichnung häufig verwenden.

3.2. Auswahl der *Teaching Languages*

Bei der Auswahl der *Teaching Languages* wurde sich darauf konzentriert, ein möglichst großes Spektrum verschiedenartiger *Teaching Languages* mit verschiedenen Paradigmen und Zielgruppen abzubilden. Ein Kriterium, das die *Teaching Language* auf jeden Fall erfüllen muss, ist, dass Computerspiele implementiert werden können – die Sprache also ein *Game-Framework* hat. Schlussendlich fiel die Wahl auf Pyret, Quorum, Scratch, Toon-Talk und Game-Changineer. Diese werden im vierten Kapitel noch genauer beschrieben. Eine grobe Einordnung, bei der das Hauptparadigma, die Typisierung und die Zielgruppe genannt wird, ist in Tabelle 3.2 zu sehen. Dabei wird die Zielgruppe als Menge mit den in Abschnitt 3.1 definierten Abkürzungen angegeben. Die eckigen Klammern geben dabei ein Intervall an, in dem mehrere Zielgruppen in aufsteigender Altersfolge hinweg enthalten sind.

| Ausgewählte <i>Teaching Languages</i> | | | |
|--|--|--------------------|-------------------|
| Sprache | Hauptparadigma | Typisierung | Zielgruppe |
| Pyret | funktional | graduell | {S2, H} |
| Quorum | objektorientiert | statisch | {S1, S2} |
| Scratch | visuell, blockbasiert | dynamisch | [K, S1] |
| Toontalk | animiert & Concurrent Constraint Logic Programming | dynamisch | [K, S1] |
| Game-Changineer | Natural Language Programming | dynamisch | {P, S1} |

Tabelle 3.2. Einordnung der ausgewählten *Teaching Languages*

3.2.1. Weitere *Teaching Languages*

Zudem gibt es eine Reihe von weiteren *Teaching Languages*, welche es nicht in die Auswahl geschafft haben.

| Weitere <i>Teaching Languages</i> | | | |
|--|-----------------------|--------------------|-------------------|
| Sprache | Hauptparadigma | Typisierung | Zielgruppe |
| Grace | objektorientiert | graduell | {S2, H} |
| Stride | framebasiert | statisch | {S1} |
| Racket | sprachorientiert | dynamisch | {S2, H} |
| Hedy | graduell | dynamisch | {P, S1} |
| Snap! | blockbasiert | dynamisch | [S1, H] |

Tabelle 3.3. Einordnung der weiteren *Teaching Languages*

Die Besonderheiten dieser Sprachen und eine Begründung, warum diese nicht ausgewählt wurden, werden im Folgenden kurz herausgestellt:

Grace

Auf der ECOOP (*European Conference on Object-Oriented Programming*) haben sich über 20 Lehrer und Forscher zusammengesetzt, um Designprinzipien für eine beginnerfreundliche objektorientierte Programmiersprache herauszuarbeiten (s. [BBN10]). Aus diesen Prinzipien ist dann die objektorientierte Programmiersprache Grace entstanden, welche große Ähnlichkeiten mit Smalltalk hat. Das Designziel war es eine kleine, simple, objektorientierte Programmiersprache zu erstellen (s. [Bla+13]), welche genutzt werden kann, um Anfängern Objektorientierung beizubringen.

Grace wurde nicht ausgewählt, da Grace kein *Game-Framework* hat.

Stride

Stride ist eine framebasierte Programmiersprache, welche in den Entwicklungsumgebungen *Greenfoot* und *BlueJ* integriert ist und Java-ähnlich ist (vgl. [Köl+20]). In einer framebasierten Programmiersprache ist jedes Anweisung oder Struktur eine unteilbares Element und anstatt Text zu bearbeiten werden Frames bearbeitet. So gibt es z.B. einen `while`-Frame und einen `if`-Frame und der Text innerhalb der Frames kann bearbeitet werden. Dies führt dazu, dass keine Syntax-Fehler, wie das Vergessen einer geschlossenen Klammern, gemacht werden können (vgl. [KBA]). Stride wurde nicht ausgewählt, da Stride sehr ähnlich zu Java ist. Die Hauptidee ist die framebasierte Bearbeitung von Quellcode. So kann Stride auch als framebasiertes Java bezeichnet werden.

Racket

Racket ist eine sprachorientierte Programmiersprache, welche auf Scheme basiert. Die Idee hinter Racket als *Teaching Language* ist die, dass Programmiersprachen selbst definiert werden können. So wurden verschiedene aufeinander aufbauende *Student Languages* definiert, welche die Komplexität nach der Empfehlung für Lernaufgaben des 4C-ID inkrementell erhöhen (vgl. [Fin21]):

- 1) **Beginning Student:** Eine kleinere Version von Racket.
- 2) **Beginning Student with List Abbreviations:** Listen können abgekürzt erstellt werden (`list` anstatt von `cons`)
- 3) **Intermediate Student:** Fügt *Local Bindings* und *Higher-Order*-Funktionen hinzu.
- 4) **Intermediate Student with Lambda:** Fügt anonyme Funktionen hinzu.
- 5) **Advanced Student:** Fügt *mutable State* hinzu.

Racket ist auch die Programmiersprache, für die die pädagogische IDE *DrRacket* entwickelt wurden ist und welche in dem Programmier-Lehrbuch *How to Design Programs* (HtDP) verwendet wird (s. [Fel+18]).

Racket wurde nicht ausgewählt, da die verschiedenen *Language-Level* eine Analyse schwierig machen. Außerdem ist die ausgewählte *Teaching Language* Pyret von Racket inspiriert und Shriram Krishnamurti, welcher schon an Racket gearbeitet hat, ist einer der Mitentwickler von Pyret.

Hedy

Hedy ist eine graduelle Programmiersprache, welche in verschiedene *Language-Levels* unterteilt ist. Dabei startet Hedy als Programmiersprache ohne syntaktische Elemente wie Klammern, Doppelpunkte und Einrückungen. Dann ändern sich die Regeln inkrementell pro Level, bis Beginner dann schlussendlich in Python programmieren (vgl. [Her20]).

Die Syntax ist somit Anfangs natürlicher und einfacher und wird dann mit der Zeit komplexer. Hedy wurde nicht ausgewählt, da die Sprache zu Python-ähnlich ist und auch kein *Game-Framework* hat.

Snap!

Snap! (s. [Sna21]) wurde von Scratch inspiriert und es ist eine blockbasierte Programmiersprache, die sich von Scratch dadurch absetzt, dass fortgeschrittene Programmierkonzepte wie Funktionen höherer Ordnung und Metaprogrammierung unterstützt werden und sich *Snap!* somit auch eher für die Erwachsenenbildung eignet.

Snap! wurde nicht ausgewählt, da das simplere Scratch als Repräsentation einer kindgerechten Programmiersprache bevorzugt wurde.

3.3. Auswahl der Spiele

Bei der Spieleanwahl kommt es darauf an möglichst unterschiedliche Spiel-Genres auszusuchen, welche verschiedene Herangehensweisen der Programmierung benötigen. Es ist außerdem vom Vorteil Spiele zu wählen, die bereits bekannt sind, sodass die Erklärung der Spiele gegebenenfalls übersprungen werden kann. Zudem sollten die Spiele ziemlich simpel sein, damit die Implementierung nicht zu viel Aufwand ist.

Es wurde sich für die Spiele *Tic-Tac-Toe*, *Flappy Bird* und *Tamagotchi* entschieden. Somit gibt es ein Strategiespiel (*Tic-Tac-Toe*), ein Jump-And-Run-Spiel (*Flappy-Bird*) und *Tamagochi*, bei dem sich Status-Werte mit der Zeit ändern. Zudem wird für jede Sprache als Einführung in das Game-Framework der Sprache *Bouncing Ball* implementiert. Hier prallt ein Ball an den Rändern des Spielfelds ab.

Es ist wichtig, dass die Spielmechaniken definiert sind, sodass überprüft werden kann, in wie weit das Spiel umgesetzt werden konnte. Hierfür werden Bedingungen definiert, welche genau so umgesetzt werden müssen.

Eine Bedingung gilt für alle Spiele, welche lautet: Nachdem das Spiel vorbei ist, soll es über die Spieloberfläche neu gestartet werden können.

3.3.1. Tic-Tac-Toe

Tic-Tac-Toe ist ein einfaches Zweipersonen-Strategiespiel, bei dem abwechselnd auf einem quadratischen 3x3 Felder großen Spielfeld, Zeichen von zwei Spielern (üblicherweise Kreuz und Kreis) gesetzt werden. Wenn ein Spieler zuerst 3 Zeichen in einer Reihe hat, so hat dieser gewonnen. Sollten in allen Felder voll sein, ohne dass 3 Zeichen in einer Reihe sind, so ist Unentschieden.

Folgende Bedingungen müssen eingehalten werden:

1. Es gibt 9 Felder angeordnet in einem 3x3-Gitter, welche am Anfang leer sind.
2. Wenn auf ein leeres Feld geklickt wird, dann wird das Zeichen des Spielers, der momentan an der Reihe ist, in das Feld gesetzt.
3. Es wird angezeigt, welche Spieler gerade an der Reihe ist.
4. Wenn auf ein volles Feld geklickt wird, dann passiert nichts.
5. Nach einem Spielzug, ist der andere Spieler an der Reihe.
6. Wenn 3 gleiche Zeichen in einer Reihen (horizontal, vertikal oder diagonal) sind, hat der Spieler des letzten Zugs gewonnen. Dieser Spieler wird angezeigt und das Spiel ist vorbei.
7. Wenn alle Felder besetzt sind, ohne dass 3 Zeichen in einer Reihe sind, so wird ein Unentschieden angezeigt und das Spiel ist vorbei.
8. Wenn das Spiel vorbei ist, dann kann kein Zeichen mehr auf ein Feld gesetzt werden.
9. Wenn das Spiel vorbei ist, kann es über die Spieloberfläche neu gestartet werden.

3.3.2. Flappy Bird

Flappy Bird ist ein Spiel, das 2013 das erste Mal in den *App-Stores* erschien und trotz simpler Spielmechanik einen großen Hype erfuhr vlg. ([Wik21b]). In diesem Spiel steuert man einen Vogel, der bei Tastendruck etwas nach oben fliegt. Dann gibt es die Hindernisse. Das sind immer zwei Röhren, welche an der gleichen Position sind und von der eine aus dem Boden kommt und die andere aus dem Himmel ragt, sodass ein Spalt übrig bleibt, wodurch der Vogel fliegen kann. Es kommen endlose dieser Hindernisse hintereinander, wobei der einzige Unterschied ist, dass sich der Spalt immer an einer zufälligen anderen Stelle befindet. Sollte der Vogel eine der Hindernisse berühren, so stürzt er ab und das Spiel ist vorbei. Für jedes erfolgreich überwundene Hindernis, erhält der Spieler einen Punkt. Die x-Position des Vogels ändert sich nicht, die Hindernisse bewegen sich auf ihn zu, sodass die Illusion erschaffen wird, dass er sich in der virtuellen Welt bewegt.

Folgende Bedingungen müssen eingehalten werden:

1. Der Vogel befindet sich anfangs mittig auf der linken Seite der Spielumgebung.
2. Der Vogel fällt standardmäßig.
3. Wenn die Leertaste gedrückt wird, so fliegt der Vogel wieder etwas nach oben.
4. Wenn der Vogel den unteren Spielfeldrand berührt, so ist das Spiel vorbei.
5. Der Vogel kann nicht aus dem oberen Spielfeldrand heraus fliegen.
6. Immer wenn der Vogel nach oben fliegt, neigt er sich nach oben und wenn er fällt, neigt er sich nach unten.
7. Die Hindernisse bestehen aus zwei vertikalen Rechtecken, von denen ein Rechteck aus dem oberen Spielfeldrand und das andere Rechteck aus dem unteren Spielfeldrand kommt und welche an der gleichen x-Position sind. Man kann es sich auch vorstellen wie ein großes Rechteck mit einem Spalt in der Mitte.
8. Der Spalt der Hindernisse ist immer gleich groß.
9. Die Hindernisse bewegen sich vom rechten Spielfeldrand nach links auf den Vogel zu. Ist das Hindernis überwunden, so verschwindet es aus dem linken Spielfeldrand.
10. Es bewegen sich immer neue Hindernisse von rechts in das Spielfeld rein. Es handelt sich also um ein unendliches Level.

11. Der Abstand zwischen den einzelnen Hindernissen ist immer gleich groß.
12. Berührt der Vogel ein Hindernis, so stürzt er ab und kann nicht mehr nach oben fliegen.
13. Für jedes überwundene Hindernis erhält der Spieler einen Punkt. Die Punktzahl wird angezeigt.
14. Wenn das Spiel vorbei ist, wird die Punktzahl nochmal größer in der Mitte der Spielumgebung angezeigt.
15. Das Spiel hat an folgenden Stellen Soundeffekte: Wenn der Vogel hochfliegt, spielt ein kurzer Flieg-Sound ab. Wenn der Vogel abstürzt spielt ein Abstürz-Sound ab. Wenn die Punktzahl erhöht wird, spielt ein Punkte-Sound ab.
16. Wenn das Spiel vorbei ist, kann es über die Spieloberfläche neu gestartet werden.

3.3.3. Tamagochi

Das originale Tamagochi wurde in Japan von dem Unternehmen Bandai entwickelt und kam das erste Mal im November 1996 auf dem Markt und erreichte 1997 an großer Popularität. Es handelt sich dabei um ein kleines uhrengroßes Elektronikspielzeug, bei dem man sich um ein virtuelles Haustier kümmert. Das Haustier hat Bedürfnisse wie schlafen, essen, trinken und Zuneigung und hat sogar eine eigene Persönlichkeit (vgl. [Wik21e]).

Ich überlege mir nun ein Spiel, das sich an dem originalen Tamagochi orientiert, die Spielmechaniken jedoch frei von mir erfunden sind. Ich versuche es nicht allzu kompliziert zu machen. Da ich mir die Spielmechaniken ad-hoc überlege, kann eine gute Spielerfahrung nicht garantiert werden.

Das Haustier hat zwei Status-Werte: Energie und Liebe. Energie und Liebe verringern sich langsam in einem konstanten Tempo. Hat das Tier keine Liebe mehr, dann ist das Spiel vorbei. Ist die Energie unter einem bestimmten Niveau, schläft das Tier und füllt seine Energie wieder etwas auf. Während das Tier schläft, kann nicht mehr mit dem Tier interagiert werden. Energie kann durch Essen wieder aufgefüllt werden. Da gibt es zum einen den Brokkoli, der die Energie auffüllt doch die Liebe verringert. Und es gibt den Kuchen, der auch die Energie auffüllt, das Tier dann aber mit einer bestimmten Wahrscheinlichkeit krank wird. Ist das Tier krank, so verringert sich seine Energie schneller. Die Krankheit hält nur für eine bestimmte Zeit an. Liebe kann aufgefüllt werden, indem mit dem Tier gespielt wird. Das Spielen verbraucht jedoch eine feste Anzahl an Energie.

Folgende Bedingungen müssen eingehalten werden:

1. Die zwei Status-Werte Energie und Liebe habe einen Wertebereich von 0 bis 100, sind am Anfang beide auf 75 und werden angezeigt.
2. Energie und Liebe verringern sich standardmäßig um jeweils 1 pro Sekunde.
3. Wenn auf den Brokkoli geklickt wird, dann verringert sich die Liebe um 10 und die Energie steigt um 15.
4. Wenn auf den Kuchen geklickt wird, dann steigt die Energie um 15.
5. Wenn auf den Kuchen geklickt wird, dann gibt es eine 30%-Chance, dass das Tier krank wird. Die Krankheit dauert 15 Sekunden und wird angezeigt.
6. Während das Tier krank ist, verringert sich seine Energie um 3 pro Sekunde.
7. Während das Tier krank ist, kann es kein Kuchen mehr essen und es kann nicht

mehr mit ihm gespielt werden.

8. Wenn die Energie kleiner gleich 10 ist, schläft das Tier für 10 Sekunden. Dabei regeneriert die Energie um 2 pro Sekunde. Außerdem ist es dann nicht mehr krank.
9. Wenn auf das Herz geklickt wird, dann wird mit dem Tier gespielt. Dabei verringert sich die Energie um 15 und die Liebe erhöht sich um 10.
10. Ist die Liebe auf 0, dann ist das Spiel vorbei.
11. Ist das Spiel vorbei wird die „Lebenszeit“ des Tiers in Sekunden angezeigt.
12. Wenn das Spiel vorbei ist, kann es über die Spieloberfläche neu gestartet werden.

3.4. Bewertungsbereiche

Es gibt schon einige Arbeiten, die sich mit der Bewertung von sogenannten *First Programming Language* (FPL) – Programmiersprachen, die zuerst beigebracht werden sollten – befassen.

So gibt Gupta ([Gup04]) eine gute Zusammenfassung über Kriterien, wie eine Sprache entworfen sein muss, damit sie einfach zu lernen ist. Er kommt zum Ende zum Schluss, dass die Zielgruppe entscheidend für die Auswahl einer Programmiersprache ist.

In einer anderen Arbeit von Farooq u. a. ([Far+14]) wurde ein Framework ausgearbeitet, um FPLs zu vergleichen und zu bewerten. Es kann dann über eine *Score*-Funktion für jede Programmiersprache einen *Score* berechnet werden, der diese bewertet. Dabei ist die Zielgruppe auf die Gruppe der Studenten festgelegt, die in der Universität eine erste Programmiersprache lernen. Die Bewertungskriterien wurden in der Arbeit von Farooq u. a. in zwei Gruppen unterteilt: technische Funktionen (*Technical Features*) und Funktionen der Umgebung (*Environmental Features*). Bei den technischen Funktionen werden höhersprachige Programmiersprachen, welche sicher sind und dem Programmierer z.B. über einen *Garbage-Collector* Arbeit erleichtert, als besonders gut bewertet. Außerdem werden streng typisierte Programmiersprachen besser bewertet als dynamische Sprachen, da diese mehr Sicherheit bieten. Für die Funktionen der Umgebung werden neben einer benutzerfreundlichen IDE besonders die Nachfrage der Industrie berücksichtigt. Da in dieser Bachelorarbeit *Teaching Languages* zielgruppenübergreifend ganzheitlich bewertet werden, die keine Relevanz in der Industrie haben müssen, eignet sich das Framework für die Bewertung nicht.

Da sich die Domaine von *Teaching Languages* angesehen wird, welche von Sprachdesignern designt wurden, die sich schon viele Gedanken in diesem Bereich gemacht haben, können einige Aspekte, die von Farooq u. a. und Gupta genannt wurden sind wie eine *höhere Programmiersprache* und *Sicherheit* (im Sinne von keinen ungültigen Speicherzugriffen usw.) als vorausgesetzt werden.

Die aufgestellten Bewertungskriterien beziehen zum einem die in Abschnitt 2.1 geschaffenen Grundlagen zum Lernen mit ein und zum anderen Kriterien, die in den Arbeiten wie von Gupta und Farooq u. a. zusammengestellt wurden, sowie zusätzliche eigene Überlegungen und auf anderen Arbeiten basierende, die besonders die Unterschiede zwischen den *Teaching Languages* hervorheben. Für die Bewertungskriterien konnten 13 Bereiche identifiziert werden:

1. Einfachheit der Implementierung von Computerspielen
2. Erlernbarkeit nach Konstruktionismus
3. Inkrementelle Einführung
4. Intuitive Syntax und Natürlichsprachlichkeit
5. Feature Uniformity und Orthogonalität
6. Unterstützung der wichtigsten Programmierkonzepte & Datenstrukturen
7. Programmierumgebung (IDE)
8. Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplin (*Clean Code*)
9. Fehlermeldungen
10. Ein hohes Abstraktionsniveau für Datentypen
11. Anwendungsbereiche und Interoperabilität mit Technologien und anderen Programmiersprachen
12. Dokumentation & Community Support

3.4.1. Einfachheit der Implementierung von Computerspielen

Hier wird genauer auf die Implementierung der Spiele eingegangen. Es wird analysiert, ob das Spiel mit allen Anforderungen implementiert werden konnte, wie aufwendig die Implementierung ist, wie viele Zeilen Code geschrieben werden musste und wo die Implementierung besonders kompliziert ist und/oder *Hacks* eingesetzt werden mussten. Umso geringer der Aufwand und die Komplexität, desto besser wird die *Teaching Language* bewertet.

Zudem gibt es Elemente, die die Implementierung eines Computerspiels vereinfachen. Diese Elemente sind ein grafischer Editor, ein richtig ausgewähltes Koordinatensystem und richtige Ankerpunkte für die Positionierung von Sprites. Obendrein gibt es weitere Elemente, die für die Implementierung von Computerspielen nicht fehlen dürfen. Dies sind das Abspielen von Audio und das Animieren von Sprites. Schließlich muss die *Teaching Language* ein Framework bieten, indem der *Game Loop* abstrahiert wird und der *Game Loop* nicht explizit implementiert werden muss, sondern sich ganz auf die Implementierung des Spiels konzentriert werden kann. In den weiteren Abschnitten wird auf die einzelnen Elemente näher eingegangen.

Grafischer Szenen-Editor

Ein grafischer Szenen-Editor vereinfacht die Implementierung von Computerspielen, indem Sprites platziert, rotiert und skaliert werden können. Die Sprites über den Code zu setzen ist etwas aufwendiger, da es länger dauern kann, bis man die richtige Position, Größe und Rotation des Sprites über den Code eingestellt hat.

Koordinatensystem

In der 2D-Grafik-Programmierung wird häufig ein linkshändiges Koordinatensystem verwendet, da dieses direkt mit der Grafik-Hardware übereinstimmt, welche Bildschirmkoordinaten verwendet. Am intuitivsten ist es allerdings, wenn Beginner die gelernten Konzepte

aus dem Mathematikunterricht anwenden können, es sich also um ein rechtshändiges Koordinatensystem handelt. Der Ursprung des Koordinatensystems kann dann entweder in der unteren linken Ecke oder in der Mitte sein. Obwohl ein Ursprung in der Mitte aus mathematischer Konvention heraus durchaus Sinn macht, ist ein Ursprung in der unteren linken Ecke praktischer, da so einfacher Code mit modularer Arithmetik geschrieben kann, um bspw. die Figur wenn sie aus dem rechten Rand rausgeht, wieder nach links zu setzen (s. Listing 3.1). Die Drehrichtung, um Sprites zu rotieren, soll nach mathematischer Konvention erfolgen. Bei einem rechtshändigen Koordinatensystem, ist diese gegen den Uhrzeigersinn und bei einem linkshändigen im Uhrzeigersinn (vgl. [Wik21a]).

```
if player.x > WIDTH / 2
    player.x = - WIDTH / 2
end
player.x = player.x % WIDTH
```

Listing 3.1. Ursprung in der Mitte

Listing 3.2. Ursprung unten links

Ankerpunkt

Mit dem Ankerpunkt eines Sprites ist gemeint, wie es relativ zu seiner Position platziert wird. Bei einem Ankerpunkt von $(0,0)$ erfolgt die Positionierung in einem rechtshändigen Koordinatensystem über die untere linke Ecke des Sprites und bei einem linkshändigen über die obere linke Ecke des Sprites. Wenn der Ankerpunkt $(0.5,0.5)$ ist, ist der Punkt mit dem das Sprite positioniert wird, die Mitte des Sprites.

Der Ankerpunkt kann entweder unveränderbar (fixiert) oder veränderbar (dynamisch) sein, wobei er dann beliebig eingestellt werden kann. Für einen fixierten Ankerpunkt machen nur die Positionen $(0,0)$ oder $(0.5,0.5)$ Sinn, da alle anderen Ankerpunkte wie $(1,1)$ eher verwirrend sind. Die Optionen des fixierten Ankerpunkts mit $(0,0)$ und dynamischen Ankerpunkt werden nun am Beispiel von *Wrapping* erkundet, wobei der Ursprung des Koordinatensystems in der unteren linken Ecke ist. Hierbei soll der Spieler, wenn er sich aus dem rechten Rand bewegt, sich am linken Rand wieder reinbewegen und andersherum. Der Spieler soll also nicht teleportiert werden.

```
# wrap links
if player.x < -player.width
    player.x = WIDTH
end
```

```
# wrap rechts
if player.x > WIDTH
    player.x = -player.width
end
```

Listing 3.3. Ankerpunkt ist $(0,0)$

```
# wrap links
if player.x < -player.width / 2
    player.x = WIDTH + player.width / 2
end
```

```
# wrap rechts
if player.x > WIDTH + player.width / 2
    player.x = -player.width / 2
end
```

Listing 3.4. Ankerpunkt ist $(0.5,0.5)$

```
# wrap links
player.setAnchor(1, 0)
if player.x < 0
    player.setAnchor(0, 0)
    player.x = WIDTH
end
```

```
# wrap rechts
player.setAnchor(0, 0)
if player.x > WIDTH
    player.setAnchor(1, 0)
    player.x = 0
end
```

Listing 3.5. dynamischer Ankerpunkt

Zwischen dem Code mit dem Ankerpunkt an (0,0) (s. Listing 3.3) und dem Ankerpunkt in der Mitte (0.5,0.5) (s. Listing 3.4) fällt auf, dass sie ziemlich gleich sind. Der Code mit dem Ankerpunkt in der Mitte ist jedoch etwas aufwendiger, da immer die halbe Spielerweite berücksichtigt werden muss.

Bei dem dynamischen Ankerpunkt (s. 3.5) ist das Praktische, dass mit dem Wert 0 die linke Seite und mit dem Wert WEITE das *Wrapping* auf der rechten Seite überprüft werden kann. Jedoch ist der Code aufwendiger, da der Anker gesetzt werden muss und man muss ähnliche Gehirngymnastik machen wie bei den anderen Codes. Ein dynamische Ankerpunkt bringt zwar viel Flexibilität, jedoch macht er es unglaublich kompliziert, wenn verschiedene Sprite-Positionen miteinander verglichen werden sollen, da berücksichtigt werden muss, wie der Anker bei den anderen Sprites gesetzt wurde. Deswegen ist ein dynamischer Ankerpunkt schlecht.

Wenn es darum geht Überlappungen von Sprites untereinander und Bildschirmpositionen wie Rändern herauszufinden, ist der Ankerpunkt (0,0) dem Ankerpunkt (0.5,0.5) überlegen. Geht es allerdings darum, Elemente zentriert zu platzieren, bspw. in die Mitte des Bildschirms, dann ist der Ankerpunkt (0.5,0.5) meistens praktischer. Da es für die Positionierung von Sprites im Idealfall schon einen grafischen Editor gibt und die initiale Position der Sprites dann nicht über Code gesetzt werden muss, ergibt insgesamt der Ankerpunkt (0,0) mehr Sinn.

GUI-Bibliothek

Desweiteren ist es für die Implementierung von Computerspielen sehr nützlich, wenn es eine GUI-Bibliothek gibt. Für einfache Spiele reicht dabei eine minimale GUI-Bibliothek, mit GUI-Elementen wie Buttons und Textboxen.

Abstraktion des *Game Loops*

Die drei Elemente des *Game Loops* (Update, Eingaben verarbeiten und Rendern) müssen durch ein Framework abstrahiert werden. In objektorientierten Sprachen wird ein Framework häufig so implementiert, dass von einer Eltern-Klasse wie „Game“ oder „GameObject“ geerbt wird und dann Methoden wie `Update(float deltaTime)` und `Start()` (für die Initialisierung) überschrieben werden. Dies wird bspw. in der Game-Engine *Unity* so gemacht.

Das Reagieren auf Eingabe wird häufig durch *Event-Driven Programming* implementiert, wobei ein bestimmtes Ereignis registriert wird, indem der Game-Engine ein *Callback* übergeben wird. Wenn das Ereignis dann eintritt, ruft die Game-Engine den *Callback* mit den Informationen zum Event auf und im *Callback* kann das Event vom Programmierer behandelt werden.

Das Rendern selbst muss in den meisten Game-Frameworks nicht selbst aufgerufen werden. Grafische Objekte werden einer Szene hinzu gefügt und werden dann an ihrer Position gerendert. Allerdings gibt es auch geringere Abstraktionen, wo explizite *Draw*-Kommandos aufgerufen werden müssen. Ein Beispiel dafür ist *Processing* ([Pro21]), wo es *Draw*-Kommandos, wie `circle(x, y, extent)` für einen Kreis oder `rect(x, y, w, h)` für ein Rechteck gibt.

Eine höhere Abstraktion macht es einfacher Spiele zu implementieren. Deswegen wird eine höhere Abstraktion als besser bewertet.

Bewertungskriterien

Insgesamt soll sich ein Bild ergeben, wie gut sich die Programmiersprache dafür eignet Computerspiele zu programmieren.

1. Konnten die Spiele mit all ihren Anforderungen implementiert werden?
2. Wie aufwendig ist die Implementierung?
3. An welchen Stellen ist die Implementierung kompliziert und mussten *Hacks* eingesetzt werden?
4. Gibt es einen grafischen Szenen-Editor, um Sprites (und GUI-Elemente) zu platzieren?
5. Ist das Koordinatensystem und die Drehrichtung nach mathematischer Konvention?
6. Ist der Ankerpunkt der Sprites gut gesetzt?
7. Gibt es eine GUI-Bibliothek?
8. Können Audio-Clips abgespielt werden?
9. Können Sprites animiert werden?
10. Wie wird der *Game Loop* abstrahiert? Wie hoch ist die Abstraktionsebene?

3.4.2. Erlernbarkeit nach Konstruktionismus

Es wird bewertet, wie gut sich die Sprache zum Erlernen nach der Lerntheorie des Konstruktionismus eignet. Dabei wird auch untersucht, wie viele Mikrokosmen die Sprache als motivierende Einführungen bietet. Im Speziellen wird als Beispiel für einen Mikrokosmos untersucht, ob die *Teaching Language* eine einfache Möglichkeit enthält *Turtle*-Grafiken zu erstellen, da herausgefunden wurde, dass sich *Turtle*-Grafiken als motivierende Einführung in die Programmierung besonders gut eignen (vgl. [BMK20]). Zudem wird geschaut, wie viele unterschiedliche Medien und Technologien eingebunden werden können. Insbesondere wird darauf geachtet, dass nicht nur virtuelle, sondern auch physische Medien wie z.B. Roboter eingebunden werden können.

Daneben ist das Konzept einer selbstoffenbarenden und offenen Umgebung ziemlich wichtig, wie Kahn anmerkt (vgl. [Kah04b]). Ist die Umgebung so gestaltet, dass der Lerner selbstständig Konzepte erlernen kann? Nach Kahn helfen besonders gut Animationen und Soundeffekte, um eine Umgebung selbstoffenbarend zu gestalten. Eine Umgebung komplett selbstoffenbar zu gestalten ist jedoch ziemlich schwierig. Deswegen sollten es auch Anleitungen geben, da wo es umbedingt notwendig ist. Dies ist auch beim Design von Videospielen ziemlich wichtig. Besonders viel Spaß macht es, Spielmechaniken selbst zu entdecken und nur da Anleitung zu haben, wo es umbedingt nötig ist.

Hinzu kommt noch das Konzept einer sicheren Umgebung. Nach Kahn ist eine Umgebung sicher, wenn Beginner sie entdecken können, ohne permanenten Schaden zu hinterlassen (vgl. [Kah04b]).

Zuletzt ist es hilfreich, wenn die Programmiersprache dynamisch typisiert ist, sodass mehr herumexperimentiert werden und schneller Prototypen erstellt werden können. Es nimmt zudem viel Zeit und Disziplin in Anspruch das Typsystem einer statisch typisierten Programmiersprache zu lernen und zu verstehen (z.B. *Generics*).

Bewertungskriterien

Somit ergibt sich folgende Liste von Bewertungskriterien:

1. Wie viele und welche Mikrokosmen als motivierende Einführung gibt es?
2. Werden *Turtle*-Grafiken unterstützt und wenn ja wie gut?
3. Wie viele unterschiedliche Medien und Technologien können eingebunden werden:
 - a) digitale Medien: Grafiken, Sound, Musik, Video, 3D-Objekte, ...
 - b) physische Medien: Roboter z.B. *Lego Mindstorms*, Mikrocontroller, Lichter, Motoren
4. Wie selbstoffenbarend ist die Umgebung?
5. Ist die Umgebung sicher?
6. Ist die Programmiersprache dynamisch typisiert?

3.4.3. Inkrementelle Einführung

Eine *Teaching Language* sollte so designt sein, dass die Konzepte der Sprache nach und nach eingeführt werden können. Es sollte möglichst einfach sein, ein simples Programm wie „Hello World“ zu erstellen. Dabei sollten nicht zu viele Konzepte benötigt werden, um ein simples Programm zu verstehen. Java ist z.B. eine Programmiersprache, die dieses Kriterium überhaupt nicht erfüllt. „Hello World“ sieht in Java folgendermaßen aus:

```
public class Main {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

Listing 3.6. „Hello World“ in Java

Um die Syntax des „Hello World“-Programms in Java voll und ganz zu verstehen, wird viel Wissen der objektorientierten Programmierung verlangt, welches für die Ausgabe an sich nicht von Belang ist:

- Was ist eine Klasse?
- Was und wofür sind Zugriffsmodifizierer?
- Was ist eine Methode?
- Was ist die Hauptmethode `main`?
- Was ist eine statische Methode?
- Was sind Parameter?
- Was bedeuten die Parameter, die der Hauptmethode übergeben werden?

- Auf `System.out.println("Hello World")!` bezogen:
 - Was ist `System`?
 - Was ist ein statisches Feld? (`.out`)
 - Wie wird eine Methode auf einem Objekt aufgerufen? (`.println("Hello World")`)

Hinzu kommt, dass in Java für eine simple Eingabe ein Objekt der Klasse `Scanner` erstellt und dann auch noch Instanziierung verstanden werden muss.

Wenn man dies nun mit dem „Hello World“ in Python vergleicht ...

```
print("Hello World!")
```

Listing 3.7. „Hello World“ in Python

... sieht man schnell, was für Anfänger weniger überfordernd ist.

Um die Einfachheit für die *Teaching Languages* zu analysieren, wird für jede Sprache ein simples Programm geschrieben. Neben der Einfachheit der Ausgabe, soll zudem die Eingabe getestet werden. Deshalb wird „Hello World“ so modifiziert, das dieses den Namen des Nutzers vorher abfragt und den Nutzer dann grüßt. Im Weiteren wird auf dieses Programm mit dem Namen „Hallo Nutzer“ verwiesen.

Darüber hinaus wird untersucht, ob die weiteren Konzepte der *Teaching Languages* schrittweise eingeführt werden können. Wie das 4C-ID angibt, ist ein effektives Lernen möglich, wenn die Schwierigkeit der *Learning Tasks* sich nach und nach steigert.

Bewertungskriterien

1. Wie einfach ist das Programm „Hallo Nutzer“?
 - a) Einfachheit der Ausgabe
 - b) Einfachheit der Eingabe
2. Können die Konzepte der Programmiersprache nach und nach eingeführt werden?

3.4.4. Intuitive Syntax und Natürlichsprachlichkeit

Intuitivität der Syntax

Die Syntax einer Programmiersprache zu beherrschen ist eine der ersten Herausforderungen, die sich Programmieranfänger stellen müssen. Eine intuitive und natürliche Syntax hilft, die intrinsische kognitive Belastung einer Programmiersprache herabzusetzen und sie besser erlernbar zu machen. Dies wurde empirisch in einer Studie von Stefik und Siebert [SS13] belegt, die im Folgenden genauer beschrieben wird. Die Studie hatte zwei Forschungsfragen:

1. Welche Wörter und Symbole finden Beginner besonders intuitiv (oder unintuitiv) in einer Allzweck-Programmiersprache?
2. Können Beginner, welche Programmiersprachen das erste Mal benutzen einfache Programme akkurater schreiben, wenn sie alternative Programmiersprachen benutzen?

Um die erste Frage zu beantworten, wurden zwei Studien an Studenten durchgeführt. In der ersten Studien sollten Beginner sowie Experten die Intuitivität von verschiedenen

Programmiersprach-Konstrukten bewerten. In der zweiten Studie sollten Beginner die Intuitivität von größeren Programmkonstrukten in neun verschiedenen Programmiersprachen (C++, Java, Smalltalk, PHP, Perl, Ruby, Go, Python und Quorum) bewerten.

Für die zweite Forschungsfrage wurden zwei weitere empirische Studien durchgeführt. In beiden von ihnen haben Beginner in den Sprachen Ruby, Java, Perl, Python, Randomo und Quorum programmiert. Dabei ist Quorum eine der *Teaching Languages*, die in dieser Arbeit untersucht wird, und Randomo ist eine Sprache, die dadurch entstanden ist, syntaktisch mit Quorum zu starten und die Schlüsselwörter dann mit zufällig generierten Zeichenfolgen zu ersetzen.

Die Studien fanden heraus, dass Variationen in der Syntax für Beginner einen Unterschied machen. Und spezifischer, dass

1. einige syntaktische Entscheidungen in bekannten Programmiersprachen intuitiver als andere sind,
2. Variationen in der Syntax die Fehlerfreiheit beeinflussen,
3. dass die Merkmale der Syntax identifiziert werden können, welche für Beginner Probleme bereiten.

Die Syntax-Konstrukte in der ersten Studie wurden in sechs Bereiche unterteilt, deren wichtigsten Ergebnisse ich im Folgenden kurz zusammenfasse.

1) Typen und Operatoren: Für den Integer-Typen bewerteten Programmierer das Wort `integer` als am meisten intuitiv und Nicht-Programmier hingegen die Wörter `number` und `integer` als gleich intuitiv. Bei Strings bewerteten Nicht-Programmierer die Wörter `characters`, `text` und `charstring` als sehr intuitiv und Programmierer die Wörter `string`, `text` und `charstring`. Bei Booleans war die Bewertung bei Nicht-Programmierern sehr inkonsistent. Zu den besten bewerteten Wörtern gehörten da `condition`, `logic` und `binary`.

Bei den Operatoren wurde herausgefunden, dass bei Zuweisungen das Symbol `=` für beide Gruppen am intuitivsten ist. Für die Verbindungen von Strings wurden die Symbole `+`, `&` und `_` für beide Gruppen als intuitivsten bewertet. Bei dem Modulus-Operator haben die Nicht-Programmierer `x = remainder of 7 / 2` als intuitiver bewertet, verglichen mit `x = 7 % 2`, was die Programmierer am intuitivsten fanden.

2) Kontrollfluss und Boolesche Operatoren: Für Bedingungen haben beide Gruppen das Wort `if` als am meisten intuitiv bewertet. Für Schleifen hingegen haben Nicht-Programmierer die Wörter, die in den meisten Programmiersprachen verwendet werden `for`, `while` und `foreach` als am unintuitivsten eingestuft und stattdessen die Wörter `again` und `loop` am intuitivsten bewertet. Bei den Programmierer waren es die Wörter `repeat` und `loop`.

Um Gleichheit zu testen, fanden Programmierer unüberraschenderweise `x == y` am intuitivsten. Nicht-Programmierer jedoch das einzelne `x = y` oder `x is y` am intuitivsten. Für das *logische Und* fanden beide Gruppen das Wort `und`, gefolgt von den Symbolen `&` und `&&` am intuitivsten. Gleichermaßen für das *logische Oder* das Wort `oder`. Für das *exklusive Oder* (XOR) wurde bei beiden Gruppen `either x or y` und `only x or only y` am höchsten bewertet.

3) Datenstrukturen: Bei Arrays zeigten Nicht-Programmierer keine Bevorzugungen

und Programmierer fanden den Ausdruck `x[]` am intuitivsten, um ein Array `x` zu erstellen. Ebenso um auf einen Index zuzugreifen, fanden die Programmierer `x[5]` am intuitivsten.

Es wurde auch die Syntax für *Generics* bewertet. Hier zeichnete sich jedoch auch keine Präferenz ab.

- 4) **Objektorientierte Programmierung:** Für das Konzept einer Klasse fanden Programmierer sowie Nicht-Programmierer die Wörter `object` und `structure` am intuitivsten. Für Vererbung wurde das Konstrukt `truck is a vehicle` (also über `is a`) am intuitivsten bewertet. Für das Konzept einer Elternklasse, haben beide Gruppen die Wörter `source`, `parent` und `foundation` am besten bewertet. Für die Selbstreferenz wurden die Wörter `self`, `myself` und `me` bei beiden Gruppen am besten bewertet. Für das Konzept von Methoden haben beide Gruppen die Wörter `procedure`, `operation` und `action` als relativ intuitiv bewertet. Um ein Ergebnis in einer Methode zurückzugeben, haben Nicht-Programmierer die Wörter `provide`, `report` und `return` am besten bewertet und Programmierer hingegen nur das Wort `return`. Für Konstanten wurde das Wort `constant` bei beiden Gruppen am besten bewertet. Nicht-Programmierer bewerteten zudem das Wort `permanent` als gleich intuitiv. Um Bibliotheken einzubinden, fanden Nicht-Programmierer die Wörter `use` und `obtain` am intuitivsten und Programmierer die Wörter `import`, `include` und `use`. Für Methodenaufrufe fanden Nicht-Programmierer zudem die Syntax `Window:close` am intuitivsten. Für das Konzept eines *Null-Pointers* bewerteten beide Gruppen das Wort `undefined` am besten. Beim Fehler-Handling (also `try`, `catch` und `throw`) fanden beide Gruppen für das Konzept von `try` die Wörter `check` und `test` am intuitivsten, für `catch` die Wörter `detect`, `error` und `problem` und für `throw` die Wörter `error`, `fix` und `alert`. Für das Konzept von `finally` wurde das Wort `finally` bei beiden Gruppen als schlecht bewertet während das Wort `regardless` sehr gut bewertet wurde.
- 5) **Eingabe, Ausgabe und Kommentare:** Für die Eingabe haben beide Gruppen die Wörter `input` und `request` als intuitiv bewertet und für die Ausgabe `display` und `output`.
- Für Kommentare fanden Nicht-Programmierer die Wörter `note` und `comment` am intuitivsten und Programmierer das Symbol `//`.
- 6) **Aspektorientierte Programmierung:** Aspektorientierte Programmierung ist dafür da, um sogenannte „cross-cutting concerns“ gut zu organisieren. Es sind damit generische Funktionalitäten gemeint, die über objektorientierte Grenzen hinweggehen, wie z.B. *Logging* (vgl. [Den+11]). Hier fanden die Autoren, dass für das Konzept des *Pointcuts* das Wort `target` ganz gut bewertet wurde. Aber insgesamt wurde von der Terminologie des aspektorientierten Programmierens sehr wenige Begriffe bei den Anfängern als gut bewertet.

Bei der zweiten Studie, bei der die Intuitivität größerer Programmkonstrukte in verschiedenen Programmiersprachen bewertet wurde, konnte bestätigt werden, dass einige Konstrukte in Programmiersprachen subjektiv als intutiver als die gleichen Konstrukte in anderen Programmiersprachen bewertet werden. Dabei wurde die Programmiersprache Quorum

als signifikant mehr intuitiv als die Programmiersprachen Go, C++, Perl, Python, Ruby, Smalltalk und PHP bewertet und erreichte eine ähnliche Signifikanz wie Java.

Die Autoren diskutieren ihre Ergebnisse und geben zu, dass sie Intuitivität nicht formal definiert haben und sie immer abhängig vom Kontext und der Kultur ist. Doch sie gehen davon aus, dass Intuitivität in Programmiersprachen erkennbare Merkmale hat. So scheint es, dass Individuen Wörter mit einer gut definierten Bedeutung im Englischen wie z.B. `repeat`, `undefined` intuitiver als metaphorische Namen wie `throw` und `catch` bewerten. Außerdem werden Symbole in der Programmierung, deren Bedeutung mit der Bedeutung in der englischen Sprache in Konflikt treten, wie z.B. der Punkt-Operator für den Methodenaufruf als weniger intuitiv bewertet (vgl. [Den+11]).

Für die zweite Forschungsfrage wurde in einer dritten und vierten Studie die Fehlerfreiheit von Nicht-Programmieren bewertet, die in verschiedenen Programmiersprachen (Ruby, Java, Perl, Python, Randomo und Quorum) aufgeteilt wurden und mit Code-Beispielen als Hilfestellung in der jeweiligen Programmiersprache versucht haben, sechs Programmieraufgaben zu lösen. Dabei wurde die Hilfestellung der Code-Beispiele für die letzten drei Programmieraufgaben entfernt. Die Programmiersprache *Randomo*, die die Gleiche ist wie Quorum, nur dass die Schlüsselwörter mit zufälligen Zeichen ersetzt wurden sind, wurde von der Idee der randomisierten kontrollierten Studien mit Placebo, wie sie auch in der medizinischen Forschung verwendet wird, inspiriert. (vgl. [Den+11]).

Die zwei Studien waren gleich aufgebaut, nur dass die zweite mit mehr Teilnehmern durchgeführt wurde (73 Teilnehmer und bei der vorherigen Studie 19). Um die Fehlerfreiheit zu bewerten wurde als Technik eine *Token Accuracy Map* (TAM) verwendet, welche ein Einschätzung der Fehlerhäufigkeit je Token liefert. Die Studien zeigten große Unterschiede zwischen der Fehlerfreiheit bei verschiedenen Programmiersprachen. Zum Beispiel wurde herausgefunden, dass Quorum-Benutzer statistisch akkurate programmieren konnten als Perl-Benutzer, hingegen Perl-Benutzer nicht akkurate programmieren konnten als Randomo-Benutzer.

Über die zweite Studie konnte bestätigt werden, dass in einigen Programmiersprachen akkurate programmiert werden konnte, als in anderen. So konnte in Quorum, Python und Ruby akkurate programmiert werden als in Java und Perl.

Zusammenfassend kann gesagt werden, dass die Intuitivität der Syntax für eine *Teaching Language* umbedingt berücksichtigt werden sollte, da aus den Studien hervorgeht, dass eine intuitive Syntax die Syntax-Barriere für Anfänger heruntersetzt. Eine intuitivere Syntax senkt die intrinsische kognitive Belastung der Programmiersprache.

Es zeichnet sich jedoch auch ein Problem ab, wenn eine Sprache ausschließlich intuitiver Syntax designet wird. Das Lernen bestimmter Begrifflichkeiten und Fachausdrücke gehören zur Programmierausbildung für die Kommunikation innerhalb des Fachbereichs hinzu. In Quorum wird beispielsweise das Wort `text` für den String-Datentyp verwendet. Die gängige Definition von Text beinhaltet jedoch, dass dieser eine Art von informativer Nachricht sendet (vgl. [Wik21m]), was bei einer Zeichenfolge nicht der Fall sein muss. Ein anderes Beispiel ist, dass Nicht-Programmierer die Wörter `condition`, `logic` und `binary` für den Booleschen Datentypen am intuitivsten bewerteten. Hier fehlt den Nicht-

Programmieren der theoretische und geschichtliche Hintergrund zur booleschen Algebra, die nach den Mathematiker George Boole benannt wurde. Auch wenn die genannten Wörter am intuitivsten erscheinen, sollte für das Sprachdesign berücksichtigt werden, dass es computertheoretisches Vokabular gibt, welches in das Repertoire jedes Programmierers gehören. In einem Programmierkurs sollten neben dem Programmieren an sich (*Learning Task*), die theoretischen Konzepte (*Supportive Information*) beigebracht werden (s. 4C-ID in Abschnitt 2.1.2). Dieses Vokabular ist besonders für Datentypen am relevantesten, hat für Syntax-Konstrukte, die für den Kontrollfluss zuständig sind, jedoch kaum eine Bedeutung mehr. Für die Syntax, die den Kontrollfluss betrifft, geht es darum intuitive und natürlichsprachliche Syntax auszuwählen, die ein möglichst gute Schnittstelle zwischen Mensch und Maschine erschafft.

Ein weiterer Nachteil intuitiver Syntax ist, dass die Schlüsselwörter, die für die Syntax verwendet werden, reserviert sind und für keine Bezeichner mehr verwendet werden können. Besonders bei Wörtern wie `text` und `action`, kann man sich leicht Anwendungsbeispiele überlegen, die diese Wörter als Bezeichner verwenden würden wollen. Bspw. könnte `action` als Variablenamen bei der Programmierung eines Spiels, bei dem der Spieler bestimmte Aktionen ausführen kann, verwendet werden. Moderne Allzweck-Programmiersprachen wie C# haben das Problem erkannt und haben sogar Syntax, um reservierte Schlüsselwörter als Bezeichner verwenden zu können. In C# kann man ein '@' vor einem reservierten Schlüsselwort schreiben, um ihn als Bezeichner verwenden zu können.

Solche Features muss eine *Teaching Language* nicht unbedingt beinhalten, da es sich um einen speziellen Ausnahmefall handelt. Alternativ kann auch ein Präfix vor den Bezeichner geschrieben werden, was jedoch die Lesbarkeit des Codes verringern kann. Auch wenn es sich um ein Problem handelt, das nicht allzu oft vorkommt, sollte es trotzdem im Sprachdesign berücksichtigt werden.

Zudem ist zu diskutieren, ob der positive Effekt der intuitiven Syntax im Englischen auf die Erlernbarkeit für Personen, die Englisch nicht als Muttersprache haben, nicht drastisch gesenkt wird. Auf die Sprachbarrieren für Nicht-Englisch-Muttersprachlern möchte ich im folgenden eingehen.

Englisch als Sprachbarriere

In einer groß angelegten Studie von Philip Guo [Guo18] wurden die Sprachbarrieren für Nicht-Englisch-Muttersprachler (das sind 95% der Weltbevölkerung) untersucht. Die Studie wurde in Form einer Umfrage mit 840 Antworten in 74 verschiedenen Muttersprachen durchgeführt. Folgende Barrieren konnten identifiziert werden:

- Probleme Lernmaterial zu verstehen
- Probleme mit der technischen Kommunikation
- Probleme Code zu lesen
- Probleme Code zu schreiben
- Schwierigkeit Englisch und Programmieren gleichzeitig zu lernen

Wenn Programmieren gleichzeitig mit Englisch gelernt werden muss, wird die extrinsi-

sche kognitive Belastung erhöht, da Konzepte mental in die eigenen Sprache umgewandelt werden müssen (vgl. [Guo18]).

Guo schlägt ein *learner centered* Design vor, welches die Motivationen und Frustrationen des Lernenden als Startpunkt für das Design von Lösungen nimmt (vgl. [Guo18]). So könnten z.B. englischsprachige Einführungsmaterialien in die Programmierung, vereinfachtes Englisch, mehr Bilder und Multimedien, kulturagnostische Code-Beispiele und eingebaute Wörterbücher verwenden werden. Weiterhin schlägt Guo folgende Ideen vor: zweisprachiges Paarprogrammieren, internationalisierte Code-Beispiele, IDE-Plugins, die Benutzern helfen bessere Bezeichner im Code zu wählen und Browser- und IDE-Erweiterungen, welche kontextuell relevante Englisch-Lernaufgaben zum jeweiligen Programmier-Arbeitsablauf anbieten.

Eine weitere Möglichkeit, um Programmieren für Nicht-Englisch-Muttersprachler einfacher zu gestalten, ist die Idee der Nicht-Englisch basierten Sprachen.

Da gibt es zum einem internationale Programmiersprachen, wie ALGOL68, dessen Standard die Internationalisierung der Sprache ermöglicht (vgl. [Wik21i]). Hier gibt es mehrere Versionen dieser Programmiersprachen in mehreren natürlichen Sprachen.

Zudem gibt eine Vielzahl von Programmiersprachen, die für eine bestimmte natürliche Sprache entwickelt wurden (s. [Wik21i]). Der Vorteil dieser Programmiersprachen gegenüber den internationalen Lösungen ist, dass diese genau an die natürliche Sprache angepasst werden können. Ein Beispiel ist *ChaScript*, eine bengalische Programmiersprache, die ECMA-Script als Zwischensprache verwendet (vgl. [Ahm+16]). Es handelt sich dabei um ECMA-Script, bei dem die Schlüsselwörter durch bengalische Wörter und die arabischen Zahlen durch bengalische Zahlen ersetzt wurden. In Benutzerfragen fanden 59% der Benutzer es einfacher Programmierkonzepte in *ChaScript* zu lernen, während 25% meinten es würde keinen Unterschied machen und der Rest (15,38%) fanden es schwerer.

Internationalisierte Programmiersprachen bieten den Vorteil, dass diese leichter für Nicht-Englisch-Sprachige zu erlernen sind, was folgende Studie belegt (s. [DH17]) (Da die Studie in der Programmiersprache *Scratch* durchgeführt wurde, wird in Abschnitt 4.3 noch näher darauf eingegangen).

Der Nachteil ist jedoch, dass Nicht-Englische-Programmiersprachen nicht weit adaptiert sind und eine internationale Kollaboration nicht möglich ist.

Ruby und David [RD16] beschreiben, dass die englische Sprache eine Wissensbarriere (*Knowledge Barrier* oder auch *Knowledge Divide*) für Nicht-Englischsprachige in der Programmierung darstellt. Mit *Knowledge Divide* ist gemeint, dass der Zugriff und die Aufnahme von Wissen ungleich verteilt ist. Englischsprachige Muttersprachler haben Zugriff auf mehr Ressourcen und können das Wissen schneller aufnehmen als Nicht-Englischsprachige. Das linguistische Wissen, das Schüler schon haben wird nicht berücksichtigt. Wenn es Schülern erlaubt wäre ihr existierendes Sprachwissen schon anzuwenden, kann der *Knowledge Divide* verringert werden (vgl. [RD16]). Ein anderes Beispiel, das einen *Knowledge Divide* erzeugt, ist der weltweite ungleiche Zugang zu Endgeräten und dem Internet.

Ruby und David schlagen als Lösung Programmiersprachen mit *Natural Language Neutrality* (NLN) vor. Dabei ist NLN eine Methode, die Methoden und Werkzeuge zur Ver-

fügung stellt, die Individuen verschiedener Muttersprachen ermöglichen in einer gemeinsamen Umgebung zu lernen, üben und zu kollaborieren, welche sprachagnostisch ist (vgl. [RD16]).

Ruby und David haben ein Werkzeug namens *Glotter* konzeptioniert, welches es ermöglicht, dass Individuen in ihrer Muttersprache programmieren und zugleich mit Individuen anderer Muttersprachen kollaborieren können. Dabei wird der Quellcode von verschiedenen Muttersprachen in eine gemeinsame Zielsprache übersetzt. *Glotter* ist dabei nicht nur neutral gegenüber den natürlichen Sprachen, sondern auch neutral gegenüber verschiedenen Programmiersprachen. Das heißt als Zielsprache kann jede beliebige Programmiersprache benutzt werden und *Glotter* kann dann für die jeweilige Programmiersprache konfiguriert werden. Dabei werden nicht nur Schlüsselwörter übersetzt, sondern auch Kommentare, die speziell annotiert werden. Glotter kann entweder in Compiler existierender Sprachen eingebaut werden oder als alleiniges Werkzeug verwendet werden (vgl. [RD16]).

Ein Problem mit *Glotter* ist, dass dieses zwar Schlüsselwörter und Kommentare übersetzt, jedoch keine Bezeichner und die Übersetzung somit unvollständig ist.

Ein weiteres interessantes Projekt in diesem Raum ist *Babylscript*. Hier werden internationale Versionen von Javascript nach Javascript kompiliert. Das interessante daran ist, dass in einer Quellcode-Datei mehrere Sprachen (also Französisch, Deutsch, Polnisch, ...) geschrieben werden können. Die Sprachen also gemischt werden können. Dieses Feature wird auf der Webseite folgendermaßen beschrieben:

„One of the unique features of Babylscript is that it allows people to write programs in a mix of different languages. A programmer can take a library written in French, mix it with their own program written in Spanish, and use code snippets they found on a Chinese help forum.“ [Iu21a]

Babylscript wird u. a.. für die Programmierlern-Webseite *Programming Basics* verwendet (s. [Iu21b]).

Syntaktische Feinheiten

Nun zum Teil, der etwas kontrovers ist, aus dem jedoch hervorgeht, dass beim Programmiersprachen-Design auch Details in der Syntax beachtet werden müssen.

McIver und Conway beschreiben in *Seven Deadly Sins of Introductory Programming Language Design* (s. [MC96]), dass Einrückungen als Syntax für Blockstrukturen für Beginner keine gute Idee ist, da Beginner aktiv das Konzept von *Scoping* üben müssen. *Whitespace*-Einrückungen für *Scoping* werden in Sprachen wie Python oder Haskell verwendet. Ein weiterer Punkt, den McIver und Conway nennen ist, dass durch explizite Syntax für den Anfang und das Ende einer Blockstruktur wie BEGIN...END und dem gleichzeitigen Einrücken das Konzept von *Scoping* einfacher zu lernen ist, weil das Konzept durch die Redundanz verstärkt wird (vgl. [MC96]).

Hinzuzufügen ist auch das Kriterium für lesbare Syntax, das Farooq u. a definiert haben: Zusammengesetzte Syntax-Konstrukte wie **if**, **while**, **for** usw. sollten konsistent sein. Außerdem stufen sie die Syntax, bei der zusammengesetzte Syntax-Konstrukte jeweils ein

spezielles Schlüsselwort haben, um den *Scope* zu beenden als besonders lesbar ein. Also bspw. `if` und `end if`, `loop` und `end loop`. Der Vorteil von dieser Syntax ist, dass Endungen von Blockstrukturen den Anfängen zugeordnet werden können. Ein anderer Vorteil ist, dass falls eine Endung vergessen wurde, der Compiler die Position, wo die Endung vergessen wurde, genau bestimmen kann. Geschweifte Klammern als Blockstruktur wie in der Familie der C-Sprachen hat den Nachteil, dass wenn eine geschweifte Klammer vergessen wurde, eine Fehlermeldung wie `Unexpected EOF` angezeigt wird, wie im Video *Bookends and curly braces* von *Context Free* demonstriert wird (vgl. [Tom21]). *Context Free* demonstriert auch, dass es Compiler – wie den Scala-Compiler – gibt, die so *smart* sind, die genaue Position wo die geschlossene geschweifte Klammer vergessen wurden, über die *Whitespace*-Einrückungen zu bestimmen.

Die Wahl der Endschlüsselwörter ist dabei auch ziemlich wichtig. Ein schlechtes unintuitives Beispiel liefert die *Bourne Shell*, wo die Endschlüsselwörter die Startschlüsselwörter rückwärts geschrieben sind: `fi .. fi`, `case ... esac` oder `do .. od`. Ein gutes Beispiel wie *Context Free* bemerkt liefert die Programmiersprache BASIC mit Kontrollstrukturen wie `FOR ... TO ... {STEP} ... NEXT` oder `DO UNTIL ... LOOP`. Hier ist bei der `FOR`-Schleife das `NEXT`-Schlüsselwort die Endung des Blocks. Die Schlüsselwörter wurden so ausgewählt, dass sie imperativen Anweisungen gleichen und somit sehr intuitiv sind (vgl. [Tom21]).

Eine andere syntaktische Feinheit ist, an welcher Position der Rückgabetyp bei Funktionsdefinitionen angegeben ist. Hier gibt es die Möglichkeit den Rückgabetyp an den Anfang oder an das Ende (*Trailing return type*) zu setzen. Es ist natürlicher den Rückgabetyp ans Ende zu setzen. Dann sieht man nämlich zuerst den Namen der Funktion, die Parameter und dann zum Schluss den Rückgabetypen. Funktionale Programmiersprachen wie *Haskell*, die *Currying* unterstützen machen das schon ganz automatisch, da es formal so korrekter ist – siehe den Vergleich zwischen C und Haskell.

```
int add(int a, int b) {
    return a + b;
}
```

Listing 3.8. add in C

```
add :: Int -> Int -> Int
add a b = a + b
```

Listing 3.9. add in Haskell

Bewertungskriterien

Die Bewertung der *Teaching Languages* im Bereich **intuitive Syntax und Natürlichsprachlichkeit** erfolgt nun auf folgenden drei Bewertungskriterien:

1. Wie intuitiv ist die Syntax basierend auf der Studie von [SS13]?
2. Ist die Programmiersprache internationalisiert und auf welche Art und Weise? Ist eine internationale Kollaboration möglich? Oder ist die Programmiersprache speziell auf eine bestimmte natürliche Sprache ausgerichtet (z.B. GermanSkript)?
3. Syntaktische Feinheiten
 - a) Intuitive Syntax für Blockstrukturen
 - b) *Trailing return type*

3.4.5. Feature Uniformity und Orthogonalität

Feature Uniformity

Eine Programmiersprache gilt als *Feature Uniform*, wenn eine Teilmenge dieser Sprache nicht in der Lage ist, alle Probleme zu lösen, die von der gesamten Sprache gelöst werden kann (vgl. [Far+14]). Oder nach Gupta sollte die Sprache eine Funktionsmenge aufweisen, welche in gewisserweise linear unabhängig ist. Es sollten nicht zu viele Möglichkeiten existieren das Gleiche zu machen. Beispielsweise ist die Programmiersprache Perl eine Sprache, die dies nicht berücksichtigt und viele Wege hat, ein und das selbe zu programmieren (vgl. [Gup04]). Gupta greift dabei auch die Begriffe *syntaktisches Synonym* und *syntaktisches Homonym* auf. Mit *syntaktischen Synonymen* sind Syntaxkonstrukte gemeint, die ein und die gleiche Funktion haben. Mit *syntaktischen Homonymen* sind Syntaxkonstrukte gemeint, die syntaktisch gleich sind, doch je nach Kontext eine andere Funktion haben.

Ein Beispiel für ein Synonym ist der *Inkrementier-* (sowie *Dekrementier-*) Operator, der in Sprachen aus der C-Familie bekannt ist. Hier kann eine Variable auf drei unterschiedlichen Wegen inkrementiert werden.

1. `x = x + 1;`
2. `x += 1;`
3. `x++;`

Ein Beispiel für ein Homonym ist das Schlüsselwort `static` in der Sprache C++. Das Schlüsselwort `static` hat in C++ abhängig davon, wo es steht drei Bedeutungen:

1. Wenn es außerhalb von Funktionen oder Klassen steht, dann steht es für eine globale Variable mit *Internal Linkage*, d.h. auf die Variable kann nur innerhalb der gleichen Datei zugegriffen werden.
2. Wenn es innerhalb einer Funktion steht, dann steht es für eine lokale statische Variable, die nur einmal initialisiert wird.
3. Innerhalb einer Klasse steht es für ein statisches Feld oder Methode.

Gupta betont auch, dass die Funktionsmenge der Sprache für Beginner möglichst klein gehalten werden sollte, da dies Beginnern hilft, sich auf das wesentliche zu konzentrieren und nicht von der Menge der Funktionen überfordert zu werden. Die Sprache sollte möglichst minimal aus wenigen Funktionsbausteinen zusammengesetzt werden, um die Lernkurve für Beginner gering zu halten.

Orthogonalität

Mit Orthogonalität ist gemeint, ob die Programmiersprache konsistenten Regeln folgt (vgl. [Far+14]). Ein Beispiel ist, dass Schlüsselwörter in der Sprache nicht als Bezeichner verwendet werden können (vgl. [Far+14]). Ein Beispiel für schlechte Orthogonalität sind die Programmiersprachen C++ und C#, bei denen das Initialisieren von lokalen Variablen nicht obligatorisch ist (vgl. [MG11]). Das macht die Sprache inkonsistent und schwerer für Beginner zu lernen.

Bewertungskriterien

In dem Bereich *Feature Uniformity und Orthogonalität* gibt es nun folgende Bewertungskriterien:

1. *Feature Uniformity*
 - a) möglichst wenige syntaktische Synonyme
 - b) möglichst wenige syntaktische Homonyme
 - c) Eine kleine simple Funktionsmenge
2. Orthogonalität (konsistente Syntax)

3.4.6. Unterstützung der wichtigsten Programmierkonzepte & Datenstrukturen

In einer Umfrage in den USA wurden 200 Leiter von Informatikprogrammen an Hochschulen befragt, was die Entscheidungsfaktoren für die Wahl einer FPL sind. Die Mehrheit gab an, die Funktionalitäten der Programmiersprache (26,19%) für die Entscheidung zu berücksichtigen. Erst danach als Zweites mit (18,81%) wurde die Einfachheit der Erlernbarkeit angegeben (vgl. [Eze18]).

In der gleichen Umfrage wurde auch befragt, welche Funktionalitäten in einer FPL am wichtigsten sind. Dabei wurden folgende Funktionalitäten am wichtigsten befunden:

- Wiederholungsstrukturen (*Repetition structure*)
- Statische Typisierung (*Static typing*)
- Das Strukturieren eines Programms in Funktionen (*Functional decomposition*)
- Kompiliert anstatt interpretiert (*A compiler*)
- Objektorientierung (*Object-Orientation*)

Für diesen Bewertungsbereich wird ausschließlich auf die Programmierkonzepte geschaut, also Wiederholungsstrukturen und das Strukturieren eines Programms in Funktionen ist für diesen Bereich relevant.

Objektorientierung wird auch rausgenommen, da die Bewertung zielgruppen- und paradigmübergreifend erfolgt. Die Wichtigkeit der Objektorientierung folgt u. a.. daraus, dass sie das weit verbreiteste Paradigma ist, das in der Wirtschaft verwendet wird.

Mit Wiederholungsstrukturen sind Programmierkonzepte wie Schleifen gemeint. Wenn dann noch Konzepte wie Bedingungen mit eingeschlossen werden gelangt man zur *strukturierten Programmierung*. Durch strukturierte Programmierung wird *Spaghetti-Code* verhindert. Der Programmcode lässt sich, wie ein Buch von oben nach unten lesen. Das Gegen teil von strukturierter Programmierung sind Programmcodes, die eine `GOTO`-Anweisung verwenden, um zu verschiedenen Stellen des Programms zu springen, was wie schon Dijkstra 1968 bemerkt die Lesbarkeit eines Programms drastisch reduziert (vgl. [Dij68]). Ebenso kann für das Strukturieren eines Programms in Funktionen als überliegendes Paradigma die *Prozedurale Programmierung* zugewiesen werden.

Die Konzepte der strukturierten und prozeduralen Programmierung sind in allen populären Programmiersprachen (DSLs ausgeschlossen) enthalten. Deshalb ist es wichtig, dass diese Konzepte auch in *Teaching Languages* vorhanden sind, sodass wenn die *Teaching Language* als FPL eingesetzt wird ein Übergang in eine populäre Programmiersprache er-

folgen kann.

Desweiteren ist es wichtig, dass die *Teaching Language* bestimmte Datenstrukturen unterstützt. Es sollten die wichtigsten Datenstrukturen wie Liste, Warteschlange (*Queue*), Stapel (*Stack*), Zuordnungstabelle (*Hashmap*) und Menge (*Set*) unterstützt werden.

Obendrein ist es wichtig, dass computertheoretische Konzepte, wie z.B. Rekursion von der Programmiersprache unterstützt werden und beigebracht werden können. Hierfür wird beispielhaft für jede *Teaching Language* untersucht, ob die Berechnung der Elemente der *Fibonacci-Folge* rekursiv nach der Funktion ...

$$f_n = f_{n-1} + f_{n-2} \text{ für } n >= 3 \quad (3.1)$$

$$f_1 = f_2 = 1 \quad (3.2)$$

... implementiert werden kann. Wenn eine Implementierung möglich ist, wird außerdem bewertet, wie einfach die Implementierung ist.

Schließlich gibt es noch das Bewertungskriterium der Mächtigkeit, auf das im Folgenden eingegangen wird:

Mächtigkeit

Es ist schwer zu analysieren, wie mächtig eine Programmiersprache ist. Ist der Compiler der Programmiersprache in der Programmiersprache selbst geschrieben, kann man aber schon sagen, dass es sich um eine mächtige Programmiersprache handelt.

Aber wie wird ein Compiler für eine Sprache in der Sprache selbst geschrieben? Es handelt sich dabei doch um ein Henne-Ei-Problem. Um dieses Problem zu lösen, gibt es den Prozess des *Bootstrapping*. Der Name *Bootstrapping* kommt vom Ausdruck „to pull oneself up by one's bootstraps“ (sich an den eigenen Stiefelschlaufen hochziehen), der im Englischen seit dem frühen 19. Jahrhundert existiert (vgl. [Wik20c]). Beim *Bootstrapping* wird der Compiler der neuen Programmiersprache in einer schon existierenden Programmiersprache implementiert (*Bootstrap-Compiler*). Schließlich wird der Compiler erneut in der neuen Programmiersprache implementiert und wird vom *Bootstrap-Compiler* kompiliert. Zum Schluss kann der Compiler in der neuen Programmiersprache sich dann selbst kompilieren. Der Compiler ist dann *self-hosting*.

Damit ein Compiler *self-hosting* ist, muss es sich um eine kompilierte Programmiersprache handeln, da eine interpretierte Sprache über eine existierenden Sprache implementiert ist und man sich so niemals von der existierenden Sprache lösen kann (vgl. [blu12]).

Das Bewertungskriterium der Mächtigkeit einer Programmiersprache davon abhängig zu machen, ob der Compiler *self-hosting* ist, ist sehr gewagt, da eine Programmiersprache mächtig und jedoch nicht *self-hosting* sein muss. Sprach-Entwickler schreiben jedoch gerne einen *self-hosting* Compiler, um zu beweisen, dass ihre Sprache mächtig ist.

Dieses Bewertungskriterium ist nur eine kleine Ergänzung zum Bewertungsbereich, welches kein großes Gewicht zum ganzen Bewertungsbereich hat.

Bewertungskriterien

Somit ergeben sich folgende Bewertungskriterien:

1. Die Sprache unterstützt folgende Konzepte der strukturierten Programmierung:
 - a) Schleifen
 - b) Bedingungen
2. Die Sprache unterstützt folgende Konzepte der prozeduralen Programmierung:
 - a) Funktionen und Funktionsaufrufe
 - b) lokale Variablen
 - c) Rekursion
3. Die Sprache unterstützt folgende Datenstrukturen:
 - a) Liste
 - b) Warteschlange (*Queue*)
 - c) Stapel (*Stack*)
 - d) Zuordnungstabelle (*Dictionary, HashMap*)
 - e) Menge (*Set*)
4. Kann die Fibonacci-Funktion rekursiv implementiert werden und wie einfach ist die Implementierung?
5. Handelt es sich bei dem Compiler der Programmiersprache um einen *self-hosting* Compiler?

3.4.7. Programmierumgebung (IDE)

Es wird zunächst auf das Konzept einer pädagogischen IDE eingegangen und die Motivation und Entwicklung dieser kurz geschildert. Schließlich werden die Kriterien einer pädagogische IDE durch weitere Kriterien ergänzt, um Bewertungskriterien für die Programmierumgebung einer *Teaching Language* zu ermitteln.

Motivation und Entwicklung Pädagogischer IDEs

In diesem Abschnitt wird auf das Konzept der pädagogischen IDEs eingegangen. Hierfür wird auf zwei unterschiedliche pädagogische IDEs eingegangen: *BlueJ* und *DrRacket*. Beide IDEs teilen überraschend viele Gemeinsamkeiten hinter ihrer Motivation und Entwicklung. Im Folgenden wird auf die Gemeinsamkeiten und Unterschiede der IDEs eingegangen.

BlueJ ist eine IDE für die objektorientierte Programmierung in Java. Hinzu kommt, dass *BlueJ* auch die *Teaching Language* Stride unterstützt. *DrRacket* hingegen ist eine IDE für die funktionale Programmierung in Racket. Vorher hieß *DrRacket* *DrScheme* und hat die funktionale Programmierung in Scheme unterstützt (vgl. [Kri20].)

Die Entwicklung von *DrRacket* (*DrScheme*) begann 1995. Shiriam Krishnamurti – der Pionier hinter *DrRacket* – beschreibt die Motivation das Projekt zu starten folgendermaßen: Seine Idee ist aus seinem Kontext entstanden, dass damals die Studenten die funktionale Programmiersprache Scheme innerhalb des Texteditors *Emacs* lernen sollten. Dabei ist *Emacs* ein sehr komplexer Editor, der über die Tastatur gesteuert wird und so müssen viele Tastenkommandos auswendig gelernt werden. Was bei Beginnern oft geschieht ist, dass sie im Editor in einem bestimmten Modus festhängen aus dem sie nicht mehr

rauskommen. Zur selben Zeit wurde C++ als Einführungsprache sehr beliebt und mit einer IDE wie *Visual C++* gab es eine komplexe grafische professionelle Programmierumgebung. Hier waren Studenten jedoch ebenfalls von der großen Anzahl an Optionen und der Komplexität überfordert. Aus diesem Problem entwickelte sich dann die Idee einer pädagogischen IDE, welche eine eigenständige grafische Anwendung sein sollte, die das Programmieren in den Vordergrund rückt und Ablenkungen und Verwirrungen reduziert (vgl. [Kri20]).

Das gleiche Problem beschreibt Michael Kölling, der Pionier hinter BlueJ 1999:

„Programming languages used are too complex and programming environments – if they exist at all – are too confusing. Some systems used for teaching were really developed for professional software engineers, making it difficult for first-year students to cope; others were not “developed” at all but grew out of historic coincidences“ [Köl99]

Eine weitere Motivation von Michael Kölling hinter BlueJ war der *Objects First*-Lehransatz.

„For teaching programming, the lesson is clear: if we want to teach object-orientation, we should do it first. The path to object-orientation through procedural programming is unnecessarily complicated. Students first learn one style of programming, then they have to “un-learn” the previously learned, before we show them how to do it “right”.“ [Köl99]

Zudem sollte nach Kölling die Programmierumgebung das Programmierparadigma wieder spiegeln (vgl. [Köl+03]). So wird in BlueJ nicht mit Quellcode sondern mit Objekten interagiert. Dafür wird das Projekt in BlueJ mittels eines UML-Klassendiagramms visualisiert und die Klassen können dann auch inspiert werden. Zusätzlich kritisiert Kölling den Einsatz von *GUI-Buildern* als pädagogisches Lerntool, weil dadurch ein falsches Bild für Studenten geschaffen wird, was es bedeutet zu Programmieren und was Objektorientierung ist (vgl. [Köl+03]).

Ein Argument gegen pädagogische IDEs

Ein großes Argument gegen pädagogische IDEs ist, dass sie den Schüler in eine Sandkasten-Umgebung setzt. Diese Sandkasten-Umgebung eignet sich zwar optimal dafür den Studenten programmieren beizubringen, spiegelt jedoch nicht die Realität des Programmierens wieder, da in der Wirtschaft komplexere sowie minimalere und technischere Werkzeuge verwendet werden. Schüler, die schon weiter fortgeschritten sind, würden die Umgebung gerne verlassen. Andere Schüler könnten an der Umgebung hängen und lieber da bleiben wollen. Deshalb ist es wichtig den Schülern transparent den Zweck einer pädagogischen IDE als Lerntool zu vermitteln. Nämlich, dass diese sich auf den Lernaspekt konzentriert und als Trittsstein in den Einstieg von professionelleren Werkzeugen genutzt wird.

So ist Kölling's Designziel hinter *BlueJ*, dass das Tool Programmieren Lernen im ersten Hochschuljahr unterstützt, es aber nicht nach dem ersten Jahr verwendet werden sollte. Er hält es ebenfalls für sehr wichtig, dass Studenten aus der Programmierumgebung

herauswachsen und gezwungen sind Erfahrungen in professionellen Programmierwerkzeugen zu sammeln. Der Übergang in professionellere Werkzeuge kann laut Kölling innerhalb einer einzigen Vorlesung, die sich proaktiv ganz dem Thema widmet, wie Konzepte aus der Beginnerumgebung in Konzepte der professionellen Umgebung übertragen werden, stattfinden (vgl. [Köl+03]).

Es handelt sich hier somit nicht wirklich um ein Argument gegen pädagogische IDEs, nur über eine Klarstellung über deren Zweck und richtigen Einsatz.

Einfachheit der Umgebung

Eines der wichtigsten Bewertungskriterien für die Programmierumgebung, die sich aus den pädagogischen IDEs herleiten ist die Einfachheit. Mit Einfachheit ist gemeint, dass die Programmierumgebung nur die nötigen Funktionen enthält, die Anfänger brauchen, um Programmieren zu lernen und auf professionelle Funktionen, die Anfänger eher durcheinander bringen, verzichtet wird. Außerdem ist es auch wichtig, dass in der IDE nur die Elemente angezeigt werden, die momentan gebraucht werden.

Um dies an einem Beispiel zu erläutern wird die professionelle IDE *Eclipse* mit der pädagogischen IDE *DrRacket* kurz verglichen. Hierfür sind Bildschirmaufnahmen der IDEs im Anhang im Abschnitt A.1 zu finden.

In *Eclipse* fällt einem zunächst die lange Werkzeugeiste auf in der 30 Icons zu finden sind. Diese kann durch Plugins noch erweitert werden und zwei der Icons (die Orangen) gehören normalerweise nicht dazu. *DrRacket* hingegen hat nur 6 Icons in der Werkzeugeiste.

Zudem fällt die Anzahl der Fenster auf. In *Eclipse* sind im Standardlayout 6 Fenster offen: Der *Package Explorer*, *Quellcode-Editor*, *Task List*, *Outline* und ein weiteres Fenster mit *Problems* sowie den angedockten *Javadoc* und *Declaration*. In *DrRacket* hingegen ist nur ein Fenster offen. Erst beim Ausführen öffnet sich unten die Ausgabe zusammen mit einem interaktiven REPL.

Etwas anderes, was Beginner im Umgang mit Eclipse zunächst durcheinander bringen könnte sind die Ansichten. Standardmäßig gibt es die *Java*-Ansicht und die *Debug*-Ansicht. Die *Debug*-Ansicht zeigt dabei auch noch andere Fenster wie das *Debug*- und das *Breakpoint*-Fenster an. Einige Fenster der *Java*-Ansicht sind nicht vorhanden und die Fenster befinden sich an anderen Positionen. Die Ansicht wird automatisch gewechselt, wenn das Programm debuggt wird. Dies kann Beginner das erste Mal ziemlich verwirren.

Zudem belasten in *Eclipse* komplexere *Features* wie *Working Sets*, mit dem man mehrere Projekte gruppieren kann, Beginner zusätzlich, da diese dieses Feature gar nicht benötigen, jedoch von der Benutzeroberfläche mit diesem Feature konfrontiert werden. Die extrinsische kognitive Belastung wird somit erhöht und es kann sich schlechter auf das Wesentliche konzentriert werden.

In *DrRacket* wird zudem, wenn mit der Maus im Editor auf ein Schlüsselwort geklickt wird, oben rechts eine Syntax-Hilfe angezeigt (s. Abb. 3.1).

```
(define id expr)           syntax
(define (head args) body ...+)

head = id
| (head args)

args = arg ...
| arg ... . rest-id

arg = arg-id
| [arg-id default-expr]
| keyword arg-id
| keyword [arg-id default-expr]
read more...
```

Abb. 3.1. Syntax-Hilfe in DrRacket

Das ist nur eines der pädagogischen Hilfsmittel von *DrRacket*. Ein anderes ist, dass wenn mit der Maus auf ein Token im Quelltext-Editor gehovert wird, über einen Pfeil angezeigt wird, wo dieses definiert ist.

Interaktivität

Ein weiteres wichtiges Kriterium, die eine Programmierumgebung für Beginner aufweisen sollte, ist eine hohe Interaktivität. Gupta erwähnt den Begriff der *Turnaround Time*, die möglichst kurz gehalten werden sollte (vgl. [Gup04]). Das ist die Zeit, die gebraucht wird, um einen Prozess oder eine Anfrage zu beenden, beispielsweise wie lange es dauert das Programm zu kompilieren. Hier haben interpretierte Sprachen einen großen Vorteil vor kompilierten Sprachen, da der Programmcode sofort ausgeführt wird. Der Nachteil von interpretierten Sprachen ist natürlich die geringere Laufzeitperformanz. Diese spielt bei *Teaching Languages* jedoch keine wichtige Rolle.

Es geht darum den Feedback-Loop für den Beginner möglichst kurz zu halten. Hierfür eignet sich das Paradigma der interaktiven Programmierung besonders. Bei der interaktiven Programmierung werden Teile eines Programms entwickelt, während dieses schon aktiv ist (vgl. [Wik20e]). Ein Werkzeug der interaktiven Programmierung ist ein *Read Eval Print Loop* (REPL). Hier kann Code eingelesen werden (*Read*), dieser wird dann vom Compiler evaluiert (*Eval*) und das Ergebnis dann ausgegeben (*Print*). Dies kann beliebig oft wiederholt werden (*Loop*). Oftmals wird auch die Funktionalität unterstützt, dass ein geschriebenes Programm in die REPL geladen und dort dann inspiziert und getestet werden kann. REPLs sind bekannt für interpretierte Skriptsprachen, können aber für kompilierte Sprachen erstellt werden, wobei sie dabei schwieriger zu implementieren sind (in der Regel über einen Interpretierer, der mit der virtuellen Maschine kommuniziert) (vgl. [Wik21k]).

Zusätzlich kann ein REPL auch noch durch ein *Notebook Interface*, wie man es bspw. von *Jupyter Notebook* kennt, erweitert werden. Es handelt sich dabei um eine virtuelle *Notebook*-Umgebung, in der ausführbare Berechnungen in ein fomatiertes Dokument eingebettet sind (vgl. [Wik21j]). Bei dieser Programmiertechnik handelt es sich des Weiteren um *literate programming*, welches 1981 von Donald Knuth eingeführt wurde. Im *literate*

embedding wird die Dokumentation eines Programms in das Programm selbst eingebettet (vgl. [Wik19]).

Ein andere Möglichkeit das Programmieren interaktiv zu gestalten ist *Live Coding* – auch *on-the-fly programming* genannt – bei welchem das Programmieren an sich zu einem integralen Bestandteil des laufenden Programms wird. Die technische Umsetzung geschieht dabei über das sogenannte *Hot Swapping*, wobei der Programmcode zur Laufzeit ohne Programmunterbrechung verändert werden kann und das veränderte Programm automatisch neu geladen wird (vgl. [Wik21g]). Bekannte Einsatzgebiete der interaktiven Programmierung sind die Programmierung von grafischen Benutzeroberflächen oder das Programmieren von Musik.

Debugging Support

Da Debugging sehr wichtig in der Informatik ist (vgl. [McC+08]), in Problemlösung (vgl. [Jon10]) und *Computational Thinking* ist (vgl. [BR+12]), ist es unverzichtbar für die Programmierumgebung, dass sie Debugging durch Werkzeuge unterstützt.

Debugging muss dabei nicht zwingend über einen visuellen Schritt-Für-Schritt-Debugger erfolgen, der bei textbasierten Sprachen oft verwendet wird. Wie sinnvoll eine Debugging-Technik ist, hängt von der Art der Programmiersprache und der Programmieraufgabe ab. Bei imperativen Sprachen ist ein visueller Schritt-Für-Schritt-Debugger ziemlich wichtig und nützlich und Programmieranfänger können über ihn viel über das Programmieren lernen. Bei deklarativen (funktionalen) Sprachen kann ein effektiveres Debugging über eine interaktive REPL erfolgen oder durch Unit-Tests, in denen das Funktionieren einer bestimmten Funktion sicher gestellt wird. Eine primitivere Form, aber ebenfalls eine sehr effektive Form des Debuggings, sind Ausgaben, sei es in der Standardausgabe oder in Log-Files. Hinzu kommen professionellere Werkzeuge des Debuggings, die ein bestimmtes Problem lösen wie *Monitoring*, *Performance Profiling* und *Memory Dumps*, die für *Teaching Languages* jedoch von keiner großer Bedeutung sind.

Unter Debugging-Support sind somit Werkzeuge gemeint, die es zum einem erlauben das Programm während und nach der Ausführung zu inspizieren und zum anderen die Richtigkeit von Programmcode zu überprüfen, damit ein Problem identifiziert und *gefxt* werden kann.

Einfachheit der Installation

Natürlich ist es auch wichtig, wie einfach die Programmierumgebung installiert werden kann und, dass möglichst wenige Probleme während der Installation auftreten.

Eine besonders gute Möglichkeit sind webbasierte IDEs, bei der keine Installation auf dem Endgerät erfolgen muss und überall – egal auf welchen Gerät programmiert werden kann. Durch Fortschritte in Internettechnologien sind webbasierte (oder auch cloudbasierter) IDEs wie auch *Cloud-Computing* mittlerweile ein großer Trend, da die Technologie dafür bereit steht und sie viele Vorteile bieten.

Einfachheit der Kollaboration

Ein weiteres Kriterium ist, wie einfach die Programmierumgebung es macht in einem Team zusammenzuarbeiten, Programme zu teilen und andere Programme zu importieren.

Für das Zusammenarbeiten im Team könnte beispielsweise ein integrierter *Git-Client* verwendet werden. Da *Git* ein sehr komplexes Versionsverwaltung-System mit vielen Features ist, sollte der *Git-Client* möglichst beginnerfreundlich nur die Features unterstützen, die Beginner für die Team-Arbeit brauchen und diese Features in einfacher Art und Weise zur Verfügung stellen.

Zudem sollten Programme öffentlich und am besten auch privat einfach geteilt werden können. So können Programmierbeginner sich Inspiration von anderen Programmen holen und sich auch Programmertipps abschauen. Zusätzlich sollten andere Programme importiert und erweitert werden können, sodass Programmieranfänger auf existierende Ideen aufbauen können.

Weitere Features zur Unterstützung des Programmierers

Aus modernen IDEs sind Features wie Syntax-Highlighting, intelligente Code-Completion, -Navigation und -Formatierer und -*Refactoring*, die den Programmierer bei der Programmieraufgabe unterstützen, bekannt.

Dabei ist die Qualität und die Einfachheit dieser Features zu berücksichtigen. Sie sollten funktional sein und nicht von der Programmieraufgabe ablenken.

Da *Teaching Languages* in einer Vielzahl von verschiedenen – auch unkonventionelleren – Programmierumgebungen eingebettet werden, kann die Implementierung dieser modernen Features sehr unterschiedlich erfolgen. Beispielsweise ist Syntax-Highlighting in der block-basierten *Teaching Language* Scratch über verschiedenfarbigen und -förmigen Blöcken, die jeweils eine andere Semantik haben, implementiert.

Bewertungskriterien

Zusammengefasst ergeben sich somit folgende Bewertungskriterien.

1. Einfachheit der Umgebung (Pädagogische IDE)
2. Interaktivität
 - a) kurze *Turnaround Time*
 - b) Elemente der interaktiven Programmierung (bspw. REPL)
 - c) Live Coding (*Hot Swapping*)
3. Debugging Support
4. Einfachheit der Installation
5. Einfachheit der Kollaboration
 - a) Programmieren im Team
 - b) Das Teilen von Programmen
6. Weitere Features ...
 - a) Syntax Highlighting
 - b) Code-Completion
 - c) Code-Navigation

- d) Code-Formatierer
- e) Code-Refactoring

3.4.8. Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplin (*Clean Code*)

Zum Programmierenlernen gehört ebenfalls dazu, den Code so zu schreiben, dass er lesbar und wartbar ist. Da ein Programmierprojekt sehr komplex werden kann und es sich auch oft um Teamarbeit handelt, sollten gute *Teaching Languages* es erlauben den Code so zu strukturieren und zu schreiben, dass er wartbar, flexibel und bugfrei bleibt. Hierfür muss die Sprache einige *Features* liefern, damit gute Programmierpraktiken umgesetzt werden können und die Sprache kann außerdem bestimmte Programmierpraktiken durch ihr Design erzwingen.

Die Sprache sollte eine modulare Programmierung unterstützen. Hierfür sollte das Programm in unabhängige, austauschbare Module separiert werden können, die jeweils eine Schnittstelle anbieten, um mit dem Modul zu interagieren. Zudem sollten von außen nicht auf Implementierungsdetails zugegriffen werden können (*Information Hiding*). In objekt-orientierten Sprachen geschieht dies meist durch Zugriffsmodifizierer einer Klasse, die den Zugriff auf eine Methode erlauben oder einschränken. In anderen Sprachen durch *Export*-Funktionen von Bibliotheken, die explizit nur das Interface exportieren und nicht die Implementierung.

Für ein Modul ist es zudem wichtig, dass dieses gut dokumentiert ist. Hierfür eignen sich Dokumentations-Kommentare aus denen eine Dokumentation generiert werden kann am besten.

Des Weiteren sollte die Programmiersprache Features enthalten, mit denen die Programmietechnik des *Design by Contract* (DBC) umgesetzt werden kann. Bei DBC wird für jede Schnittstelle einen Vertrag zwischen dem Implementierer und dem Konsumenten der Schnittstelle definiert. Durch Vorbedingungen, Nachbedingungen und der Definition von Invarianten wird sicher gestellt, dass dieser Vertrag eingehalten wird. Eine Invariante ist eine Bedingung, die sich während der Ausführung eines (Teil-)Programms nicht verändert (vgl. [Wik21h]). Die Features, damit eine Programmiersprache DBC umsetzen kann sind *Assertions* oder auch *Guards*.

Ein verwandtes Konzept zu DBC ist ein statisches Typsystem. Durch ein statisches Typsystem können bestimmte Typeigenschaften des Programms noch vor der Ausführung sicher gestellt und Bugs verhindert werden. Ein statisches Typsystem ist einem Dynamischen vorzuziehen, da die Typsicherheit des Programms noch vor der Ausführung sicher gestellt werden kann. Es gibt ein früheres Feedback und Bugs können schon vorher korrigiert werden. Außerdem ist ein stärkeres Typsystem einem schwächeren Typsystem vorzuziehen. Ein Beispiel für ein schwaches Typsystem ist Javascript. In Javascript kann der Plus-Operator auf eine Zeichenfolge und einer Zahl angewandt werden, wobei die Zahl dann automatisch in eine Zeichenfolge konvertiert wird. Die automatische Umwandlung von Typen nennt sich *Type Coercion*.

Type Coercion hat den Nachteil, dass schneller Bugs entstehen können, z.B. wenn in Javascript eine Operation wie Addition auf eine Zahl und `undefined` ausgeführt wird, dann ist das Ergebnis dieser Operation `NaN`. Dies kann schnell über ein Tippfehler geschehen, indem auf eine Eigenschaft eines Objekts zugegriffen wird, das gar nicht existiert, da diese Eigenschaft dann `undefined` ist. Da es sich um eine legale Operation handelt, wird kein Fehler ausgegeben, dass die Zahl `NaN` ist und wenn es sich um ein Zwischenergebnis handelt, löst dieser Wert dann weitere Fehler und Probleme aus. Erst beim Debuggen kann das Problem dann erkannt werden, was viel Zeit in Anspruch nehmen kann. Somit sind explizite *Casts* und Konvertierungen Impliziten vorzuziehen, sodass Bugs verhindern werden können.

Zudem sollte die Programmiersprache *Exception Handling* unterstützen. Tritt eine unerwartete Ausnahme ein, so soll diese behandelt werden können. Es gibt unterschiedliche Art und Weisen, wie eine Programmiersprache *Exception Handling* implementieren kann. In Sprachen wie Java gibt es die Syntax `throw`, mit dem Ausnahmen ausgelöst werden können. Das Programm bricht dann die Ausführung ab, bis die Ausnahme mit `try .. catch .. (finally)` behandelt wird. Andere Sprachen wie *Golang* setzen darauf, dass eine Funktion explizit einen Fehler als Wert zurückgibt und der Aufrufer, dann die Existenz eines Fehlers überprüft (vgl. [Ger11]). Die Ansätze bieten unterschiedliche Vor- und Nachteile, deswegen ist es einfach nur von Wichtigkeit, dass die Sprache einen klar definierten Ansatz hat, wie sie Ausnahmen behandelt.

Zusätzlich ist es wichtig die Korrektheit der Implementierung eines Moduls festzustellen. Hier ist es wichtig, dass die Sprache Unit-Tests unterstützt.

Für *Clean Code* ist es zudem wichtig, dass es einen *Style Guide* gibt, wie Code geschrieben werden soll. So gibt es eine feste Konvention, die Benutzer anwenden können, was die Lesbarkeit von fremden Programmen verbessert.

Schließlich gibt es noch eine Reihe von weiteren Design-Entscheidungen einer Programmiersprache, wie sie das Schreiben von unsauberen Code und Bugs verhindern kann, welche Farooq u. a aufgelisten (vgl. [Far+14]):

- 1) **Ausdrücke sollten keine Nebeneffekte haben:** Dies ist die Eigenschaft der *referentiellen Transparenz*, welche von puren funktionalen Sprachen bekannt ist. Es bedeutet, dass ein Ausdruck pur ist, und keine Nebeneffekte hat. Da dieses ein sehr strenges Kriterium ist, wird nur darauf geachtet, dass es keine Operatoren mit Nebeneffekten gibt. Operatoren mit Nebeneffekten sind Operatoren wie `++` in Programmiersprachen der C-Familie. Besonders verwirrend für Beginner ist es, dass dieser Operator als Präfix- und Postfix-Variante mit zwei verschiedenen Semantiken existiert.
- 2) **Kein Überladen von Operatoren:** Operatoren sollten nicht Überladen werden können. Dies kann nämlich zu schlechten Code führen, wo Operatoren unkonventionell überladen werden und der Code somit sehr schwer zu lesen ist. Besser ist es für eigene Typen einfach Methoden zu schreiben.
- 3) **Keine Überschattung von Variablen:** Variablen sollten nicht überschattet wer-

den können (*Variable Shadowing*). Damit ist gemeint, dass in einem inneren Bereich nicht eine neue Variable mit dem gleichen Bezeichner einer Variable aus dem äußeren Bereich erstellt werden darf, die die äußere Variable dann im inneren Bereich überschattet. Dies kann nämlich zu Bugs führen, wenn z.B. aus Versehen bei einer verschachtelten Schleife der gleiche Variablenname sowohl bei der inneren als auch bei der äußeren Schleife, verwendet wird.

Bewertungskriterien

Somit ergeben sich im diesem Bereich folgende Bewertungskriterien:

1. Der Code in der Programmiersprache lässt sich in wiederverwendbare Module organisieren.
2. Die Sprache hat Zugriffsmodifizierer oder ähnliche *Information-Hiding*-Mechanismen, um die Implementierung zu verstecken.
3. Die Sprache unterstützt Dokumentations-Kommentare und eine automatische Generierung von Dokumentation.
4. Die Sprache hat ein statisches Typsystem.
5. Die Sprache verwendet explizite Casts und Typumwandlung und keine *Type Coercions*.
6. Die Sprache hat einen klar definierten Weg um Ausnahmen zu handeln (*Exception-Handling*).
7. Die Sprache unterstützt Unit-Tests.
8. Es gibt einen *Style Guide*, der angibt wie Code geschrieben werden soll.
9. Weitere Eigenschaften für sauberen Code:
 - a) Es gibt keine Operatoren mit Nebeneffekten.
 - b) Es gibt keine Überladung von Operatoren.
 - c) *Variable Shadowing* ist nicht möglich.
 - d) Weitere ...

3.4.9. Fehlermeldungen

Fehlermeldungen spielen eine kritische Rolle, wenn es um die Benutzererfahrung für Programmieranfänger gilt.

Sie erfüllen zwei kritische Rollen (vgl. [MFK11]):

1. Sie sollten helfen den Benutzer zu einem funktionieren Programm zu verhelfen.
2. Als pädagogisches Tool sollten sie helfen, dass der Benutzer das Problem, das zu dem Fehler geführt hat, versteht.

Außerdem sollten Fehlermeldungen weitere Frustration vermeiden, indem sie leicht für den Benutzer zu verstehen sind und den Benutzer nicht auf den falschen Weg führen, um das Problem zu korrigieren (vgl. [MFK11]).

Es ist wichtig zu verstehen, dass Fehlermeldungen für Experten anders designt sein müssen als Fehlermeldungen für Beginner, um Frustrationen für Beginner zu vermeiden. Watson, Li & Godwin erkären die Wichtigkeit und das Problem von Fehlermeldungen folgendermaßen:

„Feedback is regarded as one of the most important influences on student learning and motivation. But standard compiler feedback is designed for experts - not novice programming students, who can find it difficult to interpret and understand.“ [WLG12]

Das Problem der Fehlermeldungen für Beginner wird von Marceau, Fisler und Krishnamurthi [MFK11] anhand Studentendaten innerhalb *DrRacket* ausführlich untersucht und es wird eine grundlegende Empfehlungen für den Design von Fehlermeldungen für Beginner angegeben:

Fehlermeldungen sollten keine Lösungen vorschlagen. Obwohl manche Fehler sehr wahrscheinliche Lösungen haben, decken diese Lösungen nicht alle Fälle ab. Vorgeschlagene Lösungen könnten dazu führen, dass Lernende den falschen Weg einschlagen, um den Fehler zu beheben. Zudem geben Marceau u. a.. einige Vorschläge für das Design von Fehlermeldungen für Anfänger an:

- 1) **Die Vokabeln in der Fehlermeldung vereinfachen:** Für Anfänger sind semantische Unterschiede von bestimmten Abstraktionen nicht wichtig. So schlagen die Autoren bspw. vor nicht zwischen *Prozeduren*, *Operatoren* und *Konstruktoren* zu unterscheiden, sondern alle diese Begriffe unter dem Begriff *Funktion* zusammenzufassen. Somit werden Beginner nicht mit einer Menge von Fachwörtern überfordert, die für sie zu Beginn noch nicht relevant sind.
Es ist dabei wichtig das Paradigma der Sprache zu beachten. In objektorientierten Sprachen spielen Konstruktoren beispielsweise eine so wichtige Rolle, dass sie auch einen eigenen Namen bekommen sollten.
- 2) **Bei Fehlern explizit auf Inkonsistenten hinweisen:** Es werden einige Beispiele angegeben, was damit gemeint ist: Wenn z.B. ein Fehler bei einem Funktionsaufruf auftritt, dann kann der Fehler entweder am Aufruf oder an einer falschen Definition liegen. Es ist dann wichtig, dass die Fehlermeldung beide Möglichkeiten ohne Bias angibt. Ein Vorschlag um das Problem zu beheben, ist es den Programmierer zu fragen, was er versuchen wollte zu tun und daraufhin einen passenden Fehler auszugeben.
- 3) **Studenten dabei unterstützen, Fehlermeldungsabschnitte Codeabschnitten zuordnen zu können:** Studenten sollten bestimmte Begriffe in der Fehlermeldung wie z.b. Funktionskörper Code-Abschnitten zuordnen können. Der Vorschlag ist Referenzen auf Code-Abschnitte in der Nachricht verschiedenfarbig zu machen und den dazugehörigen Code dann in der dazugehörigen Farbe anzuzeigen.
- 4) **Fehlermeldungen als integraler Bestandteil des Kurs-Designs:** Wenn ein Kurs designt wird, sollten die gleichen Begrifflichkeiten und Notationen wie in den Fehlermeldungen verwendet werden.
- 5) **Quellcode-Hervorhebungen und andere Fehler-Komponenten sollten unterrichtet werden:** Lehrer und Bücher sollten die Komponenten einer Fehlermeldung unterrichten. Es sollte dabei auch auf die Semantik verschiedener Nachrichten wie z.B. von „expected ..., found ...“ eingegangen werden.
- 6) **Es sollte einen *Style Guide* für Bibliothek-Autoren geben, wie Fehlermeldungen zu entwerfen sind.**

Dies ist jedoch nur eine Empfehlung basierend auf einer empirischen Studie. In *Compiler Error Messages Considered Unhelpful: The Landscape of Text-Based Programming Error Message Research* [Bec+19] haben die 12 Autoren einen Korpus aus 307 Papern, in denen es darum ging Programmier-Fehlermeldungen zu untersuchen und zu verbessern, zusammengestellt und diesen gründlich untersucht.

Insgesamt konnten sie 10 überlappende Bereiche für Empfehlungen, um Fehlermeldungen besser zu machen, identifizieren. Für einige Bereiche gibt es wenige evidenzbasierte Studien und auch gegensätzliche Meinungen. Die Bereiche und gegensätzliche Ansichten werden hier nochmal kurz zusammengefasst:

- 1) **Lesbarkeit erhöhen:** Ein Kriterium ist die Lesbarkeit zu erhöhen. Verschiedene Autoren verwenden verschiedene Begriffe um die Lesbarkeit zu beschreiben, wie, dass eine Nachricht verständliche, einfache Sprache oder bekanntes Vokabular enthalten muss, wie auch oben Marceau u. a. angeben. Jedoch keiner gibt genau an, wie die Lesbarkeit genau sicher gestellt oder gemessen werden kann.
- 2) **Die kognitive Belastung reduzieren:** Dies bezieht die CLT mit ein, um Fehlermeldungen zu verbessern. So lautet die generelle Empfehlung Fehlermeldungen möglichst einfach und minimal zu halten. Andere neuere Arbeiten geben spezifische Empfehlungen ab, um das Arbeitsgedächtnis möglichst nicht zu überlasten: Platziere relevante Informationen in der Nähe des fehlerhaften Codes, vermeide Redundanz, sodass der Benutzer nicht die selben Informationen mehrmals verarbeiten muss und gebe multimodales (textliches, visuelles, audiotives, ...) Feedback.
- 3) **Kontext zum Fehler angeben:** Damit ist gemeint Informationen über den Code zu geben, welcher relevant für den Fehler ist. Dies ist zum einem die Position des Fehlers im Code sowie Symbole, Bezeichner, Literale und Typen, die für den Fehler relevant sind. Eine gute Möglichkeit, um einen Fehler im Code anzuzeigen ist das Hervorheben dieses im Text-Editor. Das kennt man in modernen IDEs durch rote Kringel, die unter dem fehlerhaften Code angezeigt werden. Oder wie Marceau u. a. angeben, kann das auch noch erweitert werden, sodass der Code verschiedenfarbig gekennzeichnet wird und eine Zuordnung über die Farben zu verschiedenen Teilen der Fehlermeldung möglich ist.
- 4) **Einen positiven Ton verwenden:** Fehlermeldungen sollten nicht aggressiv, vag oder obskur formuliert werden sollten, da der Benutzer sich dadurch angegriffen fühlen oder es ihn frustrieren kann. Stattdessen sollten die Fehlermeldungen höflich, freundlich oder auch ermutigend formuliert werden. Einige Autoren geben zudem an, dass Wörter wie „illegal“, „inkorrekt“ und „ungültig“ in den Fehlermeldungen vermieden werden sollten.
Der Einbau von Humor in Fehlermeldungen wird auch diskutiert, da Beginner manchmal Werkzeuge als „kalt“ und „wertend“ wahrnehmen. Einige Autoren argumentieren gegen Humor, da dieser die Fehlermeldungen länger machen, der Humor misinterpretiert werden kann und nicht der Eindruck entstehen soll, dass ein Computer empfindungsfähig ist und denken kann.
- 5) **Beispiel-Code angeben:** Hiermit ist gemeint, dass wenn ein Fehler bezüglich eines bestimmtes Sprachfeatures auftritt, Beispielcode angezeigt wird, wie dieses Sprach-

feature benutzt werden kann. Es gibt einige evidenzbasierte Studien, die zeigen, dass der Beispielcode für Schüler hilfreich ist. In einer anderen Studie jedoch wurde jedoch gezeigt, dass Beispielcode für Beginner verwirrend sein kann, da sie denken können, dass mit dem Beispielcode ihr eigener Code gemeint ist und dann eine lange Zeit nach dem Beispielcode in ihrem Code suchen.

- 6) **Lösungen oder Tipps anzeigen:** Wenn es eine hohe Wahrscheinlichkeit gibt, dass ein Fehler durch eine bestimmte Lösung korrigiert werden kann, ist es eine gute Idee, die Lösung anzuzeigen.

Eine andere Möglichkeit ist es *Crowdsourcing* zu benutzen, um die Lösung eines Fehlers anzuzeigen. Es können z.B. Lösungen von der Webseite *Stack Overflow* eingebunden werden.

- 7) **Eine dynamische Interaktion erlauben:** Eine Möglichkeit ist, dass Fehlermeldungen dynamisch sind und die Fehlerhistorie des Benutzers mit einbeziehen.

Eine weitere Möglichkeit ist es, Fehlermeldungen so zu gestalten, dass mehr Kontext und Informationen bei Bedarf angezeigt werden – bspw. über eine *tell-me-more*-Option.

Einige Autoren geben an, dass der Compiler graduell die Hilfestellungen in Fehlermeldungen reduzieren sollte, sodass Anfänger nach und nach „richtigen“ Fehlermeldungen ausgesetzt werden.

Eine andere Idee ist das Konzept einer empathischen IDE, die auch die emotionalen Aspekte des Benutzers berücksichtigt und bspw. Fehler, die der Benutzer wiederholt macht, anders ausgibt.

- 8) **Scaffolding für Benutzer anbieten:** Mit *Scaffolding* ist gemeint, dass eine unterstützende Struktur gegeben ist, mit dem Beginner ihr Wissen aufbauen können, vergleichbar mit dem Konzept von Stützräder zum Erlernen des Fahrradfahrens. So könnten in Fehlermeldungen für Beginner eine Erklärung dafür angegeben werden, warum diese Nachricht für sie angezeigt wird.

Es wird auch gegen *Scaffolding* argumentiert mit dem Argument, dass es unmöglich ist vorherzusehen, welche Missverständnisse ein Beginner hat und die Nachricht dann inpräzise formuliert werden müsste.

- 9) **Logische Argumentation angeben:** Eine sehr neue Entwicklung im Bereich der Programmierfehlermeldungen ist es einen starken Fokus auf korrekte logische Argumentation zu legen. So können konkrete Behauptungen aufgestellt und Beweise für diese Behauptungen angegeben werden, sodass Benutzer die Fehler nachvollziehen und korrigieren können.

- 10) **Fehler zur richtigen Zeit anzeigen:** Fehler sollten zur richtigen Zeit angezeigt werden. Die neuesten Arbeiten sind der Ansicht, dass durch statische Analyse-Werkzeuge, Fehler so früh wie möglich angezeigt werden sollen. Dies kennt man wieder von modernen IDEs, die so früh wie möglich rote Kringel anzeigen.

Insgesamt ergibt sich ein komplexes Bild von Evidenzen und Meinungen, wie Fehlermeldungen für Beginner zu gestalten sind. Auf der einen Seite gibt es die Ansicht explizite Hilfsmittel für Beginner in Fehlermeldungen einzubauen, auf der anderen Seite die Be-

denken, dass diese Hilfsmittel ablenken oder verwirren könnten. Zudem die Einsicht, dass unterstützende Strukturen (*Scaffolding*) mit der Zeit abgebaut werden sollten, wie wenn ein Kind vom Stützrad auf das ungestützte Fahrrad umsteigt. Wie effektiv die Hilfsmittel sind hängt am Ende auch immer stark von den Implementierungsdetails ab. Insgesamt sollte jede „Verbesserung“ von Fehlermeldungen genau abgewägt werden und eine Entscheidung sollte sich auf evidenzbasierte Studien stützen.

Eine Kriterium, das noch nicht genannt wurden ist, ist die Lokalisierung von Fehlermeldungen, sodass Fehlermeldungen in die Muttersprache des Beginners übersetzt wird, sodass die Fehlermeldungen besser verstanden werden können.

Ebenso ist zu berücksichtigen, welche neuen Möglichkeiten durch die Fortschritte in *Machine Learning* sich für Programmierfehlermeldungen eröffnen. So geben Becker u. a an, dass die Verbesserung von Fehlermeldungen in der Zukunft basierend auf Entwicklungen in verwandten Bereichen möglicherweise nicht mehr erforderlich ist (vgl. [Bec+19]). Becker u. a verweisen dann auf die neuesten Entwicklungen in dem Bereich der Fehlermeldungen, wo *Crowdsourcing* und *Machine Learning* verwendet wird, um automatisiertes Feedback zu geben.

Bewertungskriterien

Die Bewertungskriterien für Fehlermeldungen setzen sich aus den Verbesserungsvorschlägen zusammen, welche relativ unumstritten und es schon gut Belege für ihre Wirksamkeit gibt.

1. Es soll eine einfache Sprache und vereinfachtes programmiertechnisches Vokabular verwendet werden.
2. Die Fehlerposition im Code soll angezeigt werden und gut bestimbar sein.
3. Die Fehlermeldungen enthält für Anfänger unterstützende Elemente bspw.:
 - a) Fehlermeldungenabschnitte und der korrespondierende Code werden farblich gekennzeichnet.
 - b) Eine Lösung wie der Fehler behoben werden kann, wenn eine Lösung mit einer sehr hohen Wahrscheinlichkeit identifiziert werden kann.
4. Die Fehlermeldung darf keine störende Elemente enthalten, die die Fehlermeldung unnötig verlängern oder den Benutzer verwirren könnten.
5. Die Fehlermeldung darf den Benutzer keine Lösung vorschlagen, die ihn dazu veranlasst einen falschen Weg einzuschlagen.
6. Die Fehlermeldung sollte in einem positiven Ton verfasst sein.
7. Die Fehlermeldung sollte allen Kontext und alle Informationen anzeigen, die benötigt werden, um den Fehler zu beheben.
8. Die Fehlermeldung sollte lokalisiert sein.

3.4.10. Ein hohes Abstraktionsniveau für Datentypen

Eine *Teaching Language* soll eine höhere Programmiersprache sein, in der es einen *Garbage Collector* gibt, der den Speicher automatisch verwaltet und der Benutzer sich nicht mit *Low-Level*-Details wie z.B. Pointerarithmetik auseinander setzen muss.

Ein hohes Abstraktionsniveau kann für eine *Teaching Language* vorausgesetzt werden, es sei denn es handelt sich um eine *Teaching Language*, die genau diese *Low-Level*-Details beibringen möchte.

Nur sollen ebenso die Datentypen ein hohes Abstraktionsniveau aufweisen. In *Low-Level*-Sprachen wie C, wird auf eine Zeichenfolge über einen Char-Pointer zugegriffen. Der Pointer zeigt auf das erste Zeichen und um die ganze Zeichenfolge zu durchlaufen, erhöht man die Speicheradresse des Pointers solange, bis man auf den Nullterminator \0 stößt.

Eine *Teaching Language* für Anfänger sollte sich nicht mit diesen *Low-Level*-Details befassen. Für den String-Datentypen sollte der *Unicode*-Standard verwendet werden, sodass alle möglichen Zeichen unterstützt werden. Der Benutzer sollte nicht mit Details wie der Kodierung in Kontakt kommen. Zeichenfolgen-Operationen wie Verkettungen sollten einfach funktionieren. Auch die Ausgabe von Zeichenfolgen sollte *Unicode* unterstützen.

Gleichermaßen sollten Zahlen auf einer hohen Abstraktionsebene implementiert sein. In vielen Programmiersprachen gibt es unterschiedliche Datentypen wie long und int für Ganzzahlen und float und double für Fließkommazahlen. In diesen Sprachen hat der Programmierer große Kontrolle darüber, wie viel Speicher eine Zahl verwendet, muss sich jedoch auch genau überlegen welchen Datentypen er nimmt und sich mit Problemen wie *Integer-Overflow* oder der Ungenauigkeit der Fließkommazahlen herumschlagen. Öffnet man beispielsweise den *Node.js* REPL und gibt folgendes ein ...

```
> 0.2 + 0.1
0.3000000000000004
> 0.2 + 0.1 === 0.3
false
```

Listing 3.10. Fließkommaarithmetik in Javascript

... sieht man schnell, was das Problem ist. In Javascript ist der Zahlentyp nämlich einfach als 64-Bit-Fließkommazahl implementiert.

Eine Lösung dafür bietet die Langzahlarithmetik. Hier wird für eine Zahl im Speicher so viel Speicher verwendet, wie benötigt wird. Der Benutzer muss sich nicht mehr darum kümmern, welcher Datentyp für eine Zahl verwendet wird. Die *Teaching Language* sollte also nur einen Zahlentyp haben, welcher über Langzahlarithmetik implementiert ist.

Bewertungskriterien

1. Die Sprache hat einen String-Datentyp auf hohem Abstraktionsniveau, welcher den *Unicode*-Standard implementiert. Zeichenfolgen werden auch richtig in *Unicode* ausgetauscht.
2. Die Sprache hat einen einzigen Zahlentyp, welcher über die Langzahlarithmetik implementiert ist.

3.4.11. Anwendungsbereiche und Interoperabilität mit Technologien und anderen Programmiersprachen

Die *Teaching Language* soll möglichst viele Anwendungsbereiche und eine Interoperabilität mit anderen Technologien und Programmiersprachen besitzen.

Beispiel für Anwendungsbereiche neben Computerspielen sind:

- 2D-Grafik
- 3D-Grafik
- GUI-Programming
- Audio, Musik, Audiomixing, Signalverarbeitung
- Datenanalyse (*Data Science*)
- Machine Learning
- Netzwerkprogrammierung
- Web-API
- Roboter, IOT, Mikrocontroller
- Datenbanken

Auch ist es interessant, für welche Plattformen Anwendungen geschrieben werden können. Laufen Anwendungen im Web, als Desktopanwendung, als App auf dem Smartphone oder sogar in eingebetteten System wie Mikrocontrollern?

Obendrein ist es auch noch wichtig zu wissen, ob die Sprache interoperabel mit anderen Sprachen ist. Damit ist gemeint, ob sie Bibliotheken aus anderen Sprachen wie z.B. C oder Java aufrufen kann.

Zuletzt gibt es auch noch die Möglichkeit, dass die Sprache in eine anderer Programmiersprache transpiliert werden kann, sie also in eine andere Sprache übersetzt werden kann.

Bewertungskriterien

In diesem Bewertungsbereich wird auf die Quantität der Möglichkeiten geschaut und nicht auf die Qualität. So kann eine *Teaching Language* einen Anwendungsbereich zwar unterstützen, es wird jedoch nicht genauer untersucht, wie gut die Implementierung ist.

1. Wie viele und welche Anwendungsbereiche erlaubt die Sprache?
2. Auf welchen Plattformen kann ein Programm in der Sprache ausgeführt werden?
3. Ist die Sprache interoperabel mit anderen Sprachen oder kann sie in andere Sprachen transpiliert werden?

3.4.12. Dokumentation & Community Support

Eine gute Dokumentation, die die Sprache sowie die Standardbibliothek gut erklärt, ist in jeder Programmiersprache wichtig. So können Konzepte, die noch unklar sind jederzeit nachgeschlagen werden.

Eine große und hilfreiche Community, in der erfahrene Programmierer Beginnern helfen, ist für eine *Teaching Language* besonders hilfreich. Dazu zählen offizielle und inoffizielle Foren und Gruppen in denen sich ausgetauscht werden kann.

Außerdem sind Tutorials, die die Sprache einführen wichtig. Besonders gut ist es, wenn die Tutorials die Sprache anhand eines spannenden Beispiels einführen oder es mehrteilige Tutorials gibt, die das Programmieren einer komplexeren Anwendung zeigen.

Zudem kommt hinzu, dass Dokumentation, Tutorials und Beispiele internationalisiert und in mehreren Sprachen verfügbar sein sollte, sodass es keine Sprachbarriere gibt.

1. Wie qualitativ ist die Dokumentation?
2. Wie groß/hilfreich ist die Community?
3. Gibt es offizielle/inoffizielle Foren und/oder Gruppen?
4. Gibt es Beispiele und Tutorials und wenn ja welche und wie viele?
5. Ist die Dokumentation, Tutorials, Beispiele und Lehrmaterial internationalisiert und in mehreren Sprachen verfügbar?

3.5. Bewertungsbereichübergreifende Eigenschaften einer Programmiersprache und Reflexion

In diesem Abschnitt werden einige Eigenschaften einer Programmiersprache, die mehreren Bewertungsbereichen zuzuordnen sind, beschrieben und dabei die korrespondierenden Bewertungskriterien reflektiert.

3.5.1. Syntaktischer Zucker

Dieser Abschnitt greift Kriterien der Bereiche **Inkrementelle Einführung** (Abs. 3.4.3), **Intuitive Syntax und Natürlichsprachlichkeit** (Abs. 3.4.4) und **Feature Uniformity und Orthogonalität** (Abs. 3.4.5) auf und reflektiert diese.

Syntaktischer Zucker macht eine Programmiersprache nicht mächtiger, sondern das Ziel ist es sie lesbarer und ausdrucksfähiger zu machen. Er macht die Sprache für den menschlichen Gebrauch „süßer“ (vgl. [Wik21]). Somit unterscheidet sich syntaktischer Zucker nicht wesentlich von einem Makro. Die Funktionalität Makros erstellen zu können, macht eine Programmiersprache mächtiger und erlaubt es Benutzern eigenen syntaktischen Zucker für die Programmiersprache zu definieren. So wie ein Marko expandiert wird, wird auch syntaktischer Zucker expandiert. Dies wird *Desugaring* (zu Dt. *Entsüßen*) genannt. Dabei werden die versüßten Syntax-Konstrukte in die Kern-Syntaxkonstrukte der Sprache transformiert. Ein Beispiel dafür gibt Felleisen in seinem Paper von 1991 *On the expressive power of programming languages* ([Fel91]) an. So kann ein `repeat .. until`-Konstrukt in ein `while`-Konstrukt überführt werden:

(**repeat** *s* **until** *e*) *is expressible as* (*s*; **while** $\neg e$ **do** *s*)

(s. [Fel91])

In seinem Paper beschreibt Felleisen formal wie syntaktischer Zucker von Features, die die Sprache mächtiger machen (z.B. Erstellen von Makros) unterschieden werden können.

Eine *Teaching Language* für Beginner sollte nicht allzu mächtig sein, da die Sprache dadurch komplexer und schwerer zu erlernen ist (vgl. Abs. 3.4.5). Syntaktischer Zucker macht eine Sprache jedoch nicht mächtiger, sondern les- und schreibbarer und macht sie für Beginner somit zugänglicher (vgl. Abs. 3.4.4).

Ein Beispiel für gut eingesetzten syntaktischen Zucker geben Crestani und Sperber in ihrem Bericht *Experience Report: Growing Programming Languages for Beginning Students* anhand der Programmiersprache Scheme an. Die Funktionsdefinition kann hier mit einem expliziten Lambda definiert werden

```
(define f
  (lambda (x)
  ...)

... oder ohne Lambda

(define (f x)
  ...)
```

... geschrieben werden, was syntaktischer Zucker ist. Crestani und Sperber vergleichen dabei die pädagogischen Herangehensweisen von den funktionalen Einführungsprogrammierkursen *How to Design Programming* (HtDP) von Felleisen entwickelt ([Fel+18]) und von ihrem Einführungskurs *Die Macht der Abstraktion* (DMdA) ([KS07]).

Die Schreibweise einer Funktionsdefinition wurde dabei in HtDP ohne explizites Lambda und in DMdA mit explizitem Lambda gewählt. Die Autoren begründen ihre Wahl dadurch, dass wenn Funktionen höherer Ordnung eingeführt werden, das Lambda direkt schon bekannt ist und ansonsten das Konzept des syntaktischen Zucker erklärt werden müsste. Ein Argument gegen die Entscheidung von DMdA ist, dass es die Einführung der Sprache für Beginner komplizierter macht, da es syntaktische Elemente gibt, die erst später verstanden werden müssen (vgl. Abschnitt 3.4.3).

Syntaktischer Zucker fällt in die Gruppe der syntaktischen Synonyme, die Gupta für eine *Teaching Language* negativ bewertet (vgl. Abs. 3.4.5), da statt dem syntaktischen Zucker auch die ungezuckerte Syntax verwendet werden kann. Als Beispiel wurde der Inkrementieroperator `++` aus der Programmiersprache C angegeben. Es ist schwer zu entscheiden, ob es sich bei dem Inkrementieroperator um syntaktischen Zucker handelt. Nach Felleisen handelt es sich um syntaktischen Zucker, wenn die Syntax (wie ein Makro) in eine Teilemenge der Sprache expandiert werden kann und diese Expandierung die Korrektheit und Semantik aller Programme bewahrt (vgl. [Fel91]). So kann der Inkrementieroperator in der Präfix-Variante als syntaktischer Zucker gesehen werden (s. Listing unten), jedoch nicht in der Postfix-Variante, da es keine korrekte Transformierung gibt, die die Semantik des Postfix-Operators übernimmt.

```
int x = 0;
printf("%d", ++x); // 1
```

Listing 3.11. Präfix-`++` in C

```
int x = 0;
printf("%d", x = x + 1); // 1
```

Listing 3.12. Desugaring des Präfix-`++` in C

Für die Bewertung der *Feature Uniformity* in Abschnitt 3.4.5 sollte abgewogen werden, ob die Vorteile des syntaktischen Zuckers der besseren Les- und Schreibbarkeit sowie einer inkrementellen Einführung, die Nachteile des syntaktischen Synonyms überwiegen.

3.5.2. Statische vs. Dynamische Typisierung

Im Bewertungsbereich **Erlernbarkeit nach Konstruktionismus** wird für eine dynamische Typisierung argumentiert wohingegen im Bewertungsbereich **Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplin (Clean Code)** für ein statische Typisierung wird.

Dynamische Typisierung ist besonders gut fürs *Prototyping* und erlaubt es sehr flexibel zu sein und Programme auszudrücken, die so in einer statisch typisierten Programmiersprache nicht möglich sind. Eine statische Typisierung hingegen stellt einen großen Teil der Korrektheit eines Programms sicher und kann von Tools verwendet werden, um den Programmierer z.B. durch *Code-Completion* zu unterstützen.

Graduelle Typisierung verbindet dynamische und statische Typisierung miteinander und ist somit optimal für eine *Teaching Language* geeignet. Bei der graduellen Typisierung sind Typannotationen optional und typisierte Teile des Programms werden statisch überprüft und während der Laufzeit können dann noch dynamische Typ-Überprüfungen erfolgen.

Es ist unklar, woher die Idee der graduellen Typisierung stammt. Zudem gibt es eine Reihe von weiteren Begriffen wie *quasi-static*, *optional* oder *pluggable* Typisierung, die auf das gleiche Konzept hinauswollen.

Ein Nachteil der graduellen Typisierung ist, dass die Implementierung dieser komplizierter als die der statischen Typisierung ist. Und die der statischen Typisierung ist schon kompliziert.

4. Implementierung und Bewertung der Teaching Languages

4.1. Pyret

4.1.1. Kurze Einführung in Pyret

Pyret ist eine funktionale, graduell typisierte Programmiersprache, welche eigens dazu geschaffen wurden ist, funktionale Programmierung und Skripting beizubringen (vgl. [Pyr21a]).

4.1.2. Beschreibung der Programmierumgebung

Pyret kann in einer Web-Umgebung, sowie über ein *NPM-Package* in *Node.js* innerhalb der Kommandozeile ausgeführt werden. Zudem gibt es für *Visual Studio Code* eine Pyret-Erweiterung, die jedoch bisher nur Syntax-Highlighting unterstützt. Über das *NPM-Package* können aber bislang keine interaktiven Anwendungen erstellt werden, da die benötigten Bibliotheken für interaktive Anwendungen dort nicht unterstützt werden. Deshalb und weil die Web-Umgebung sowieso die bessere Erfahrung für Programmieranfänger bietet, wird sich hier ausschließlich auf die Web-Umgebung fokussiert.

Die Web-Umgebung ist in einen linken und rechten Bereich unterteilt. Links kann der Pyret-Quellcode eingegeben werden und rechts ist ein interaktiver Bereich. Dabei hat dieser interaktive Bereich mehrere Funktionen:

- Ausgabe des ausgeführten Programms
- *Read Eval Print Loop* (REPL): Nach dem Ausführen des Programms, ist das Programm in die REPL geladen, sodass Funktionen, Konstanten und Variablen des geschriebenen Programms inspiziert werden können.
- Ausgabe von Unit-Test-Ergebnissen: In Pyret gibt es dedizierte Syntax, um geschriebene Funktionen, direkt nach der Definition zu testen. Falls Unit-Tests geschrieben wurden sind, werden sie beim Ausführen des Programms auch automatisch ausgeführt und die Ausgabe dieser in der Konsole angezeigt.

4.1.3. Interaktive Anwendungen in Pyret

Pyret verwendet das funktionale reaktive Programmierparadigma, um interaktive grafische Anwendungen zu erstellen. Reaktives Programmieren erleichtert die deklarative Entwicklung von ereignisgesteuerten Anwendungen, indem der Programmierer sich darauf konzentriert „was zu tun ist“ und die Sprache (bzw. die Bibliothek) automatisch verwaltet, wann es zu tun ist (vgl. [Bai+13]).

Funktionale reaktive Programmierung ist sehr populär. Ein bekanntes Beispiel für die Popularität ist die Javascript-Bibliothek *ReactJS*, mit der Web-Benutzerschnittstellen ent-

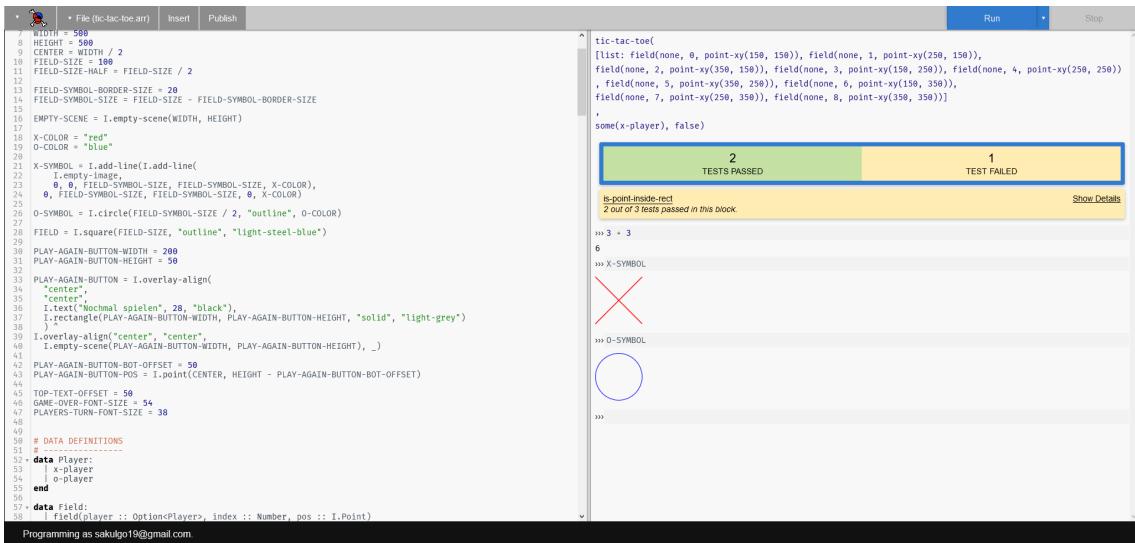


Abb. 4.1. Programmierumgebung in Pyret

wickelt werden können und welche *Open Source* ist und von Facebook verwaltet wird (vgl. [Agg18]).

Um eine interaktive Anwendung in Pyret zu erstellen, wird die `world`-Bibliothek verwendet, welche ein Teil der Standardbibliothek ist. Dabei wird die Funktion `big-bang` der Bibliothek aufgerufen, um die „World“ (die interaktive Anwendung) zu erstellen. Die Funktion bekommt als ersten Parameter einen Startzustand für die Welt und als zweiten Parameter eine Liste von *Event-Handlern* übergeben. Die Liste muss nicht vollständig sein – es können also nur die *Event-Handler* verwendet werden, welche benötigt werden:

```

1 import world as W
2
3 W.big-bang(init,
4   [list:
5     W.on-tick(<expr>),
6     W.on-tick-n(<expr>),
7     W.on-mouse(<expr>),
8     W.on-key(<expr>),
9     W.to-draw(<expr>),
10    W.stop-when(<expr>)
11  ]
12 )

```

Listing 4.1. Weltenerstellung in Pyret

Ein *Event-Handler* ist eine Funktion, die aufgerufen wird, wenn das Event eintritt. Die unterschiedlichen Events sind über die Namen ziemlich selbsterklärend. Am wichtigsten sind die Events `on-tick` und `to-draw`. Das Event `on-tick` wird immer ausgelöst, wenn der Zustand der Welt aktualisiert werden soll. Dies geschieht standardmäßig mit 28 FPS. Dem Event wird als *Event-Handler* eine Funktion übergeben, die den vorherigen Welt-Zustand als Parameter übergeben bekommt und den neuen Welt-Zustand zurückgibt. Das Event `to-draw` wird immer ausgelöst, wenn die Welt gezeichnet werden soll. Hier wird wieder der Welt-Zustand als Parameter übergeben und als Rückgabewert wird etwas vom Typ `Image`

erwartet, das dann gezeichnet werden kann.

Die `image`-Bibliothek stellt Funktionen bereit, um Bilder zum einen zu erstellen (z.B. `circle`, `rectangle`, `image-url`), Bilder zu transformieren (z.B. `rotate`, `scale`, `flip-horizontal`) und Bilder zu positionieren und zu kombinieren (z.B. `place-image`, `overlay`, `underlay`). Es wird das funktionale Paradigma eingehalten, es handelt sich bei den Funktionen also um pure Funktionen, bspw. wird bei der Funktion `rotate` kein Bild verändert (mutiert), sondern es wird ein ganz neues rotiertes Bild zurückgegeben.

Um den Quellcode übersichtlich zu halten, habe ich mir eine Programmstruktur überlegt, welche die Verwendungsweise der `world`-Bibliothek berücksichtigt:

- 1) **Definition von Konstanten:** Hier werden Konstanten definiert.
- 2) **Definition von Datentypen:** Hier werden benutzerdefinierte Datentypen definiert.
- 3) **Erstellen des initialen Welt-Zustands:** Hier wird der initiale Welt-Zustand erstellt, welcher der `bigbang`-Funktion als ersten Parameter übergeben wird.
- 4) **Hilfefunktionen:** Hier werden Funktionen definiert, welche nicht mit der Spiele-Logik zusammenhängen.
- 5) **Zeichenfunktionen:** Hier werden Funktionen definiert, welche die Welt zeichnen und von dem `to-draw`-Event aufgerufen werden.
- 6) **Updatefunktionen:** Hier werden Funktionen definiert, welche die Welt updaten und von dem `on-tick`-Event aufgerufen werden.
- 7) **Event-Handlers:** Hier werden Funktionen definiert, welche auf Events, wie Mausklicks oder Tastendrücke reagieren.
- 8) **Welten-Erstellung:** Hier werden die Teile zusammengefügt und die `bigbang`-Funktion aufgerufen.

4.1.4. Datendefinitionen in Pyret

Datendefinitionen sehen in Pyret so aus, dass hinter dem Schlüsselwort `data` der Typname geschrieben wird und dann Konstruktoren folgen. Dabei kann ein Konstruktor verschiedene Felder haben (s. 4.2). Es handelt sich um sogenannte *Algebraische Datentypen*, die auch in anderen funktionalen Programmiersprachen wie Haskell verwendet werden. Bei Algebraischen Datentypen wird unterschieden zwischen:

- Summentypen
- Produkttypen

(vgl. [Wik20b])

Summentypen heißen so, da die Anzahl möglicher Werte für diesen Datentyp sich aus der Summe der Typvarianten berechnen lässt und sind in Pyret die verschiedenen Konstruktoren zusammengenommen.

Produkttypen heißen so, da die Anzahl möglicher Werte für diesen Typ aus dem Produkt der Feldtypen sich zusammensetzt, und sind in Pyret ein einzelner Konstruktor.

Dies wird im Folgenden kleinen Beispiel anhand der Modellierung eines Schaltkreises in Pyret an einem Beispiel erläutert:

```

1  data Schalter:
2    | auf
3    | zu
4  end
5
6  data Reihe:
7    | reihe(a :: Schalter, b :: Schalter)
8    | schalter(s :: Schalter)
9  end
10
11 data Parallel:
12   | parallel(a :: Reihe, b :: Reihe)
13 end

```

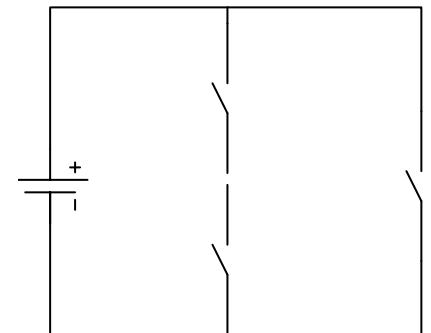


Abb. 4.2. Beispielschaltung

Listing 4.2. Schaltungs-Code in Pyret

Die Beispielschaltung in Abb. 4.2 wird in Pyret mit `parallel(reihe(auf, auf), schalter(auf))` erstellt. Es stellt sich jetzt die Frage, wie viele verschiedene Schaltungen mit dem Datentyp `Parallel` erstellt werden kann. Hierfür wird die Notation $|T|$ für die Anzahl aller Werte für den Typen T verwendet. Jetzt kann die Anzahl der Werte für den Datentypen `Parallel` berechnet werden:

$$\begin{aligned}
 |Schalter| &= |auf| + |zu| \\
 &= 1 + 1 = 2 \\
 |Reihe| &= |reihe| + |schalter| \\
 &= |Schalter| * |Schalter| + |Schalter| \\
 &= 2 * 2 + 2 = 6 \\
 |Parallel| &= |parallel| \\
 &= |Reihe| * |Reihe| \\
 &= 6 * 6 \\
 &= 36
 \end{aligned}$$

Es können also 36 verschiedene Schaltungen mit der Datenstruktur `Parallel` erstellt werden. Man würde Schaltungen nicht wie oben im Listing 4.2 modellieren, da man mit dieser Modellierung nur 36 Varianten von Schaltungen erstellen kann. Dieses Beispiel wurde nur so gewählt, um eine endliche Anzahl von Schaltungs-Varianten zu haben. Vielmehr würde man eine Schaltung wie in Listing 4.3 modellieren. Hier kann eine Schaltung, entweder ein Schalter, eine Reihen- oder eine Parallelschaltung sein. Da Schaltungen somit beliebig unendlich verschachtelt werden können gilt dann $|Schaltung| = \infty$.

```

1  data Schalter:
2    | auf
3    | zu
4  end
5

```

```

6   data Schaltung:
7     | schalter(s :: Schalter)
8     | reihe(a :: Schaltung, b :: Schaltung)
9     | parallel(a :: Schaltung, b :: Schaltung)
10    end

```

Listing 4.3. Verbesserter Schaltungs-Code

4.1.5. Bouncing Ball

Mit der in Abschnitt 4.1.3 definierten Code-Struktur in Pyret ist der Code für die Implementierung des *Bouncing Balls* recht gut verständlich:

```

1 import world as W
2 import image as I
3
4 # 1. Konstanten
5 WIDTH = 500
6 HEIGHT = 300
7 SCENE = I.empty-scene(WIDTH, HEIGHT)
8 RADIUS = 10
9 BALL = I.circle(RADIUS, "solid", "red")
10 SPEED-X = 10
11 SPEED-Y = 10
12
13 # 2. Datentypen
14 data Ball:
15   ball(x :: Number, y :: Number, vx :: Number, vy :: Number)
16 end
17
18 # 3. Initialer Weltzustand
19 INIT_BALL = ball(WIDTH / 2, HEIGHT / 2, SPEED-X, SPEED-Y)
20
21 # 4. Hilfefunktionen
22 fun clamp(n :: Number, min :: Number, max :: Number) -> Number:
23   num-max(min, num-min(max, n))
24 end
25
26 # 5. Zeichenfunktion
27 fun draw(b :: Ball) -> I.Image:
28   I.place-image(BALL, b.x, b.y, SCENE)
29 end
30
31 # 6. Updatefunktion
32 fun update(b :: Ball) -> Ball:
33   vx = if (b.x <= RADIUS) or (b.x >= (WIDTH - RADIUS)):
34     b.vx * -1 else: b.vx end
35
36   vy = if (b.y <= RADIUS) or (b.y >= (HEIGHT - RADIUS)):
37     b.vy * -1 else: b.vy end
38
39   x = clamp(b.x + vx, RADIUS, WIDTH - RADIUS)
40   y = clamp(b.y + vy, RADIUS, HEIGHT - RADIUS)
41

```

```

42     ball(x, y, vx, vy)
43 end
44
45 # 7. Event-Handlers
46 # keine Event-Handler
47
48 # 8. Welten-Erstellung
49 W.big-bang(INIT_BALL, [list:
50     W.on-tick(update),
51     W.to-draw(draw)
52 ])

```

Listing 4.4. Bouncing Ball in Pyret

Der Hilfunktion `clamp` wird eine Zahl, ein Minimum und ein Maximum übergeben. Wenn die Zahl kleiner als das Minimum ist, wird das Minimum zurückgegeben. Wenn die Zahl größer als das Maximum ist, wird das Maximum zurückgegeben. Ansonsten wird die Zahl unverändert zurückgegeben.

In der `draw`-Funktion, wird ein Bild zurückgegeben, indem das `BALL`-Bild an der Position des Balles auf die Szene `SCENE` platziert wird.

In der `update`-Funktion, wird die x-Geschwindigkeit invertiert, falls der Ball am linken oder rechten Rand der Szene ist. Das gleiche wird mit der y-Geschwindigkeit gemacht. Schließlich werden die Geschwindigkeiten auf die Position addiert. Hierbei wird die Hilfsfunktion `clamp` verwendet, sodass die Ball-Position niemals außerhalb der Szene ist. Am Schluss wird ein neuer Ball mit der aktualisierten Position und Geschwindigkeit zurückgegeben.

4.1.6. Tic-Tac-Toe

Der komplette Quellcode ist im Anhang (s. C.3) zu finden.

Konstanten

Zuerst werden die Konstanten definiert. Dazu gehören die Weite und die Höhe der Zeichenoberfläche in Pixeln, die Farben der Spieler, die Symbole der Spieler, die Tic-Tac-Toe-Felder und der Button, mit dem man am Ende das Spiel neu starten kann.

Für die Konstanten, welche Bilder sind, kann die interaktive Konsole in Pyret sehr gut benutzt werden, um zu sehen, was der Code produziert, da diese die Bilder direkt ausgibt.

Für das X-Symbol wird mit der Funktion `add-line` der `image`-Bibliothek zwei Linien auf ein leeres Bild (`empty-image`) hinzugefügt.

```

>>> I.add-line(
    I.add-line(I.empty-image, 0, 0, FIELD-SYMBOL-SIZE, FIELD-SYMBOL-SIZE, X-COLOR),
    0, FIELD-SYMBOL-SIZE, FIELD-SYMBOL-SIZE, 0, X-COLOR)

```



Abb. 4.3. Code für das X-Symbol in der Konsole

Da ineinander verschachtelte Funktionsaufrufe nicht so gut lesbar sind, kann der Code für das X-Symbol mittels *Curried Application Expressions* kombiniert mit dem *Chaining-Application-Operator* \wedge umgeschrieben werden:

```
1 X-SYMBOL = I.empty-image  $\wedge$ 
2   I.add-line(_, 0, 0, FIELD-SYMBOL-SIZE, FIELD-SYMBOL-SIZE, X-COLOR)  $\wedge$ 
3   I.add-line(_, FIELD-SYMBOL-SIZE, 0, 0, FIELD-SYMBOL-SIZE, X-COLOR)
```

Currying funktioniert in Pyret so, dass für beliebige Argumente eines Funktionsaufrufs ein Unterstrich $_$ eingefügt werden kann. Der Funktionsaufruf gibt dann eine Funktion zurück, bei der die Argumentwerte noch eingefügt werden müssen.

Mittels dem *Chaining Application Operator* \wedge , können Funktionsaufrufe mit nur einem Argument umgeschrieben werden: $e1 \wedge e2$ ist äquivalent zu $e2(e1)$. Links steht ein Ausdruck, der auf die rechte Funktion angewandt wird. Das Ganze ist linksassoziativ, sodass beliebig lange Funktionsanwendungsketten geschrieben werden können.

Die Umschreibung bringt den Vorteil, dass der Code besser lesbar wird, da die Operationen die nacheinander ausgeführt werden nun nicht mehr von innen nach außen, sondern von links nach rechts zu lesen sind, was die natürliche westliche Leserichtung ist. Der Nachteil ist natürlich, dass funktionale Konzepte wie *Currying* für Programmieranfänger noch recht schwer zu verstehen sind, sodass eine gewisse Erfahrung in funktionaler Programmierung vorausgesetzt wird.

Da es in Pyret nur die `image`-Bibliothek für grafische Elemente gibt, und keine Bibliothek für GUI-Elemente, muss der Button über Bilder erstellt werden. Hierfür wird dem Rechteck den Button-Text und noch einen Rahmen mittels einer `empty-scene` hinzugefügt, welche immer umrandet ist:

```
1 PLAY-AGAIN-BUTTON =
2 I.rectangle(200, PLAY-AGAIN-BTN-HEIGHT, "solid", "light-grey")  $\wedge$ 
3 I.overlay-align("center", "center", I.text("Nochmal spielen", 28, "black"), _)  $\wedge$ 
4 I.overlay-align("center", "center", I.empty-scene(PLAY-AGAIN-BTN-WIDTH,
5   PLAY-AGAIN-BTN-HEIGHT), _)
```

Listing 4.5. Erstellung eines Buttons in Pyret



```
>>> I.rectangle(200, PLAY-AGAIN-BTN-HEIGHT, "solid", "light-grey")  $\wedge$ 
I.overlay-align("center", "center", I.text("Nochmal spielen", 28, "black"), _)  $\wedge$ 
I.overlay-align("center", "center", I.empty-scene(PLAY-AGAIN-BTN-WIDTH, PLAY-AGAIN-BTN-HEIGHT), _)

Nochmal spielen
```

Abb. 4.4. Pyret: Ausführung des Button-Codes in der REPL

Falls mehrere Buttons in dem Spiel benötigt würden, würde es Sinn ergeben, eine Funktion zu schreiben, die verschiedene Buttons erzeugen kann.

Datendefinitionen

In Listing 4.6 sind die Daten-Definitionen für Tic-Tac-Toe:

```

1 # DATA DEFINITIONS
2 # -----
3 data Player:
4   | x-player
5   | o-player
6 end
7
8 data Field:
9   | field(player :: Option<Player>, index :: Number, pos :: I.Point)
10 end
11
12 data Tic-Tac-Toe:
13   | tic-tac-toe(
14     fields :: List<Field>,
15     player :: Option<Player>,
16     game-over :: Boolean)
17 end

```

Listing 4.6. Daten Definitionen für Tic-Tac-Toe

`Player` definiert die 2 möglichen Spieler für Tic-Tac-Toe.

`Field` stellt ein einzelnes Tic-Tac-Toe-Feld dar, welches von einem Spieler besetzt werden kann oder leer ist. Deshalb wurde der Datentyp `Option` verwendet. Wenn das Feld leer ist, dann hat dieser den Wert `none` und ansonsten `some(<Player>)`. Ein Feld hat außerdem noch einen Index, wobei die 9 Tic-Tac-Toe-Felder von 0 bis 8 durchnummieriert sind und eine Position auf dem Spielfeld.

`Tic-Tac-Toe` ist die Datenstruktur für den ganzen Weltenzustand. es enthält die 9 Felder in einer Liste, den Spieler der gerade dran ist und ob das Spiel vorbei ist oder nicht. Für den Spieler wurde als Typ wieder eine `Option<Player>` verwendet, da dieses Feld auch angibt, wer am Ende gewonnen hat und dann auf `none` gesetzt wird, falls Unentschieden ist.

Initialisieren des Welten-Zustands

In Listing 4.7 wird der initiale Zustand von dem Tic-Tac-Toe-Spiel gesetzt.

```

1 # INITIALIZE GAME STATE
2 # -----
3
4 # Berechne die Positionen für die Tic-Tac-Toe-Felder
5 # die in einem 3x3-Gitter angeordnet sind.
6 FIELD-POSITIONS = for pos from [list:
7   I.point(-1,-1), I.point(0,-1), I.point(1,-1),
8   I.point(-1, 0), I.point(0, 0), I.point(1, 0),
9   I.point(-1, 1), I.point(0, 1), I.point(1, 1)
10 ]):
11
12   I.point(
13     CENTER + (pos.x * FIELD-SIZE),
14     CENTER + (pos.y * FIELD-SIZE)

```

```

15      )
16 end
17
18 # Initialisiere die Felder aus den Positionen.
19 # Weise ihnen ihren Index und ihre Position zu.
20 INIT-FIELDS =
21   map_n(lam(index, pos): field(None, index, pos) end, 0, FIELD-POSITIONS)
22
23 # Initialisiere den initialen Game-State. Der X-Spieler beginnt.
24 INIT-TIC-TAC-TOE = tic-tac-toe(INIT-FIELDS, some(x-player), false)

```

Listing 4.7. Initialisieren des Weltenzustands

Zuerst werden die Feldpositionen für jedes Feld berechnet. Hierfür wird die `for`-Syntax verwendet, welche syntaktischer Zucker ist. Die `for`-Syntax kann verwendet werden, um eine Funktion, die als ersten Parameter eine Funktion erwartet, umzuschreiben.

Bei `INIT-FIELDS` in Zeile 20 wird mit `map-n` die vorher berechneten Positionen mit einem von 0 startendem Index in die Felder transformiert:

```
map_n(lam(index, pos): field(None, index, pos) end, 0, FIELD-POSITIONS)
```

Listing 4.8. map-n mit Lambda

Als erstes Argument wird ein Lambda übergeben, welches die Position, sowie den Index in ein Feld transformiert. Dies kann mit der `for`-Syntax umgeschrieben werden, sodass kein explizites Lambda benötigt wird:

```
INIT-FIELDS = for map_n(index from 0, pos from FIELD-POSITIONS):
    field(None, index, pos)
end
```

Listing 4.9. Umschreibung mit `for`-Syntax

Die Umschreibung verbessert wieder die Lesbarkeit des Codes, da ein Inline-Lambda immer etwas verbos ist.

Am Ende wird dann mit den Feldern der ganze initiale Status des Tic-Tac-Toe-Spiels erstellt (s. Listing 4.7, Z. 24).

Zeichen-Funktionen

Für die `draw`-Funktion werden erst die Tic-Tac-Toe-Felder gezeichnet und in eine Variable `background` gespeichert. Anschließend wird überprüft, ob das Spiel vorbei ist oder nicht. Wenn Das Spiel vorbei ist, dann wird über den Tic-Tac-Toe-Feldern noch der Game-Over-Text, sowie der Button, mit dem man nochmal spielen kann, gezeichnet (s. Abb. 4.5). Ansonsten wird über den Hintergrund lediglich noch der

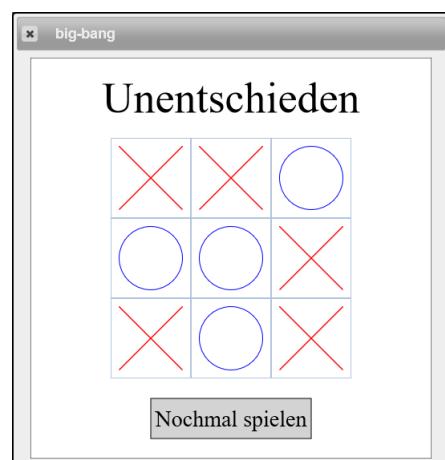


Abb. 4.5. Pyret: Unentschieden in Tic-Tac-Toe

Text gezeichnet, der angibt welcher Spieler an der Reihe ist.

```

1 fun draw(t :: Tic-Tac-Toe) -> I.Image:
2   background = draw-tic-tac-toe-fields(t.fields)
3   if t.game-over:
4     background ^
5       I.place-image(draw-game-over-text(t), CENTER, TOP-TEXT-OFFSET, _)
6       I.place-image(PLAY-AGAIN-BUTTON, PLAY-AGAIN-BUTTON-POS.x,
7                     PLAY-AGAIN-BUTTON-POS.y, _)
8   else:
9     background ^
10    I.place-image(draw-players-turn-text(t), CENTER, TOP-TEXT-OFFSET, _)
11  end
12 end
```

Listing 4.10. Zeichenfunktion für Tic-Tac-Toe

Die Funktion `draw-tic-tac-toe-fields` ist besonders interessant und sieht folgendermaßen aus:

```

1 fun draw-tic-tac-toe-fields(fields :: List<Field>) -> I.Image:
2   for fold(scene from EMPTY-SCENE, f from fields):
3     I.place-image(draw-field(f), f.pos.x, f.pos.y, scene)
4   end
5 end
```

Listing 4.11. Verwendung der `fold`-Funktion, um die Felder über einen Hintergrund hinzuzufügen

Die `fold`-Funktion ist eine Funktion, die in jeglichen funktionalen Programmiersprachen vertreten ist. Sie reduziert eine Liste von Werten, auf ein einzelnes Element. Beispielsweise kann sie verwendet werden, um die Summe einer Liste von Zahlen zu berechnen. Hier wird sie verwendet, um alle Tic-Tac-Toe-Felder auf den Hintergrund (`EMPTY-SCENE`) zu platzieren. Die Feld-Bilder werden über `draw-field(f)` erzeugt und mittels `I.place-image` auf die jeweilige Position platziert.

Update-Funktionen

Bei Tic-Tac-Toe wird dem `on-tick-Handler` nichts zugewiesen, weil ein Welten-Update als Reaktion auf einen Mausklick erfolgt. Somit muss nur der Maus-*Event-Handler* berücksichtigt werden.

Event-Handler

Nur der Maus-*Event-Handler* wird zugewiesen. Hierfür wird folgende Funktion geschrieben:

```

1 fun on-mouse(t :: Tic-Tac-Toe, mx :: Number, my :: Number, event :: String)
2 -> Tic-Tac-Toe:
```

```

3
4  if event == "button-down":
5      if t.game-over:
6          on-click-replay-button(t, mx, my)
7      else:
8          on-click-field(t, mx, my)
9      end
10 else:
11     t
12 end
13 end

```

Listing 4.12. Maus-Eventhandler

Wenn die Maustaste gedrückt wird, wird überprüft, ob das Spiel vorbei ist oder nicht. Wenn das Spiel vorbei ist, dann muss nur überprüft werden, ob der Button, mit dem man neu startet, gedrückt wurden ist. Die Funktion `on-click-replay-button` überprüft dies und gibt den initialen Weltenzustand zurück, wenn der Button geklickt wurden ist. Ansonsten wird einfach der bisherige Weltenzustand zurückgegeben.

Wenn das Spiel nicht vorbei ist, dann muss überprüft werden, ob ein bestimmtes Feld angeklickt wurde und der Weltenzustand dann gegebenenfalls aktualisiert werden. Dies macht die Funktion `on-click-field`, welche so definiert ist:

```

1 fun on-click-field(t :: Tic-Tac-Toe, mx :: Number, my :: Number)
2 -> Tic-Tac-Toe:
3     # Finde ein leeres Feld, auf das mit der Maus geklickt wurde
4     clicked-field = for find(f from t.fields):
5         is-none(f.player) and
6         is-point-inside-rect(mx, my, f.pos.x, f.pos.y, FIELD-SIZE, FIELD-SIZE)
7     end
8
9     cases Option<Field> clicked-field:
10        | some(f) => set-field(t, f)
11        | none => t
12    end
13 end

```

Listing 4.13. Funktion, die auf einen Feld-Klick reagiert

Zuerst wird überprüft, ob es ein leerer Feld gibt, auf das geklickt wurden ist. Hierfür wird die `find`-Funktion verwendet, der eine Funktion übergeben wird, die für ein Feld zurückgibt ob es leer ist und angeklickt wurde. Die Funktion gibt dann eine Option zurück, die `none` ist, falls ein solches Feld nicht gefunden werden konnte. Die Helperfunktion `is-point-inside-rect` prüft, ob ein Punkt innerhalb eines Rechtecks ist. Mit dieser kann überprüft werden, ob die Mausposition innerhalb des Felds ist. Dies macht es etwas aufwendig für den Programmierer. Die `world`-Bibliothek hat nur die Möglichkeit einen Maus-Klick mit seiner Position zu erkennen, aber nicht die Möglichkeit mitzuteilen, auf was geklickt wurden ist. Dies muss der Programmierer selbst implementieren.

Es wird dann Pattern-Matching mit der Syntax `cases` verwendet, um zu gucken, ob etwas in der Option enthalten ist (`some(f)`) oder kein Feld gefunden werden konnte (`none`).

Konnte kein Feld gefunden werden, wird einfach der bisherige Welten-Zustand zurückgegeben. Falls auf ein Feld geklickt wurden ist, wird die Funktion `set-field` aufgerufen, welche den neuen Weltzustand dann zurückgibt. Diese Funktion sieht so aus:

```

1 #|
2   Wird aufgerufen wenn ein Tic-Tac-Toe-Feld gesetzt wird.
3   Erstellt den neuen Tic-Tac-Toe-State mit dem gesetzten Feld.
4   Evaluiert ob der Spieler, der das Feld klickt gewonnen hat oder,
5   ob Unentschieden ist.
6 |
7 fun set-field(t :: Tic-Tac-Toe, f :: Field) -> Tic-Tac-Toe:
8   # Setze den Spieler, der gerade dran ist auf das Feld
9   new-f = field(t.player, f.index, f.pos)
10  # Update die Felder
11  new-fields = L.set(t.fields, f.index, new-f)
12
13  # Überprüfe ob das Spiel vorbei ist und updatet den Spieler.
14  {game-over; next-p} = if check-player-won(new-fields, t.player):
15    # Der Spieler der gewonnen hat, wird zurückgegeben
16    {true; t.player}
17  else if check-full(new-fields):
18    # Unentschieden, deshalb wird 'none' als Spieler zurückgegeben
19    {true; none}
20  else:
21    # Der nächste Spieler ist dran
22    {false; alternate-player(t.player)}
23 end
24
25 tic-tac-toe(new-fields, next-p, game-over)
26 end

```

Listing 4.14. `set-field`-Funktion

Zuerst wird ein neues Feld erstellt, welches den Spieler, der gerade dran ist, enthält. Mittels `L.set` wird das neue Feld an die Index-Position des alten Felds eingefügt. `L.set` gibt dann eine Liste neuer Felder zurück, wobei nur eins unterschiedlich ist als vorher – nämlich das, dass neu eingefügt wurden ist.

Mit den neuen Felder wird nun überprüft, ob der Spieler der gerade auf ein Feld geklickt hat, gewonnen hat. Ansonsten wird überprüft, ob alle Felder voll sind, weil dann Unentschieden ist. Ansonsten wird einfach der Spieler gewechselt. Die einzelnen Zweige des `if`-Ausdrucks geben jeweils ein Tupel mit zwei Werten zurück `{<game-over>; <next-player>}`. Das zurückgegebene Tupel wird in Zeile 14 destrukturiert, sodass die Werte des Tupels direkt in den Variablen `game-over` und `next-p` gespeichert sind.

Am Ende der Funktion wird dann ein neuer Tic-Tac-Toe-Zustand zurückgegeben.

Erstellung der Welt

Am Ende sind dann alle Funktionen geschrieben, sodass die Welt erstellt werden kann:

```
1 # Erstelle das Tic-Tac-Toe-Spiel
```

```

2 W.big-bang(INIT-TIC-TAC-TOE, [list:
3   W.to-draw(draw), # Installiere den Draw-Handler
4   W.on-mouse(on-mouse)]) # Installiere den Mouse-Update-Handler

```

Listing 4.15. Der Big-Bang des Tic-Tac-Toe-Universums

4.1.7. Bewertung

Einfachheit der Implementierung von Computerspielen

Da Pyret das Abspielen von Audio nicht unterstützt, konnte die Anforderung, das bei *Flappy Bird* Sound spielen soll, nicht implementiert werden. Ansonsten konnten die Spiele einwandfrei implementiert werden.

Subjektiv gesehen, hielt sich der Aufwand der Implementierungen der Spiele in Pyret für mich in Grenzen. Ich fand es deutlich weniger aufwendiger als die Implementierung in *ToonTalk*.

Die Implementierungen in Pyret sind jedoch durchaus komplexer als in der Sprache Quorum, was an der geringeren Abstraktionsebene des Game-Frameworks liegt. So werden Elemente keiner Szene hinzugefügt und werden automatisch gezeichnet, sondern müssen explizit über `draw` gezeichnet werden. In der `draw`-Funktion werden über die Funktionen der `Image`-Bibliothek die grafischen Elemente komponiert (auf die richtige Art und Weise zusammengefügt), sodass ein ganzes Bild entsteht, das ausgegeben werden kann. Computerspiele über funktionale reaktive Programmierung zu erstellen, ist nicht weit verbreitet.

Ein weitere Schwierigkeit ist es, dass es keine GUI-Bibliothek gibt und auch keine Möglichkeit auf ein Mausklicks für ein Bild zu horchen. Mouse-Events können über das `on-mouse`-Event nur global registriert werden. Es wird dem Programmierer überlassen Code zu schreiben, um herauszufinden auf welche grafische Komponente geklickt wurde. Hierfür wurde für Tic-Tac-Toe und Flappy-Bird, die Helper-Funktion `is-point-inside-rect` verwendet, die überprüft, ob ein Punkt sich innerhalb eines Rechtecks befindet. Diese Funktion wird dann mit der Maus-Klick-Position als Punkt und der Position, der Weite und Höhe des Bilds aufgerufen.

```

fun is-point-inside-rect(
    px :: Number, py :: Number,
    rx :: Number, ry :: Number,
    rw :: Number, rh :: Number
) -> Boolean:
    w-half = rw / 2
    h-half = rh / 2

    l = rx - w-half
    r = rx + w-half
    t = ry - h-half
    b = ry + h-half

    (px >= l) and (px <= r) and (py >= t) and (py <= b)

```

```
end
```

Listing 4.16. `is-point-inside-rect`

Das Problem mit dieser Funktion ist, dass das Bild entkoppelt wird. Anstatt das Bild zu übergeben, werden die Position des Bilds und dessen Dimensionen übergeben. Die Problematik ist, dass in einem Bild nicht dessen Position gespeichert wird, da die Positionierung der Bilder in der `draw`-Funktion erfolgt.

Um dieses Problem zu lösen, wurde für die Implementierung von Tamagochi ein neuer Datentyp `Positioned-Image` erstellt, um ein Bild mit einer Position zu kapseln. Dieser Datentyp wurde in eine eigene Datei ausgelagert (s. Anhang C.5). Es werden zwei Methoden `contains-point` und `draw` für diesen Datentyp erstellt. `draw` wird verwendet um das Bild über ein anderes Bild an der Position zu zeichen und `contains-point`, um zu überprüfen, ob das Bild einen bestimmten Punkt enthält.

Eine weitere Schwierigkeit hat sich bei der Implementierung von Tamagochi gezeigt (s. Anhang C.6). Bei Tamagochi müssen Werte abhängig von der Zeit durchgängig verändert werden. Die `on-tick`-Handler, der den Zustand updatet, bekommt allerdings nur den Weltzustand und nicht die vergangene Zeit zum letzten Frame als Parameter übergeben. In der Dokumentation steht sehr versteckt, dass die FPS standardmäßig auf 28 gesetzt sind. Die Tamagochi-Werte können dann pro Frame über die Formel $v = v + \Delta v * (1/FPS)$ geändert werden, wobei Δv für die Änderung pro Sekunde steht. Als ich allerdings den Wert 28 für die FPS pro Sekunde verwendet habe, haben sich die Werte deutlich langsamer als erwartet verringert. Ich habe dann nach einigem Ausprobieren den Wert 21 als FPS-Wert gewählt, was zu einem befriedigenden Ergebnis nach der Spezifikation geführt hat.

Neben diesen Schwierigkeiten sind die Implementierungen in Pyret insgesamt für Beginner als komplexer als in bspw. Quorum zu bewerten. Funktionale Programmierung setzt ein bestimmtes Denken voraus, das Beginnern oftmals schwer fällt.

Es gibt keinen grafischen Editor, um grafische Elemente zu platzieren. Dafür gibt es den REPL, mit dem grafische Elemente inspiert werden können.

Das Koordinatensystem ist linkshändig und die Drehrichtung ist gegen den Uhrzeigersinn und verstößt somit gegen die mathematische Konvention.

Der Ankerpunkt der Grafiken ist in der Mitte platziert. Da es keinen grafischen Editor gibt, um Elemente zu platzieren, ist dies eine gute Wahl. Allerdings macht es Code zum Überprüfen von Überlappungen komplizierter.

Pyret bietet von sich aus keine Möglichkeit Sprites zu animieren. Aber genauso, wie für ein klickbares Bild die Datenstruktur `Positioned-Image` erstellt wurde, könnte auch eine Datenstruktur, um Sprites zu animieren, erstellt werden. Es würde die Implementierung jedoch zusätzlich verkomplizieren. Besonders weil es keinen Erweiterungsmechanismus wie Vererbung gibt und ein animiertes Sprite, dann nicht wie ein `Image` behandelt werden kann.

Erlernbarkeit nach Konstruktionismus

Neben Computerspielen und Grafik-Programmierung hat Pyret außerdem eine ausgezeichnete Unterstützung für *Data Science* (vgl. [Pol+18]). So können Daten aus *Google-*

Spreadsheets in Tabellen geladen werden. Besonders bemerkenswert ist die zusätzliche Syntax, die für Tabellen-Datentypen angewandt werden kann. So gibt es SQL-artige Syntax, um Spalten auszuwählen oder Syntax wie `sieve` um Daten zu filtern, `transform` um Tabellen zu transformieren und `order` um Tabellen zu sortieren. Tabellen werden in dem REPL als Tabellen ausgegeben (vgl. [Pyr21d]). Zudem können Diagramme und Plots erstellt werden, wobei *Google Charts* als Backend verwendet wird und in der Standardbibliothek gibt es ein Statistik-Modul (vgl. [Pyr21b]).

Ansonsten bietet Pyret keine weitere Möglichkeiten, um andere Mikrokosmen zu entdecken.

So werden auch keine *Turtle*-Grafiken unterstützt. Da das Programmieren von *Turtle*-Grafiken imperativ ist, sollte es sehr schwer fallen eine Unterstützung zu implementieren.

Was die Einbindung von unterschiedlichen Medien und Technologien betrifft, beschränkt sich Pyret auf Bilder, Tabellen, Farben, Diagramme und Plots.

Durch den REPL, kann die Programmiersprache etwas selbstständig erkundet werden. Jedoch ist das Lesen von Dokumentation und Lehrmaterial oder ein Lehrer umbedingt notwendig, um tiefer in Pyret einsteigen zu können. Die Syntax offenbart sich nicht von selbst.

Die Programmierumgebung selbst als Web-Umgebung ist ziemlich sicher. Der Benutzer kann nichts kaputt machen oder zerstören.

Da Pyret eine graduelle Typisierung verwendet, können Prototypen gut erstellt werden und Beginner müssen sich nicht mit der Komplexität von Typen befassen.

Ein Extra, welches Pyret zum Konstruktionismus bietet, ist, dass die Sprache ein *Theme* hat – nämlich ein Piraten-*Theme*. So ist das Logo der Sprache ein Totenkopf und das *Theme* spiegelt sich in vielen Details der IDE und der Präsentation wieder. Wenn z.B. alle Unit-Tests erfolgreich ausgeführt wurden, so wird z.B. „Looks shipshape, both tests passed, mate!“ ausgegeben. Oder wenn die IDE lädt, dann wird als Ladetext u. a. „Assembling the crew“, „Latching the portholes“ oder „Calibrating the compass“ angezeigt. Die Dateiendung von Pyret-Quellcode ist `.arr`, da Piraten „Arr!“ machen. Zudem wird im Github-Repository darauf verwiesen, dass die Verwendung von *PirateSpeak* ([Man06]) für Kommentare und Bug-Reports empfohlen wird.

Inkrementelle Einführung

Pyret hat eine einfache Ausgabe-Funktion mit dem Namen `print`, jedoch gibt es keine Funktion für eine Eingabe während der Laufzeit. Eingaben geschehen über den REPL. Da Pyret auch lokal auf dem Computer installiert und über die Kommandozeile ausgeführt werden kann, können somit keine Konsolenanwendungen wie z.B. *Text Adventures* geschrieben werden, die aktiv die Benutzereingabe während der Laufzeit benötigen.

```
fun hallo-nutzer(name):
    print("Hallo " + name)
```

```
end
```

Listing 4.17. „Hallo Nutzer“ in Pyret

In dem Buch *Programming and Programming Languages* ([Kri+20]) wird die Programmiersprache Pyret Schritt für Schritt eingeführt. Im Buch sind sehr viele Beispiele enthalten, von welchen viele je ein einziges Konzept in Pyret atomar betrachten. Über die REPL können Beginner die Beispiele schnell ausführen und nachvollziehen.

Intuitive Syntax und Natürlichsprachlichkeit

Die Syntax in Pyret ist an einigen Stellen intuitiv und an anderen Stellen weniger intuitiv. So ist die Syntax für die binären Operatoren intuitiv, bspw. `and` für das *logische Und*, `or` für das *logische Oder* oder der Testoperator `is` für Unit-Tests. Insgesamt gibt es sehr viele verschiedene Syntax und syntaktischen Zucker, mit den sich Code besser ausdrücken lässt.

An anderen Stellen, ist Pyret-Code weniger intuitiv. Im Folgenden wird ein kurzes Beispiel anhand der Tamagochi-Implementierung gegeben:

```

1  clicked-symbol = SYMBOLS.find(
2      lam(sym):
3          sym.img.contains-point(mouse-pos) and
4          not(sym.disabled-debuffs.any(has-pet-debuff(pet, _)))
5      end)
6
7      cases (Option<Tamagochi-Symbol>) clicked-symbol:
8          | some(symbol) =>
9              block:
10                 new-health = clamp(pet.love + symbol.love-change, 0, 100)
11                 new-energy = clamp(pet.energy + symbol.energy-change, 0, 100)
12                 new-status = symbol.status-update(pet.status)
13
14                 tama-pet(new-health, new-energy, new-status, pet.life-time)
15             end
16         | none => pet
17     end
```

Listing 4.18. Pyret: Aktivierung eines Tamagochi-Symbols

In diesem Code-Abschnitt wird überprüft, ob ein Tamagochi-Symbol angeklickt wurde, welches nicht deaktiviert ist. Anschließend wird das Symbol aktiviert. In Zeile 2, ist ein Lambda zu sehen. Lambdas beginnen mit dem `lam`- und enden mit dem `end`-Schlüsselwort. Alternativ gibt es jedoch noch eine Kurzschreibweise über geschweifte Klammern. Außerdem wird in Zeile 4 der `any`-Methode ein Lambda `has-pet-debuff(pet, _)` übergeben, was syntaktischer Zucker für den Lambda-Ausdruck `lam(x): has-pet-debuff(pet, x)` ist. Es gibt also schon drei verschiedene Art und Weisen, wie Lambdas erstellt werden können. Dies ist etwas kritisch zu sehen, da Beginner von der großen Anzahl von *syntaktischen Synonymen* verwirrt werden könnten. Zum einen hilft syntaktischer Zucker wie die `for`-Syntax Code besser auszudrücken, auf der anderen Seite macht zu viel syntaktischer Zucker es für Beginner komplizierter.

Des weiteren ist die Syntax des Pattern-Matching über `cases` (in Zeile 7) schwierig. Da die Typisierung graduell ist und der Typ des Ausdrucks, der *gemacht* werden soll nicht zwingend bekannt ist, muss der Typ in den runden Klammern angegeben werden, damit die statische Überprüfung erfolgen kann. Dies zeigt einen Nachteil der graduellen Typisierung auf, da Syntax dadurch komplizierter werden kann.

Zudem kommt Syntax hinzu, die nur in seltenen Fällen gebraucht wird, wie z.B. die `block`-Syntax in Zeile 9, um auf der rechten Seite des Pattern-Matchings mehrere Zeilen zu schreiben, wobei nur die letzte als Wert zurückgegeben wird.

Ein Problem, welches während der Implementierungen mit der Syntax aufgetreten ist, ist, dass Daten-Konstruktoren vom Stil genauso geschrieben werden wie Variablen und der Konstruktorname nicht mehr als Bezeichner verwendet werden kann.

Die Syntax in Pyret ist nicht internationalisiert.

Pyret's Syntax bietet einige wohlüberlegte Feinheiten wie einen *Trailing return type*. Besonders positiv ist, dass das Minus-Zeichen - innerhalb von Bezeichnern verwenden werden kann und dieses sollte nach *Style-Guide* so oft wie möglich verwendet werden. So gibt es keine Diskussionen über *Camel Case*, *Snake Case* usw. und das Minus-Zeichen ist intuitiver und einfacher zu lesen und zu tippen als die anderen Alternativen.

Feature Uniformity und Orthogonalität

Pyret hat viele syntaktische Synonyme, aber alle sind syntaktischer Zucker. Ein Beispiel ist der Lambda-Ausdruck, der sich im vorherigen Abschnitt angesehen wurde.

Es konnten keine syntaktischen Homonyme identifiziert werden.

Die Sprache bietet eine große Funktionsmenge an, welche jedoch kaum Überschneidungen miteinander haben, isoliert betrachtet werden können und je ihren Zweck und Nutzen haben und nach und nach eingeführt werden können (vgl. Abs. 4.1.7).

Pyret's Syntax ist sehr konsistent. Auf der Homepage wird angegeben, dass ein Designziel der Syntax und Semantik von Pyret es ist, die Ersetzbarkeit von äquivalenten Ausdrücken so weit wie möglich zu berücksichtigen (vgl. [Pyr21a]).

Unterstützung der wichtigsten Programmierkonzepte & Datenstrukturen

Pyret bietet keine Schleifen im imperativen Sinne, sondern aus der Sichtweise der funktionalen Programmierung. So werden Schleifen als Rekursion implementiert. Zudem gibt es die `for`-Syntax als syntaktischen Zucker für die Iteration. Um ein Beispiel zu geben, ist hier eine eigene Implementierung der funktionalen `map`-Funktion:

```

1 fun my-map(transform-func, lst):
2   doc: "transforms elements of a list with a transform-function"
3   cases(List) lst:
4     | empty => empty
5     | link(first, rst) =>
6       link(transform-func(first), my-map(transform-func, rst))
7   end
```

```

8 where:
9   # Aufruf ohne syntaktischen Zucker
10  my-map(lam(x): x * x end, [list: 1, 2, 3]) is [list: 1, 4, 9]
11
12  # Aufruf mit syntaktischen Zucker
13  for my-map(x from [list: 1, 2, 3]):
14    x * x
15  end is [list: 1, 4, 9]
16 end

```

Listing 4.19. Pyret: eigene Implementierung der `map`-Funktion

Für Bedingungen bietet Pyret eine `if .. else if .. else`-Syntax, sowie ein äquivalente Syntax über `ask`. Außerdem gibt es die `when`-Syntax, welcher kein Ausdruck, sondern ein *Statement* ist (also wie ein einzelnes `if` in anderen Sprachen).

Die Elemente der prozeduralen Implementierung werden komplett unterstützt.

Von den Datenstrukturen enthält Pyret weder einen *Stack* noch eine *Queue* in der Standardbibliothek. Die Hashmap wird nur eingeschränkt unterstützt, da es einen `StringDict`-Typen gibt, der jedoch nur Strings als Schlüssel haben kann. *Sets* und *Lists* werden gut unterstützt.

Die Implementierung der Fibonacci-Funktion ist in Pyret sehr einfach:

```

1 fun greater-zero(n :: Number) -> Boolean:
2   n > 0
3 end
4
5 fun fib(n :: Number%{greater-zero}):
6   doc: "returns the n'th term of the fibonacci sequence"
7   if (n == 1) or (n == 2):
8     1
9   else:
10     fib(n - 2) + fib(n - 1)
11   end
12 end

```

Listing 4.20. Pyret: rekursive Fibonacci-Funktion

Der Parameter `n` wird mit dem Typ `Number%{greater-zero}` annotiert. Neben dem Typen `Number`, können hinter dem Prozentzeichen eine Reihe von *Assertions* stehen (in diesem Falle `greater-zero`), welche während der Laufzeit überprüft werden. Dabei sind die *Assertions* als Prädikatfunktionen definiert.

Der Pyret-Compiler ist *self-hosting* (vgl. [Pyr21e]).

Programmierumgebung (IDE)

Die Programmierumgebung von Pyret ist sehr einfach. Sie entstand aus den jahrelangen Erfahrungen des *DrRackets*-Designs (vgl. [Pyr21e]).

Die *Turn Around Time* ist nicht hoch. Nachdem auf dem *Run*-Button geklickt wurde, muss etwa noch eine Sekunde gewartet werden. Wenn das Programm ausgeführt wurde, kann

mit dem Programm in dem REPL interagiert werden. Der REPL ist ein sehr wichtiger Bestandteil der IDE in Pyret. Die Ausgabe von Bildern, Tabellen, Diagrammen, Farben usw. ist grafisch und auch die Ergebnisse der Unit-Tests werden in diesem Interagierungsbereich ausgegeben.

Es handelt sich nicht um *Live Coding*, da bei einer Änderung der Quellcodedatei, das Programm neu ausgeführt werden muss.

Unit-Tests sind in Pyret ein integraler Bestandteil. Es gibt dedizierte Syntax für Unit-Tests. *Debugging* kann also entweder über die Unit-Tests geschehen oder über den REPL. Wenn z.B. ein Fehler über die Unit-Tests gefunden wurden ist, kann über den REPL die Ursache herausgefunden werden. Die Umgebung bietet somit einen hohe Debugging-Unterstützung, die für eine funktionale Programmiersprache genau richtig gewählt ist.

Da es sich um eine webbasierte IDE handelt, gibt es keine Installation und es kann mit dem Programmieren sofort losgelegt werden.

Für die Kollaboration lassen sich Pyret-Programme über eine *Google-Cloud*-Integration einfach veröffentlichen und teilen. Ist ein Programm veröffentlicht, kann es über das `import`-Statement und einem Link zum veröffentlichtem Programm importiert werden (s. Anhang C.6).

Es gibt keine Quellcodeverwaltungs-Integration. Das Arbeiten an größeren Programmen im Team muss selbst organisiert werden.

Die IDE unterstützt Syntax-Highlighting und Code-Formatierung, jedoch keine Code-Completion, -Navigation und -Refactoring. Die Code-Formatierung ist sehr gut umgesetzt, da der Code beim Eintippen automatisch eingerückt wird.

Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplin (*Clean Code*)

Pyret-Code ist in Modulen organisiert und es können selbst Module erstellt werden, welche über `import` oder `include` importiert werden können. Über die `provide`-Syntax kann innerhalb eines Moduls angegeben werden, welche Symbole und Typen exportiert werden sollen (s. Anhang C.5) und somit wird ein *Information Hiding*-Mechanismus implementiert.

Pyret unterstützt Dokumentationskommentare über die `doc`-Syntax. Es gibt aber kein Werkzeug, mit dem daraus eine Dokumentation generiert werden kann und die IDE nutzt diese Kommentare nicht, um sie innerhalb der IDE anzuzeigen.

Pyret unterstützt statische Typisierung über die graduelle Typisierung. Jedoch weist der Typprüfer Lücken auf – so werden z.B. parametrisierte Typen (*Generics*) nicht Typgeprüft (vgl. [Pyr21c]).

Pyret unterstützt *Assertions*, um Argumente, die einer Funktion übergeben werden, zu überprüfen.

In Pyret gibt es keine *Type-Casts* und keine *Type Coercions*. Umwandlungen in andere

Typen geschieht immer explizit über eine Funktion wie z.B. `string-to-number`.

In Pyret können Fehler verschiedener Typen über `raises` geworfen werden. Es gibt aber kein Konstrukt, um geworfene Fehler aufzufangen.

Dafür gibt es jedoch einige Syntax, um in Unit-Tests zu überprüfen, ob ein Fehler geworfen wurde und diesen überprüfen kann:

- `expr raises exn-string`: Überprüft, ob der Ausdruck einen bestimmten *Exception-String* wirft.
- `expr does-not-raise`: Überprüft, ob der Ausdruck keine Ausnahme wirft.
- `expr raises-other-than exn-string`: Überprüft, ob eine Ausnahme einen String anders als der angegebene String wirft.
- `expr raises-satisfies pred`: Überprüft, ob der Ausdruck eine Ausnahme wirft, welche ein bestimmtes Prädikat erfüllt.
- `expr raises-violates pred`: Überprüft, ob der Ausdruck eine Ausnahme wirft, welche ein bestimmtes Prädikat verletzt.

Pyret bietet ausgezeichnete Unterstützung für Unit-Tests. Für Unit-Tests gibt es dedizierte Syntax. So gibt es den `check`-Block, indem Tests definiert werden können. Bei Funktionsdefinitionen kann am Ende ein `where`-Block stehen, in dem Tests für die Funktion definiert werden. Zudem gibt es zahlreiche Test-Operatoren, welche in der Syntax von Pyret eingebaut sind, z.B. 5 `satisfies is-odd`, um zu testen, dass die 5 ungerade ist.

Die Tests werden automatisch mit dem Ausführen des Codes ausgeführt und die Ergebnisse im Interaktionsbereich übersichtlich angezeigt.

Pyret hat einen ausführlichen *Style-Guide*, der genau angibt wie Pyret-Code geschrieben werden soll und was es zu beachten gibt (s. [Pyr21c]).

Als funktionale Sprache ist der Standardmodus, dass es keine Nebeneffekte gibt. Es können jedoch auch veränderbare Variablen über das `var`-Schlüsselwort erstellt und mit dem Zuweisungsoperator `:=` neu zugewiesen werden.

Gleichermaßen ist *Variable Shadowing* standardmäßig nicht möglich, kann jedoch durch das explizite Schlüsselwort `shadow` aktiviert werden.

Durch diese Entscheidungen wird in Pyret automatisch defensiv programmiert und bestimmte *Features*, die die Code-Qualität gefährden könnten, können mit zusätzlicher Syntax verwendet werden. Pyret sieht also ein, dass *Features* wie *Variable Shadowing* und *Mutation* durchaus ihren praktischen Nutzen haben und ist in der Hinsicht sehr pragmatisch und erlaubt diese.

Operatoren müssen durch Leerzeichen von dem Ausdruck abgetrennt sein, sodass lesbarer Code entsteht. Zudem gibt es keine Operator-Rangfolge und verschiedene Operatoren müssen mit Klammern abgetrennt werden. Die Entscheidung wird von den Sprachdesignern folgendermaßen begründet:

„Pyret takes the firm stance that since every operator has its own quirks, it does not make sense to create a complex, hard-to-predict set of rules for how different operators interact. Instead, it uses just one single rule, with the easy use of parentheses to resolve any unintended behaviors.“ [Pyr21b]

Operator-Überladungen sind erlaubt. Ein Ausdruck wie `a + b` ist syntaktischer Zucker für `a._add(b)` und ebenso sind die anderen Operatoren definiert.

Es gibt keinen unären Minus-Operator. Um eine Zahl zu negieren, musste in den Implementierungen der Ausdruck `0 - n` verwendet werden, was etwas „unbequem“ ist.

Fehlermeldungen

Da die ganzen Erfahrungen von *DrRacket* und den Forschungen daraus über Fehlermeldungen (vgl. [MFK11]), welche bei der Aufstellung der Bewertungskriterien im Abschnitt 3.4.9 sich genauer angesehen wurden, in die Implementierung von Pyret integriert wurden sind, hat Pyret eine ausgezeichnete Unterstützung für pädagogische Fehlermeldungen. Dennoch werden manchmal Fehlermeldungen ausgegeben, die etwas verwirrend sind, z.B.

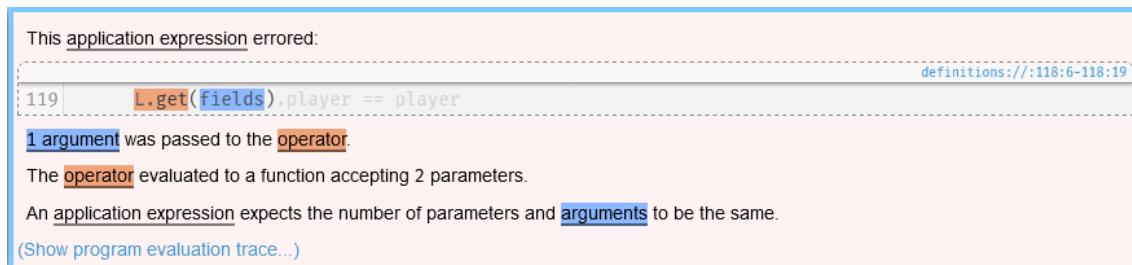


Abb. 4.6. Pyret: verwirrende Fehlermeldung

Hier sollte über die `L.get`-Funktion ein bestimmtes Element aus einer Liste über einen Index gelesen werden, jedoch wurde der Index als Argument vergessen. Es ist verwirrend, dass auf die Funktion `L.get` mit „operator“ verwiesen wird und dieser Operator erst in eine Funktion mit zwei Parametern umgewandelt wird („The operator evaluated to a function accepting 2 parameters.“). Es gibt keinen Operator für die `L.get`-Funktion. Es ist zwar gut, dass die Fehlermeldungen transparent sind und erklärt wird, was „hinter der Szene“ passiert, jedoch nützt dies nichts, wenn die Erklärung gar nicht dazu passt.

Hier ist eine anderes Beispiel für eine schlechte Fehlermeldung:



Abb. 4.7. Pyret: schlechte Fehlermeldung

Es handelt sich um einen Fehler, der innerhalb eines Lambdas geworfen wurde. Hier wird jedoch kein Kontext gegeben, wo der Fehler aufgetreten ist.

Ansonsten sind die Fehlermeldungen in Pyret jedoch sehr gut aufbereitet, wie folgendes positives Beispiel zeigt:

```

The Option annotation
definitions://:35:20-35:40
36 | field(symbol :: Option<Field-Symbol>, index :: Number, pos :: I.Point)
was not satisfied by the value
x-player
which was sent from around
definitions://:111:19-111:55
112 | some(f) => set-field(fields, f.index, x-player)
(Show program evaluation trace...)

```

Abb. 4.8. Pyret: gute Fehlermeldung

Das programmiertechnische Vokabular ist sehr präzise und weniger vereinfacht, dafür sind diese Begrifflichkeiten und der dazugehörige Code in der gleichen Farbe hinterlegt, sodass Anfänger genau wissen, was gemeint ist. Der fehlerhafte Code wird zudem auch im Editor farblich markiert.

Die Sprache der Fehlermeldungen ist weder positiv noch negativ, sondern neutral. Es werden viele Informationen und zudem Erklärungen in den Fehlermeldungen gegeben und der *Stack-* bzw. *Evaluation-Trace*, kann bei Bedarf angezeigt werden. Außerdem werden keine Lösungen vorgeschlagen, wie Fehlermeldungen zu berichtigen sind.

Ein hohes Abstraktionsniveau für Datentypen

Die Zeichenfolgen in Pyret, sowie der Editor und die Ausgabe unterstützen Unicode. Aber es gibt einen Bug, bei dem das Unicode-Zeichen U+2284A (ein chinesisches Zeichen) nicht unterstützt wird und ein Fehler ausgegeben wird (vgl. [jsw20]).

Pyret's Zahlen verwenden Langzahlarithmetik, wobei es eine Optimierung gibt, bei der Integer, die kleiner als $2^{53} - 1$ sind, Javascript's Standardzahlentyp verwenden (`double`). Zusätzlich gibt es neben diesen `Exactnums` auch `Roughnums`, welche Javascript's `double` verwenden. Das Literal für `Roughnums` ist eine Zahl wo eine Tilde davorgeschriften ist, z.B. `~3.14159` (vgl. [Pyr21b]).

Der Standardmodus ist somit Langzahlarithmetik und bei Bedarf können unpräzise Zahlen verwendet werden.

Anwendungsbereiche und Interoperabilität mit Technologien und anderen Programmiersprachen

Anwendungsbereiche:

- 2D-Grafik
- Simulationen, interaktive Programme und Computerspiele
- Data Science
 - Tabellen (können von *Google-Sheets* importiert werden)
 - Statistiken
 - Plots und Diagramme (*Google-Charts* als Backend)

Ansonsten gibt es keine Interoperabilität mit anderen Technologien oder anderen Programmiersprachen.

Dokumentation & Community Support

Die Dokumentation ist hervorragend (s. [Pyr21b]). Um zu starten gibt es dort ein *Getting-Started-Guide*, der die wichtigsten Konzepte von Pyret kurz und präzise einführt. Daran angeschlossen folgt ein motivierendes Tutorial *A Flight Lander Game*, bei dem ein Spiel implementiert wird, bei dem ein Flugzeug abstürzt und man es auf dem Land und nicht im Wasser landen muss.

Getting Started und *A Flight Lander Game* sind dabei eher auf fortgeschrittene Programmierer zugeschnitten, da die Konzepte nur sehr knapp erklärt werden.

Es gibt jedoch das Lehrbuch *Programming and programming languages* ([Kri+20]), das auf Programmieranfänger zugeschnitten ist und die Konzepte aufeinander aufbauend erklärt. Dabei ist das Buch Online kostenlos als Web-Buch verfügbar.

Die Community scheint eher klein zu sein. Es gibt eine Mailing-Liste für Neuigkeiten und eine *Google*-Gruppe für Unterstützung und Diskussionen. Außerdem können Bugs und Probleme im *Github*-Repository mitgeteilt werden.

4.2. Quorum

4.2.1. Kurze Einführung in Quorum

Quorum bezeichnet sich selbst als evidenzbasierte Programmiersprache, welche evidenzbasiert menschliche Faktoren in ihrem Design berücksichtigt (vgl. [Quo]).

Für das Design von Quorum wurden einige Studien herangezogen, welche die Design-Entscheidungen empirisch belegen. Es wurden Studien aus folgenden Bereichen herangezogen:

- Statische vs. Dynamische Typisierung
- Software Transactional Memory (STM)
- Syntax (s. Studie in Abs. 3.4.4)
- Vererbung
- Blockbasiert vs. Textbasiert
- Compiler Error Design

Es wird nicht näher auf die Studien eingegangen. Die Studie zur Syntax wurde allerdings bei dem Bewertungsbereich **Intuitive Syntax und Natürlichsprachlichkeit** genau beschrieben.

Eine weitere Philosophie der Programmiersprache ist, dass sie inklusiv ist und Individuen mit Behinderungen berücksichtigt werden. So hat die Syntax der Programmiersprache keine arkanischen Zeichen, wie Semikolon oder geschweifte Klammern.

Zudem ist Quorum eine pure objektorientierte Programmiersprache.

4.2.2. Beschreibung der Programmierumgebung

Es gibt eine eigens entwickelte IDE für Quorum mit dem Namen *Quorum Studio*. Diese ist jedoch nur für die Betriebssysteme *Windows* und *Mac OS X* verfügbar. *Quorum*

Studio besitzt alle Funktionen, die von einer modernen IDE erwartet werden. Im Besonderen wurde jedoch auf die Zugänglichkeit (*Accessibility*) geachtet. So gibt es beispielsweise Funktionen wie *Smart Zoom*, *Smart Navigation* und eine anpassbare Screen-Reader und Braillezeilen-Unterstützung, sodass Blinde die IDE verwenden können.

Bei der Verwendung von *Quorum Studio*, musste ich für mich leider feststellen, dass die IDE hinderlich ist, da folgende Probleme bei mir auf einem recht schnellen Computer mit dem Betriebssystem *Windows 10* aufgetreten sind:

- Die IDE ist öfters abgestürzt.
- Compiler-Fehler im Editor, welche rot unterkragt sind wurden manchmal nur teilweise angezeigt oder sind komplett ausgefallen.
- Schlechte Performanz: spürbare Verzögerung bei Eingabe von Text bis zur Ausgabe auf den Bildschirm
- *Intellisense* sehr langsam und an dieser Stelle (während des Wartens auf *Intellisense*) ist *Quorum Studio* auch am häufigsten abgestürzt.

Ich möchte hier betonen, dass diese Fehler ausschließlich bei mir aufgetreten sind und ich den Sachverhalt nicht weiter untersucht habe.

Grundlegend ist zu sagen, dass Performanz und Bug-Freiheit für die *Usability* immer kritisch sind und diese nichtfunktionalen Anforderungen zusammen mit den funktionalen Anforderungen unbedingt berücksichtigt werden sollten.

Der Quorum-Compiler kann jedoch auch ohne Quorum Studio heruntergeladen werden. Hier wird der Quellcode über folgenden Befehl in der Kommandozeile kompiliert:

```
quorum Main.quorum *.quorum
```

Hierfür muss der Installationsordner von Quorum zuerst zur Umgebungsvariable *Path* hinzugefügt werden. Dabei handelt es sich bei *quorum* unter Windows um eine Batch-Datei, die folgenden Befehl enthält:

```
java -jar "%~dp0Quorum.jar" -library "%~dp0Library" %*
```

Wenn das obere *quorum Main.quorum *.quorum* ausgeführt wird, so wird Folgendes ausgeführt:

```
java -jar "C:\Program Files\Quorum\Quorum.jar"
-library "C:\Program Files\Quorum\Library" Main.quorum *.quorum
```

Weil die Implementierung der Spiele bei mir in Quorum immer aus mehreren Dateien bestanden, da es wie in Java die Einschränkung gibt, dass eine Datei nur eine Klasse enthalten darf, habe ich bei der Eingabe in der Kommandozeile nach der Hauptklasse *Main.quorum*, die anderen Dateien mittels Wildcards über **.quorum* aufgeführt.

Unglücklicherweise klappt dies nicht bei dem Tic-Tac-Toe-Spiel, weil aus mir noch unbekannten Gründen folgender Fehler auftritt:

```
I cannot execute this statement, as we have already returned from this action.
```

Kompiliert man Tic-Tac-Toe hingegen, indem alle Dateien einzeln aufgelistet werden mit

```
quorum Main.quorum Field.quorum
```

... gibt es keine Probleme.

4.2.3. Interaktive Anwendungen in Quorum

Eine interaktive Anwendung in Quorum zu erstellen ist recht simpel. Hierfür erbt die Hauptklasse `Main` einfach nur von der Klasse `Game` und die Aktionen (Methoden) `CreateGame`, sowie `Update(number seconds)` werden überschrieben.

```

1  use Libraries.Game.Game
2
3  class Main is Game
4      action Main
5          StartGame()
6      end
7
8      action CreateGame
9          // Initialisiere hier das Spiel
10     end
11
12     action Update(number seconds)
13         // Ändere hier die Spiel-Objekte
14     end
15 end

```

Listing 4.21. leere Spiele-Klasse

Das Programm startet mit der `Main`-Aktion, welche die Aktion `StartGame` der `Game`-Klasse aufruft, welche ein Fenster erstellt und den *Game Loop* startet. Vor dem Aufruf von `StartGame`, können außerdem Spiel-Einstellungen, wie z.B. die Fenstergröße oder der Fenstername, gesetzt werden.

Die Aktion `CreateGame` wird am Ende der `StartGame`-Aktion ausgeführt. Hier können Spiele-Resourcen wie Grafiken und Töne geladen werden und grafische Objekte dem Spiel mit der `Add`-Funktion hinzugefügt werden.

Die `Update`-Funktion wird jeden Frame aufgerufen und hier können Objekte animiert oder eine bestimmte Spiele-Logik implementiert werden.

4.2.4. Bouncing Ball

Über folgende 67 Zeilen Code wird *Bouncing Ball* in Quorum implementiert:

```

1  use Libraries.Game.Game
2  use Libraries.Game.Graphics.Drawable
3  use Libraries.Game.Graphics.Color
4  use Libraries.Compute.Vector2
5  use Libraries.Sound.Audio
6
7  class Main is Game
8      constant integer WIDTH = 500
9      constant integer HEIGHT = 300
10     constant integer BALL_RADIUS = 10
11     constant integer BALL_DIAMETER = BALL_RADIUS * 2
12
13     integer xSpeed = 100
14     integer ySpeed = 100

```

```

15      Audio bleepAudio
16      Drawable ball
17
18      action Main
19          SetScreenSize(WIDTH, HEIGHT)
20          StartGame()
21      end
22
23      action CreateGame
24          // Erstelle einen roten Kreis und platziere in in der Mitte des Fensters
25          Color color
26          ball:LoadFilledCircle(BALL_RADIUS, color:Red())
27          ball:SetPosition(WIDTH / 2, HEIGHT / 2)
28          // Füge den Ball zum Spiel hinzu
29          Add(ball)
30
31          // Lade den Audio-Clip
32          bleepAudio:Load("bleep.wav")
33      end
34
35      action Update(number seconds)
36          x = ball:GetX()
37          y = ball:GetY()
38
39          // Pralle vom linken Rand ab
40          if x < 0
41              ball:SetX(0)
42              xSpeed = xSpeed * -1
43              bleepAudio:Play()
44          // Pralle vom rechten Rand ab
45          elseif (x + BALL_DIAMETER) > WIDTH
46              ball:SetX(WIDTH - BALL_DIAMETER)
47              xSpeed = xSpeed * -1
48              bleepAudio:Play()
49          // Pralle vom unteren Rand ab
50          elseif y < 0
51              ball:SetY(0)
52              ySpeed = ySpeed * -1
53              bleepAudio:Play()
54          // Pralle vom oberen Rand ab
55          elseif (y + BALL_DIAMETER) > HEIGHT
56              ball:SetY(HEIGHT - BALL_DIAMETER)
57              ySpeed = ySpeed * -1
58              bleepAudio:Play()
59      end
60
61          // Skaliere die Geschwindigkeit mit den vergangen Sekunden
62          // und bewege den Ball
63          Vector2 velocity
64          velocity:Set(xSpeed, ySpeed)
65          velocity:Scale(seconds)
66          ball:Move(velocity)
67      end

```

```
68 end
```

Listing 4.22. Bouncing Ball in Quorum

Es folgt hier noch eine kurze Erklärung des Codes mit Zeilenangaben:

1. Importierung von Klasse (Z. 1 bis 5)
2. Erstellung von Konstanten (Z. 8)
3. Erstellung von Instanzvariablen (Z. 13 bis 16)
4. Haupt-Aktion `Main` (Z. 18 bis 21)
5. `CreateGame`
 - a) Initialisieren des Balls (Z. 24 bis 28)
 - b) Hinzufügen des Balls zum Spiel (Z. 29)
 - c) Laden des Audio-Clips (Z. 32)
6. `Update`
 - a) Holen der Ball-Koordinaten (Z. 38 bis 39)
 - b) Pralle von den Rändern ab (Z. 39 bis 59)
 - c) Bewege den Ball (Z. 61 bis 66)

4.2.5. Flappy Bird

`CreateGame`

Die `CreateGame`-Aktion von Flappy Bird sieht folgendermaßen aus:

```
1 action CreateGame
2     // set Icon
3     File iconFile
4     iconFile:SetPath(
5         "../../../../assets/flappy-bird/sprites/bluebird-midflap.png")
6     SetApplicationIcon(iconFile)
7
8     // Enable Gravity
9     EnablePhysics2D(true)
10    SetGravity2D(0, -GRAVITY)
11
12    LoadAudio()
13
14    CreateBackground()
15    CreatePipes()
16    CreateBird()
17    CreateScoreLabel()
18    CreateGameOverLabel()
19    CreatePlayAgainButton()
20
21    // listen to keyboard input
22    AddKeyboardListener(me)
23
24    // check bird and pipe collision
25    AddCollisionListener(me)
```

```
26 end
```

Listing 4.23. Flappy Bird in Quorum: CreateGame

In Zeile 2 bis 6 wird das Anwendungs-Icon gesetzt. Anschließend wird in Zeile 8 bis 10 die 2D-Physik der Game-Engine aktiviert. In Zeile 12 werden über `LoadAudio` die Audio-Clips geladen. Die Aktionen `CreateBackground` bis `CreateBird` von Zeile 14 bis 16 laden Sprites und positioniert sie auf den Bildschirm. Die Aktionen `CreateScoreLabel` bis `CreatePlayAgainButton` von Zeile 17 bis 19 erstellen GUI-Elemente, wobei der Button zum erneuten Spielen, sowie das Game-Over-Label am Anfang über den Aktionsaufruf `element:Hide()` unsichtbar gemacht wird.

Durch die Aktion `AddKeyboardListener(me)`, wird die `Main`-Klasse über Tastatureingaben informiert. Ebenso wird durch die Aktion `AddCollisionListener(me)`, die `Main`-Klasse über 2D-Physik-Kollisionen informiert. Um das Hinzufügen von *Listeners* zu der `Main`-Klasse besser zu verstehen, sehen wir uns an, von welchen Klassen die `Main`-Klasse erbt:

```
class Main is Game, KeyboardListener, CollisionListener2D, Behavior
```

Die `Main`-Klasse erbt von den Klassen:

- **Game:** Die Klasse von der geerbt werden muss, um in Quorum ein Spiel zu erstellen.
- **KeyboardListener:** Die `Main`-Klasse muss ein `KeyboardListener` sein, damit wir über `AddKeyboardListener(me)` angeben können, dass sie auf Tastatureingaben horchen soll. Hierfür überschreibt die `Main`-Klasse, die Aktion `PressedKey(KeyboardEvent event)`, auf die ich später noch eingehe.
- **CollisionListener2D:** Hier ist es ebenso, dass wir angeben, dass die `Main`-Klasse ein `CollisionListener2D` ist, sodass wir 2D-Physik-Kollisionen abhorchen können, indem wir die Aktion `BeginCollision(CollisionEvent2D event)` überschreiben.
- **Behavior:** Die `Main`-Klasse, ist ebenso ein Behavior und überschreibt die `Run`-Aktion. Der Button zum erneuten Spielen bekommt nämlich in der Aktion `CreatePlayAgainButton` als Behavior mit `playAgainButton:SetBehavior(me)` die `Main`-Klasse übergeben. Wenn der Button dann angeklickt wird, wird die `Run`-Aktion der `Main`-Klasse ausgeführt, die das Spiel neu startet.

Sprite Animationen

Für Flappy Bird habe ich bei Quorum entschieden, den Vogel zu animieren. Hierfür müssen mehrere Sprites geladen werden, die in festen Intervallen wechseln.

Dafür habe ich dann eine neue Klasse `SpriteAnimation` erstellt, die genau das tut, weil es in Quorum noch keine solche Klasse gab. Dabei erweitert die Klasse, die `Drawable`-Klasse, sodass ein Objekt dieser Klasse mit der `Add`-Aktion ganz einfach einem Spiel hinzugefügt werden kann.

```
1 use Libraries.Containers.Array
2 use Libraries.Game.Graphics.Drawable
3
4 class SpriteAnimation is Drawable
5   private Array<Drawable> sprites
6   private number fps = 4
7
```

```

8   private number elapsedTime = 0
9   private integer spriteIndex = 0
10
11  action SetFPS(number fps)
12    me:fps = fps
13  end
14
15  action GetFPS returns number
16    return fps
17  end
18
19  action GetSprites returns Array<Drawable>
20    return sprites
21  end
22
23  action Create(Array<Drawable> sprites, number fps)
24    me:sprites = sprites
25    me:fps = fps
26    if sprites:GetSize() = 0
27      alert("There must be at least one sprite!")
28    end
29    Load(sprites:Get(spriteIndex))
30  end
31
32  action Update(number seconds)
33    elapsedTime = elapsedTime + seconds
34    if elapsedTime >= 1/fps
35      spriteIndex = (spriteIndex + 1) mod sprites:GetSize()
36      Load(sprites:Get(spriteIndex))
37      elapsedTime = elapsedTime - 1/fps
38    end
39  end
40 end

```

Listing 4.24. Klasse SpriteAnimation

Der Klasse wird über die `Create`-Aktion ein Array von Sprites übergeben, sowie die FPS, die angeben wie schnell zwischen den einzelnen Sprites gewechselt wird.

Außerdem hat die Klasse zwei private Felder mit dem Namen `elapsedTime` und `frameIndex`. `elapsedTime` ist ein Zähler für die vergangene Zeit, welche kontinuierlich in der `Update`-Aktion erhöht wird. `spriteIndex` gibt den Index des Sprites in dem Array an, welches momentan angezeigt wird. Wenn in der `Update`-Aktion die vergangene Zeit größer gleich die Zeit pro Sprite ist, dann wird der `spriteIndex` modular um eins hochgezählt, das neue Sprite an dem `spriteIndex` geladen, und die vergangene Zeit zurückgesetzt.

In der `CreateBird`-Aktion der `Main`-Klasse wird der Vogel, welcher eine Instanz der `SpriteAnimation`-Klasse ist, dann erstellt.

```

1  action CreateBird
2    Drawable midFlap
3    midFlap:Load("../..../assets/flappy-bird/sprites/bluebird-midflap.png")

```

```

4     Drawable upFlap
5     upFlap:Load("../assets/flappy-bird/sprites/bluebird-upflap.png")
6     Drawable downFlap
7     downFlap:Load("../assets/flappy-bird/sprites/bluebird-upflap.png")
8
9     Array<Drawable> sprites
10    sprites:Add(midFlap)
11    sprites:Add(upFlap)
12    sprites:Add(midFlap)
13    sprites:Add(downFlap)
14
15    bird:Create(sprites, 3)
16    bird:SetPosition(50, HEIGHT / 2)
17
18    // the bird reacts to gravity
19    bird:EnablePhysics(true)
20    bird:SetResponsive()
21    FLY_VELOCITY:Set(0, 200)
22    FALL_VELOCITY:Set(20, -400)
23    bird:SetAngularVelocity(-1.5)
24
25    Add(bird)
26 end

```

Listing 4.25. Erstellung des Vogels in Quorum

Hierfür werden zuerst die einzelnen Sprites `midFlap`, `upFlap` und `downFlap` geladen und dann ein Sprite-Array erstellt, in dem die Sprites in folgender Reihenfolge hinzugefügt werden: `midFlap`, `upFlap`, `midFlap`, `downFlap` (s. 4.9).

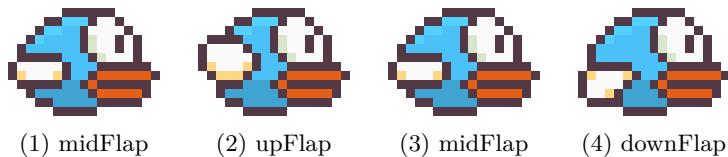


Abb. 4.9. Sprite-Animation des Vogels

In Zeile 15 wird der animierte Vogel dann mit der `Create`-Aktion erstellt. Anschließend wird die Position des Vogels gesetzt. Mit `bird:EnablePhysics(true)` wird die 2D-Physik des Vogels aktiviert, sodass er von der Physics-Engine berücksichtigt wird. Mit `bird:SetResponsive()` wird der Physics-Engine mitgeteilt, dass die simulierten Kräften wie die Gravitation oder Kollisionskräfte auf den Vogel wirken. Die andere Option wäre `bird:SetNonResponsive()`, bei der ein Objekt von Kräften nicht beeinflusst werden würde. Die Physik müsste für ein solches Objekt dennoch mit `EnablePhysics()` eingeschaltet werden, damit Kollisionen erkannt werden.

In Zeile 23 wird dem Vogel eine leichte Rotation im Uhrzeigersinn gegeben, sodass dieser sich durchgehend nach rechts dreht und sich so dann nach unten neigt.

Update-Aktion

```

1  action Update(time)
2      if not gameOver
3          // Animations Update
4          bird:Update(time)
5
6          // Der Vogel kann nicht aus dem oberen Rand rausfliegen
7          if bird:GetY() + bird:GetHeight() > HEIGHT
8              bird:SetY(HEIGHT - bird:GetHeight())
9              // Das Spiel ist vorbei,
10             // wenn der Vogel unten aus dem Bildschirm fällt
11             elseif bird:GetY() < -bird:GetHeight()
12                 GameOver()
13
14
15         bird:SetRotation(math:MinimumOf(bird:GetRotation(), 80))
16
17         // Die Röhren werden nach rechts neu versetzt und
18         // der Spieler bekommt dann einen Punkt
19         if topPipe:GetX() < -topPipe:GetWidth() and bird:IsResponsive()
20             pointAudio:Play()
21             score = score + 1
22             scoreLabel:SetText("Score: " + score:GetText())
23             ResetPipes()
24
25     end
26 end

```

Listing 4.26. Update-Aktion für Flappy-Bird

In Zeile 4 wird die Sprite-Animation aktualisiert. Schließlich wird überprüft, ob der Vogel über dem oberen Rand des Bildschirms hinaus ist. Wenn dies der Fall ist, wird er wieder zurück an den oberen Rand gesetzt. Anschließend wird geprüft, ob der Vogel unten aus dem Bildschirm gefallen ist und dann gegebenenfalls die Aktion `GameOver` aufgerufen. In Zeile 15 wird sicher gestellt, dass der Vogel höchstens 80° nach unten zeigt. In Zeile 19 wird überprüft, ob die zwei Röhren, die von oben und unten aus der Zeichenoberfläche ragen, sich aus der linken Seite der Zeichenoberfläche rausbewegen. Wenn ja, dann wird ein AudioClip gespielt, die Punktzahl um eins erhöht, da das Hindernis ja nun überwunden ist und die Röhren in `ResetPipes()` wieder in zufälliger Höhe nach rechts gesetzt.

Negativ zu bemerken an dieser Stelle ist, dass es in Quorum kein `return` ohne Rückgabewert gibt. So ist alles in der `Update`-Aktion in der Bedingung `if not gameOver` (s. Z. 2) und es kann nicht folgendermaßen kodiert werden:

```

action Update(time)
    if gameOver
        return
    end

    // ...

```

Außerdem gibt es in Quorum auch nicht die praktischen Anweisungen, die aus anderen

Programmiersprachen wie Java bekannt sind: `break`, um aus einer Schleife auszubrechen und `continue`, um den Rest der Schleifen-Iteration zu überspringen.

Auf Tastatureingaben reagieren

Da `Main` von der Klasse `KeyboardListener` erbt und sich selbst mit `AddKeyboardListener(me)` als *Listener* hinzufügt, kann in der Aktion `PressedKey` überprüft werden, ob die Leertaste gedrückt wird.

```

1 action PressedKey(KeyboardEvent event)
2     if bird:IsResponsive() and event:keyCode = event:SPACE
3         wingAudio:Play()
4         bird:SetLinearVelocity(FLY_VELOCITY)
5         bird:SetRotation(-70)
6     end
7 end
```

Listing 4.27. Pressed-Key für Flappy-Bird

Zudem wird überprüft, ob der Vogel *responsive* ist, also noch mit keiner Röhre kollidiert ist und nicht runterfällt. Wenn der Vogel nach oben fliegen soll, wird ein kurzer Audio-Clip gespielt, die Geschwindigkeit des Vogels so gesetzt, dass er entgegen der Schwerkraft nach oben beschleunigt wird und die Rotation des Vogels so gesetzt, dass er nach oben geneigt ist.

Schließlich gibt es noch den Kollisions-Handler, der immer aufgerufen wird, wenn eine Kollision auftritt.

```

1 action BeginCollision(CollisionEvent2D event)
2     if bird:IsResponsive()
3         hitAudio:Play()
4         bird:SetNonResponsive()
5         bird:SetLinearVelocity(FALL_VELOCITY)
6         bird:SetRotation(80)
7     end
8 end
```

Listing 4.28. Quorum: Flappy-Bird - Kollisions-Handler

Da es nur zwei Objekte gibt, die in dem Spiel miteinander kollidieren können (den Vogel und die Röhren), muss nicht geprüft werden, welche Objekte kollidiert sind. Wenn der Vogel dann noch nicht kollidiert ist (`bird:IsResponsive()`), wird eine Audio-Clip für die Kollision gespielt, der Vogel wird auf *nonresponsive* gesetzt, sodass die Schwerkraft nicht mehr für ihn gilt. Dann wird seine Geschwindigkeit nach unten und seine Rotation so gesetzt, dass er sich nach unten neigt. Wenn der Vogel nun mit einer Röhre kollidiert, fällt er im schnellen Tempo nach unten.

4.2.6. Bewertung

Einfachheit der Implementierung von Computerspielen

In Quorum konnten alle Spiele mit allen Anforderungen erfüllt werden.

Der Aufwand der Implementierungen war subjektiv gesehen relativ gering. Der Aufwand war geringer als bei der Programmiersprache Pyret.

Bei Tic-Tac-Toe war das Behandeln eines Klicks eines Feldes kompliziert. In der Aktion `CreateTicTacToe` der `Main`-Klasse werden 9 Felder erstellt und für jedes Feld mit `field:AddMouseListener(me)` die `Main`-Klasse als `MouseListener` registriert (s. Anhang C.9, Z. 93).

Der Event-Handler der `Main`-Klasse kann dann `GetSource` auf dem Event aufrufen, um das Feld, das geklickt wurden ist, zu bekommen und dieses in die `Feld`-Klasse casten:

```
action ClickedMouse(MouseEvent event)
    if not gameOver and event:IsClicked() and event:IsLeftButtonEvent()
        Item source = event:GetSource()
        Field field = cast(Field, source)
        if field:GetSymbol() = ""
            field:SetSymbol(player)
        // ...
    end
```

Das Problem war, dass wenn auf das Symbol eines schon gesetzten Felds geklickt wurden ist, `event:GetSource()` nicht das Feld, sondern das Symbol zurückgegeben hat und der Cast ungültig war.

Um dieses Problem zu lösen, wird in der `Field`-Klasse ebenso ein Event-Handler für das Klicken auf das Feld registriert und mit `event:SetSource(me)` wird die Eventquelle mit dem Feld überschrieben (s. Anhang C.10, Z. 54).

Wenn Quorum anonyme Methoden unterstützen würde, wäre die Implementierung einfacher gewesen, da die `Feld`-Referenz direkt verwendet hätte werden könnte und man keinen *Cast* bräuchte. In etwa so:

```
Field field
field:AddMouseListener(event =>
    if not gameOver and event:IsClicked() and event:IsLeftButtonEvent()
        HandleFieldClick(field)
)
```

Ansonsten ist noch zu erwähnen, dass die Implementierung von Tamagochi 6 Klassen-Dateien benötigt hat (die anderen beiden Spiele haben nur 2 benötigt).

Quorum hat einen grafischen Szenen-Editor (s. [Lan20]). Diesen habe ich jedoch nicht benutzt, da er in *Quorum Studio* integriert ist. Ich habe ihn kurz ausgetestet und würde die *Usability* als nicht so hoch bewerten.

Das Koordinatensystem ist rechtshändig, die Drehrichtung gegen den Uhrzeigersinn und somit alles nach mathematischer Konvention. Der Ankerpunkt ist mit unten links gut gesetzt.

Es gibt eine umfangreiche GUI-Bibliothek mit Steuerelementen wie Buttons, Textboxen,

Dialogen, Menüleisten, Spreadsheets, Toggle-Buttons usw. (s. [Quo18]).

Audio kann geladen und abgespielt werden.

Für die Animation von Sprites war keine Klasse vorhanden, konnte jedoch relativ einfach selbst implementiert werden, indem die `Drawable`-Klasse erweitert wird, wie ich es bei *Flappy Bird* gemacht habe (s. Anhang C.12).

Die Abstraktion vom *Game Loop* geschieht über Vererbung. Dem Spiel werden dabei grafische Elemente hinzugefügt, die automatisch dargestellt werden. In der `Update(number time)`-Funktion können die grafischen Elemente dann z.B. bewegt werden. Besonders praktisch für Physik basierte Spiele, wie Flappy-Bird ist die Physics-Engine. Durch sie kann u. a. Gravitation für Elemente simuliert, Kräfte auf Objekte angewandt und Physik-Kollisionen erkannt werden. In Pyret und Scratch musste die Gravitation dagegen selbst implementiert werden.

Erlernbarkeit nach Konstruktionismus

Es gibt viele motivierende Anwendungsbereiche in denen Quorum verwendet werden kann. Dies liegt an der großen Standardbibliothek von Quorum (s. [Quo18]), welche Klassen bereitstellt, welche eine Basisstruktur für eine bestimmte Art von Anwendung vorgibt (s. Tabelle 4.1).

In der Standardbibliothek gibt es sogar die Klasse `TurtleGame` mit der man *Turtle-Grafiken* erstellen kann. Beim Ausprobieren dieser Klasse, kam allerdings die Fehlermeldung:

Error reading file:

```
C:\Users\lulug\Documents\Quorum Studio\TurtleGraphics\media\code\Grid.png (Absolute)
```

Die Klasse hat also als Voraussetzung, dass bestimmte Bilder im Projektordner abgelegt sind. Leider konnte ich nicht herausfinden, wo ich diese Bilder runterladen kann. Außerdem war diese Klasse in der Dokumentation nicht gut dokumentiert und ich habe den Versuch abgebrochen ein Beispiel zum Laufen zu bringen.

Quorum bindet viele verschiedene Technologien ein, die das Lernen der Programmierung für Beginner motivierend macht. Neben *Lego Mindstorm* gibt es auch die Möglichkeit mit Quorum ein Spiel für Android zu entwickeln. Weitere Beispiele sind das Erstellen von Musik oder das Synthesisieren von Audio.

Ein weiteres Beispiel ist, dass es neben dem `output`-Kommando, um Text auszugeben, ein `say`-Kommando gibt, welches den Text über *Text-to-Speech* ausgibt.

Neben 2D-Grafiken und -Physics, unterstützt Quorum auch 3D-Grafiken und -Physics. So können auch 3D-Modelle geladen und 3D-Spiele erstellt werden.

Die Standardbibliothek unterstützt das Einlesen, Manipulieren, Schreiben und Ausgeben von folgenden Datenformaten: SVG, JSON, HTML, XML und MIDI.

Zudem gibt es eine gute Unterstützung für *Data Science*. So gibt es die Klasse `DataFrame` in welche Daten geladen werden können. Es gibt ein umfangreiches Statistik-Paket und es können Plots und Diagramme erstellt werden.

| Klasse/n | Beschreibung |
|---|--|
| Libraries. Science. Astronomy. Skynet | Diese Klasse wird verwendet, um über das Internet und das Token Ihres Skynet-Kontos Anweisungen an das Skynet-Roboterteleskop-Netzwerk zu senden. Wenn Sie kein Konto haben, wenden Sie sich für Informationen an Skynet (https://skynet.unc.edu/help/contact). |
| Libraries. Sound. Music | Diese Klasse generiert Musik aus dem Standard Music Instrument Digital Interface (MIDI). Diese Klasse kann zum Spielen einzelner Noten und Akkorde sowie zum Komponieren von Mehrspur-Songs verwendet werden. Diese Klasse abstrahiert einen Großteil der MIDI-Schnittstelle, was die Komposition von Songs erheblich vereinfacht. |
| Libraries. Robots. Lego | Klassen wie Battery, Button, ColorSensor, Screen als Abstraktion, um auf die LEGO Mindstorms EV3 Elemente zugreifen. |
| Libraries. Data. Formats. ScalableVectorGraphics | Klassen zum Erstellen von SVG-Grafiken. |
| Libraries. Curriculum. TurtleProgram. TurtleGame | Klasse zum Erstellen eines Spiels mit dem <i>Turtle</i> -Grafiken erstellt werden können. |

Tabelle 4.1. Quorum: Klassen, die eine Entdeckung eines bestimmten Bereich ermöglichen (s. <https://quorumlanguag.com/libraries.html>)

Insgesamt ist die Standardbibliothek mit 1121 Klassen, verteilt in 69 Paketen sehr umfangreich.

Die Programmierumgebung von Quorum ist weniger offenbarend. Für die Einführung sind explizite Instruktionen notwendig. Jedoch hat *Quorum Studio* eine intelligente Autovervollständigung und es können die Methoden innerhalb der IDE für eine Klasse angezeigt werden.

Da die Programmierumgebung lokal auf dem Computer installiert werden kann und in Quorum potenziell schädliche Programme geschrieben werden können, ist die Umgebung nicht ganz so sicher. Es gibt allerdings auch die Möglichkeit Quorum eingeschränkt im Web auszuführen.

Quorum ist statisch typisiert, hat allerdings *Type-Inference*, wo der Typ einer Variable aus dem zugewiesenen Ausdruck automatisch bestimmt werden kann.

Inkrementelle Einführung

Das „Hallo Nutzer“-Programm in Quorum sieht folgendermaßen aus.

```
text name = input("Hallo, wie heisst du?")
```

```
say "Hallo " + name + "!"
```

Über einen Textprompt wird der Benutzer zur Eingabe aufgefordert und der Benutzer wird dann mit `say` über *Text-To-Speech* begrüßt. Es hätte ebenso gut das `output`-Kommando verwendet werden können, um den Text auszugeben.

Obwohl Quorum eine objektorientierte Programmiersprache ist, ist keine Klasse notwendig um ein Quorum-Programm auszuführen. Dabei geht Quorum technisch so vor, dass wenn keine Klasse vorhanden ist, der Code automatisch in eine `Main`-Aktion einer `Main`-Klasse eingefügt wird.

Außerdem gibt es die speziellen Kommandos `input`, `output` und `say`, die es ohne Erstellung eines Objekts ermöglichen Benutzereingaben einzulesen und etwas dem Benutzer auszugeben. So erlaubt es Quorum zuerst die Elemente der strukturellen Programmierung wie Variablen, Bedingungen und Schleifen mit den primitiven Datentypen zu erlernen. So muss nicht von Anfang an Objektorientierung eingeführt werden und die kognitive Belastung ist geringer. In Quorum gibt es allerdings keine Funktionen, die unabhängig von einer Klasse definiert werden können. So muss die prozedurale Programmierung spätestens mit der objektorientierten Programmierung eingeführt werden.

Da Quorum *Type-Inference* hat, müssen mit Variablen auch nicht direkt mit Datentypen (Annotationen) eingeführt werden. So hat ein Lehrer große Freiheiten einen Kurs zu implementieren.

Intuitive Syntax und Natürlichsprachlichkeit

Quorum's Syntax-Design-Entscheidungen basieren auf den Ergebnissen der im Abschnitt 3.4.4 zusammenfassten Studie ([SS13]) über intuitive Syntax. Die Ergebnisse der Studie flossen direkt in das Design von Quorum ein. Dadurch ist Quorum's Syntax besonders intuitiv für Beginner.

Die Syntax ist jedoch nicht internationalisiert.

Es werden auch syntaktische Feinheiten berücksichtigt. Bei Aktionen steht der Rückgabetyp über `returns` am Ende der Deklaration (*Trailing return type*).

Was besonders auffällt ist, dass es kein Token für den Anfang einer Blockstruktur gibt, eine Blockstruktur jedoch mit dem Token `end` beendet wird. Dies wird konsistent so angewendet für den Beginn von Aktionen sowie den Beginn von Syntaxkonstrukten wie `if <bool-expr>` oder `repeat until <bool-expr>`. Da es keine Syntax gibt, um einen Block zu öffnen, könnte es schwieriger für Beginner sein, das Konzept von *Scopes* zu erlernen.

Feature Uniformity und Orthogonalität

Die einzige Syntax, die in Quorum als syntaktisches Synonyme zählen könnten sind die Schleifen. Es gibt die `repeat <n>`-Schleife, die `n` mal durchläuft, dann gibt es die `repeat until <bool-expr>`-Schleife, die solange durchläuft bis der Ausdruck wahr ist und die `repeat while <bool-expr>`-Schleife, die solange durchläuft wie der Ausdruck wahr ist. Die `while`-Schleife wäre ausreichend, um die anderen Schleifen ausdrücken zu können.

Jedoch bietet die `repeat <n>`-Schleife den Vorteil, dass keine Zählervariable erstellt werden muss. Bei der `repeat <n>`-Schleife handelt es sich nicht um syntaktischen Zucker, da

sie nicht einfach durch eine `while`-Schleife ersetzt werden kann, da das Erstellen einer Zählervariable vonnöten ist. Es könnte schon eine Variable mit dem gleichen Namen im *Scope* deklariert wurden sein und das Programm würde nicht mehr kompilieren (Beweis nach Felleisen [Fel91]).

Es konnten keine syntaktischen Homonyme identifiziert werden.

Die Sprache hat für eine objektorientierte Sprache eine sehr kleine, simple Funktionsmenge. Es gibt keine abstrakten Klassen oder Schnittstellen. Abstrakte Klassen oder Schnittstellen können dennoch so in der Art über Klassen und Vererbung implementiert werden. Es wird ein Beispiel aus der Tamagochi-Implementierung gegeben, indem die Implementierung mit einer potenziellen Implementierung in Java verglichen wird.

Bei Tamagochi fügt die `Main`-Klasse dem Haustier ein `PetListener` hinzu, um auf Änderungen der Energie, der Liebe und des Status des Haustiers zu reagieren (s. Anhang C.13). Der `PetListener` ist eine normale Klasse in Quorum (s. Listing 4.29), in Java würde man es jedoch über ein Interface implementieren (s. Listing 4.30):

```
class PetListener
    action OnPetLoveChanged(number love) end
    action OnPetEnergyChanged(number energy) end
    action OnPetStatusChanged(integer status) end
end
```

Listing 4.29. `PetListener` in Quorum

```
interface PetListener {
    public void onPetLoveChanged(double love);
    public void onPetEnergyChanged(double energy);
    public void onPetStatusChanged(int status);
}
```

Listing 4.30. `PetListener` in Java

Nun kann die `Main`-Klasse dieses Interface implementieren. Dies geschieht in Quorum über Mehrfachvererbung und in Java über `implements`:

```
class Main is Game, PetListener, Behavior
```

Listing 4.31. Implementierung des `PetListener` in Quorum

```
public class Main extends Game implements PetListener, Behavior
```

Listing 4.32. Implementierung des `PetListener` in Java

Einen semantischen Unterschied gibt es. Java erzwingt die Implementierung der Schnittstelle und bei Quorum müssen die Aktionen der `PetListener`-Klasse nicht zwingend überschrieben werden.

Bemerkenswert ist, dass Quorum auf Klassen-Konstruktoren komplett verzichtet. Wenn Objekte erstellt werden, dann können sie über Aktionen initialisiert werden. Dies ist eine gewagte Entscheidung, da das Initialisieren von Objekten über einen Konstruktor ein sehr wichtiges Konzept in der Objektorientierung ist.

Es gibt außerdem keine statischen Variablen oder Aktionen. Dies kann unpraktisch sein, da wenn z.B. auf eine Konstante einer Klasse zugegriffen werden soll, zunächst eine Instanz erstellt werden muss.

Quorum reduziert den Funktionssatz von größeren objektorientierten Sprachen wie Java auf das Nötigste.

Insgesamt hat Quorum eine sehr konsistente Syntax. Die einzige Inkonsistenz, die identifiziert werden konnte, sind die Kommandos `output`, `say`, `input` und `alert`. Diese Kommandos haben eine eigene spezielle Syntax. Bei `output` und `say`, muss der Text der ausgegeben werden nicht in runden Klammern stehen. Bei Aktions-Aufrufen jedoch, werden Parameterklammern erforderlich. Außerdem ist es inkonsistent, dass bei dem Kommando `input` und `alert` wiederum Klammern benötigt werden.

Diese Kommandos haben ihre eigenen Regeln. Der Benutzer selbst kann keine Kommandos erstellen und muss sie einfach als gegeben hinnehmen.

Die konsistente Syntax von Quorum, fällt bei der Verwendung von generischen Datentypen auf. In Quorum ist es möglich primitive Datentypen als Typargument bei *Generics* anzugeben, z.B. `List<integer>`. Dies ist in Java nicht möglich. Wie in Java, hat Quorum Klassen, die einen *Wrapper*, um die primitiven Datentypen bilden und Referenztypen sind, wie z.B. die Klasse `Integer`. Anders als in Java aber, kann als Typargument der primitive Datentyp verwendet werden. Wenn ein primitives Element dann z.B. einer Liste hinzugefügt wird, dann wird dieses automatisch in ein Objekt der Klasse umgewandelt. Dies nennt man *Autoboxing* (vgl. [Lan19b]).

Unterstützung der wichtigsten Programmierkonzepte & Datenstrukturen

Die Elemente der strukturierten und prozeduralen Programmierung werden unterstützt.

Die Datenstrukturen sind in dem Paket `Libraries.Containers` zu finden. Es gibt die Klassen `List`, `Queue`, `Stack`, `HashTable` und mehr. Das einzige was fehlt ist eine Klasse für ein *Set* (Menge). Allerdings kann dafür auch die `HashTable` verwendet werden, indem nur die Schlüssel beachten werden. Beim Einfügen kann dann einfach `undefined` als Wert verwendet werden.

Die rekursive Fibonacci-Funktion kann einfach implementiert werden:

```
action Fib(integer n) returns integer
    if n <= 0
        alert(n + " muss größer 0 sein!")
    elseif n = 1 or n = 2
        return 1
    else
        return Fib(n-1) + Fib(n-2)
    end
end
```

Listing 4.33. Quorum: Fibonacci-Funktion

Der Compiler von Quorum ist *self-hosting*.

Programmierumgebung (IDE)

Die Bewertung der IDE ist in Quorum etwas schwieriger, da es drei verschiedene Umgebungen gibt:

- 1) **Quorum-Studio:** *Quorum Studio* ist eine Desktopanwendung und enthält alle Funktionalitäten, um Quorum-Anwendungen zu erstellen (s. Anhang A.1). Sie ist bei mir öfters abgestürzt und die *Usability* war nicht so gut. Deshalb habe ich mich für Umgebung 3 (Kommandozeile) entschieden.
- 2) **Web-Editor und -Runner:** Hier handelt es sich, um einen sehr minimalistischen Editor (s. Abb. 4.10). Der Code wird serverseitig in Javascript kompiliert, zurückgegeben und dann ausgeführt. Wenn auf *Run Program* geklickt wird, wird ein neuer Tab geöffnet, indem der Javascript-Code ausgeführt und die Ausgabe angezeigt wird. Mit diesem Editor können nur Anwendungen geschrieben werden, für die nur eine Quellcode-Datei verwendet wird. Da *Bouncing Ball* in nur einer Datei implementiert wurden ist, habe ich versucht nach dem Entfernen der Audio-Zeilen, diesen im Web-Editor auszuführen. Allerdings kommt dann die Fehlermeldung:

I did not understand:

```
Line 58, Column 13: I could not find an action named 'Move' in the class
'Libraries.Game.Graphics.Drawable' with the parameter
Libraries.Compute.Vector2
```

Es scheint so als würde der Web-Editor eine andere API benutzen.

- 3) **Kommandozeile:** Diese Umgebung habe ich schlussendlich mit dem Editor *VS-Code* verwendet, um die Spiele zu implementieren.

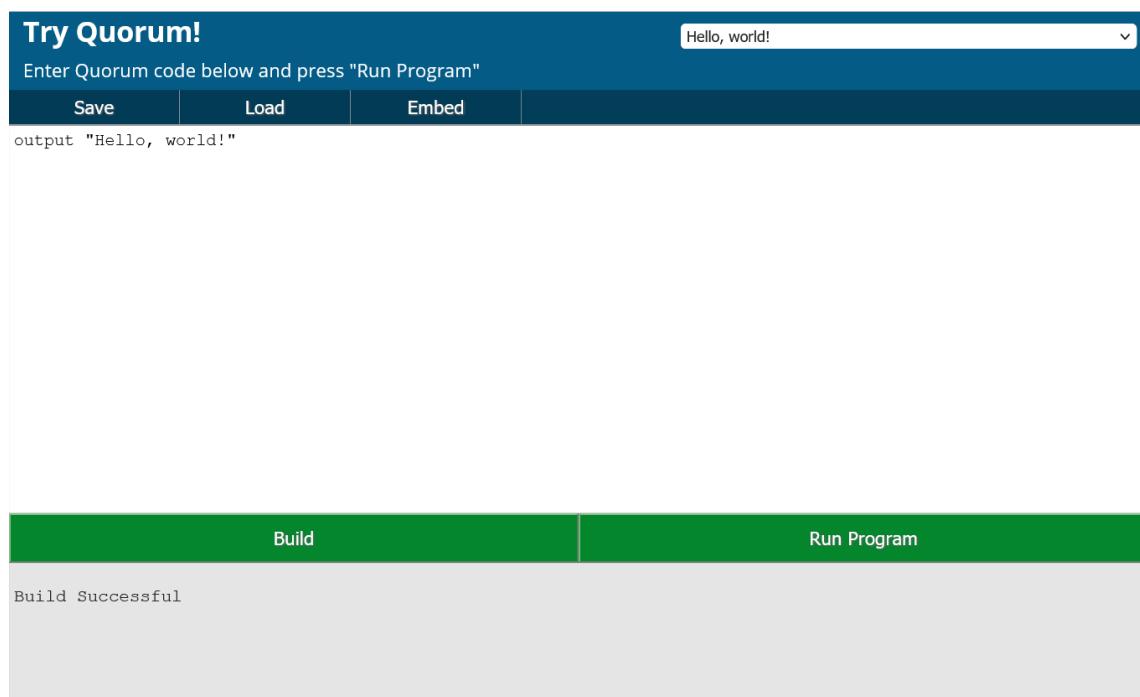


Abb. 4.10. Quorum's Web-Editor

Im Weiteren werden alle dieser 3 Umgebungen berücksichtigt. Der *Usability*-Faktor kommt in den Bewertungskriterien nicht vor, da eine isgesamte *Usability* sehr schwer zu

bewerten ist. Da die *Usability* von *Quorum-Studio* meiner Erfahrung nach eher gering ist, versuche ich dies bei der Bewertung zu berücksichtigen. Schlussendlich kommt es immer sehr auf Implementierungsdetails an, diese werden hier jedoch nicht genauer untersucht.

Quorum-Studio sowie der *Web-Runner* sind sehr einfach aufgebaut und bieten eine große Übersichtlichkeit.

Die *Turnaround Time* ist bei allen Umgebungen etwas gleich lang. Die Kompilierung bei kleineren Programmen dauert etwa eine Sekunde. Bei dem *Web-Runner* ist noch der *Netzwerk-Overhead* da, zeitlich dauert es jedoch nicht viel länger als bei den lokalen Umgebungen.

Es gibt keinen REPL und auch kein *Live Coding*. Somit ist die Interaktivität mit -1 (unzureichend erfüllt) zu bewerten.

Quorum Studio hat einen visuellen interaktiven Debugger. Diesen habe ich jedoch nicht verwendet und kann nicht viel dazu sagen. Die Symbolleiste von *Quorum-Studio* hat in der Werkzeugeiste Symbole für *Step In*, *Step Over* und *Step Out* und *Shortcuts* für diese Funktionen. Somit hat der Debugger auf jeden Fall die wichtigsten Funktionen.

Unit-Tests werden ebenfalls unterstützt. Der nächste Bewertungsbereich wirft einen genaueren Blick darauf.

Die lokale Installation geschieht über einen *Installer*. Es kann entweder *Quorum Studio* oder die Konsoleninstallation installiert werden. Zudem kann Quorum auch ohne die Installation über den *Web-Editor* ausgeführt werden. Hierbei kann auch ein Konto erstellt werden, sodass Programme gespeichert und geladen werden können, wenn man eingeloggt ist.

Für die Teamarbeit bietet *Quorum-Studio* eine *Git*-Integration, wobei diese nur die wichtigsten Funktionen von *Git* enthält. Es wurde kein genaueren Blick auf die *Git*-Integration geworfen.

Um Code zu teilen, kann der *Web-Runner* mit dem Code in eigene Webseiten eingebettet werden. Der *Web-Runner* bietet eigens einen *Embed*-Button, um den Code für die Einbettung zu generieren.

Quorum-Studio hat moderne Funktionen wie Syntax-Highlighting, Code-Completion und -Refactoring. Es lassen sich Getter und Setter generieren. Jedoch gibt es keine Code-Navigation und -Formatierung.

Der *Web-Editor* hat gar keine dieser modernen Funktionen. Es gibt noch nicht einmal eine Zeilennummerierung.

Viele Programmiersprachen bieten Plugins für IDEs wie z.B. für *VS-Code* an. Es kann Syntax-Highlighting, aber auch moderne Features wie Code-Completion und -Navigation über das *Language Server Protocol* (LSP) (s. [Mic21a]) implementiert werden.

Dabei ist die Implementierung von Syntax-Highlighting je nach Grammatik der Programmiersprache nicht kompliziert und geschieht bei *VS-Code* über die Definition einer *TextMate-Grammar* (s. [Ltd21]). Ein *Language Server* über das LSP zu erstellen ist dagegen eine sehr komplexe Angelegenheit, da ein inkrementeller Compiler benötigt wird, der

die Änderungen von einzelnen Quellcodedateien akzeptiert (vgl. [Wik20d]).

Quorum bietet keine *Plugins* für existierende IDEs an. Da eine IDE ein sehr komplexes Produkt ist, sollten Sprachentwickler, wenn sie eine IDE-Unterstützung für ihre Sprache haben wollen, sich überlegen nicht das LSP zu implementieren, sodass die Sprache in einer schon existierenden IDE als Plugin verwendet werden kann.

VS Code basiert auf den *Monaco Editor*, welcher *Open Source* ist, in Webseiten verwendet werden kann und ebenfalls das LSP unterstützt (s. [Mic21b]). Ein solcher Editor könnte also gut verwendet werden, um eine Web-IDE, dessen Editor schon viele Funktionen bereitstellt, zu entwickeln.

Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplin (*Clean Code*)

Code wird in Quorum in Klassen organisiert. Klassen können wiederum im Paketen zusammengefasst werden (wie in Java). Einzelne Klassen von anderen Paketen können dann über `use` importiert werden. Es können auch alle Klassen eines Pakets über `all` am Ende importiert werden, z.B. `use Libraries.Containers.all` um alle *Container*-Klassen zu importieren.

Quorum bietet über einen *Information Hiding*-Mechanismus über die Zugriffsmodifizierer `private` und `public`. Der Zugriffsmodifizierer `private` ist dabei der Standard und somit wird automatisch defensiv programmiert. Auf private Instanzvariablen können jedoch auch erbende Klassen zugreifen. Es gibt keinen Zugriffsmodifizierer wie `protected` in Java. (vgl. [Lan19a]).

Quorum unterstützt Dokumentationskommentare, hat jedoch keine dedizierte Syntax für diese Kommentare und verwendet einfach mehrzeilige Kommentare.

Hier ist ein Beispiel der Dokumentation der `Set`-Methode der `Array`-Klasse:

```
/*
   This action sets the item at a given location in the indexed object to a new item.

   Attribute: Parameter location The index or location the value will be stored at.

   Attribute: Parameter value The item to be added to the indexed object.

   Attribute: Example
   use Libraries.Containers.Array
   Array<integer> myArray
   myArray:SetSize(10)
   myArray:Set(0, 22)
*/
action Set(integer location, Type value)
```

Listing 4.34. Dokumentationskommentare in Quorum

Hier sieht man, dass es bestimmte Attribute wie `Parameter` oder `Example` gibt, aus denen dann automatisch Dokumentation generiert werden kann. Die Standardbibliotheks-Dokumentation ist automatisch generiert und in der Standardbibliothek ist die Klasse

`DocumentationGenerator` zu finden, welche eine Dokumentation von Quellcode-Dateien generieren kann (vgl. [Quo18]).

Es konnte jedoch keine Dokumentation gefunden werden, die beschreibt welche Attribute es für die Dokumentation gibt und wie genau die Spezifikation dieser Dokumentationskommentare aussieht. Zudem wird bei der Auto vervollständigung in *Quorum Studio* diese Dokumentation nicht genutzt, um mehr Informationen anzuzeigen.

Quorum ist statisch typisiert. In Quorum gibt es explizite *Casts* über die `cast(<type>, <expr>)`-Syntax. *Type-Coercions* gibt es bei der Konkatenation von Strings, bei der primitive Datentypen automatisch in eine Zeichenfolge konvertiert werden. Dies gilt nur für primitive Datentypen und nicht für Objekte. Ebenso kann eine Operation von einem `integer` und `number`-Ausdruck angewendet werden, wobei der Typ der evaluierten Operation dann `number` ist. Es gibt also relativ wenige *Type Coercions*, welche in einem Kontext verwendet werden, welcher Sinn macht und den Code weniger verbos machen.

Quorum unterstützt *Exception-Handling* auf die selbe Art und Weise wie Java. Die Schlüsselwörter sind `check (try)`, `detect (catch)` und `always (finally)` und `alert (throw)`.

Die Standardbibliothek von Quorum verfügt über die Klassen `Libraries.Testing.Tester` und `Libraries.Testing.Test` um Unit-Tests zu definieren und auszuführen. Dabei werden über die `Test`-Klasse Test-Objekte erstellt, die einer Instanz des `Tester`s hinzugefügt werden. Der `Tester` führt die Tests dann aus und gibt zurück, welche Tests erfolgreich oder unerfolgreich sind. Um Tests zu erstellen, würde man also von der `Test`-Klasse erben, die Run-Methode überschreiben und Test-Methoden wie `Check(number value, number expected)` aufrufen. Dann würde man ein Objekt dieser Testklasse erstellen und dieses dem `Tester` hinzufügen. Der `Tester` kann dann am Ende mit mehreren Tests ausgeführt werden (s. [Quo18]).

Quorum-Studio unterstützt keine grafische Ausgabe der Tests. Die Test-Ausgabe beschränkt sich so auf Text.

Quorum verfügt über einen *Style-Guide* (s. [Lan19c]). Dieser ist aber allerdings sehr kurz gehalten. Dafür bezieht er Regeln mit ein, die Quorum-Code besonders für Beginner lesbar gestalten (s. [Lan19c]), wie z.B.

- Do not use acronyms (e.g., awk, sed, ascii, io)
- Do not use computer science or technical jargon (e.g., virtual void abs() = 0;, BTree, or LinkedList)
- Do not make names excessively verbose (e.g., ThisActionIsReallyAwesomeButHardToType())

In Quorum gibt es keine Operatoren mit Nebeneffekten, Operatoren können nicht überladen werden und Variablen können nicht überschattet werden.

Fehlermeldungen

Hier sind einige Beispiele für Fehlermeldungen in Quorum:

`Main.quorum, Line 75, Column 13: I could not locate a type named bool.`

Did you forget a use statement?

I did not understand:

```
Line 58, Column 13: I could not find an action named 'Move' in the class
'Libraries.Game.Graphics.Drawable' with the parameter
Libraries.Compute.Vector2
```

```
Main.quorum, Line 3, Column 8: Cannot assign a value of type "text" to
a variable of type "integer".
```

Es wird einfaches Vokabular verwendet, welches zur Syntax von Quorum passt, z.B. `action`, statt `method`.

Die Fehlerposition wird angezeigt. In *Quorum Studio* werden Fehlermeldungen rot unterkragt. Die textuelle Ausgabe zeigt nicht den fehlerhaften Code mit der Fehlerposition an. Dies ist durchaus verbesserungswürdig.

Es werden wenige unterstützende Informationen angezeigt. Um die Ausgabe des *Stack Traces* bei einem Laufzeitfehler zu überprüfen, wurde folgendes kleines Programm mit Rekursion geschrieben:

```
class Main
    action Main
        Run(0)
    end
                Error: x greater 100
    action Run(integer x)      file: Test.quorum,  class: Main,  action: Run,  line: 8
        if x > 100              file: Test.quorum,  class: Main,  action: Run,  line: 10
            alert("x greater 100") file: Test.quorum,  class: Main,  action: Run,  line: 10
        end                      ... (noch 99 mal)
        Run(x + 1)              file: Test.quorum,  class: Main,  action: Main,  line: 3
    end
end
```

Listing 4.35. Quorum: 100 mal Rekursion dann Fehler

Das Programm rekursiert 100 mal und wirft dann einen Fehler. Die Fehlermeldung ist auf der rechten Seite zu sehen. Das Problem an der Fehlermeldung ist, dass bei dem *Stack-Trace* der rekursive Aufruf 100 Mal wiederholt wird und die Ausgabe so 100 Zeilen lang ist. Es hätte besser designt sein können, z.B. so wie ich die Fehlermeldung oben über `... (noch 99 mal)` abgekürzt habe.

Dies ist ein störendes Element.

Quorum verwendet einen positiven Ton in den Fehlermeldungen. Der Compiler wird personalisiert und die Fehlermeldungen werden aus der Sicht des Compilers in der ersten Person Singular ausgegeben.

Bei sehr sicheren Fehlerquellen werden Lösungen vorgeschlagen, wobei der Compiler die mögliche Ungewissheit über das Wort `think` ausdrückt, z.B.:

`Main.quorum, Line 12, Column 12:`

`In order to return a value of type "integer" from this action,`

I think we need the phrase "returns integer" before the body.

Die Fehlermeldungen sind nicht lokalisiert.

Ein hohes Abstraktionsniveau für Datentypen

Der primitive `text`-Datentyp in Quorum unterstützt zwar Unicode, *Quorum Studio* und die Ausgabe im *Web-Runner* jedoch nicht.

Quorums Zahlentypen verwenden für `integer` eine 32-Bit-Ganzzahl und für `number` eine 64-Bit-Fließkommazahl. Da Quorum nicht nur in *Java-Bytecode* kompiliert werden kann, sondern auch in Javascript, gehe ich davon aus das für Javascript für beide Datentypen einfach der `number`-Typ verwendet wird (es gibt keine Sprachspezifikation und der Quellcode ist zwar *Open Source*, ich konnte aber keinen Code für die Javascript-Kompilierung finden).

In der Standardbibliothek gibt es zudem keine Klasse für Langzahlarithmetik (s. [Quo18]).

Anwendungsbereiche und Interoperabilität mit Technologien und anderen Programmiersprachen

Quorum ist eine sehr entwickelte *Teaching Language* und hat eine große Standardbibliothek, welche viele Funktionen abdeckt. Es gibt keinen Quorum-Paketmanager, in dem Quorum-Pakete verteilt werden könnten. Dafür bietet Quorum alleine schon in der Standardbibliothek eine riesige Auswahl an Paketen und Klassen, welche mehr als genug mögliche Anwendungsbereiche für den Unterricht abdecken (s. Anhang A.3). Diese große Standardbibliothek ist ein enormer Vorteil für eine *Teaching Language*, da dadurch die Komplexität eines Paketmanagers oder die manuelle Einbindung von Bibliotheken wegfällt.

Was allerdings fehlt ist ein Paket für Datenbanken.

Da Quorum in *Java Bytecode* kompiliert und Java so gut wie überall laufen kann, kann Quorum auf vielen verschiedenen Plattformen laufen.

Quorum selbst gibt Beispiele und Tutorials für folgende Plattformen:

- Konsolenanwendungen
- grafische Desktopanwendungen
- Spieleentwicklung auf Android-Smartphones
- LEGO Mindstorm EV3 Roboter

Zudem kann Quorum auch nach Javascript transpilieren und es können interaktive und grafische Programme für den Webbrowser geschrieben werden.

Quorum ist interoperabel mit Java und kann andere Java-Bibliotheken aufrufen. Dies funktioniert durch Plugins. Hierfür gibt es Systemaktionen, die über `system action` erstellt werden. Die Implementierung der Systemaktionen geschieht dann in Java in dem Paket `plugins.quorum.Libraries.Mine`. Die Bibliothek kann dann in ein jar-Bündel kompiliert werden und bei der Kompilierung von Quorum mit eingebunden werden (vgl. [Lan19d]).

Dokumentation & Community Support

Die Standardbibliothek von Quorum ist nahezu vollständig dokumentiert (s. [Quo18]). Ein Problem an der Dokumentation der Standardbibliothek ist, dass sie schlecht navigierbar ist. Es gibt keine Seiten- und Suchleiste. Die Suche kann jedoch über die Browsersuche erfolgen. Doch eine Seitenleiste, wo die Pakete aufgelistet sind und bei Bedarf aufgeklappt und die Klassen angezeigt werden könnten, wäre sehr praktisch.

Quorum bietet eine exzellente Einführung und Vertiefung in weitere Bereiche über die Referenzseite (s. [Lan20]). Dort gibt es 12 Kapitel:

- A. Introduction to Quorum
- B. Advanced Quorum
- C. Scene Editing
- D. Graphics and Media
- E. Game Physics
- F. Audio
- G. User Interface
- H. LEGO Robots
- I. Reading and Writing Data
- J. Network
- K. Mobile
- L. Other Resource

Außerdem gibt es noch bestimmte *Tracks* die das Erlernen von von Quorum innerhalb eines bestimmten Anwendungsbereichs ermöglichen:

- 1. Core Track
- 2. Visual Track
- 3. Audio Track
- 4. Robotic Track
- 5. Computer Science Principles Track
- 6. Teacher Track

Zudem partizipiert Quorum in *Hour of Code* (s. [Cod15]) – eine Initiative von *Code.org* in der einstündige Programmieranfänger-Kurse für Kinder angeboten werden. Hier bietet Quorum zwei Tutorials an: *Code With Mary* und *Astronomy*.

Die Beispiele und Tutorials in Quorum sind nicht internationalisiert.

Quorum bietet ein Google- sowie eine Facebook-Gruppe und eine Mailingliste an, um mit der Community in Kontakt zu treten.

4.3. Scratch

4.3.1. Kurze Einführung in Scratch

Scratch ist eine blockbasierte visuelle Programmierumgebung, die sich hauptsächlich an Kinder und Jugendliche in der Altersgruppe 8 bis 16 richtet. Das Scratch-Projekt begann 2003 am *MIT Media Lab* und wurde 2007 veröffentlicht. Dabei baut Scratch auf die kon-

struktionistischen Ideen von *Logo* und *Etoys* auf (vgl. [Mal+10]). Die Idee hinter Scratch ist es an persönlichen Projekten zu arbeiten, wie animierte Geschichten und Spiele und gleichzeitig Programmieren zu lernen (vgl. [Mal+10]). Ein Scratch-Projekt besteht dabei aus Skripts und aus Medien wie Bilder und Töne. Eine größere Auswahl an Bildern und Tönen ist in Scratch bereits vorhanden, Bilder und Töne können jedoch auch hochgeladen werden oder gar in Scratch selbst erstellt oder aufgenommen werden.

4.3.2. Beschreibung der Programmierumgebung

Das Programmieren funktioniert dabei durch das Aneinanderreihen von visuellen Blöcken, die ineinander einrasten und zusammen ein Skript ergeben. Dabei ist ein Skript immer auf einem Objekt aktiv. Dabei gibt es auch noch den Hintergrund, die sogenannte Bühne, welche auch ein Skript haben kann. Die Blöcke sind in verschiedene Farben unterteilt, die jeweils für einen anderen Bereich zuständig sind. So sind alle Blöcke, die etwas mit Bewegung zu tun haben blau und alle Blöcke, die etwas mit Steuerung und Logik zu tun haben orange.

Auch ist die Form von Blöcken relevant: So gibt es Startblöcke, die einen gerundeten Bogen haben und bei denen es nicht möglich ist, etwas darüber einzurasten (siehe 4.11a). Diese werden für Ereignisse verwendet. Dann gibt es Kontroll-Blöcke wie Schleifen, die mehrere Blöcke enthalten können (b). Die Prozedur-Blöcke führen eine bestimmte Prozedur aus (c). Es gibt rautenförmige Bedingungsblöcke (d), die in Kontrollblöcke wie 'falls <Bedingung> dann' eingesetzt werden können (e). Schließlich gibt es noch gerundete Blöcke, wie z.B. 'Zufallszahl von (A) bis (B)' (f) oder Operatoren wie '+' (g), welche Werte darstellen.

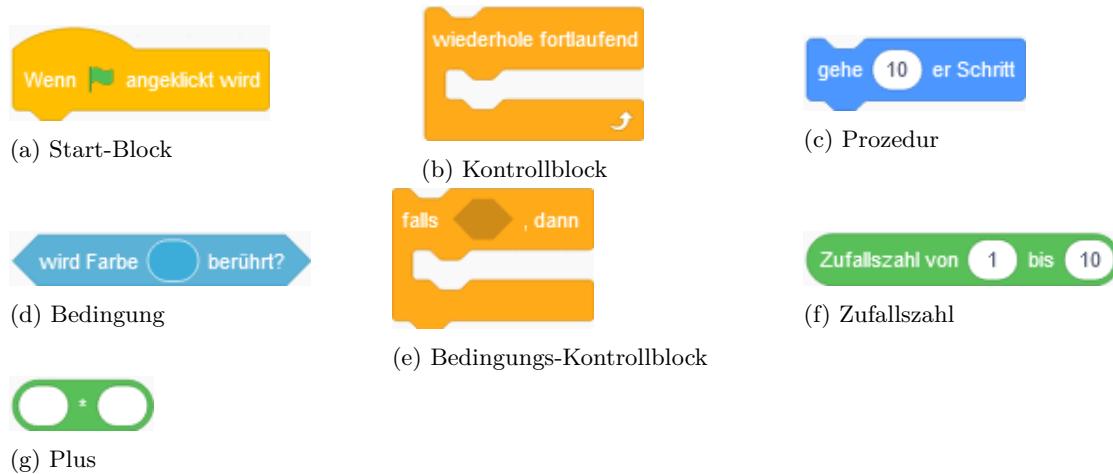
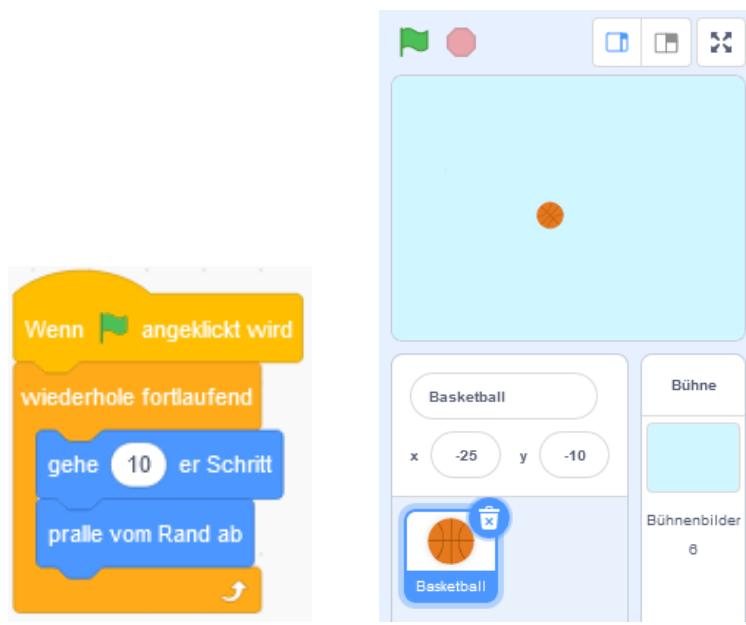


Abb. 4.11. Scratch: Blockarten

Scratch 3.0 verwendet die von Google entwickelte Open-Source-Bibliothek mit dem Namen *Blockly*, mit welcher man eigene blockbasierte, visuelle Programmiersprachen erstellen kann (s. [Pas19]).

4.3.3. Bouncing Ball

Bouncing Ball ist in Scratch sehr einfach zu implementieren. Hierfür wird zuerst ein Ball hinzugefügt und die Rotation des Balls auf 45 °gesetzt. Das Skript kann geschrieben werden, indem in einer Unendlich-Schleife die zwei blauen Bewegungs-Blöcke `gehe 10er Schritt` und `pralle vom Rand ab` platziert werden. Besonders praktisch ist der Block `pralle vom Rand ab`, da dieser Block automatisch den Winkel des Balls verändert, wenn der Ball an den Rand der Bühne stößt. Der Block `gehe 10er Schritt` bewegt den Ball 10 Pixel in die Orientierungsrichtung des Balls. Wenn der Ball sich schneller oder langsamer bewegen soll, kann die Schrittgröße vergrößert oder verringt werden.



(a) Skript für den Ball

(b) Bühne mit Ball

Abb. 4.12. Scratch: Bouncing Ball

Der Nachteil ist, dass über `pralle vom Rand ab` nicht programmieren werden kann, dass bspw. ein Ton gespielt wird, wenn der Ball den Rand berührt. Es gibt leider keinen Bedingungs-Block, der prüft, ob ein Objekt am Rand ist. Diese Anforderung müsste man also über Bedingungen implementieren, die die Position des Balls abfragen.

4.3.4. Tamagochi

Um das Tamagochi-Spiel in Scratch zu erstellen, werden zunächst alle grafischen Elemente auf die Bühne platziert. Dabei gibt es die 3 Symbole (Herz, Brokkoli und Kuchen), das Haustier, sowie ein Button, der jedoch nur sichtbar ist, wenn das Spiel vorbei ist.



Abb. 4.13. Scratch: Tamagochi

Außerdem gibt es noch die 2 Balken, deren Füllung die Werte anzeigen. Die Füllung der zwei Balken ist besonders interessant. Diese wurde nämlich durch die *Malstift*-Erweiterung für Scratch erstellt. Diese erlaubt es ein Objekt malen zu lassen, sodass es eine Spur hinter sich zieht. Für das Objekt das malt, habe ich das Bild eines Bleistifts gewählt und die Sichtbarkeit des Bleistifts ausgeschaltet. Die Balken selbst sind keine Objekte, sondern wurden einfach auf den Hintergrund gemalt. Vorher hatte ich die Balken als Objekte, doch das Problem war, dass der *Malstift* die Spur nur auf den Hintergrund malen kann und auf keine Objekte.

Nun sehen wir uns den Code des Haustiers genauer an (s. Abb. 4.14):

Die rötlichen Blöcke sind sogenannte *eigene Blöcke*. Scratch erlaubt es nämlich eigene Blöcke zu erstellen, welche aus anderen Blöcken zusammengesetzt sind. Diesen Blöcken können beliebig viele Parameter übergeben werden. So sind diese Blöcke analog zu Funktionsdefinitionen. Der einzige Nachteil ist, dass sie keinen Wert zurückgeben können. Es gibt jedoch einen *Hack* der verwendet werden kann, um das Problem der fehlenden Rückgabewerte zu lösen. So kann ein selbst definierter Block den eigentlichen Rückgabewert in eine globale Variable schreiben. Nachdem die Methode aufgerufen wurde, kann dann auf die globalen Variable zugegriffen werden, um auf den *Rückgabewert* zuzugreifen. Dies wurde mit dem benutzerdefinierten Block `clamp` so gemacht.

Der `clamp`-Block in Abb. 4.16 begrenzt den übergebenen Wert zwischen einem Minimum und Maxi-

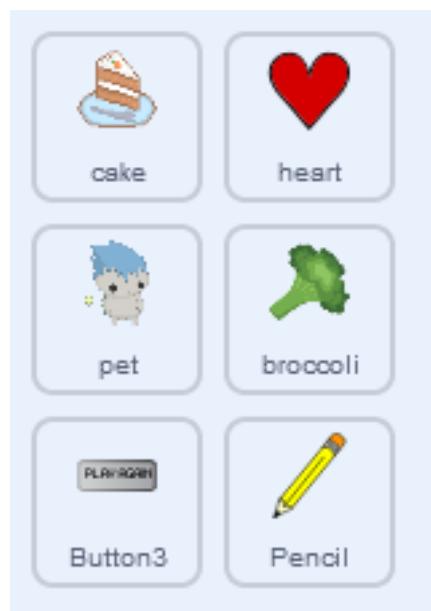


Abb. 4.15. Scratch: Tamagochi – Objekte

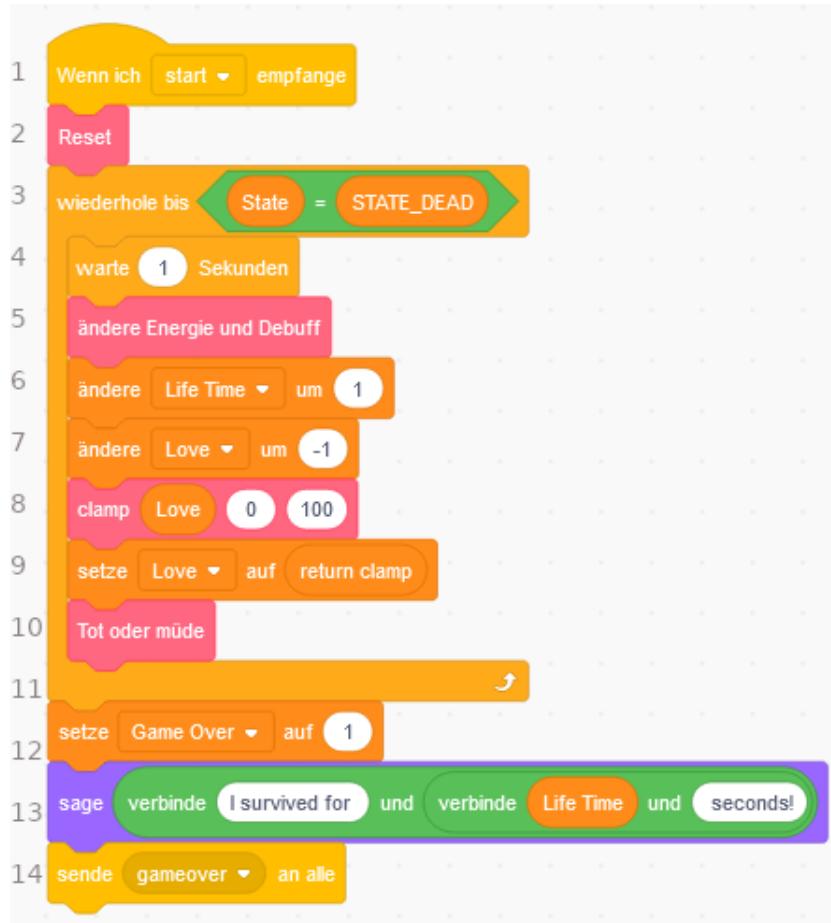


Abb. 4.14. Scratch: Tamagochi – Start-Ereignis des Haustiers

rum. Der Block wird benutzt, um die *Liebe* sowie die *Energie* des Haustiers zwischen $[0, 100]$ zu begrenzen. Die globale Variable `return clamp` wird in Abb. 4.16 gesetzt. In dem Start-Block des Haustiers in Zeile 8, wird der `clamp`-Block dann aufgerufen und die Variable `Love` des Haustiers dann auf die RückgabevARIABLE gesetzt.

Da es sich um einen *Hack* handelt, hat dieser wohl oder übel auch seine Nachteile. In Scratch wird nämlich ganz automatisch mit hoher Nebenläufigkeit programmiert. So laufen die Skripts für jedes Objekt in ihren eigenen *Thread* (vgl. [Wik20a]) und auch innerhalb eines Objekt können mehrere Skripts gleichzeitig laufen. Und wenn mehrere *Threads* auf eine gemeinsame Variable zugreifen, entstehen *Race Conditions*. Und Scratch hat keine Blöcke für Synchronisation und wechselseitigen Ausschlüssen von Threads, die diese *Race Conditions* verhindern könnten.

Jedoch kann für jede Variable angegeben werden, ob sie global in allen Objekten

Abb. 4.16. Scratch: Tamagochi – Definition des *Clamp-Blocks*

verfügbar ist, oder nur lokal in einem Objekt. Außerdem sind Blockdefinitionen sowieso nur innerhalb des Objekts verfügbar, indem sie definiert wurden sind, was natürlich den Nachteil bringt, dass der Block, wenn er auch in anderen Objekten eingesetzt werden soll, kopiert werden muss. Wenn also darauf geachtet wird die RückgabevARIABLEN lokal zum Objekt zu erstellen, kann die Gefahr von *Race Conditions* erheblich verringert werden. Außerdem ist überhaupt die Gefahr von *Race Conditions* in Scratch zu diskutieren. Die neuste Version (Scratch 3.0) wurde nämlich in Javascript implementiert und Javascript ist *Single-Threaded*. Die sogenannte *Scratch-VM* führt den blockbasierten Code aus und einen kurzen Blick in den Quellcode zeigt, dass der Code wirklich nach dem *Thread*-Modell angelegt ist (s. ??). Es sieht jedoch nicht danach aus, dass *Working Threads* verwendet wurden sind, sodass Scratch *Single-Threaded* sein müsste. Auch wenn es sich nicht um wirkliche *Threads* handelt, bleibt die Gefahr, dass ein anderes Skript die RückgabevARIABLE überschreibt, bevor sie gelesen werden konnte.

Zurück zum Code in Abb. 4.14: Ursprünglich hatte ich den Code in den Blöcken **Reset** in Zeile 2, **ändere Energie** und **Debuff** in Zeile 5 und **Tot oder müde** in Zeile 10 nicht in Blöcke ausgelagert. Für die Dokumentation habe ich diese dann doch in Blöcke ausgelagert, da das Bild des ganzen Blocks nicht auf eine Seite gepasst hat. Die Auslagerung ist auch nicht sonderlich sauber im Sinne von *Clean Code*. Wie an den Blocknamen erkennbar ist, haben sie mehrere Verantwortlichkeiten. Dies hätte gewiss besser gelöst werden können, jedoch wollte ich an der Struktur des Codes nichts mehr ändern.

An dieser Problematik zeichnet sich ein Nachteil von visuellen Programmiersprachen ab. Da sie nicht in Textform sind, ist die Dokumentation schwieriger und sie können auch nicht einfach kopiert und eingefügt werden.

Da ich das Tamagochi in Scratch nach der Implementierung in *Pyret* implementiert habe, habe ich mich beim Programm-Design sehr stark an der Implementierung in *Pyret* orientiert. So habe ich folgende Variablen angelegt:

- **State:** Der Status des Haustiers. Dieser kann einer der drei folgenden Werte annehmen:
 - STATE_NORMAL: Das Haustier ist im normalen Zustand.
 - STATE_DEBUFF: Das Haustier hat einen Debuff, welcher besondere Auswirkungen hat. Der eigentliche Debuff ist in der Variable **Debuff** gespeichert.
 - STATE_DEAD: Das Haustier ist verstorben.
- **Debuff:** Der Debuff des Haustiers. Dieser kann folgende zwei Werte annehmen:
 - DEBUFF_ASLEEP: Das Haustier schläft.
 - DEBUFF_SICK: Das Haustier ist krank.
- **Debuff Time:** Zähler, der die verbleibende Zeit des Debuffs runterzählt.
- **Love:** Die Liebe des Haustiers.
- **Energy:** Die Energie des Haustiers.
- **Life Time:** Zählt die Lebenszeit des Tiers hoch.
- **Game Over:** Gibt an, ob das Spiel vorbei ist. Da es in Scratch keine *Booleans* gibt, wird eine Zahl (0 für falsch und 1 für wahr) verwendet.

Die groß geschriebenen Variablen wie z.B. STATE_NORMAL sind Konstanten. Weil es in Scratch jedoch keine Unterscheidung zwischen Konstanten und veränderbaren Variablen gibt, sind diese als normale Scratch-Variablen definiert, die ganz zu Beginn des Spiels auf ihre konstanten Werte gesetzt werden (s. C.22 im Anhang). Der Code ist recht selbsterklärend. Angelegte Variablen wie Love, Life Time und Energy werden nach den für das Spiel aufgestellten Regeln verändert. Wichtig für das Timing ist auch der Block in Zeile 4 Warte 1 Sekunde, sodass pro Sekunde die entsprechende Werte nach den Regeln verändert werden.

Der definierte Block ändere Energie und Debugf in Abb. 4.17 ändert die Energie des Haustiers abhängig von dem Status des Haustiers. Außerdem wird in Zeile 10 die Debugf Time verringert. Wenn diese dann auf 0 ist, wird der Status wieder auf den normalen Status zurückgesetzt. Immer wenn der Status geändert wird, senden wir ein StateChanged-Ereignis an alle Objekte, sodass die Objekte auf den geänderten Status reagieren können. So ändert das Haustier beispielsweise sein Aussehen basierend auf seinen Status (s. Anhang C.24b). Bei Scratch können über Kostüme einem Objekt mehrere Aussehen verliehen werden. Über den Block wechselt Kostüm kann den zwischen den einzelnen Kostümen entweder über den Kostüm-Namen oder den Index gewechselt werden. Um Objekte zu animieren, kann ein Kostüm in einer Schleife mit Pausen dauerhaft gewechselt werden. Dies habe ich so bei der Implementierung von Flappy Bird gemacht, um den Vogel zu animieren (s. C.11c).

Jedes der drei Symbole reagiert auf 2 Ereignisse:

1. Wenn diese Figur angeklickt wird
2. das o.g. StateChanged-Ereignis

Wenn auf das Symbol geklickt wird, dann wird der gewünschte Effekt des Symbols ausgeführt, indem die Variablen des Haustiers geändert werden. Wenn sich der Status des Haustiers ändert, dann wird das Symbol je nach Status entweder aktiviert oder deaktiviert. Deaktiviert wird es z.B. wenn das Haustier schläft oder je nach Symbol — bspw. kann das Haustier kein Kuchen essen, wenn es schon krank ist. Das Aktivieren und Deaktivieren der Symbole geschieht über Kostüme. So hat jedes Symbol eine deaktivierte Variante, bei der das Symbol durchgekreuzt ist.

Die Funktionsweise des Symbol-Codes wird am Beispiel des Kuchen-Symbols gezeigt, da dieses mit der zufälligen Krankheit einen komplizierteren Effekt hat.



Abb. 4.17. Scratch: Tamagochi – Ändere Energie und Debugf

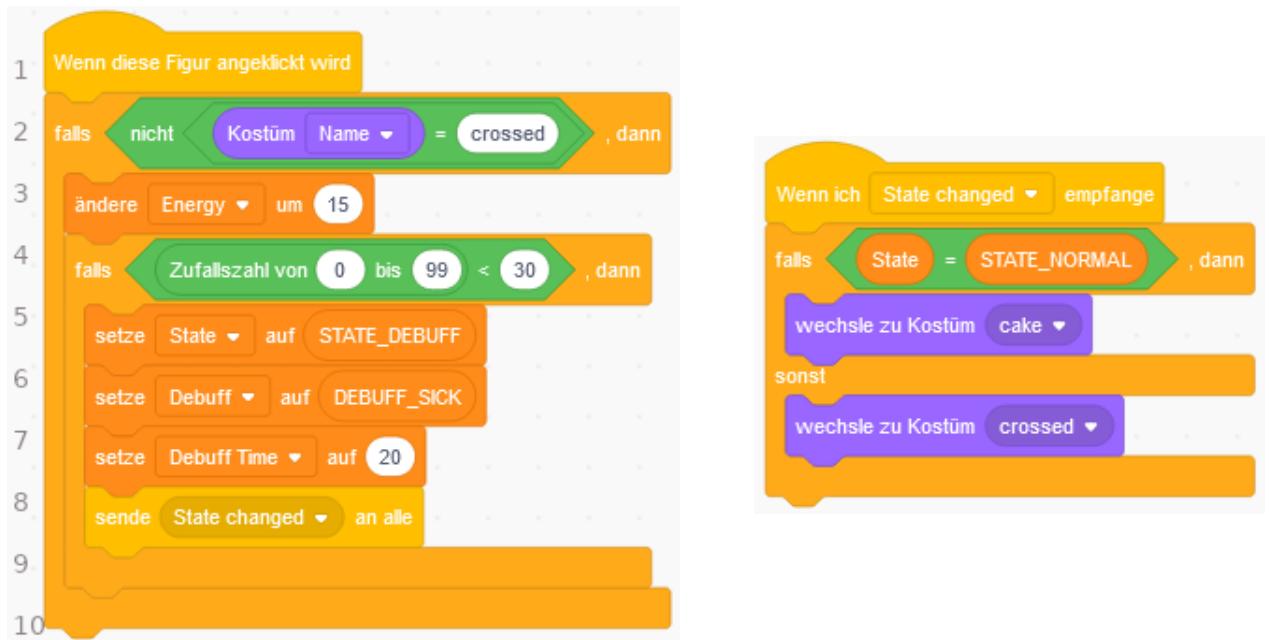


Abb. 4.18. Scratch: Tamagochi – Blöcke für das Kuchen-Symbol

Wenn der Kuchen angeklickt wird, wird in Zeile 2 zunächst überprüft, ob der Kuchen aktiviert ist. Der Kuchen ist aktiviert wenn das *Kostüm* des Kuchens nicht gleich *crossed* ist. Schließlich wird der Effekt des Kuchens angewandt. So wird die Energie um 15 erhöht und für den zufälligen Effekt des Krankwerdens von 30% wird eine Zufallszahl von 0 bis 99 verwendet, welche kleiner 30 sein muss. Wurde der krankmachende Effekt ausgelöst, wird in den Zeilen 5 bis 7, der Status des Tiers auf *STATE_DEBUFF*, der Debuff auf *DEBUFF_SICK* und die *Debuff Time* auf 20 gesetzt, da dies die Zeit ist, die die Krankheit andauert. Da der Status geändert wurde wird zum Schluss dann das *State-changed*-Ereignis an alle Objekte geschickt.

4.3.5. Bewertung

Einfachheit der Implementierung von Computerspielen

Die Computerspiele konnten mit allen Anforderungen in Scratch implementiert werden.

Der Aufwand der Implementierung war subjektiv nicht höher als der Aufwand in Pyret.

Die Implementierung von Tic-Tac-Toe war jedoch etwas aufwendiger. Es wurde so implementiert, dass wenn auf ein leeres Tic-Tac-Toe-Feld geklickt wird, das Symbol des Spielers in eine Liste mit 9 Elementen an dem Feld-Index eingetragen wird (s. Anhang C.6a). Dieser Code musste dann für jedes Feld kopiert werden und so modifiziert werden, dass das erste Feld oben links das erste Element der Liste setzt, bis zum letzten Feld unten rechts, das das neunte Element setzt. Bei jedem Klick werden die Elemente der Liste dann überprüft und geguckt, ob es einen Gewinner gibt oder es Unentschieden ist. Hierbei wird einfach für jede der 8 Gewinn-Reihen überprüft, ob diese das gleiche Spieler-Symbol enthalten.

Die Positionierung der grafischen Objekte in Scratch geschieht über den grafischen Editor. Elemente können hier nur positioniert werden. Die Größe und Rotation des Objekts kann

in den darunterliegenden Eigenschafts-Fenster geändert werden.

Es handelt sich um ein rechtshändiges Koordinatensystem mit dem Ursprung in der Mitte. Die Drehrichtung ist dabei entgegen mathematischer Konvention im Uhrzeigersinn. Der Ankerpunkt eines grafischen Objekts ist die Mitte. Dies macht es insgesamt schwer Code zu erstellen, der die Position von bestimmten Objekten abfragt. Um bei Flappy-Bird die Röhre von links nach rechts zu setzen, musste folgender Code geschrieben werden: `falls x-Position < -255, dann (s. Anhang C.13a)`. Um die -255 herauszufinden, habe ich die Röhre aus den linken Bildschirmrand bewegt und geguckt, welche Position sie dann hat.

Wenn der Ursprung in der Mitte ist, dann gibt es negative wie positive Koordinatenwerte. Die kognitive Belastung wäre geringer, wenn der Ursprung in der Ecke wäre und nur mit positiven Koordinatenwerten umgegangen werden müsste.

Es gibt keine GUI-Bibliothek in Scratch. GUI-Elemente wie z.B. Buttons können jedoch über Objekte mit Animationen und Kostümen erstellt werden.

In Scratch können Klänge und Musik abgespielt werden. Hierfür gibt es eine eigene Blockkategorie mit dem Namen „Klang“. Es gibt Blöcke zum Abspielen und Stoppen von Klängen und zum Modifizieren der Lautstärke und der Höhe von Klängen.

Die Animation von Sprites muss selbst implementiert werden und kann über das durchgängige Wechseln von Kostümen in einer Schleife implementiert werden (s. Anhang C.11c).

Die Abstraktion vom *Game Loop* ist nicht ganz so hoch. Es muss sich nicht darum gekümmert werden Elemente zu zeichnen, sie müssen jedoch aktualisiert werden. Der Update-Loop muss selbst implementiert werden. Wenn z.B. ein Objekt sich von links nach rechts bewegen soll, wird eine Schleife benötigt, in der die die x-Position des Objekts erhöht wird.

Eingaben wie Klicks können über die Ereignis-Blöcke erfasst werden. Hier bietet Scratch eine sehr hohe Abstraktion.

Erlernbarkeit nach Konstruktionismus

Es können Erweiterungen für Scratch über Javascript geschrieben werden (vgl. [Das+15]).

Es gibt eine offizielle *Scratch-Extension-Library* mit folgenden Erweiterungen:

- Musik
- Malstift
- Video-Erfassung
- Text zu Sprache
- Übersetzen
- Makey Makey (Ein Elektronik-Kit, mit dem alles zu einer Taste gemacht werden kann)
- *micro:bit* (ein pädagogischer Microcontroller)
- LEGO Mindstorms EV3
- LEGO BOOST
- LEGO Education WeDO 2.0

- Go Direct Force & Acceleration (Kraft- und Beschleunigungssensoren)

Zudem gibt es noch die experimentellen Erweiterungen, wie *Arduino*-Integration, ein *Sound-Synthesizer* und viele mehr (s. [Scr21b]).

Es können viele pädagogische Spielzeuge mit Scratch verbunden werden. Besonders die Integration von physischen *Gadgets* ist besonders hoch. Der Scratch-Editor erlaubt es nicht nur Grafiken selbst zu erstellen und im Projekt zu verwenden, sondern hat auch einen integrierten Sound-Editor.

Turtle-Grafiken können über die *Malstift*-Erweiterung erstellt werden.

Die Umgebung von Scratch ist selbstoffenbarend. Da im Scratch-Editor auf der linken Seite die unterschiedlichen Blöcke angezeigt werden, kann mit diesen einfach herumexperimentiert werden. Zusätzlich wird, wenn ein Block aufgehoben wird, visuelles Feedback gegeben, wo dieser Block eingefügt werden kann (vgl. [Mal+10]) und das Programm kann wie ein Puzzle zusammengesetzt werden. Dies führt dazu, dass Scratch sehr gut ohne weitere Hilfe selbstständig entdeckt werden kann.

Als Web-Editor ist die Umgebung ziemlich sicher. Projekte werden auf der Plattform gespeichert, können aber auch heruntergeladen und in den Desktop-Editor importiert werden.

Scratch ist dynamisch typisiert und um Scratch zu lernen, muss sich nicht mit Datentypen auseinandergesetzt werden.

Inkrementelle Einführung

Das „Hallo Nutzer“-Programm in Scratch sieht folgendermaßen aus:



Abb. 4.19. Scratch: „Hallo Nutzer“

Für Beginner, die gerade erst in Scratch einsteigen, denke ich, dass das Erstellen von „Hallo Nutzer“ ohne Anleitung eher schwer ist. Es werden Blöcke aus vier verschiedenen Bereichen benötigt, welche erst einmal gefunden werden müssen. Besonders die Verschachtelung des *verbinde*-Blocks, um „Hallo“ mit dem Namen und einem Ausrufezeichen zu verbinden, dürfte für Beginner etwas schwieriger sein.

Wichtig für Beginner sind die sieben Blockarten und wie sie miteinander verbunden werden. Das Programm „Hallo Nutzer“ besteht aus drei Blockarten. Es können also Programme geschrieben werden, die eine kleine Teilmenge dieser Blockarten verwenden. Da es ziemlich viele Blöcke in Scratch gibt, können diese dann nach und nach eingeführt oder erkundet werden.

Intuitive Syntax und Natürlichsprachlichkeit

Scratch-Scripts bestehen aus Blöcken, die aneinander gefügt werden und Kontrollstrukturen und Ausdrücke repräsentieren. Das *Drag-and-Drop*-System verhindert ein Verbindung von Blöcken, die keinen Sinn macht. Das macht die Syntax in Scratch sehr intuitiv, da ein Programm wie ein Puzzle zusammengefügt werden kann.

Scratch ist in über 60 verschiedene Sprachen übersetzt (vgl. [Scr21a]). Dabei sind die Blöcke alle einzeln übersetzt wurden.

Im folgenden wird auf die Studie eingegangen, welche über Scratch durchgeführt wurde und Evidenz dafür bieten, dass Programmierkonzepte in der Muttersprache schneller erlernt werden können (s. [DH17]).

In der Studie wurden 90859 Projekte analysiert und dabei das kumulative Repertoire an Blöcken von Benutzern analysiert. Wenn Benutzer in ihrem nächsten Projekt mehr Blöcke verwenden als im vorherigen Projekt, steigt das kumulative Repertoire um die Anzahl der neu verwendeten Blöcke (vgl. [DH17]). Es wird davon ausgegangen, dass die kumulative Repertoire höher ist, wenn Scratch schneller erlernt wird.

Schließlich wird zwischen den Projekten unterschieden, in welchen der Benutzer die Muttersprache als *Interface*-Sprache verwendet hat oder eine andere Sprache.

Die Ermittlung der Muttersprache des Benutzers erfolgte daraus, welches Land die Benutzer bei der Anmeldung angegeben haben und welche Sprachen in diesem Land am häufigsten gesprochen wurde. Bei der Erstellung von Projekten wird die Sprache gespeichert, in dem das Projekt angelegt wurde. So konnte in der Studie die *Interface*-Sprache ermittelt werden.

Das Ergebnis der Studie war, dass ein positiver Effekt von Lokalisierung auf die Rate, in der das kumulative Blockrepertoire ansteigt, festgestellt werden konnte.

Die Forscher geben an, dass es sich hierbei jedoch um eine Korrelation und keine Ursache handelt und es einige Effekte gibt, die nicht berücksichtigt wurden, wie z.B. dass die dominierende Sprache in Scratch Englisch ist.

Feature Uniformity und Orthogonalität

Ein Designziel von Scratch war es von vorne rein, die Menge der Blöcke zu reduzieren, welches damit begründet wird, dass das Hinzufügen von einem Block größere Kosten hat, als bei einer textbasierten Programmiersprache, da dieses Raum in der Block-Palette einnimmt (vgl. [Mal+10]).

Es wurde sorgfältig darauf geachtet, wenige neue Blöcke zu erstellen und Blöcke zusammenzufassen, die in einem Block mit einem *Drop-Down*-Menü zusammengefasst werden können (vgl. [Mal+10]).

Zusätzlich können neue Blöcke durch Erweiterungen hinzugefügt werden, sodass Scratch bei Bedarf um neue Funktionen erweitert werden kann.

Es gibt jedoch z.B. drei verschiedene *Schleifen*-Blöcke:

1. `wiederhole <n> mal`
2. `wiederhole fortlaufend`
3. `wiederhole bis`

Und für jeden der Rechenoperatoren gibt es auch einen separaten Block. Für mathematische Funktionen wie Wurzel oder Sinus gibt es allerdings nur einen Block, wo die Funktion über ein *Drop-Down*-Menü ausgewählt werden kann.

Es scheint so, dass zugunsten der *Usability* sich oft für mehrere Blöcke entschieden wurde.

Außerdem ist ein Maß für die *Feature Uniformity* nicht die Anzahl der Blöcke, sondern die Art der Blöcke. Und hier beschränkt sich Scratch auf die sieben Blockarten (s. 4.11). Zudem kommt insgesamt das Objektmodell von Scratch hinzu. In Scratch kommunizieren Objekte miteinander, indem sie asynchrone Nachrichten senden und Scratch hat somit eine hohe Nebenläufigkeit. Außerdem sind Variablen und Listen sowie benutzerdefinierte Blöcke weitere Funktionen von Scratch.

Insgesamt fällt es schwer *Feature Uniformity* für Scratch zu bewerten, da es Blöcke gibt, die neue Sprachfeatures hinzufügen und andere Blöcke, die nur eine neue Aktion hinzufügen.

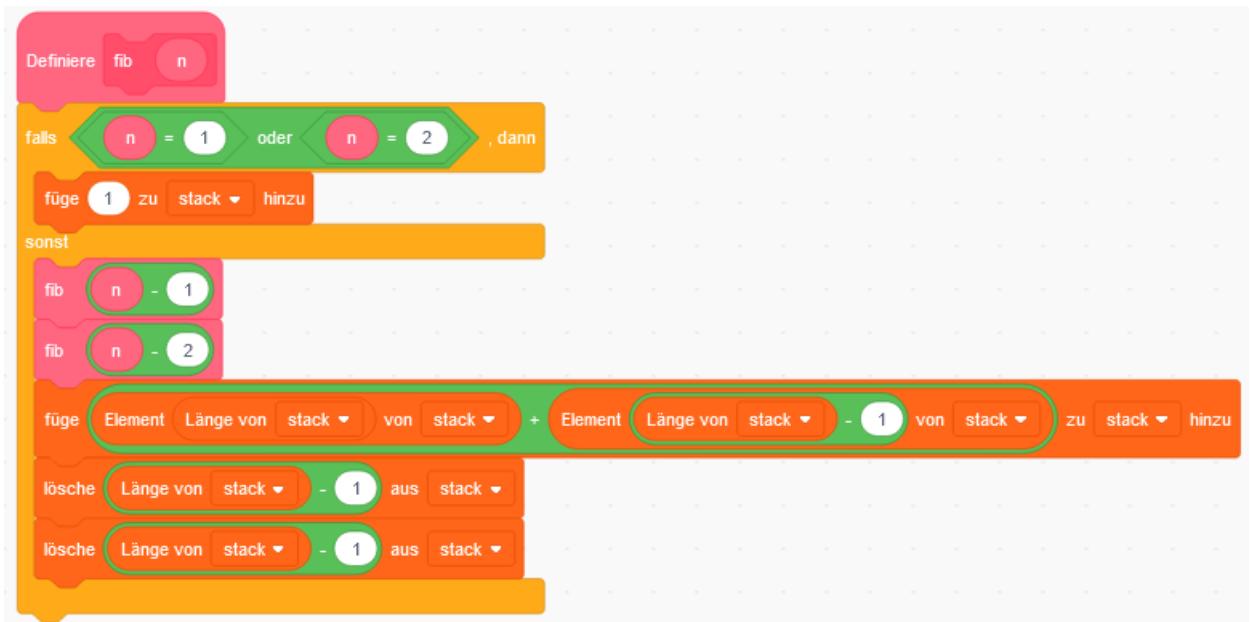
Da es eine blockbasierte Programmiersprache ist in dem die Blöcke aneinander gefügt werden müssen, bleibt kein Raum für inkonsistente Syntax. Scratch's Syntax ist so sehr konsistent.

Unterstützung der wichtigsten Programmierkonzepte & Datenstrukturen

Scratch hat Schleifen- und Bedingungsblöcke und erfüllt somit die Anforderungen der strukturierten Programmierung

Die prozedurale Programmierung unterstützt Scratch über die benutzerdefinierten Blöcke nicht ganz, da keine Rückgabetypen unterstützt werden. Zudem gibt es in Scratch auch nur globale Variablen und keine lokale Variablen. Rekursion wird unterstützt, ist aber weniger sinnvoll, da es keinen Rückgabetypen gibt.

Eine rekursive Implementierung der *Fibonacci*-Funktion ist in Scratch möglich, jedoch sehr mühsam und nur unter Verwendung eines *Hacks*:

Abb. 4.20. Scratch: rekursive *Fibonacci*-Funktion

Hierbei wird eine Liste mit dem Namen `stack` angelegt, welche als künstlicher *Stack* dient. Da Methodenblöcke keinen Rückgabetypen haben, wird das Ergebnis des rekursiven *Fibonacci*-Aufrufs auf den *Stack* geschoben. Der Aufrufer macht die zwei rekursiven Aufrufe und kann dann die zwei obersten Elemente des *Stacks* zusammenaddieren. Nach der Addition werden diese zwei Elemente als Zwischenergebnisse nicht mehr benötigt und werden aus dem *Stack* entfernt.

Die Implementierung war so schwer, dass ich sie zunächst in Javascript gemacht und dann auf Scratch übertragen habe:

```

1 // globaler Stack
2 const stack = new Array();
3
4 function stackFib(n) {
5     if (n === 1 || n === 2) {
6         stack.push(1);
7     } else {
8         stackFib(n - 1);
9         stackFib(n - 2);
10        stack.push(stack.pop() + stack.pop());
11    }
12 }
13
14
15 function fib(n) {
16     stack.length = 0;
17     stackFib(n);
18     return stack.pop();
19 }
20
21 // Test

```

```
22 console.log(fib(5) === 8);
```

Listing 4.36. äquivalenter Javascript-Code zur Implementierung der *Fibonacci*-Funktion in Scratch

Dabei bildet die Funktion `stackFib` den oberen Scratch-Skript ab. Die Funktion `fib` dient nur als Schnittstelle und leert den *Stack* vorsichtshalber vorher und gibt dann das Element zurück. Genauso wird im Scratch vor dem Ausführen des benutzerdefinierten Blocks, die Elemente des Stacks gelöscht (s. Anhang C.1a).

Bei den Datenstrukturen unterstützt Scratch nur die Liste. Diese kann zwar auch als *Stack* oder *Queue* verwendet werden, was allerdings aufwendig ist, da die Funktionalitäten über Listen-Operationen implementiert werden müssen.

Programmierumgebung (IDE)

Die IDE in Scratch ist ziemlich übersichtlich und einfach gehalten und lädt Benutzer zum Programmieren ein.

Bei Scratch handelt es sich um eine *Live-Coding*-Umgebung, was durch Maloney u. a. bestätigt wird:

„A key feature of Scratch is that it is always live [...]. There is no compilation step or edit/run mode distinction. Users can click on a command or program fragment at any time to see what it does. In fact, they can even change parameters or add blocks to a script while it is running. By eliminating potentially jarring mode switches and compilation pauses, Scratch helps users stay engaged in testing, debugging, and improving their projects.“ [Mal+10]

Scratch bietet somit eine hohe Interaktivität und Debugging-Kapazität. Des weiteren können die Werte angelegter Variablen in Scratch auf der Oberfläche angezeigt werden, was eine weitere Debugging-Möglichkeit bietet.

Es kann jedoch nicht Schritt-Für-Schritt debuggt werden. Das Programm kann zwar angehalten werden und die Blöcke, die momentan laufen werden, angezeigt werden, doch es fehlt die Feingranularität eines Schritt-Für-Schritt-Debuggers.

Als Web-IDE ist keine Installation notwendig.

Projekte können in Scratch mit einem einfachen Button-Klick veröffentlicht und geteilt werden. Zudem können Projekte *geremixt* werden, also vom Ersteller übernommen und dann abgeändert werden.

Außerdem gibt es *Studios*, in denen Benutzer mehrere Projekte innerhalb einer Gruppe teilen können. Dabei haben die meisten Studios ein spezifisches Thema (vgl. [Wik20a]).

Die Scratch-Homepage hat einen *Showcase*, in dem interessante oder kreative Projekte und Studios vorgestellt werden.

Zuletzt gibt es die Möglichkeit Code-Abschnitte in den *Rucksack* zu importieren, welche dann über das Projekt hinaus in anderen Projekten verwendet werden können.

Wenn man die modernen Funktionen einer textbasierten IDE auf Scratch überträgt, unterstützt Scratch *Syntax-Highlighting* (die Blöcke werden in unterschiedlichen Farben und

Formen angezeigt) und einen Code-Formatierer, da es eine Funktion gibt, die die Blöcke übersichtlich aneinanderreihrt.

Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplin (*Clean Code*)

Scratch-Skripts werden immer pro Objekt angelegt. Jedes Objekt hat somit seinen eigenen Code, welches nur das Objekt betrifft und kann Nachrichten an andere Objekte senden. Wenn ein Objekt das gleiche tun soll, wie ein anderes Objekt, bleibt nichts anderes übrig, als den Code vom einem Objekt zum anderen Objekt zu kopieren. So wurde bei der Implementierung von Tic-Tac-Toe für jedes Feld der gleiche Code kopiert und leicht abgeändert, sodass jedes Feld einen anderen Index verwendet, um das Spieler-Symbol in die Liste einzufügen (s. Anhang C.6a).

Es können zwar eigene Methodenblöcke definiert werden, die sind jedoch dann nur in dem Objekt verfügbar, wo sie definiert wurden sind.

Zudem kann ein Objekt eine Kopie von sich selbst erstellen. Dabei bietet Scratch jedoch keine Möglichkeit, bestimmte Parameter der Kopie abzuändern.

Dies macht Scratch-Skripts von der *Clean-Code*-Sicht her ziemlich unordentlich. Da Code nicht geteilt werden kann, gibt es eine große Redundanz von Code. Zudem kann jedes Objekt anderen Objekten beliebige Nachrichten asynchron schicken (es geht auch synchron, sodass das aufrufende Objekt wartet, bis das Objekt die Aktion abgeschlossen hat) und der Code kann so ziemlich komplex und schwer wartbar werden, da es keine Hierarchie von Objekten und Aufrufen gibt.

Zum *Information Hiding* bietet Scratch an, Variablen zu erstellen, auf welche nur innerhalb eines Objekts zugegriffen werden kann. Es gibt somit objektglobale und totalglobale Variablen. Dies ist immerhin etwas, jedoch zu wenig, um bei einem komplexen Projekt nicht die Kontrolle zu verlieren.

In Scratch gibt es zwar Kommentare in Form von grafischen Klebezetteln, diese können jedoch beliebig verschoben werden und nicht an einem bestimmten Block angefügt werden. Wenn ein Block bewegt wird und ein Kommentar danebensteht, muss also daran gedacht werden, den Kommentar ebenso zu bewegen.

Scratch ist dynamisch typisiert und verwendet eine *Fail-Soft*-Strategie. Dies wird von Malone folgendermaßen beschrieben:

„Scratch also strives to eliminate run time errors by making all blocks be failsoft. Rather than failing with an error message, every block attempts to do something sensible even when presented with out-of-range inputs.“ [Mal+10]

Wenn also eine ungültige Operation ausgeführt wird, dann gibt Scratch keinen Fehler zurück, sondern macht einfach irgendetwas, „*was Sinn macht*“.

Zum Beispiel setzt der zweite Block hier die Variable einfach auf 0.



Abb. 4.21. Scratch: Ungültige Operation, die jedoch *soft failed*

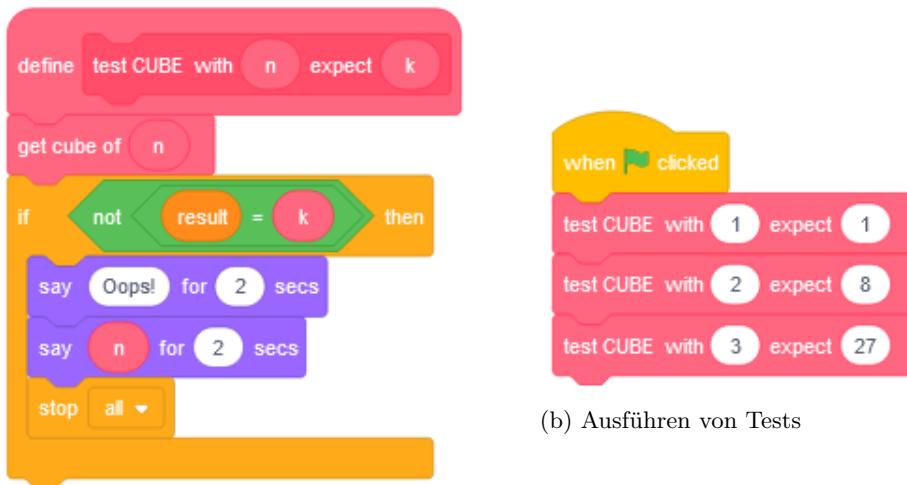
Oder bei der Addition von einer Zahl und einem Text, wird einfach die Zahl zurückgeben. Ein anders Beispiel ist die Division durch 0, welche in Scratch Unendlich zurückgibt.

Diese Entscheidung macht Scratch sehr fehleranfällig während der Laufzeit. Es wird die große Interaktivität von Scratch erreicht, indem der Aspekt der Sicherheit eines korrekt ausgeführten Programms komplett ignoriert wird.

Dadurch, dass Objekte abgetrennt voneinander sind und über unparametrisierten Nachrichten kommunizieren, kann argumentiert werden, dass Fehlerzustände nicht so weit propagiert werden können. Die Verwendung von globalen Variablen widerspricht diese Aussage jedoch wieder.

Durch diese *Fail-Soft*-Strategie gibt es in Scratch keine Ausnahmen, welche behandelt werden müssten.

Scratch hat keine Unterstützung für Unit-Tests, es können jedoch Blöcke so zusammengefügt werden, sodass sie Unit-Tests repräsentieren:



(a) Test-Block für die CUBE-Funktion

Abb. 4.22. Scratch: Unit-Tests

Scratch verfügt nicht über einen *Style-Guide*, der angibt wie Scratch-Skripts geschrieben werden sollen.

Fehlermeldungen

In Scratch gibt es keine Fehlermeldungen. Es gibt zum einem keine Syntax-Fehler, da Blöcke nur so zusammepassen wie es Sinn macht. Maloney u. a. beschreiben dies über die LEGO-Analogie:

„When people play with LEGO® bricks, they do not encounter error messages. Parts stick together only in certain ways, and it is easier to get things right than wrong. The brick shapes suggest what is possible, and experimentation and experience teaches what works. Similarly, Scratch has no error messages. Syntax errors are eliminated because, like LEGO® bricks, blocks fit together only in ways that make sense.“ [Mal+10]

Zudem gibt es auch keine Laufzeitfehler, da wie oben beschrieben eine *Fail-Soft*-Strategie verwendet wird, um eine hohe Interaktivität zu gewährleisten.

Dadurch schafft es Scratch Fehlermeldungen völlig zu eliminieren, was ein großer Vorteil für Programmier-Anfänger ist, da dadurch die intrinsische kognitive Belastung reduziert wird und Scratch so einfacher zu lernen ist.

Ein hohes Abstraktionsniveau für Datentypen

Die *Scratch VM* verwendet als Zahlentyp einfach Javascript-Zahlen (64-Bit-Fließkommazahlen), was durch folgenden Quellcode der *Scratch-VM* belegt wird, welcher ein Wert in Scratch in eine Zahl *castet*:

```
static toNumber (value) {
    // If value is already a number we don't need to coerce it with
    // Number().
    if (typeof value === 'number') {
        // Scratch treats NaN as 0, when needed as a number.
        // E.g., 0 + NaN -> 0.
        if (Number.isNaN(value)) {
            return 0;
        }
        return value;
    }
    const n = Number(value);
    if (Number.isNaN(n)) {
        // Scratch treats NaN as 0, when needed as a number.
        // E.g., 0 + NaN -> 0.
        return 0;
    }
    return n;
}
```

Listing 4.37. Quellcode von Scratch-VM: <https://github.com/LLK/scratch-vm/blob/develop/src/util/cast.js>

Ebenso wird für den String-Datentypen, Javascript-Strings verwendet, welche *Unicode* unterstützen. In der Scratch-Umgebung können *Unicode*-Strings eingegeben und ausgegeben werden.

Anwendungsbereiche und Interoperabilität mit Technologien und anderen Programmiersprachen

Für die Anwendungsbereiche wird auf den vorherigen Abschnitt **Erlernbarkeit nach Konstruktionismus** verwiesen.

Scratch kann über die Hilfsanwendung *Scratch Link* auf Hardware zugreifen. Dabei geschieht die Kommunikation vom Web-Browser zu *Scratch Link* über *Web Sockets* (vgl. [Tea21]).

Scratch kann außerdem über *Third-Party-Tools* wie dem *Sulfurous-Player*, der Scratch-Projekte nach Javascript kompiliert auf Smartphones innerhalb einer *Web-View* ausgeführt werden (s. [Pro20]).

Zudem gibt es das *Leopard*-Projekt, dass es erlaubt Scratch-Projekte in Javascript zu kompilieren und Scratch-Projekte auf Webseiten zu verwenden (s. [Sul16]).

Dokumentation & Community Support

Scratch hat eine große Community. So groß, dass Scratch 2020 Platz 20 im TIOBE-Index erreicht hat (vgl. [TIO20]). Es gibt über 50 Millionen Projekte, die in Scratch erstellt wurden sind und jeden Monat kommen eine Millionen neue Projekte hinzu (vgl. ebd.).

Paul Jansen, der CEO von TIOBE, erklärt den Rang von Scratch folgendermaßen und beschreibt wie Scratch das Rennen der *Teaching Languages* gegen *Alice* gewonnen hat:

„*Since computers are getting more and more an integral part of life, it is actually quite logical that languages to teach children programming are getting popular. Some years ago there was competition between Scratch and Alice which language would become the new "Logo" programming language of the modern ages. Alice is now at position #90 of the TIOBE index so it seems clear who has won. Possible reasons why this happened is that Scratch is easier to learn (a critical success factor in this field) and Scratch is sponsored by companies such as Google and Intel.*“ [TIO20]

Ein großer Bestandteil von Scratch ist die Community. Projekte können kommentiert und geteilt werden. Zudem werden kurierte Projekte auf der *Homepage* ausgestellt. Obendrein finden in der Community Wettbewerbe statt, in denen ein Projekt innerhalb einer kurzen Zeit zu einem bestimmten Thema erstellt werden muss.

Offiziell gibt es das *Scratch-Forum* in dem sich über Scratch ausgetauscht werden kann, es gibt allerdings auch noch zahlreiche andere Gruppen und Foren auf den verschiedensten *Social Media* Kanälen.

Zur Dokumentation von Scratch gibt es ein großes *Scratch-Wiki*, das von der Community verwaltet wird und in 22 Sprachen übersetzt ist (s. [Wik20a]).

Das Problem bei der Dokumentation von Scratch-Skripts ist, dass Scratch-Skripts nicht einfach *Copy-Pasted* werden können, da es eine visuelle Programmiersprache ist. In dieser Arbeit wurden für die Dokumentation Bildschirmaufnahmen gemacht. Das Scratch-Wiki

hat jedoch ein Plugin mit dem Namen *ScratchBlocks4*, welches eine textuelle Beschreibung von Scratch-Blöcken (mit fester Syntax wie eine Programmiersprache) in `<scratchblocks>...</scratchblocks>`-Tags in visuelle Scratch-Blöcke umwandelt (s. [Con21]).

Durch die Popularität von Scratch gibt es zahlreiche Webseiten, Tutorials und Bücher, z.B. das Buch „*Scratch Programming Playground: Learn to Program by Making Cool Games*“ [Swe16] oder „*The everything kids' Scratch coding book : learn to code and create your own cool games*“ [Ruk18], welche sich an die jüngere Generation (8-13) richten.

Zudem gibt es auch einige deutschsprachige Bücher, wie „*Der kleine Hacker: Programmieren für Einsteiger. Mit Scratch schnell und effektiv programmieren lernen.*“ oder „*Programmieren lernen mit der Maus: Der Start in die Programmierung mit Scratch*“ (vgl. [Joh19]).

4.4. ToonTalk

4.4.1. Kurze Einführung in ToonTalk

ToonTalk ist eine animierte Programmiersprache und Lernumgebung, deren Design und Implementierung 1992 (in dem selben Jahr als Microsoft Windows 3.1 veröffentlichte) startete und 1995 das erste Mal veröffentlicht wurde. Nach drei Hauptreleases stoppte die Entwicklung dann 2009 und ToonTalk wurde Open-Source und kostenlos verfügbar gemacht. 2014 startete dann mit *ToonTalk Reborn* eine Neuentwicklung. Inspiriert von webbasierten Projekten wie NetLogo oder dem Modelling4AllProject, welche darauf abzielten agentenbasierte Modellierung über das Web für alle leicht erlernbar und verfügbar zu machen, sollte auch ToonTalk für das Web neu implementiert werden (vgl. [Kah14]). Der Name ToonTalk stammt daher, dass der Nutzer durch (Car)Toons mit dem Computer spricht (Talk).

In meiner Recherche habe ich mir zuerst das neuere *ToonTalk Reborn* angesehen und versucht, das erste Spiel Tic-Tac-Toe darin zu implementieren. Es ist mir jedoch sehr schwer gefallen, da mir das Programmierparadigma *Concurrent Constraint Logic Programming* hinter ToonTalk noch nicht bekannt war und die *Usability* der Webseite auch ziemlich frustrierend war.

Ich habe dann eine E-Mail (s. Anhang A.1) an Ken Kahn (den Designer und Entwickler von ToonTalk) gesendet. Er antwortete mir dann mit einem Rat mir das originale ToonTalk (das nur auf Windows läuft) anzusehen. Dies habe ich dann auch gemacht. Das praktische an dem originalen ToonTalk sind die Puzzle-Games in welchen die Elemente von ToonTalk nach und nach eingeführt werden. Trotzdem habe ich mir mit ToonTalk immer noch sehr schwer getan. Die Implementierung von *Bouncing Ball* in ToonTalk war noch recht simpel. Besonders weil es in ToonTalk eine Anleitung in Video-Form gibt, in der zwei Kinder Ping-Pong programmieren und ich den *Bouncing Ball* dann einfach nachprogrammieren konnte.

Doch ich habe zwei Tage für die Implementierung von Tic-Tac-Toe gebraucht und es war wahrlich *Hard Fun*. Aus Zeitmangel habe ich die zwei anderen Spiele dann nicht mehr programmiert. Meiner Erfahrung nach würde ich sagen, dass die Implementierung von

Flappy Bird und *Tamagochi* in ToonTalk durchaus möglich ist.

Meine Schwierigkeiten mit ToonTalk deuten nicht darauf hin, dass es sich um eine schlechte *Teaching Language* handelt. Ich fand es schwer, weil mir das Programmierparadigma und die Art und Weise der Programmierung noch unbekannt war und die kognitive Belastung für mich dementsprechend sehr hoch war. Nachdem ich mich eine Weile mit ToonTalk beschäftigt habe, kann ich sagen das mir die Ideen und Konzepte von ToonTalk sehr gut gefallen. Natürlich versuche ich im Weiteren eine objektive Sichtweise zu bewahren.

Kahn beschreibt die Idee und die Motivation hinter ToonTalk folgendermaßen:

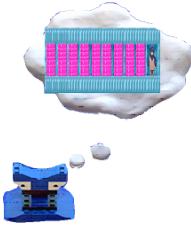
„ToonTalk started with the idea that perhaps animation and computer game technology might make programming easier to learn and do (and be more fun). Instead of typing textual programs into a computer, or even using a mouse to construct pictorial programs, ToonTalk allows real, advanced programming to be done from inside a virtual animated interactive world.“ [Kah04b].

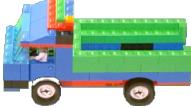
Es geht also schon um die konstruktionistische Idee, dass Lernen Spaß machen soll. Zugleich betont er, dass fortgeschrittene Programmierung („real, advanced programming“) mit ToonTalk möglich ist.

4.4.2. Beschreibung der Programmierumgebung

In ToonTalk werden abstrakte Konzepte der Informatik auf animierte Objekte in einer virtuellen Spielwelt abgebildet.

| Abstraktion | Konkretisierung in ToonTalk | Grafik |
|-------------------------------|---|---|
| Programm | Die ganze Stadt entspricht einem Programm in ToonTalk. |  |
| Prozess | In der Stadt können mehrere Häuser stehen. Jedes Haus entspricht einem Prozess. Außerdem gibt es da auch noch die Rückseite von Bildern, welche auch eigene Prozesse sind. |  |
| Programmfragment, Prozedur | Roboter entsprechen Programmfragmenten. Roboter werden programmiert, indem man ihnen Boxen gibt und man ihnen dann vorführt, was sie machen sollen. Diese Technik ist als <i>Programming by Demonstration</i> (PbD) bekannt. Nachdem man sie trainiert hat, kann man ihnen dann eine Box geben, auf welche sie dann die Operationen ausführen, die man ihnen gezeigt hat. |  |

| | | |
|---|--|---|
| Guards oder Vorbedingungen | <p>In den Gedankenblasen eines Roboters steht, wann dieser ein Programm ausführt. Wenn einem Roboter eine Box gegeben wird, dann führt dieser nämlich wiederholt die trainierten Operationen aus, bis die Box die Bedingungen in seiner Gedankenblase nicht mehr erfüllt. Die Bedingungen können über <i>Dusty</i> den Staubsauger bearbeitet werden.</p> <p>Mehrere Roboter können zu einem Team aus Robotern zusammengeführt werden. Die Roboter stehen dann hintereinander in einer Reihe. Der Box wird dann von vorne nach hinten immer wieder durchgereicht, bis die Box die Bedingungen eines Roboters erfüllt und dieser die dann bearbeitet.</p> |  |
| Container, Liste, Array, Tupel | Boxen können Dinge enthalten. Boxen können aneinander gefügt und beliebig viele Löcher haben und verschachtelt werden. |  |
| Datei-System, Persistierung, Dictionary | Notizbücher können verwendet werden um Objekte zu speichern. Jeder Benutzer in ToonTalk hat ein persönliches Notizbuch. Dort sind standardmäßig schon Sachen wie Bilder gespeichert. Da Notizbücher Notizbücher enthalten können, sind sie auch analog zu einer Ordner-Struktur. Notizbücher können auch als Wörterbuch zum Nachschlagen verwendet werden. Wenn die linke Seite eines Notizblatts mit Text beschriftet ist und man dann einen Text auf das Notizbuch fallen lässt, dann wird automatisch die Seite aufgeschlagen, wo der Text der linken Seite gleich ist. |  |
| Zahlen | Zahlen haben in ToonTalk eine unbegrenzte Genauigkeit. Wenn man eine Zahl auf eine andere Zahl fallen lässt, dann wird diese addiert. Man kann jedoch auch andere Operatoren wie +, -, *, /, und = über die Tastatur eingeben und verwenden. |  |
| Text | Text in Toontalk kann über die Tastatur eingegeben werden, wenn ein Text-Feld aufgenommen wurden ist. Wenn man Text auf anderen Text fallen lässt, dann wird der Text zusammengefügt. |  |

| | | |
|-------------------------|--|---|
| Bild, Sprite, Animation | Bilder in ToonTalk können auch animiert sein. Bilder haben eine Rückseite. Mit der 'F'-Taste können Bilder umgedreht (geflippt) werden. Wenn Bilder umgedreht werden, erscheint ein Notizblock mit Fernbedienungen für das Bild. Über Fernbedienungen können Eigenschaften des Bilds gelesen und verändert werden. Beispielsweise kann die Position und die Größe gesetzt oder ob das Bild ein anderes Objekt berührt abgefragt werden. In Toontalk gibt es schon einige Bilder, die verwendet werden können. Es können jedoch auch neue Bilder importiert werden. Wenn ein Bild auf ein Bild fallen gelassen wird, dann wird das Bild in das untere Bild eingefügt. Die Teile, aus die ein Bild besteht, kann auch über die Fernbedienung gelesen und verändert werden. |  |
| Vergleich | Mit der Waage können Buchstaben und Zahlen miteinander verglichen werden. Zur Verwendung muss sie in eine Box platziert werden. Links und Rechts von der Box werden dann weitere Boxen angefügt, in die die Elemente reinkommen, die verglichen werden sollen. Es können Zahlen und Text miteinander verglichen werden. Wenn das linke Element größer ist, dann ist die linke Seite unten. Wenn das rechte Element größer ist, die rechte Seite und sonst ist sie ausgeglichen. Sie kann dann als Bedingung für Roboter verwendet werden. |  |
| Prozess-Spawning | Über den Truck kann ein neuer Prozess gespawnt werden. Wenn einem Truck ein Roboter und eine Box gegeben wird, dann fährt der Truck mit dem Roboter weg und baut ein neues Haus (Prozess), indem der Roboter dann diese Box bearbeitet. |  |
| Prozess-Terminierung | Wenn eine Bombe gezündet wird, dann wird das Haus zerstört und der Prozess beendet. |  |

| | | |
|------------------------------------|--|---|
| Message-Sending (Channel) | Einem Vogel können Nachrichten übergeben werden, welche er dann zu seinem Nest bringt. Ein Vogel ist also immer mit einem Nest assoziiert. Wenn ein Nest erstellt wird, dann schlüpft daraus automatisch ein Vogel, der Nachrichten zum Nest bringen kann. Ein Nest kann auch kopiert werden, sodass nicht nur <i>Unicasts</i> sondern auch <i>Multicasts</i> möglich sind. Ein Nest kann sich in einem anderen Haus befinden. Und es gibt sogar Langdistanzvögel, welche über das Netzwerk zu anderen Nestern auf anderen Computern fliegen kann. |  |
| Message-Receiving (Channel), Queue | Der Vogel bringt Nachrichten in ein Nest, welche dann von Robotern verarbeitet werden können. Neue Nachrichten werden immer unter die Nachrichten gelegt, welche bereits im Nest sind. Es handelt sich somit um eine Warteschlange mit dem FIFO-Prinzip. |  |

Tabelle 4.2. ToonTalk: Zuordnung von Computer-Abstraktionen auf Objekte

Der Spieler steuert eine Programmier-Persona, die mit verschiedenen Werkzeugen Programme erstellt, ausführt, debuggt und verändert. Der Spieler kann sich frei in der Stadt bewegen. Er kann auch in den Helikopter steigen und über die Stadt fliegen, um die Stadt aus der Vogel-Perspektive zu sehen. Wenn er unten auf dem Boden ist, kann er in Häuser reingehen. Um dann Programme zu erstellen, kann der Spieler sich dann mit der linken Maustaste hinknien. Die Ansicht wechselt dann von der *Third-Person* in die Ego-Perspektive. Der Spieler kann dann über seine Hand verschiedene Werkzeuge und Objekte verwenden, bewegen und manipulieren. Dabei stehen ihm folgende Werkzeuge zur Auswahl, die jeweils mit einer bestimmten Taste herbeigerufen werden können:

| Funktion | Taste | Grafik |
|--|-------|---|
| Dusty der Staubsauger kann Objekte einsaugen, um diese zu löschen. Ihn verwendet man sehr häufig um hinter sich aufzuräumen . Das ist vergleichbar wie wenn man C programmiert und Speicher auf dem Heap mit <code>delete</code> wieder freigeben muss. Außerdem kann er auch Objekte wegradieren. Dies ist für die Bearbeitung der Bedingungen von Robotern umbedingt von Nöten. Wegradiieren meint, wenn man bspw. die Zahl auf einem Zahlen-Block wegradiert, dann gilt als Bedingung nur noch, dass es eine Zahl sein muss, aber welche Zahl genau es ist, ist egal. Das selbe kann auch für Text oder Blöcke gemacht werden. Ein Roboter kann den Datentyp somit als Bedingung bekommen. | F2 |  |

| | | |
|--|----|--|
| Mit dem Zauberstab können Objekte kopiert werden. Der Zauberstab hat zwei Modi. Den normalen Kopiermodus und den Selbstkopiermodus. Wenn ein Roboter trainiert wird, kann dieser auch den Zauberstab aufnehmen und Objekte kopieren. Außerdem kann er sich selbst kopieren, um rekursive Prozesse zu starten. | F5 |  |
| Pumpy die Pumpe kann verwendet werden, um Objekte zu vergrößern oder zu verkleinern. Sie hat mehrere Modi. So kann sie Objekte größer, kleiner, breiter, schmäler, höher, niedriger, sowie auf eine Standardgröße wieder zurücksetzen. | F3 |  |
| Der Werkzeugkasten enthält alle Objekte, die ein Spieler braucht, um Programme zu erstellen, auszuführen und zu debuggen. | F6 |  |
| Marty der Marsianer ist der persönliche Assistent des Spielers und erklärt dem Spieler die Werkzeuge und Dinge die er mit den Objekten tun kann. Außerdem sagt er auch Bescheid, falls Fehler auftreten, z.B. wenn man einen Roboter eine Box gibt, die gar nicht zu ihm passt. | F1 |  |

Tabelle 4.3. ToonTalk: Werkzeuge

4.4.3. Concurrent Constraint Logic Programming und Actor Model

In diesem Abschnitt wird *Concurrent Constraint Logic Programming* (CCLP) anhand von ToonTalk beschrieben.

Die Essenz von CCLP ist, dass eine Berechnung aus der Interaktion nebenläufiger Agenten besteht, welche miteinander über Bedingungen (Constraints) kommunizieren (vgl. [SR89]).

Beim Designen eines Algorithmus geht es darum, ein Mechanismus zu spezifizieren, welcher bei einer bestimmten Eingabe mit Vorbedingungen eine Ausgabe mit Nachbedingungen erzeugt. So gibt man in ToonTalk einem Roboter eine Box als Eingabe, welche bestimmte Bedingungen erfüllen muss. Dieser führt dann Operationen (also den Algorithmus) auf der Box aus und erzeugt eine Ausgabe mit einer Nachbedingung. Die Roboter schauen also nach, ob ihre Vorbedingung erfüllt ist, dann führen sie ihren Code aus. Um mit anderen Robotern zu kommunizieren, werden Vögel benutzt, die zu Nestern fliegen und dann womöglich die Bedingung so verändern, dass ein Roboter angestoßen wird zu arbeiten.

ToonTalk verwendet das *Actor-Model*, jeder Roboter (bzw. Roboter-Team) arbeitet in einem eigenen Prozess. Die Kommunikation mit anderen Robotern in anderen Prozessen

funktioniert über *Message-Passing* (die Vögel, die Nachrichten zu den Nestern bringen). Es gibt keinen *Shared State* zwischen den Häusern (Prozessen).

4.4.4. Bouncing Ball

Als Erstes benötigen wir das Bild eines Balls. Dieses können wir aus dem Bilder-Notizbuch entnehmen. Wenn das Bild des Balls mit der F-Taste umgedreht wird, sehen wir die Rückseite des Balls und es erscheint ein Notizbuch mit Sensoren und Fernbedienungen für den Ball.

So kann die vertikale Geschwindigkeit (*Up-Speed*) und die horizontale Geschwindigkeit (*Right-Speed*) des Balls über die Fernbedienungen gesetzt werden.

Dann gibt es da noch den Kollisions-Sensor des Balls (s. Abb. 4.23). Dieser zeigt als Animation an, ob der Ball mit einem Objekt kollidiert oder nicht.

Nun kann ein Roboter trainiert werden, der die Geschwindigkeit mit -1 multipliziert, wenn eine Kollision auftritt, sodass die Richtung invertiert wird. Dafür erstellen wir eine Box mit zwei Löchern: Links der Kollisionsdetektor der eine Kollision anzeigt und rechts die Geschwindigkeit und trainieren den Roboter dann auf diese Eingabe. Nachdem die Geschwindigkeit mit -1 multipliziert wurden ist, deaktivieren wir noch den Kollisionsdetektor, indem wir mit der Hand über ihn gehen und die '+'-Taste drücken. Dadurch wird der Kollisionsdetektor für einen kurzen Moment auf *keine Kollision* gesetzt. Wenn wir den Detektor nicht deaktivieren würden, dann könnte der Ball hängen bleiben, da durchgängig die Geschwindigkeit invertiert werden würde, weil der Ball nicht vom Fleck kommt und immer noch eine Kollision entdeckt wird.

Diesen Roboter kopieren wir mit dem Zauberstab und platzieren dann die zwei Roboter auf die Rückseite des Bilds. Bei dem Sensoren gibt es nicht nur einen Kollisionsdetektor, sondern auch zwei Weitere. Der eine erkennt horizontale und der andere vertikale Kollisionen. Nun können wir uns daraus zwei Eingaben zusammenbauen. Eine Eingabe mit dem horizontalen Detektor und der horizontalen Geschwindigkeit und eine Eingabe mit dem vertikalen Detektor und der vertikalen Geschwindigkeit. Wenn wir die beiden Eingaben nun den zwei Robotern auf der Rückseite des Bilds geben, haben wir unseren *Bouncing Ball*. Wenn dieser horizontal kollidiert, wird dann von dem einen Roboter die horizontale Geschwindigkeit invertiert und bei einer vertikalen Kollision vom anderen Roboter die vertikale Geschwindigkeit invertiert.

Schließlich können wir noch eine Arena für den Ball bauen. Hierfür holen wir uns ein Rechteck und fügen an der Seite des Rechtecks Grenzen hinzu, indem wir ein Rechteck mit einer anderen Farbe an den Kanten des Rechtecks fallen lassen.

Jetzt können wir den Ball der Arena hinzufügen.

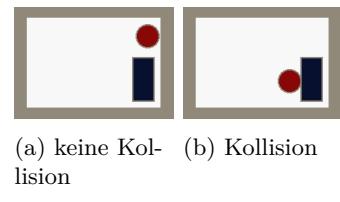


Abb. 4.23. ToonTalk: Kollisionssensor

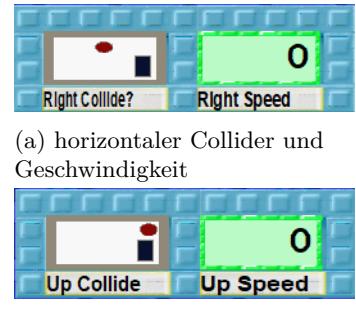
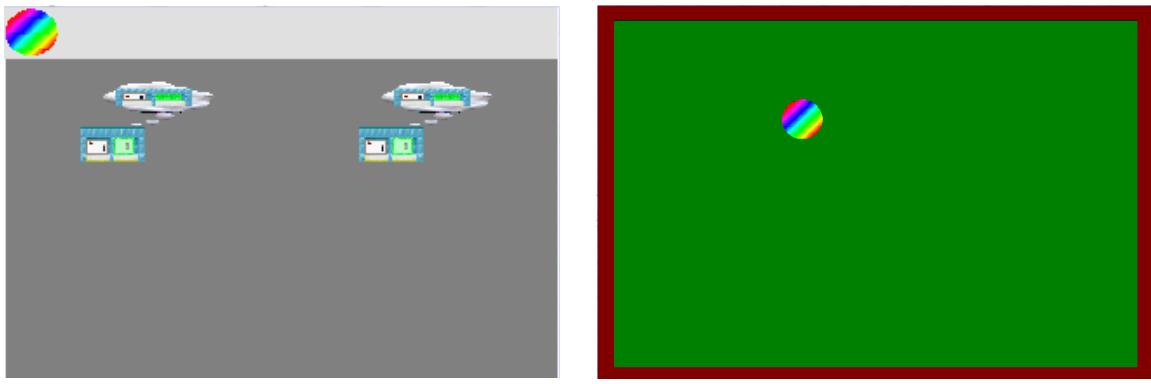


Abb. 4.24. ToonTalk: Bouncing-Ball Eingaben



(a) Rückseite des Bouncing Balls

(b) Bouncing Ball in der Arena

Schließlich können wir auch noch Sound hinzufügen, wenn der Ball kollidiert. Hierfür trainieren wir einen Roboter auf eine Box mit zwei Eingaben. Links ein Kollisionsdetektor, der eine Kollision anzeigt und rechts ein Sound. Nun trainieren wir den Roboter so, dass er den Sound abspielt.

Wenn wir diesen dann wieder auf die Rückseite des Balls platzieren und ihm als Eingabe den allgemeinen Ball-Kollisionsdetektor geben und dazu einen Sound, dann spielt er diesen Sound ab, wenn der Ball kollidiert.

4.4.5. Tic-Tac-Toe

Das Tic-Tac-Toe-Spiel in ToonTalk besteht aus mehreren Objekten, die über Vögel miteinander kommunizieren. Anhand des Sequenzdiagramms in Abb. 4.26 wird die Implementierung in ToonTalk beschrieben.

Aufgrund der nebenläufigen Natur von ToonTalk ist die Interkommunikation zwischen den Objekten immer asynchron, was im Sequenzdiagramm an den nicht ausgefüllten Pfeilen zu erkennen ist. In ToonTalk fliegt dann immer ein Vogel von einem Objekt zum anderen.

Die Interaktion beginnt damit, dass der Spieler auf ein UI-Feld klickt. Ist dieses leer, wird ein Vogel an das Turn-Objekt gesendet. Das Turn-Objekt speichert, welcher Spieler gerade dran ist und wechselt die Spieler. Das Turn-Objekt kann dann das Symbol des Spielers, der momentan dran ist, zurückschicken. Dann wechselt es den Spieler und schickt der Anzeige eine Nachricht um den nächsten Spieler anzuzeigen. Wenn das UI-Feld das Symbol dann erhalten hat, kann es seinen Text setzen. Schließlich wird das Symbol in den Feldern eingetragen, indem ein Vogel es zu der dem UI-Feld korrespondierenden Position in der Felder-Box bringt. Roboter überprüfen, ob das Spiel vorbei ist und wenn ja, wird ein Gewinner-Text an die Turn-Komponente gesendet. Diese leitet den Text an die Anzeige weiter, damit dieser angezeigt wird und merkt sich dann anschließend, dass das Spiel vorbei ist, sodass keine Reaktion mehr erfolgt, wenn auf ein leeres Feld geklickt wird.

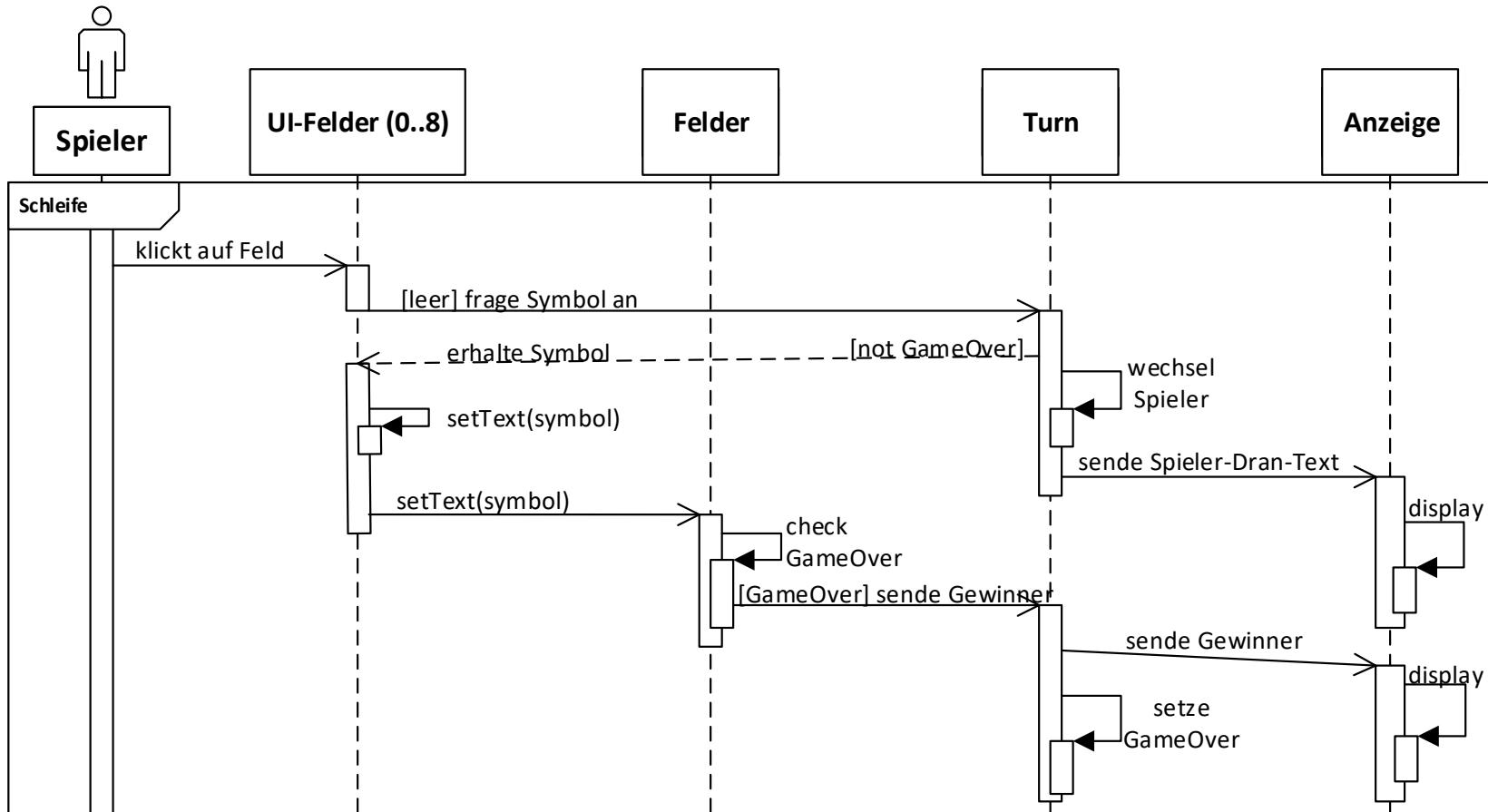


Abb. 4.26. ToonTalk: Tic-Tac-Toe – Sequenzdiagramm

Nun wird genauer auf die einzelne Implementierung der Objekte eingegangen:

Spieler (Maus)

Für das Klicken auf dem Feld müssen wir uns etwas besonderes überlegen. Die Sache ist ja die, dass der Spieler seine „Programmier-Persona“ kontrolliert, welche durch Klicken auf Objekte mit diesen interagiert. Deshalb müssen wir uns eine Maus programmieren, welcher der Spieler steuern kann. Mit der Taste F9 kann der Spieler seine Hand verstecken. Die Idee hinter der Maus ist, dass wenn der Spieler seine Hand versteckt, er dann die Maus steuert. Als Maus wird ein Bild von der Hand des Spielers verwendet (s. Abb. 4.27a).

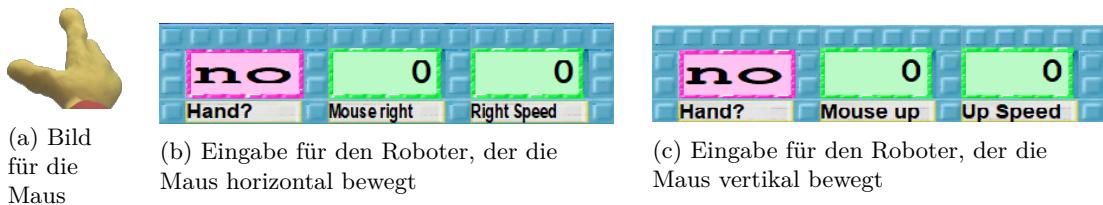


Abb. 4.27. ToonTalk: Maus für Tic-Tac-Toe

Wir trainieren uns einen Roboter, der wenn die Hand des Spielers unsichtbar ist, die Geschwindigkeit des Maus-Bildes auf die Maus-Geschwindigkeit setzt, sodass wir das Bild der Maus bewegen.

Wir brauchen zwei Roboter für die horizontale und vertikale Geschwindigkeit. Deshalb kopieren wir den trainierten Roboter und setzen ihn zusammen mit dem anderen Roboter auf die Rückseite des Maus-Bildes. Nun können wir den zwei Robotern die Eingabe-Boxen in Abb. 4.27 b) und c) übergeben. Jetzt können wir das Bild wieder richtig herum drehen und wir haben unsere Maus.

UI-Feld

Das UI-Feld ist ein Rechteck, in welchem ein Text steht. Der Text ist am Anfang auf '-' gesetzt, was für ein leeres Feld steht. Für das Feld werden zwei Roboter trainiert:

1. Ein Roboter um den Klick zu registrieren und das Turn-Objekt zu benachrichtigen.
2. Ein zweiter Roboter, um den Text zu setzen und die Felder zu benachrichtigen.

Zuerst hatte ich die zwei Roboter so trainiert, das sie unabhängig auf zwei verschiedene Eingaben operieren. Als ich die UI-Felder jedoch am Ende auf das Tic-Tac-Toe-Feld eingesetzt habe, ist ein Roboter auf der Rückseite des Felds immer verloren gegangen und das Spiel hat nicht funktioniert. Deswegen agieren die beiden Roboter nun in einem Team zusammen auf einer Eingabe-Box.



(a) Eingabe, wenn nicht drauf geklickt wird



(b) Eingabe, wenn drauf geklickt wird

Abb. 4.28. ToonTalk: Tic-Tac-Toe - Eingabe-Box des UI-Felds

Diese Eingabe-Box ist in zwei Varianten in Abb. 4.28 abgebildet und ich erkläre nun die einzelnen Slots:

1. **Parts:** Das ist die Box, die angibt welche Teile das UI-Feld enthält. Da das Feld Text enthält, wird dieser Text angezeigt und kann auch verändert werden.
2. **Left Click:** Globaler Sensor, der angibt, ob die linke Maustaste geklickt wird.
3. **Touching Who?:** Wenn ein Objekt das Feld berührt, wird hier das Bild des Objekts angezeigt.
4. **myBird:** Dieser Vogel gehört zum Feld und wird für die Rückkommunikation des Turn-Objekts verwendet.
5. **turnBird:** Dieser Vogel kommuniziert mit dem Turn-Objekt.
6. **myNest:** Das Nest von *myBird*.
7. **indexBird:** Der Vogel trägt das Symbol an die korrespondierende Position in das Felder-Objekt ein.

Der erste Roboter wird nun so trainiert, dass er bei einem Klick (s. Abb. 4.28a) das Turn-Objekt benachrichtigt. Hierfür übergibt er dem *turnBird* die Box mit *myBird*, welcher dann zum Turn-Objekt fliegt. Das Turn-Objekt kann dann über den *myBird* das Symbol zurückschicken. Dieses landet dann in *myNest*.

Der zweite Roboter aktiviert sich dann, wenn etwas im Nest liegt. Er wird so trainiert, dass er den Text der *Parts*-Box auf das Symbol setzt, das im Nest liegt. Dann übergibt er dem *indexBird* das Symbol.

Da man für jedes der 9 Felder individuell die Eingabe-Box erstellen muss, habe ich diesen Vorgang durch einen Roboter automatisieren lassen. Dieser erstellt das UI-Feld, sodass es funktional ist. Er konstruiert das UI-Feld. Der Konstruktor-Roboter bekommt als Eingabe das noch unfunktionale UI-Feld, den Index-Vogel, den Turn-Vogel, das Roboter-Team für das UI-Feld und einen Vogel für die Rückgabe übergeben (s. ??).



Abb. 4.29. ToonTalk: Tic-Tac-Toe – Eingabe für den Feld-Konstruktor-Roboter

Nun trainieren wir den Roboter so, dass er die Eingabe-Box für das Feld wie in Abb. 4.28 zusammenbaut. Dafür muss er das Feld umdrehen und aus dem Sensor-Notizblock die

Parts und *Touching Who?*-Sensoren (bzw. Fernbedienungen) rausuchen. Nachdem er die Box konstruiert hat, platziert er die Roboter mit der Box als Eingabe auf der Rückseite des UI-Felds. Schließlich übergibt er das UI-Feld dem Rückgabe-Vogel. Am Ende beendet er seinen Prozess, indem er die Bombe zündet.

Jetzt können wir diesen Konstruktor-Roboter in ein Truck setzen und ihm die Eingabe übergeben. Der Truck fährt dann los und wir bekommen das funktionale UI-Feld zurück, das der Rückgabe-Vogel in das Rückgabennest liefert (s. 4.30).



Abb. 4.30. ToonTalk: Tic-Tac-Toe - UI-Feld-Konstruktor

Turn

Das Turn-Objekt sieht folgendermaßen aus:



Abb. 4.31. ToonTalk: Tic-Tac-Toe - Turn-Objekt

Auf dem Objekt sind drei Roboter trainiert, die in einem Team zusammengefügt sind. Zwei der Roboter machen inhaltlich das Selbe. Sie sind nämlich dafür da, dem UI-Feld das Symbol zurückzuschicken und die Anzeige zu ändern. Sie werden aktiviert, wenn sich eine Box mit einem Vogel auf dem Turn-Nest befindet. Für den einen Roboter gilt als Bedingung das das X-Symbol in der Turn-Variable vorhanden sein muss. Der Roboter schickt dann das X-Symbol zurück, ändert dann die Turn-Variable auf 'O' und sendet der Anzeige über den Vogel eine Nachricht mit dem Text 'O ist dran!'. Der andere Roboter sendet das 'O' zurück, setzt die Turn-Variable auf 'X' und sendet der Anzeige einen Vogel mit dem Text 'X ist dran!'.

Der dritte Roboter aktiviert sich, wenn sich etwas im win-Nest befindet. Die Nachricht wird kopiert und der Anzeige geschickt. Nachdem der anzeigen-Vogel wieder zurück ist,

wird dieser dann gelöscht. Die Nachricht wird kopiert anstatt sie zu verschieben, weil die zwei vorherigen Roboter als Bedingung haben, dass das win-Nest leer ist, sodass kein Feld mehr geklickt werden kann, wenn GameOver ist. Wenn wir den Anzeige-Vogel nicht löschen würden, würde der Game-Over-Roboter am Ende immer weiter arbeiten, weil er als Bedingung hat, dass der Anzeige-Vogel existieren muss.

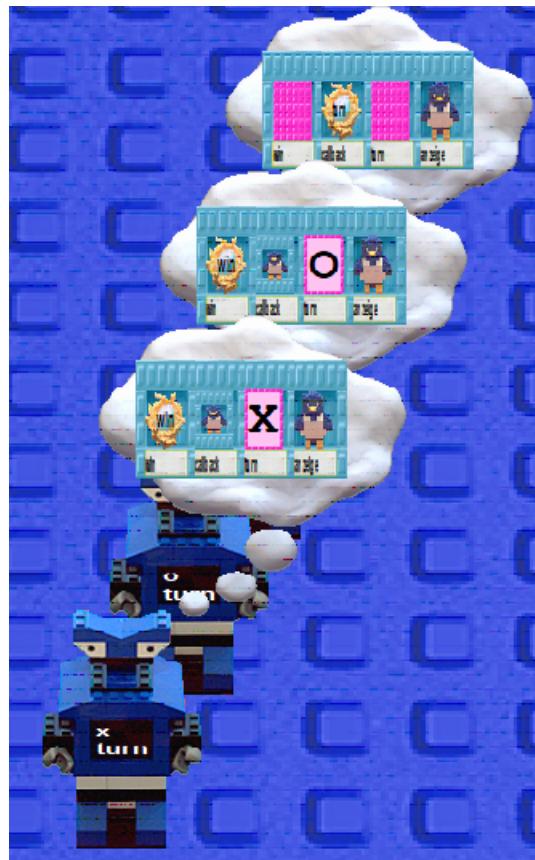


Abb. 4.32. ToonTalk: Tic-Tac-Toe - Die Roboter des Turn-Objekts

Anzeige

Die Anzeige ist ziemlich simpel, sie besteht aus einem länglichen Rechteck auf dem Text steht. Es arbeitet ein Roboter in dem Anzeige-Objekt. Immer wenn ein Text in das Nest gelegt wird, setzt dieser den Text der Anzeige auf diesen Text.

Felder

In den Feldern wird der interne Zustand des Tic-Tac-Toe Spiels verwaltet – also welches Symbol auf welchem Feld steht. Die Felder besteht aus einer Box mit 10 Löchern. In den ersten 9 Löchern sind die Nester der Index-Vögel. Die Index-Vögel selber befinden sich in den UI-Feldern. Wenn ein Feld geklickt wird, wird dem Index-Vogel im UI-Feld das Symbol gegeben. Dieser bringt es dann in sein Nest in dem Felder-Objekt.



(a) Anzeige

(b) Eingabe für den Roboter

Abb. 4.33. ToonTalk: Tic-Tac-Toe - Anzeige

In dem letzten Loch befindet sich der Gewinner-Vogel. Dieser benachrichtigt das Turn-Objekt, wenn es einen Gewinner gibt.



Abb. 4.34. ToonTalk: Tic-Tac-Toe - Felder

Um zu überprüfen, ob es einen Gewinner gibt, werden 16 Roboter für jede mögliche Gewinn-Konfiguration trainiert. Es gibt 3 vertikale, 3 horizontal und 2 diagonale Gewinnmöglichkeiten. Also insgesamt 8 - Und dann nochmal für jeden Spieler, also insgesamt 16.

Zusätzlich gibt es noch einen Roboter, der Unentschieden erkennt. Dieser wird aktiviert wenn alle Felder voll sind.

Die Roboter machen nichts weiteres als dem Gewinner-Vogel den Gewinner-Text zu übergeben, welcher die Nachricht dann an das Turn-Objekt sendet.

Zuerst habe ich alle 17 Roboter in ein Team zusammengetan. Dann hat das Spiel jedoch nicht funktioniert. Es sah so aus als würden sie sich gegenseitig blockieren. Vielleicht gibt es einfach eine Obergrenze von der Anzahl an Robotern, die man zusammen in ein Team bringen kann.

Ich habe mich dann für acht 2er-Teams entschieden - für jede Gewinn-Konfiguration unabhängig vom Spieler ein Team. Ein Roboter überprüft, ob 'X' gewonnen hat und der andere, ob 'O' gewonnen hat.

Der Unentschieden-Roboter arbeitet alleine.

Das Felder-Objekt wird dann so oft kopiert, wie es von Nötigen ist, sodass jedes Team ein Objekt hat worauf es arbeitet.



(a) 2er-Team, das die erste Horizontale überprüft



(b) Der Unentschieden-Roboter

Alle Objekte zusammenfügen

Am Ende können alle Objekte zusammengefügt werden. Hierfür verwenden wir wieder ein Rechteck, welches das ganze Tic-Tac-Toe Spiel enthält. Die UI-Felder, die Anzeige und die Maus werden in dem Rechteck platziert. Auf der Rückseite des Rechtecks werden alle restlichen Roboter platziert. Dazu zählen die acht 2-er-Team Roboter, der Unentschieden-Roboter und die 3er-Team Roboter des Turn-Objekts. Dann bekommen alle Roboter die Eingabe zugewiesen auf welche sie achten sollen.

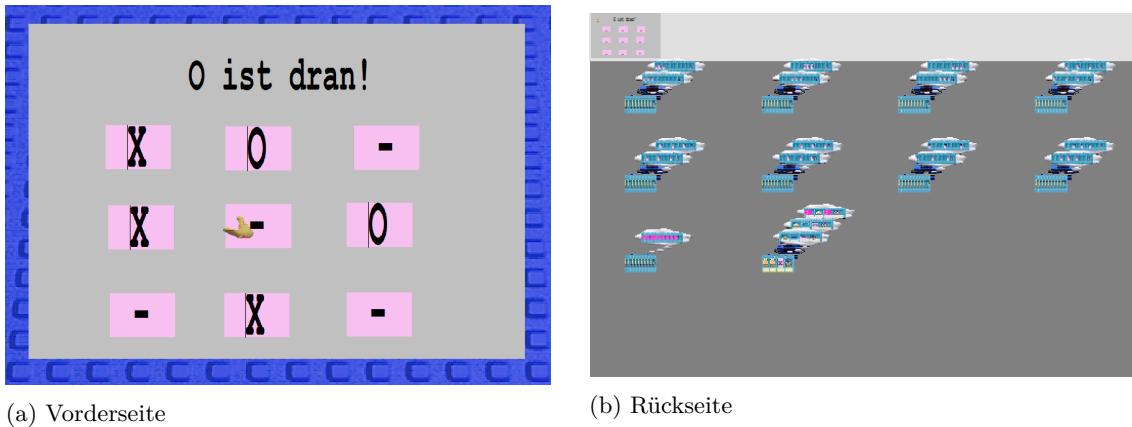


Abb. 4.36. ToonTalk: fertiges Tic-Tac-Toe

4.4.6. Bewertung

Einfachheit der Implementierung von Computerspielen

Da es sehr aufwendig ist, ein Spiel in ToonTalk zu implementieren, wurde nur Tic-Tac-Toe implementiert. Bei Tic-Tac-Toe wurden alle Anforderungen außer, dass über die Spielfläche neu gestartet werden kann, erfüllt. Es kann allerdings einfach das fertige Tic-Tac-Toe-Spiel gelöscht und ein neues aus dem Notizbuch kopiert werden.

Die Implementierung war sehr komplex. Dies kann allerdings daran liegen, dass ich mit dem Paradigma des *Concurrent Constraint Logic Programming* und mit dem Konzept einer animierten Programmiersprache noch nicht so vertraut war und die Steuerung von ToonTalk erst erlernen musste.

Ich musste mir zuerst ein Programm-Konzept für Tic-Tac-Toe auf dem Papier überlegen, das ich dann auf ToonTalk übertragen konnte.

Ein *Hack* war z.B., dass zuerst eine Maus erstellt und mit der F9-Taste, die Spiele-Hand deaktiviert werden musste, um dann mit der erstellen Maus Tic-Tac-Toe zu spielen.

Zudem kamen viele weitere kleinere Problem hinzu, die ich mit *Trial and error* in den Griff bekommen habe.

Da man sich mit einer virtuellen Figur in einer Spielwelt bewegt, Dinge zusammenbaut und miteinander verbindet, wird ein grafischer Szenen-Editor nicht mehr benötigt, da man sich bereits in der Szene befindet.

Das Koordinatensystem ist rechtshändig. Wenn ein Bild eine positive Y-Geschwindigkeit hat, dann bewegt es sich nach oben und bei einer positiven X-Geschwindigkeit nach rechts.

Der Koordinatenursprung befindet sich in einem Raum unten in der Mitte. Bilder können nicht rotiert werden.

Es gibt keine GUI-Bibliothek.

In ToonTalk gibt es Sound-Felder, die einen Klang abspielen, wenn auf sie gezeigt und die Leertaste gedrückt wird. Es können `WAV`-Sound-Dateien in ToonTalk importiert werden. Außerdem können auch MIDI-Dateien abgespielt werden (vgl. [Kah16]).

ToonTalk unterstützt Bilder und Animationen (über GIF-Dateien) und es lassen sich auch Bilder und Animationen importieren.

Der *Game Loop* in ToonTalk muss nicht abstrahiert werden. Sobald man ToonTalk spielt befindet man sich selbst im *Game Loop*.

Erlernbarkeit nach Konstruktionismus

Neben Grafiken und Sound konnte ToonTalk über das *Windows Media Control Interface* verschiedene Medien wie Sound, Musik und Videos abspielen. Die Betonung liegt auf „konnte“, da diese Schnittstelle zugunsten der *DirectX-API* eingestellt wurde (vgl. [Ban10]).

Außerdem können für ToonTalk Erweiterungen erstellt werden. So gibt es bspw. die *RCX-Extension* über welche in ToonTalk mit *LEGO-Mindstorm*-Robotern kommuniziert werden kann. Über einen besonderen Vogel kann dann mit den Robotern kommuniziert werden:

„From a programming point of view, the RCX extension is a special bird that can fly ‘outside’ of ToonTalk and communicate with the RCX. To execute a command on the RCX, e.g. turn on the motors, boxes that contain commands are given to the bird. If a response is needed, e.g. a sensor reading, then the box will include another bird that will bring back the response, thereby achieving communication from the RCX back to the computer.“ [Web]

Da ToonTalk schon etwas älter ist, ist zu testen, ob die Erweiterung für neuere Versionen von LEGO-Robotern noch funktioniert.

Turtle-Grafiken können in ToonTalk nicht erstellt werden.

ToonTalk ist eine ziemlich selbstoffenbarenden Umgebung. Zum Beispiel enthält ToonTalk Boxen und schon kleine Kinder können entdecken, wie Boxen bewegt werden oder wie Dinge in Boxen reingelegt oder wieder rausgenommen werden.

Zudem, werden Aktionen durch Sound-Effekte und Animationen unterstützt. Wenn beispielsweise ein Ding über eine Box gehalten wird, dann wackelt dieses, um anzudeuten, dass es dort reingelegt werden kann. Wenn der Benutzer dann klickt, wird eine Animation und ein Sound-Effekt abgespielt. Das Ding schrumpft auf die Größe des Boxslots und wird dann eingefügt.

Außerdem gibt es dann noch *Marty*, ein persönlicher virtueller Assistent, der gefragt werden kann, wenn bestimmte Mechaniken nicht verstanden werden und der diese dann

erklärt.

ToonTalk ist auch ein teilweise sichere Umgebung, da Aktionen wieder rückgängig gemacht werden können, z.B. *Dusty* Dinge wieder ausspucken kann oder *Bammer die Maus* kann Elemente auch wieder auseinanderhämtern.

Nur ist es bei mir mehrfach vorgekommen, dass ToonTalk *gecrasht* ist, weswegen ich vorsichtshalber so oft wie möglich gespeichert habe.

In ToonTalk gibt es keine Kompilierzeit und da alles während der Laufzeit geschieht, müsste ToonTalk dynamisch typisiert sein. Roboter können jedoch über ihre *Constraints* Elemente bestimmter Typen erwarten. Ein Roboter kann beispielsweise so trainiert sein, dass er nur auf eine Box, in der eine Zahl drin ist, operiert. Deswegen hat ToonTalk auch Elemente eines strengen Typsystems.

Inkrementelle Einführung

Für das Programm „Hallo Nutzer“ kann einem Roboter eine Box mit einem Namen gegeben werden. Dann trainiert man den Roboter so, dass er ein „Hallo“ an den Anfang des Namens setzt und ein Ausrufezeichen ans Ende.

Programmierkonzepte können in ToonTalk nacheinander eingeführt werden, was das *Puzzle-Game* demonstriert, welches die Konzepte von ToonTalk von einfachen Themen wie das Erstellen von Boxen zu komplexeren Themen, wie das Erstellen eines neuen Prozesses, nach und nach einführt.

Intuitive Syntax und Natürlichsprachlichkeit

In ToonTalk gibt es keine Syntax. Roboter werden über *Programming by demonstration* trainiert, indem man ihnen an einem Beispiel zeigt, was sie genau machen sollen.

Somit braucht die Syntax auch nicht internationalisiert zu werden. Ein wichtiger Bestandteil von ToonTalk ist allerdings *Marty*, welcher einem die Funktionen von ToonTalk erklären kann. ToonTalk ist jedoch nur in Englisch und Schwedisch erhältlich, sodass *Marty* nutzlos für Leute ist, die diese Sprachen nicht sprechen.

Feature Uniformity und Orthogonalität

ToonTalk's Funktionen und computerbasierte Abstraktionen sind so ausgewählt, dass es keine Überschneidungen zwischen ihnen gibt. Jedes Element in ToonTalk hat seinen eigenen Nutzen und kann von keinem anderen Element ersetzt werden. Um Nachrichten zwischen Objekten zu senden gibt es z.B. nur das Vogel-Nest-Paar. Dabei kann dann ein *Multicast* erzeugt werden, indem das Nest mit dem Zauberstab kopiert wird. Die Funktionen ergänzen sich so gegenseitig und wenn alle zusammengenommen werden, können mächtigere Programme erstellt werden.

Die Benutzung der Werkzeuge ist konsistent. Jedes Werkzeug lässt sich dadurch verwenden, dass es aufgehoben wird. Die Taste zum Ausführen des Werkzeugs ist gleich und der

Modus des Werkzeugs – sofern es einen hat – lässt sich auf die gleiche Art und Weise ändern.

Unterstützung der wichtigsten Programmierkonzepte & Datenstrukturen

Die Elemente der strukturierten Programmierung in ToonTalk werden folgendermaßen umgesetzt:

Die Schleife wird dadurch umgesetzt, dass ein Roboter wiederholt sein trainiertes Programm auf eine Box ausführt, bis diese nicht mehr mit den Bedingungen des Roboters übereinstimmt. Dies ist also äquivalent zu einer `do ... while`-Schleife.

Bedingungen werden durch die Bedingungen des Roboters implementiert. Dabei steht jeder Slot in einer Box für eine Bedingung, die erfüllt sein muss. Dabei kann eine Bedingung über *Dusty* auch rausgelöscht werden. Die Bedingungen in einer Box sind mit einer AND-Verknüpfung miteinander verbunden. Alle müssen erfüllt sein, damit der Roboter mit der Arbeit beginnt.

Dadurch, dass mehrere Roboter mit unterschiedlichen Bedingungen in einem Team zusammengefasst werden können, wird eine OR-Verknüpfung implementiert. Wenn die Bedingungen des ersten Roboters nicht übereinstimmen, dann wird der nächste Roboter ausprobiert usw. .

Über die Roboter-Teams kann auch NOT implementiert werden. Hierfür erstellt man einen Roboter, der die Bedingung hat, wann etwas nicht ausgeführt werden soll. Dieser Roboter macht dann einfach nichts. Diesen Roboter setzt man dann vor den anderen Roboter, der etwas ausführen soll, wenn die Bedingung nicht zutrifft. Wenn man dem Team jetzt die Bedingung gibt, die nicht erfüllt sein soll, dann blockiert der erste vordere Roboter den zweiten Roboter und nichts passiert. Andernfalls wird der zweite Roboter aktiviert.

Die prozedurale Programmierung wird in ToonTalk über Roboter, Trucks und Nester implementiert. In einen Truck, kann man einen Roboter setzen, der eine bestimmte Funktion ausführt. Der Roboter ist äquivalent zum Funktionskörper. Damit der Truck mit dem Roboter losfährt und einen neuen Prozess erstellt (Funktionsaufruf), muss man ihm noch eine Box mitgeben, an die der Roboter arbeiten soll (Parameter).

In einem Prozess sind alle erstellten Werte lokal. Innerhalb eines Prozesses können weitere Prozesse aufgerufen oder sogar der gleiche Prozess kopiert und aufgerufen werden. Es können somit rekursiv neue Prozesse erzeugt werden.

Roboter räumen automatisch hinter sich auf. Sie benutzen *Dusty*, um alle Werte die sie während ihrer Aktion erstellen am Ende der Aktion aufzusaugen – ein automatischer *Garbage Collector*.

Prozesse (Häuser) müssen jedoch selbständig zerstört werden. Hierfür wird die Bombe verwendet. Wenn eine Funktion in ToonTalk ausgeführt werden soll, welche einmalig etwas berechnen soll, dann ist es üblich, dass der trainierte Roboter am Ende die Bombe nutzt um seinen Prozess zu zerstören, sodass es keine stillgelegten Prozesse gibt, die unnötig Ressourcen verschwenden.

Über diese Funktionen kann in ToonTalk dann die rekursive *Fibonacci*-Funktion implementiert werden. Für die Implementierung werden vier Roboter benötigt. Zwei der Ro-

boter sind für die Basisfälle der *Fibonacci*-Funktion zuständig. Der dritte Roboter ist für den Rekursionsschritt verantwortlich und ruft die zwei rekursiven Prozesse auf. Der letzte Roboter addiert die Ergebnisse der rekursiven Aufrufe und gibt das Ergebnis zurück.



Abb. 4.37. ToonTalk: Roboter für die *Fibonacci*-Funktion

Die *Fibonacci*-Funktion in ToonTalk hat allerdings eine sehr schlechte Performanz. Dies liegt an der hohen Nebenläufigkeit. Anders als bei der synchronen prozeduralen Programmierung, wo sich der Funktionsstack bei einem rekursiven Aufruf auf- und wieder abbaut, werden in ToonTalk alle Prozesse zuerst erstellt und dann wieder abgebaut. Die Implementierung schafft es nicht ein *Fibonacci*-Zahl über der vierzehnten zu berechnen, da es einfach zu viele Prozesse gibt. Die Anzahl der Prozesse für die nte Zahl kann über folgende rekursive Formel berechnet werden:

$$|p|_n = |p|_{n-1} + |p|_{n-2} + 2 \text{ für } n >= 3 \quad (4.1)$$

$$|p|_1 = |p|_2 = 1 \quad (4.2)$$

Dies ist sehr ähnlich zur Definition der *Fibonacci*-Reihe, beim der rekursiven Definition wird allerdings noch +2 addiert, da der Roboter, der die rekursiven Prozesse erstellt und der Roboter, der addiert, je noch ihren eigenen Prozess haben. Es kann dann berechnet werden, dass für die 14. *Fibonacci*-Zahl 1129 simultane Prozesse benötigt werden.

ToonTalk unterstützt die Datenstrukturen Liste, *Queue* und *HashMap*. Eine Liste wird durch die Boxen implementiert. Boxen können erweitert werden, indem man Boxen anfügt und sie können auch wieder getrennt werden. Wenn man eine Box auf eine Zahl n fallen lässt, dann wird die Box zerbrochen, sodass die erste Teil der Box n Elemente enthält.

Eine *Queue* wird durch das Nest implementiert. Nachrichten, die Vögel bringen werden immer ganz unten in das Nest gelegt.

Das Notizbuch, kann als *HashMap* verwendet werden, wobei die Schlüssel auf Zeichenfolgen begrenzt sind.

Programmierumgebung (IDE)

Die Umgebung in ToonTalk ist ziemlich einfach gehalten. Jedoch gibt es viele Werkzeuge und Funktionen, die zuerst erlernt werden müssen. Man kann die Werkzeuge je mit einem bestimmten Tasten-Kürzel zu sich rufen. Wenn die Alt-Taste gedrückt wird, werden aber alle Werkzeuge mit ihrem *Shortcut* angezeigt. Um die Werkzeuge zu verwenden, verwendet man die Leertaste und der Modus der Werkzeuge kann entweder über die Tastatur oder auch über einen Mausklick geändert werden, was es einfacher macht.

Die Interaktivität in ToonTalk ist extrem hoch. Durch *Programming by demonstration* sieht man direkt, was das Programm macht. Es gibt keinen Wechsel zwischen Programmierung und Programmausführung. Man befindet sich in der Laufzeit selbst.

Durch diese hohe Interaktivität und dadurch das Programmabläufe animiert sichtbar gemacht werden, kann ein ToonTalk-Programm gut debuggt werden. Das Problem ist, dass Dinge außer Kontrolle geraten können, wenn ein Fehler gemacht wurden ist. Hierfür gibt es jedoch eine Zeitreisefunktion, mit der man rückspulen und vor spulen kann, um Operationen rückgängig zu machen. Außerdem können aller Roboter über die F8-Taste aus- und angeschaltet werden. Falls die Dinge zu sehr außer Kontrolle geraten, kann man somit alle Roboter ausschalten, um die Ordnung dann wieder herzustellen.

ToonTalk kann nur auf Windows installiert werden und es wird ein *Installer* zu Verfügung gestellt.

ToonTalk-Programme können über *Copy-Paste* aus ToonTalk exportiert werden. Wenn man ein Objekt in ToonTalk hält und dann STRG+C drückt, dann wird dieses als Text in die Zwischenablage gespeichert. Ebenso kann ein ToonTalk-Programm, welches als XML-Datei gespeichert ist mit STRG+V in ToonTalk importiert werden. Dadurch können Programme leicht miteinander ausgetauscht werden.

Diese Funktion hat das *Playground*-Projekt ermöglicht. Hier wurde eine ganze Stadt in ToonTalk mit Computerspielen gefüllt, welche von Kindern erstellt wurden sind (vgl. [Kah04b]).

Zudem ist eine weitere Möglichkeit der Kollaboration, dass über Langdistanzvögel andere ToonTalk-Programme auf anderen Rechnern ausgeführt werden können.

Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplinen (*Clean Code*)

In ToonTalk lassen sich Programme gut organisieren. Roboter und Objekte, die bestimmte Programme ausführen, können in Notizbücher gespeichert und immer wieder verwendet werden.

Roboter können auf der Rückseite von Bildern plaziert und dadurch versteckt werden (*Information Hiding*). So lassen sich komplexe Objekte erstellen, deren Funktionsweise versteckt wird.

Objekte können jedoch nicht so leicht miteinander verbunden werden. Bei Tic-Tac-Tac gibt es bspw. die Anzeige. Die macht nicht anderes an den Text anzuzeigen, den ein Vogel in Nest legt. Die Anzeige kann dann auch losgekoppelt von Tic-Tac-Toe in anderen

Projekten verwendet werden. Hierfür muss aber die Anzeige umgedreht werden und dem auf der Rückseite platzierten Roboter die richtige Box gegeben werden. Da der Roboter anzeigt, was er für eine Box braucht, gibt es eine Art implizite Dokumentation. Wenn ein Objekt umgedreht wird, gibt es vielleicht Roboter, die irgendetwas internes machen und andere Roboter, die für die Verbindung (die Schnittstelle) wichtig sind. Diese zwei Arten von Robotern können nicht voneinander unterschieden werden (höchstens durch Namenskonventionen). Bei komplexeren Objekten mit mehreren Robotern, dürfte es also sehr schwer fallen, diese richtig zu verbinden.

Die Dokumentation in ToonTalk kann über mehrere Maßnahmen erfolgen. Robotern sollten geeignete Namen gegeben werden. Die Slots einer Box können Namen gegeben werden. Es können Text-Felder zur Dokumentation verwendet werden. Programmteile sollten in Notizbücher mit geeignetem Namen abgespeichert werden.

Es gibt also Funktionen, durch welche Programme in ToonTalk dokumentiert werden können. So etwas wie Dokumentations-Kommentare, aus der eine Dokumentation generiert werden kann, gibt es nicht.

ToonTalk ist dynamisch typisiert mit einem strengen Typsystem. Robotern unterscheiden z.B. zwischen Text- und Zahlenfeldern. Es gibt Operationen die für die verschiedenen Objekte definiert sind. Wenn man z.B. eine Zahl auf einen Text z.B. fallen lässt, dann erhöht sich der *Code-Point* des letzten Zeichen des Texts um die Zahl. Andersherum kann kein Text auf eine Zahl fallen gelassen werden. Ungültige Operationen werden also nicht zugelassen.

Typumwandlungen oder *Casts* gibt es in ToonTalk so nicht. Ein Text kann nicht in eine Zahl konvertiert werden.

Ebenfalls gibt es kein *Exception-Handling* und keine Unit-Tests. Unit-Tests können allerdings über Elemente wie die Waage und Robotern implementiert werden.

Zudem gibt es auch keinen *Style-Guide* für ToonTalk.

Fehlermeldungen

Wenn in ToonTalk irgendwelche Fehlerzustände entstehen, dann gibt *Marty* dem Benutzer Bescheid. Ein möglicher Fehler ist, dass ein Roboter oder ein Team von Robotern eine Box bearbeiten, die von den Bedingungen her gar nicht dazu passt.

Ein anderer Fehlerzustand ist, wenn mit *Dusty* (versehentlich) ein Werkzeug eingesaugt wurden ist und dieses Werkzeug dann herbeigerufen werden soll. *Marty* macht dann darauf auch aufmerksam.

Marty gibt auch eine Fehlermeldung aus, wenn versucht wird, durch 0 zu dividieren. Dabei bleibt der Wert unverändert und *Marty* gibt einfach die Fehlermeldung aus.

Ungültige Operationen können in den meisten Fällen gar nicht ausgeführt werden und falls es eine ungültige Operation gibt, dann wird diese abgefangen und *Marty* gibt dann den Fehler aus. Dabei erklärt *Marty* den Fehler in einfacher und positiver Sprache. Wenn ein Roboter eine ungültige Box hat, dann blinkt die Bedingung des Roboters rot auf, um den Benutzer darauf aufmerksam zu machen.

Insgesamt werden in ToonTalk nur wenige Fehler angezeigt, da es keine Syntax-Fehler und auch nur wenige Operationen gibt, die einen Fehler erzeugen könnten.

Fehlermeldungen sind nicht lokalisiert.

Ein hohes Abstraktionsniveau für Datentypen

ToonTalk unterstützt die Verarbeitung, die Eingabe und das Anzeigen von *Unicode-Zeichen*. Außerdem wird auch Text, der von rechts nach links gelesen wird, unterstützt.

Zahlen in ToonTalk unterstützen die Langzahlarithmetik. Es werden große Ganzzahlen sowie exakte rationale Zahlen unterstützt.

Es wurden sich eine besondere Anzeige für lange Zahlen überlegt. Für rationale Zahlen mit sich wiederholenden Dezimalstellen wurde sich ein neues Zahlenformat mit dem Namen *Shrinking Digits* überlegt, damit diese nicht Unendlich lang sind:

„We solved this problem by inventing a new number format called the Shrinking Digits format. The idea is to display the fractional part as a decimal that uses a decreasingly smaller font size. Visually this indicates that there is more to the number – you just can't see it since it is too small.“ [Kah04a]

Bei besonders großen Zahlen gibt es keinen besonderen Trick, wie sie angezeigt werden. Bei *ToonTalk Reborn* hingegen, wurde sich eine neue Anzeigemöglichkeit für besonders große Zahlen überlegt, bei der die Ziffern in der Mitte der Zahl immer kleiner werden.

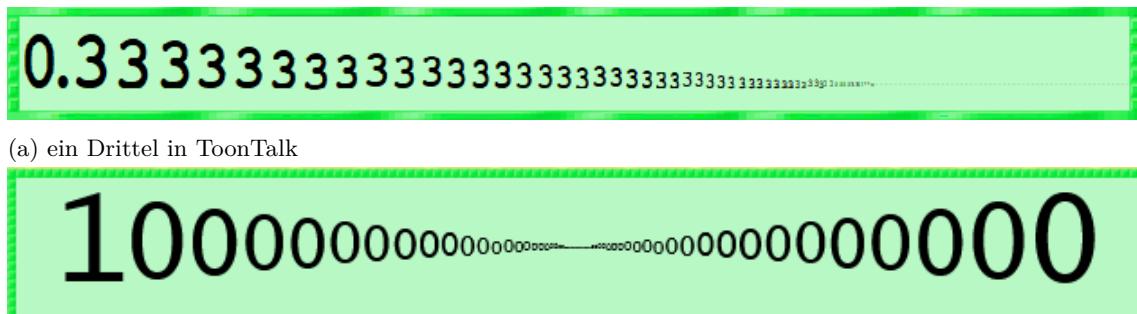


Abb. 4.38. ToonTalk: Anzeige von langen Zahlen

Anwendungsbereiche und Interoperabilität mit Technologien und anderen Programmiersprachen

Anwendungsbereiche:

- 2D-Grafiken
- Spiele
- Musik und Sound
- LEGO Mindstorm Roboter

ToonTalk läuft auf Windows-Desktoprechnern und das neuere *ToonTalk Reborn* läuft im Web.

Ein ToonTalk-Programm kann in ein *Java-Applet* übersetzt werden, welches im Browser

ausgeführt werden kann. Hierfür steckt man in ToonTalk einen Roboter, ein Bild oder ein Truck ein, drückt dann auf Pause und klickt dann auf „Save It, Make Java Applet, and Show Web Page“. Dann öffnet sich automatisch der Browser, indem das Programm ausgeführt wird (vgl. [Kah]).

Dokumentation & Community Support

ToonTalk hat eine ausführliche Dokumentation, welche nur auf Englisch verfügbar ist (s. [Kah16]). Außerdem könnte die Dokumentation etwas lesbarer und navigierbarer gestaltet sein.

ToonTalk's Community ist klein und es gibt keine offiziellen Gruppen oder Foren. Kehn Kahn liegt ToonTalk jedoch noch sehr am Herzen und er ist sehr hilfsbereit (s. Email A.1).

Durchs ToonTalk's *Puzzle-Game* werden die Elemente von ToonTalk sehr gut eingeführt. Zusätzlich gibt es ein Video-Tutorial, das zeigt wie zwei Kinder zusammen *Ping-Pong* erstellen.

Außerdem gibt es einige Beispiele und Demos. So enthält das Haupt-Notizbuch ein Beispiel-Notizbuch, das einige Beispiele enthält. Über das Hauptmenü können einige Demos gestartet werden, die z.B. das Erstellen eines Bank-Accounts in ToonTalk oder das Programmieren der *Fibonacci*-Reihe vorstellen. Hier ist die Zeitreise-Funktion dann aktiviert, sodass die einzelnen Schritte nachvollzogen werden können.

Die Dokumentation und die Beispiele sind nicht internationalisiert.

4.5. Game Changineer

4.5.1. Kurze Einführung in Game Changineer

Game-Changineer verwendet *Natural Language Programming* (NLP), um englischsprachige Programmanweisungen in ein 2D-Computerspiel zu übersetzen (vgl. [Hsi18]).

Dabei wird eine sogenannte *Constrained Natural Language* (CNL) verwendet. Diese ist eine Teilmenge der natürlichen Sprache, die geschaffen wird, indem die Grammatik und die Vokabeln begrenzt werden. Dadurch können Mehrdeutigkeiten und die Komplexität der Sprache verringert werden (vgl. [Wik21f]).

Die wichtigste Einschränkung, mit der der Designer von Game-Changineer Michael S. Hsiao die englische Sprache begrenzt hat ist, dass jeder Satz *Objekte* miteinbezieht. Dabei ist ein Objekt als Entität in einem Videospiel definiert, wie z.B. ein Charakter, oder eine Punktzahl. Deshalb haben die Autoren ihre CNL als *Object Oriented Natural Language* (OONL) bezeichnet (vgl. [Hsi18]).

Hsiao gibt an, dass 20 Zeilen dieser OONL in mehr als 1800 Zeilen Spielecode konvertiert werden können. Dabei ist der Ansatz, dass zuerst Zwischencode (IR-Code) generiert wird. So wird z.B. der Satz „When a rabbit collides with a fox, the rabbit dies.“ in den Zwischencode „if collide(rabbit, fox), die(rabbit)“ übersetzt. Aus diesem Zwischencode aus, kann dann in einer beliebigen Programmiersprache das Spiel generiert werden. Für die Game-Changineer-Webseite wird der Zwischencode in Javascript umgewandelt, wobei die

ProcessingJS Bibliothek verwendet wird (vgl. [Hsi18]).

4.5.2. Beschreibung der Programmierumgebung

Bei der Programmierumgebung handelt sich um eine Webanwendung. Es kann sich dort angemeldet werden. Es kann aber auch ohne Anmeldung ein Spiel erstellt werden. Angemeldet, können Programme gespeichert und geladen werden. Auf der linken Seite der Webseite befindet sich ein *Canvas*, indem das Spiel ausgeführt wird. Über dem *Canvas* werden Fehlermeldungen und Warnungen angezeigt und wie gut die Anweisungen verstanden wurden sind.

Auf der rechten Seite, kann ein Titel für das Spiel eingeben werden. Schließlich folgt ein Textbox, in der die Anweisungen für das Spiel geschrieben werden können. Hier ist es wichtig, dass jede Anweisung mit einem Punkt endet. Darunter gibt es drei Eingaben:

1. **Execute:** Wenn auf diesen Button geklickt wird, wird der Programmcode an den Server geschickt, der das Spiel nach Javascript kompiliert. Links wird dann nach kurzer Zeit das kompilierte Spiel angezeigt.
2. **No Music/With Music:** Select-Box, die angibt, ob Musik für das Spiel eingeschaltet werden soll oder nicht.
3. **No Image/With Chosen Image:** Select-Box, die angibt ob ein Hintergrundbild für das Spiel verwendet werden soll. Das Hintergrundbild kann rechts als Datei ausgewählt werden.

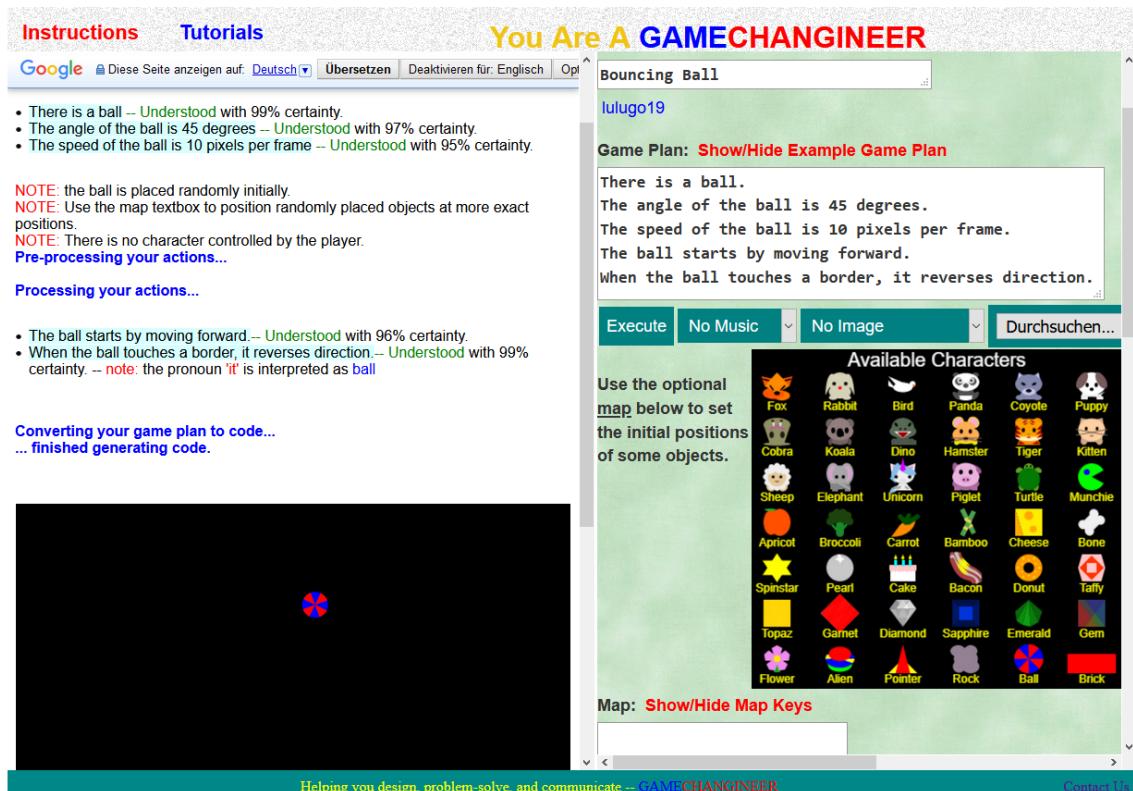


Abb. 4.39. Game-Changineer-Webseite

Schließlich folgt ein Bild aller Charaktere des Spiels und darunter eine weitere Textbox, bei der es sich um einen Map-Editor handelt. Jedem Charakter wird ein Zeichen zugeordnet

und über den Map-Editor können dann die Zeichen an beliebigen Positionen eingetragen werden, um Objekte im Spiel zu platzieren.

Es können Objekte auch über Anweisungen im Text auf der Map platziert werden, wobei jedoch keine genaue Position angegeben werden kann wie in Abb. 4.40 illustriert.

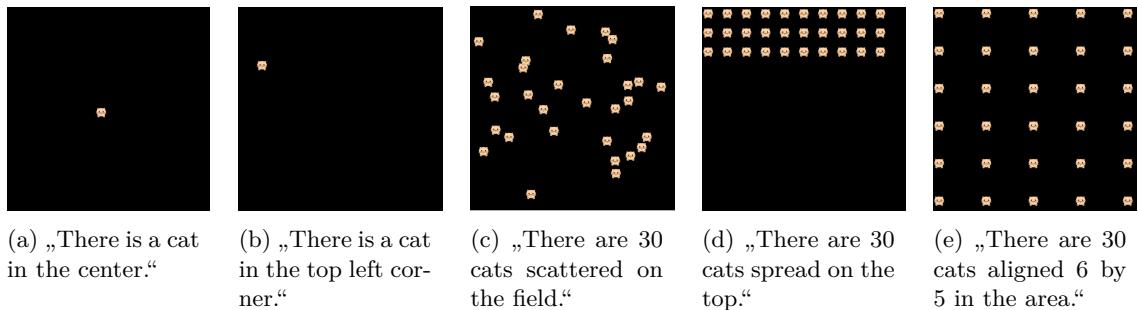


Abb. 4.40. Postionierung durch Anweisungen

4.5.3. Bouncing Ball

Der Code für *Bouncing Ball* in Game-Changineer besteht nur aus fünf Zeilen.

```
1 There is a ball.
2 The initial angle of the ball is 45 degrees.
3 The speed of the ball is 10 pixels per frame.
4 The ball starts by moving forward.
5 When the ball touches a border, it reverses direction.
```

Listing 4.38. Bouncing Ball in Gamechangineer

Über Zeile 1 wird dem Compiler mitgeteilt, dass es einen Ball gibt. Da keine Position angeben wird, wird der Ball zufällig auf der Karte platziert. In Zeile 2 und 3 wird der initialen Winkel und die Geschwindigkeit des Balls angegeben. In Zeile 4 wird angegeben, dass sich der Ball am Anfang nach vorne bewegt und in Zeile 5, dass der Ball seine Richtung umkehrt, wenn er den Spielfeldrand berührt.

Ein Game-Changineer-Programm lässt sich längst nicht so einfach schreiben wie es scheint, sondern kann ziemlich frustrierend sein, wenn genaue Anforderungen gestellt werden. Das liegt zum einen an der OONL, die nur bestimmte Satzstrukturen erlaubt und es auch solche Beschränkungen gibt, dass ein Nebensatz nur aktiv das Objekt des Hauptsatz verändern kann. So darf z.B. nicht geschrieben werden „When the ball touches a border, the carrot turn yellow“. Es wird jedoch immer darauf hingewiesen, was falsch geschrieben wurde ist. So wird folgende Diagnose für den falsch geschriebenen Satz ausgegeben:

```
-- DIAGNOSE: the characters in the antecedent and consequent must be the same for a
border event. Have you considered using attributes? Example:
- When the ball reaches a border, it becomes happy for 0.1 seconds.
- When the ball is happy, the carrot ...
```

Zum anderen liegt die Schwierigkeit an der Ungenauigkeit der Sprache und dass man sich eindenken muss, wie dies Sätze interpretiert werden können. Wenn z.B. in Zeile 4 statt „The ball starts by moving forward.“, „The ball moves forward.“ geschrieben wird, dann funktioniert das Programm nicht mehr. Wenn der Ball dann an den Spielfeldrand stößt,

kehrt er für eine Splitsekunde die Richtung um, ändert die Richtung dann jedoch wieder direkt auf 45 °. Mit dem Satz „The ball moves forward.“ wird nämlich angegeben, dass der Ball sich dauerhaft nach vorne bewegen soll. Wie weiß der Compiler wo vorne ist? Das erkennt er an den initialen Winkel. Nun stellt der Compiler sicher, dass der Ball sich durchgängig in die selbe Richtung bewegt.

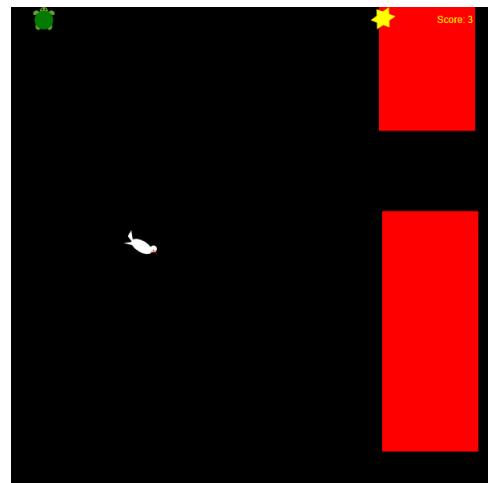
4.5.4. Flappy Bird

Um *Flappy Bird* in Game-Changineer zu implementieren wird eine Schildkröte als Timer verwendet. Die Schildkröte bewegt sich nach rechts und immer wenn sie den rechten Bildschirmrand erreicht, werden die Hindernisse gespawnt und die Schildkröte dann wieder nach links gesetzt. Um die Punktzahl immer dann hochzuzählen, wenn der Vogel ein Hindernis überwunden hat, gibt es dann noch einen Stern, der so platziert ist, dass die Schildkröte den Stern immer passiert, wenn der Spieler ein Hindernis überwunden hat.

Um die Objekte zu platzieren, wird der Map-Editor verwenden und es werden folgende Zeichen eingegeben:

```
-----s--T-
-----
-----
-----
-----
-----
-----
-----
-----b-----
```

(a) Map im Map-Editor



(b) Flappy Bird in Game-Changineer

Die Minus-Zeichen stehen für leeren Raum, das 'T' steht für die Schildkröte, das 's' für den Stern und das 'b' für den Vogel. Der Vogel wird auf der linken Seite vertikal mittig auf dem Spielfeld platziert. Da jedoch keine Objekte mehr unter dem Vogel platziert werden, muss die Map unterhalb des Vogels nicht bearbeitet werden.

Für den Turtle-Timer wird folgender Code geschrieben:

```
1 // Setting up the turtle timer.
2 The turtle moves right.
3 The speed of the turtle is 4 pixels per frame.
4 When the turtle reaches the right border, it wraps around
   and it becomes yellow for 0.1 second.
5 When the turtle is yellow, the turtle inserts[4, 600, 600, 0, 130] a brick
6   and the bricks are notified for 0.1 second.
7 When the brick is notified, it inserts[2, 400] a brick.
```

Aus Zeile 4 und 5 geht hervor, dass immer wenn die Schildkröte den rechten Spielrand erreicht, sie wieder ganz nach links gesetzt wird und dann für eine Zehntelsekunde gelb

wird. Nun kann der Zustand der gelben Schildkröte in den darunterliegenden *When*-Satz verwendet werden, um das Hindernis — in diesem Fall ein *brick*, welcher einfach ein Block ist — einzufügen.

Mittels dem Verb `insert` kann ein Objekt andere Objekte in das Spiel einfügen. In den eckigen Klammern werden Parameter eingegeben, welche die Einfügeposition des Objekts in der Welt bestimmen. Dabei gibt es folgende Möglichkeiten:

- `insert[0, <p>]`: Fügt ein Objekt `<p>` Pixel oberhalb des einfügenden Objekts ein.
- `insert[1, <p>]`: Fügt ein Objekt `<p>` Pixel rechts vom einfügendem Objekt ein.
- `insert[2, <p>]`: Fügt ein Objekt `<p>` Pixel unterhalb des einfügenden Objekts ein.
- `insert[3, <p>]`: Fügt ein Objekt `<p>` Pixel links vom einfügendem Objekts ein.
- `insert[4]`: Fügt ein Objekt an einer zufälligen Position hinzu.
- `insert[4, <xs>, <xe>, <ys>, <ye>]`: Fügt ein Objekt an einer zufälligen Position innerhalb eines Rechtecks hinzu, dessen Grenzen von den Parametern angegeben wird.

Die Schildkröte fügt mit `inserts[4, 600, 600, 0, 130] a brik` einen Block an der x-Position 600 und an einer zufälligen y-Position zwischen 0 und 130 ein. Nach dem Einfügen wird der Block sofort über `the bricks are notified for 0.1 second` benachrichtigt. Der benachrichtigte Block fügt dann unter sich mit einem Abstand noch einen Block ein (Zeile 8), sodass es zwei Blöcke gibt – ein Block oben, ein Block unten und eine Lücke in der Mitte.

Anschließend muss definiert werden, wie die Blöcke aussehen und wie sie sich verhalten sollen:

```
1 // Defining the bricks.
2 There are 0 bricks.
3 The size of the bricks is 300.
4 The angle of the bricks is 90.
5 The bricks are moving left at 200 pixels per frame.
6 When the position_x of the brick is less than 0, it disappears.
```

Um Objekte definieren zu können, die erst während der Laufzeit eingefügt werden, wird in Zeile 1 dem Compiler mitgeteilt, dass es Blöcke gibt. Da die Blöcke standardmäßig sehr klein sind und horizontal ausgerichtet sind, wird die Größe und der Winkel gesetzt, sodass die Blöcke größer und vertikal ausgerichtet sind. Dann wird angegeben, dass die Blöcke sich nach links auf den Vogel zu bewegen sollen. Hier fällt sicher auf, dass die Geschwindigkeit mit `200 pixels per frame` sehr hoch gesetzt ist. Doch beim Ausführen bewegen sich die Blöcke viel langsamer. Ich denke, dass liegt an einem Bug, der im Zusammenhang mit dem Ändern der Größe steht. In originaler Größe sind die gesetzten Geschwindigkeiten der Objekte nämlich wie erwartet. Mit der letzten Zeile wird angegeben, dass die Blöcke verschwinden sollen, wenn sie sich außerhalb des linken Spielrands bewegen.

Die Anweisungen zum Erhöhen der Punktzahl sind Folgende:

```
1 // Increasing score
2 When the turtle touches the spinstar, the spinstar becomes blue for 0.1 second.
3 When the spinstar is blue, the score increases by 1.
```

Wenn die Schildkröte den Stern berührt wird der Stern für eine Zehntelsekunde blau und die Punktzahl erhöht sich um eins. Dies wurde so umständlich gemacht weil durch folgende simplere Anweisung ...

- 1 When the turtle touches the spinstar, the score increases by 1.

... der Punktzahl sich um mehr als eins erhöht, weil die Schildkröte für eine gewisse Zeit den Stern durchgängig berührt und für jeden Frame in dieser Zeit, die Punktzahl erhöht wird. Der *Hack* oben funktioniert jedoch. Hier wird die Punktzahl nur um eins erhöht. Rein logisch von den Sätzen betrachtet ist es schwer eine Erklärung zu finden, warum der *Hack* funktioniert. Sätze können unterschiedlich interpretiert werden und wie genau der Compiler die Sätze interpretiert ist oft unerwartet.

Zum Schluss müssen noch Anweisungen angegeben werden, wie der Vogel sich verhält:

- 1 The player controls the bird.
- 2 The bird falls.
- 3 The size of the bird is 40.
- 4 The bird is fit.

In Zeile 1 wird dem Compiler mitgeteilt, dass der Spieler den Vogel kontrollieren kann. Dann wird angegeben, dass der Vogel durchgängig runterfällt. Die Größe wird dann geändert und danach wird dem Vogel das Prädikat *fit* zugewiesen. Wenn der Vogel ein Hindernis berührt, dann ist dieser nicht mehr *fit* und er kann nicht mehr kontrolliert werden.

Nun zum letzten Stück des Codes:

- 1 When space is pressed, the bird becomes energized for 0.1 second and the angle of the bird equals 0.
- 2 When the fit bird is energized, it jumps by 5 pixels per frame.
- 3 When the direction_y of the bird is less than 0,
- 4 the angle of the bird decreases by 3.
- 5 Otherwise, the angle of the bird increases by 3.
- 6 When the angle of the bird is greater than 80, the angle of the bird equals 80.
- 7 When the bird collides with a brick, it is not fit.
- 8 When the bird is not fit, it moves down at 600 pixels per frame.
- 9 When the bird is not fit, the angle of the bird increases by 1.
- 10 When the bird touches the bottom border, game over.
- 11 When the bird lands on a brick, game over.

Wenn die Leertaste gedrückt wird, dann wird dem Vogel kurz Energie verliehen (*energized*). Diese Energie kann er jedoch nur nutzen, wenn er auch *fit* ist. Dann springt er nämlich (Zeile 2). Die Codezeilen 3 bis 5 lassen den Vogel rotieren, sodass er sich nach unten neigt, wenn er fällt und nach oben, wenn er fliegt.

Wenn der Vogel mit einem Block kollidiert, ist er nicht mehr *fit*. Wenn er nicht *fit* ist bewegt er sich sehr schnell nach unten und neigt sich auch nach unten. Das Spiel ist vorbei, wenn der Vogel den unteren Spielfeldrand oder einen Block berührt.

Das Anzeigen von Game-Over und das Neustarten braucht nicht programmiert zu werden. Diese Funktionalität baut Game-Changineer schon automatisch in das Spiel ein, wie Abb. 4.42 zeigt.



Abb. 4.42. Game-Changineer: Game-Over

4.5.5. Bewertung

Einfachheit der Implementierung von Computerspielen

Es konnten nicht alle Anforderungen alle Spieler erfüllt werden. Bei Tic-Tac-Toe wird nicht der Gewinner oder Unentschieden angezeigt. Außerdem kann auch noch weiter gespielt werden, wenn ein Spieler gewonnen hat. Es war nicht möglich den Gewinnzustand zu überprüfen. Nur wenn Unentschieden ist, wird das Spiel automatisch beendet. Dies geschieht durch die Zeile „When all the topazes are gone, game over.“ (s. Anhang C.23 Z. 7). Bei *Flappy Bird* konnten die Sound-Effekte nicht implementiert werden. Tamagochi konnte überraschenderweise vollständig implementiert werden.

Die Implementierungen waren aufwendig. Dies lag zum einen an der Unschärfe der natürlichen Sprache, welche auf verschiedene Weisen interpretiert werden kann und zum anderen, dass Game-Changineer bestimmte Restriktionen erzwingt, wie ein Satz zu schreiben ist und diese Restriktionen sich ziemlich willkürlich anfühlen. Außerdem sind Bugs aufgetreten – das Anklicken von Objekten z.B. funktioniert nicht so gut. Ich habe das Game-Changineer-Team öfters um Hilfe gefragt, wenn ich nicht wusste, wie ich etwas lösen konnte (s. Anhang A.2).

Auf der anderen Seite können simplere Spiele ziemlich schnell implementiert werden, da einige Zeilen ausreichen, um ein simples Spiel zu erstellen. Durch Herumexperimentieren können schnell verschiedene Spieleideen ausgetestet werden.

Viele der Implementierungsdetails wurden über *Hacks* umgesetzt. Ein *Hack* ist z.B. eine Schildkröte als Timer zu benutzen. Ein anderer *Hack* ist, dass Attribute von Objekten für eine Splitsekunde eingeschaltet werden, um eine Aktion einmalig auszulösen. So wird die Schildkröte in *Flappy Bird*, wenn sie rechts den Spielfeldrand berührt kurz gelb, damit sie dann einen Block spawnnen kann.

Über den Map-Editor können Objekte auf der Map platziert werden. Dabei ist die *Usabi-*

lity nicht so gut, da einfach Zeichen in ein Textfeld eingegeben werden müssen. Außerdem können Objekte auch nicht rotiert werden. Jedoch ist dies die einzige Möglichkeit, um die Position von Objekten präzise festzulegen, da über die Sprache nur eine ungefähre Position angegeben werden kann.

Das Koordinatensystem ist rechtshändig und die Drehrichtung im Uhrzeigersinn. Dies passt so zum rechtshändigen Koordinatensystem. Der Richtung eines Objekts kann allerdings nicht über einen negativen Winkel angegeben werden. Der Ankerpunkt bei Objekten ist die obere linke Ecke.

Spiele können zwar mit einer nicht änderbaren Hintergrundmusik erstellt werden, jedoch können keine eigenen Klänge geladen und abgespielt werden.

Viele der Sprites sind animiert, es können jedoch keine eigenen Animationen und Sprites verwendet werden.

Die Abstraktion des *Game Loops* ist sehr hoch. Das Spiel wird mit natürlicher Sprache auf einer sehr hohen Ebene beschrieben und es gibt keine Möglichkeit in den *Game Loop* einzugreifen.

Erlernbarkeit nach Konstruktionismus

Neben Computerspielen können in Game-Changineer auch Simulationen wie zu wissenschaftlichen Experimenten oder Kunst und Animationen erstellt werden. Es gibt zudem noch ein weiteres Projekt von den Machern von Game-Changineer mit dem Namen *Song-Inspirineer*, wo Musik komponiert werden kann.

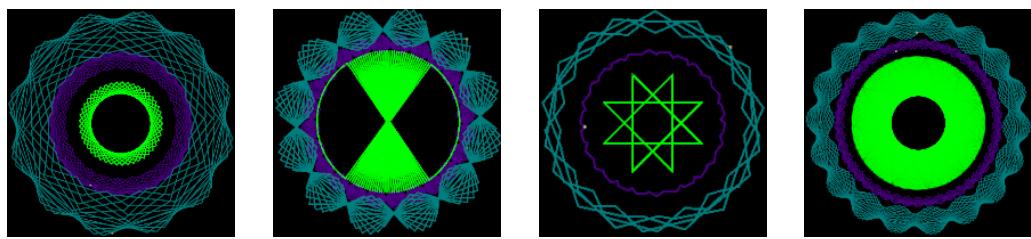
In Game-Changineer können *Turtle*-Grafiken erstellt werden, indem Objekten eine Spur und Anweisungen zur Bewegung gegeben werden, wodurch sie Muster erzeugen. Bei folgendem Code, kann durch das Ändern der Parameter eine Reihe von interessanten Bildern erstellt werden (s. Abb. 4.43).

```

1 The size of the turtle is 3.
2 The turtle is traced with green.
3 When the turtle is not dead, it moves forward 100 pixels.
4 When the turtle is not dead, the angle of the turtle increases by 88 degrees.
5
6 The size of the rabbit is 3.
7 The speed of the rabbit is 15 pixels per frame.
8 The rabbit is traced with indigo.
9 The rabbit revolves around the turtle.
10
11 The size of the hamster is 3.
12 The speed of the hamster is 30 pixels per frame.
13 The hamster is traced with teal.
14 The hamster revolves around the turtle.

```

Listing 4.39. Game-Changineer: Code für *Turtle*-Grafiken

Abb. 4.43. Game-Changineer: *Turtle*-Grafiken

Es können nur die vorgegebenen Sprites in Game-Changineer verwendet werden. Abgesehen vom Hintergrundbild, können keine eigenen Medien eingebunden werden.

Die Umgebung ist relativ selbstoffenbarend. Nachdem eingeführt wurden ist, wie Sätze geschrieben werden können, kann viel ausprobiert und herumexperimentiert werden. Unten auf der Seite werden Vokabeln für Aktionen, Richtungen, Farben und Spielmechaniken gegeben, die einfach ausprobiert werden können. Außerdem können auch andere Wörter ausprobiert wird. Wenn ein Wort nicht verstanden wird, dann wird eine Warnung und ein Vorschlag ausgegeben, wie das Problem behoben werden kann.

Als Webanwendung handelt es sich auch um eine sehr sichere und in sich geschlossene Umgebung. Da es keine automatische Speicherung gibt, muss jedoch daran gedacht werden, erstellte Projekte zu speichern.

Inkrementelle Einführung

Die Konzepte in Game-Changineer können nach und nach eingeführt werden. So sind die Tutorials gegliedert in Beginner-, Mittel- und Fortgeschrittenenstufe (s. [Gam21]).

Das Programm „Hallo Nutzer“ kann in Game-Changineer nicht erstellt werden, da es keine Möglichkeit einer Eingabe sowie einer textuellen Ausgabe während der Laufzeit gibt. Außerdem existiert kein String-Datentyp in Game-Changineer.

Intuitive Syntax und Natürlichsprachlichkeit

Die Syntax ist durch die Verwendung der natürlichen Sprache sehr intuitiv. Bei einfachen Spielen, lässt sich die Syntax wie eine Spielbeschreibung lesen. Bei komplexeren Spielen wie Tamagochi ist dies nicht mehr gegeben, da *Hacks* eingesetzt werden. Zudem können Adjektive ungenau verwendet werden, um einen bestimmten Zustand auszudrücken, der dann auch angezeigt wird. Wenn bei Tamagochi das Haustier z.B. grün ist, dann ist es krank und wenn es blau ist, dann schlafes (s. Anhang C.27).

Game-Changineer gibt es nur in der englischen Sprache. Die Syntax ist somit nicht internationalisiert.

Feature Uniformity und Orthogonalität

Es gibt in Game-Changineer viele verschiedene Arten, um das Gleiche auszudrücken (Synonyme). Verschiedene Verben werden auf ein und die selbe Aktion zurückgeführt. Zum Beispiel können die Verben „touch“, „collide“, „hit“ alle verwendet werden, um zu überprüfen, ob zwei Objekte miteinander kollidieren.

Um noch ein erweitertes Beispiel zu geben, kann sich folgender Code angeschaut werden, der ein Schaf ganz schnell von blau nach rot blinken lässt:

```
1 There is a sheep.
2 The initial state of the sheep is 0.
3 When the state of the sheep is 0, then the sheep turns red.
4 When the sheep whose state is equal 0 is red, it becomes blue.
```

Listing 4.40. Game-Changineer: Blinkendes Schaf

Für den dritten Sätzen hätte auch geschrieben werden können: „When the state of the sheep is equal to 0, it becomes red.“ Es gibt mehrere Wörter um ein Boolesches Attribut für ein Objekt einzuschalten: „turns“, „becomes“, „is“, „changes to“.

Der vierte Satz kann auch komplett folgendermaßen umgeschrieben werden: „When the state of the red sheep is equals 0, it becomes blue.“ So gibt es sehr viele Möglichkeiten, wie Sätze formuliert werden können. Im Nebensatz kann ein Objekt des Hauptsatzes entweder über „it“ referenziert oder es kann der Name des Objekts verwendet werden.

Es konnten keine Homonyme in Game-Changineer identifiziert werden.

Die Funktionsmenge in Game-Changineer ist eher groß. Es gibt viele Satzkonstruktionen, welche neue Funktionen einführen. Beispielweise kann über `with <p> percent probability` eine Aktion nur mit einer bestimmten Wahrscheinlichkeit ausgelöst werden, oder über `for <n> seconds`, kann ein Attribut für eine bestimmte Zeit eingeschaltet werden. Ein anders Beispiel ist die `insert[...]`-Syntax mit der Objekte zur Laufzeit eingefügt werden können.

Die `insert`-Syntax ist auch ein Beispiel für eine Inkonsistenz in der Syntax. Alle Elemente der Syntax außer der `insert`-Syntax sind natürlichsprachig. Die `insert`-Syntax verwendet eckige Klammern in denen Zahlenparameter eingetragen werden müssen. So ähnlich würde man es auch in typischen Programmiersprache machen. Warum konnte die Syntax des Einfügens nicht über natürliche Sprache, wie z.B. „the rabbit inserts a ball 20 pixels below“, implementiert werden?

Unterstützung der wichtigsten Programmierkonzepte & Datenstrukturen

Es gibt keine explizite Schleife in Game-Changineer. Die Anweisungen sind jedoch im *Game Loop* eingebettet. Deswegen kann der *Game Loop* selbst als Schleife verwendet werden. Hier ist beispielsweise ein Programm, das die Summe der Zahlen 1 bis 100 berechnet:

```
// Initialisiere.
There is a sheep.
The initial sum of the sheep is 0.
The sum of the sheep is displayed.
The initial num of the sheep is 1.

// Berechne die Summe der Zahlen 1 bis 100.
The sum of the sheep increases by the num of the sheep.
The num of the sheep increases by 1.
When the num of the sheep is greater 100, game over.
```

Listing 4.41. Game-Changineer: Summe von 1 bis 100

Bedingungen können über `When ... , then`-Sätze erstellt werden.

Game-Changineer unterstützt keine prozedurale Programmierung. Dadurch kann die *Fibonacci-Reihe* nur iterativ und nicht rekursiv implementiert werden, wobei die iterative Version aufwendig und unintuitiv ist (s. Anhang C.21).

Game-Changineer unterstützt gar keine Datenstrukturen. Man könnte noch argumentieren, dass eine Liste unterstützt wird, da mehrere Objekte eines gleichen Typs erstellt werden können und dann über die eckigen Klammern auf bestimmte Elemente zugegriffen werden kann:

```
There are 10 rabbits.  
Rabbit[0] moves down. // Der erste Hase bewegt sich nach unten  
Rabbits[1,3] move right. // Der zweite und vierte Hase nach rechts  
Rabbits[4,9] move left. // Der fünfte und der 10 nach links
```

Zudem können neue Objekte eingefügt und entfernt werden.

Programmierumgebung (IDE)

Die Programmierumgebung in Game-Changineer ist einfach gehalten, könnte von der *Usability* jedoch noch etwas besser sein. Der Map-Editor ist ein einfacher Texteditor, die Buttons sind etwas unübersichtlich platziert und es gibt blinkenden Text. Das Design ist nicht modern und sieht so aus, wie von frühen ersten Webseiten.

Die Interaktivität ist nicht so hoch. Da der Programmcode serverseitig kompiliert wird, muss ca. 1 Sekunde gewartet werden, bis es eine Ausgabe gibt. Auch Fehlermeldungen werden erst angezeigt, nachdem das Programm kompiliert wurden ist.

Es gibt keine gute Möglichkeit, um zu Debuggen. Um Anweisungen zu testen, ist das Beste, das man machen kann, Objekte auf bestimmte Farben zu setzen, sodass man sieht, ob die Anweisung korrekt ist.

Es gibt keine direkte Möglichkeit, um Programme zu teilen. Es gibt allerdings ein *Community-Showcase*, in dem viele Projekte von Benutzern ausgestellt sind. Möglicherweise kann man sein Projekt auch dort ausstellen lassen.

Von modernen IDE-Funktionen her, gibt es nur eine *Code-Completion*. Bei der Eingabe von Sätzen, werden mögliche Vorschläge angezeigt.

Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplin (*Clean Code*)

Code kann in Game-Changineer nicht in Modulen organisiert werden. Es gibt überhaupt keine Möglichkeit, um Code zu organisieren. Ein Game-Changineer-Programm ist in sich geschlossen und kann nicht mit anderen Programmen verbunden werden. Auch die anderen Eigenschaften von *Clean Code* erfüllt Game-Changineer nicht.

Immerhin gibt es eine Art *Style-Guide* mit Tipps wie möglichst klare Spielpläne geschrieben werden. Dabei sind die drei Tipps (vgl. [Gam21]):

1. Verwende keine mehrdeutigen Wörter.

2. Versuche Sätze einfach zu halten.
3. Konzentriere dich auf einen einzelnen *Frame*.

Fehlermeldungen

Die Fehlerausgabe in Game-Changineer ist sehr ausgeklügelt. Es gibt fünf verschiedene Arten von Ausgaben:

- NOTE
- CHECK
- WARNING
- PROBLEM SOLVE
- DIAGNOSE

Ich kann keinen semantischen Unterschied zwischen 'CHECK' und 'NOTE' erkennen. Beide geben Nachrichten aus, welche den Benutzer auf Besonderheiten aufmerksam machen oder erklären, wie ein bestimmter Satz interpretiert wird. Ebenso kann ich auch keinen semantischen Unterschied zwischen 'PROBLEM SOLVE' und 'DIAGNOSE' erkennen.

Warnungen werden für Dinge ausgegeben, die eventuell Probleme bereiten und vom Nutzer nicht beachtet wurden. 'PROBLEM SOLVE'- und 'DIAGNOSE'-Nachrichten werden für Dinge ausgegeben, für die der Nutzer an seinen Anweisungen etwas ändern muss, um das Programm ausführbar zu machen. Bei dieser Art von Nachrichten werden zudem Beispiele angezeigt, wie ein Satz zu konstruieren wäre.

Für jeden Satz wird ausgegeben mit wie viel Prozent Wahrscheinlichkeit, dieser verstanden wurden ist und jede diagnostische Nachricht wird pro Satz ausgegeben, sodass sie eins zu eins Sätzen zugeordnet werden können.

Bei einigen Nachrichten ist eine *Text-To-Speech*-Ausgabe vorhanden, bei der die Nachricht vorgelesen wird. Es ist jedoch nicht ersichtlich warum bei einigen Nachrichten diese Funktion vorhanden ist und bei anderen nicht. Dies ist etwas störend.

Insgesamt ist die Ausgabe der diagnostischen Nachrichten sehr verbos und es werden zu viele Informationen auf einmal angezeigt. Besser wäre es, wenn mehr Informationen bei Bedarf angezeigt werden könnten und/oder diese übersichtlicher dargestellt würden.

Das Vokabular der Fehlermeldungen ist sehr technisch und kompliziert, was folgende Beispiele zeigen:

```
-- NOTE: The execution order of turtle and brick may result in
       setting the Boolean attribute of the brick late by one frame cycle.

-- NOTE: the Boolean attribute in the consequent will be updated in
       the cycle/frame immediately after the antecedent event.
```

Die Sprache der Fehlermeldungen ist neutral und Fehlermeldungen sind nicht lokalisiert.

Ein hohes Abstraktionsniveau für Datentypen

Es gibt keinen String-Datentypen, der *Unicode* unterstützen könnte. Im Editor selbst können nur ASCII-Zeichen eingegeben werden. Wenn ein ungültiges Zeichen eingegeben wird,

ist die Umrandung des Editors rot und bei der Ausführung wird ein Fehler ausgegeben. Für die Zahlen werden Javascript-Zahlen (64-Bit-Fließkommazahlen) verwendet.

Anwendungsbereiche und Interoperabilität mit Technologien und anderen Programmiersprachen

Es gibt nicht noch mehr Anwendungsbereiche, als die schon im Abschnitt 4.5.5 genannten.

Dokumentation & Community Support

Die Dokumentation und Tutorials wurden in einem erstellt (s. [Gam21]). Die Dokumentation ist ausführlich, könnte jedoch etwas übersichtlicher und navigierbarer gestaltet sein.

Zudem gibt es eine Reihe von Video-Tutorials und eine große Anzahl an Beispiel-Programmen.

Außerdem gibt es ein Quiz, in dem man einfache Sätze nach und nach eingibt, um ein simples Spiel zu erstellen oder ein Spiel zu erweitern. Nach jedem erfolgreichen Schritt, wird die nächste Aufgabe angegeben.

Auch bei *Hour of Code* [Cod15] ist Game-Changineer vertreten. Dabei können drei verschiedene Spiele unter Anleitung programmiert werden.

Es gibt keine offiziellen Gruppen oder Foren. Das Game-Changineer-Team kann allerdings über eine Form angeschrieben werden und es ist sehr hilfsbereit, wie ich gemerkt habe.

5. Zusammenfassung und Fazit

5.1. Vergleich und Zusammenfassung der *Teaching Languages*

Im Anhang ist eine Rangliste der *Teaching Languages* pro Bewertungsbereich enthalten (s. B.6). Die Rangliste zeigt sehr gut auf, dass jede *Teaching Language* sowohl Stärken als auch Schwächen hat. Es gibt keine *Teaching Language*, die in allen Bereichen besonders gut oder schlecht abschneidet. So belegt Pyret in dem Bereich „Programmierumgebung (IDE)“ den ersten Platz und den letzten Platz im Bewertungsbereich „Intuitive Syntax und Natürlichsprachlichkeit“. Scratch belegt den ersten Platz im Bereich „Konstruktionismus“ und zusammen mit Game-Changineer den letzten Platz im Bereich „Clean Code“.

Jede der untersuchten *Teaching Languages* hat besondere Stärken, welche einzigartig für diese sind, und welche die Entwicklung einer weiteren *Teaching Language* inspirieren kann. Die Stärken der *Teaching Languages* sind jeweils auf die Zielgruppe dieser zugeschnitten. Zudem hat jede *Teaching Language* auch Schwächen, welche entweder aus dem Kompromiss mit den Stärken der Sprache oder aus Implementierungsdetails resultieren. Das Entwickeln einer *Teaching Language* ist ein höchst komplexes Projekt und es müssen auch die Kosten des Aufwands für eine bestimmte (technische) Entscheidung berücksichtigt werden. So gibt es bei einigen der *Teaching Languages* noch Punkte, die verbessert werden könnten.

Bei Pyret ist eine der Stärken der interaktive Bereich, welcher als REPL dient und zudem die Ergebnisse der Unit-Tests, Fehlermeldungen und Ausgaben ausgibt, wobei bei der Ausgabe und Darstellung besonders menschliche Faktoren hervorragend berücksichtigt werden. Zudem ist die Integration von Unit-Tests in die Sprache selbst ein hervorstechendes Merkmal. Hierdurch wird die Hürde, Unit-Tests zu schreiben, herabgesetzt und das Schreiben von Unit-Tests kann direkt von Anfang an als integraler Bestandteil der Software-Entwicklung gelehrt werden. Pyret erlaubt das Erlernen von professionellen Programmierpraktiken und es empfiehlt sich ein Einsatz an Hochschulen. Es könnte beispielsweise in einem Kurs wie „Paradigmen der Programmierung“ verwendet werden, um funktionale Programmierung anhand der grafischen interaktiven Programmierung interessanter zu gestalten.

Die große syntaktische Auswahl in Pyret, der Fokus auf *Clean Code* und die funktionale Programmierung, macht diese jedoch für absolute Anfänger schwerer zu erlernen. Bei Pyret überwiegt der Instruktionismus stark den Konstruktionismus und ein exploratives Erlernen der Sprache ist daher nicht besonders gut möglich.

In Quorum ist Inklusion eines der hervorstechenden Merkmale. Es wurde besonders darauf geachtet, Menschen mit Behinderungen miteinzubeziehen. So unterstützt die IDE *Quorum Studio* Screen-Readers und Braillezeilen. Außerdem wird über das Kommando `say` *Text-To-*

Speach als Standardfunktion unterstützt. Des Weiteren ist die Syntax intuitiv und enthält keine schwierig zu tippenden Zeichen. Zusätzlich ist die Standardbibliothek sehr umfangreich, sodass die verschiedensten Projekte umgesetzt werden können und die Komplexität eines Paket-Managers wegfällt. Bei der Sprache selbst, fällt besonders die Reduzierung von objektorientierten Funktionalitäten auf die Wesentlichsten auf, sodass die Sprache „klein“ bleibt, jedoch immer noch relativ mächtig ist.

Die Schwäche von Quorum ist die Programmierumgebung. *Quorum Studio* als selbsterstellte IDE hat *Usability*-Probleme, welche unter anderem durch eine geringere Performance hervorgerufen werden. Der *Web-Runner* ist zudem sehr minimal und die Interaktivität wird durch die serverseitige Kompilierung eingeschränkt.

Scratch zeichnet sich durch seine blockbasierte visuelle Programmiersprache aus, welche das intuitive Erstellen eines Programms fördert, da ein Programm wie ein Puzzle zusammengesetzt wird. Außerdem wird das Element der Fehlermeldungen durch eine *Soft-Fail*-Strategie komplett eliminiert und die Programmierumgebung weist als *Livecode*-Umgebung eine hohe Interaktivität auf. Bei Scratch überwiegt der Konstruktionsmus stark den Instruktionismus. Programmierkonzepte können durch die große Intuitivität der Sprache und Umgebung selbstständig erkundet werden. Zusätzlich ist die Community von Scratch riesig und kreative Projekte können einfach geteilt und erweitert werden.

Scratch scheitert allerdings an dem Anspruch einer professionellen Programmiersprache, sauberen Code mit dieser zu schreiben und Code in Module zu organisieren. Da es nur globale Variablen gibt und Funktionen keinen Rückgabetyp haben, können computertheoretische Konzepte wie ein Programm zur Ausgabe der *Fibonacci*-Folge nur sehr schlecht umgesetzt werden.

In ToonTalk wird fortgeschrittene Programmierung in einem Videospiel ermöglicht. Programmietechnische Abstraktionen werden auf Elemente in einer virtuellen Welt abgebildet. Als animierte Programmiersprache und durch *Programming by Demonstration*, können Programmierkonzepte selbstständig erkundet werden.

Diese Stärken von ToonTalk bringen ebenso Nachteile mit sich. Bei herkömmlichen Programmiersprachen kann einzelner Code aus einer „Vogelperspektive“ heraus modifiziert und geändert werden. Bei ToonTalk müssen Roboter neu trainiert werden, wenn ein Fehler gemacht wurde. Dabei muss dann auch Code, der vorher schon programmiert wurde, erneut erstellt werden. Zudem gibt es nicht diese „Vogelperspektive“ aus der man Code an beliebigen Stellen verändern kann, vielmehr befindet man sich im Code selbst. Das Programmierparadigma des *Concurrent Constraint Logic Programming* (CCLP) ist außerdem nicht sehr populär und die integrierte *Concurrency* bringt auch Nachteile wie Performanceeinbußen mit sich, wie die Implementierung der rekursiven *Fibonacci*-Funktion zeigt. Die Originalität von ToonTalk ist wahrscheinlich mitunter ein Grund, warum es nicht besonders populär ist.

Game-Changineer beeindruckt mit seiner natürlichen Programmiersprache, mit der mit wenigen Zeilen Code interessante Spiele erstellt werden können. Simple Spielen können sehr einfach entwickelt werden und Game-Changineer lädt den Nutzer dazu ein, herum zu experimentieren und interessante Spielideen umzusetzen.

So ist es eher unintuitiv, dass Game-Changineer den vierten Platz im Bereich „Einfachheit der Implementierung von Computerspielen“ belegt, obwohl Game-Changineer genau dafür gemacht ist, Computerspiele zu erstellen und zahlreiche interessante von Kindern erstellte Spiele auf der Webseite ausgestellt sind. Das liegt daran, dass Game-Changineer besonders gut daran ist, simple Computerspiele zu erstellen, welche spezifische Mechaniken wie die Bewegung und die Kollision von Sprites umfassen. Spiele, die allerdings andere Mechaniken haben (wie z.B. Tic-Tac-Toe) können in Game-Changineer nicht implementiert werden.

Die Natürlichsprachlichkeit von Game-Changineer bringt zudem auch Nachteile mit sich, da es viele Inkonsistenzen in der Syntax gibt und es an der Präzision fehlt, Problemlösungen formal zu definieren und nach bestimmten Anforderungen umzusetzen. Game-Changineer eignet sich somit nur dafür, Programmierung und *Computational Thinking* auf eine leicht erlernbare und kreative Art und Weise einzuführen und skaliert nicht in die professionelle Software-Entwicklung.

Zuletzt möchte ich noch auf die kognitive Belastung zu sprechen kommen. Die kognitive Belastung wurde explizit nicht bewertet, da diese schwer anhand von Bewertungskriterien festzulegen ist und der persönliche *Bias* eine objektive Bewertung leicht verzerren könnte. Scratch und ToonTalk beispielsweise eliminieren die Notwendigkeit Syntax auswendig zu lernen. Zudem eliminiert Scratch Fehlermeldungen und ToonTalk reduziert diese erheblich. Dadurch wird die intrinsische kognitive Belastung reduziert und die Sprache sollte einfacher zu erlernen sein. Für jemanden wie mich jedoch, der viel Programmiererfahrung hat und sehr vertraut mit den Elementen der Syntax und Fehlermeldungen von typischen Programmiersprachen ist, waren diese Sprachen schwerer zu erlernen, da diese Sprachen von meinem kognitiven Modell abwichen.

5.2. Reflexion von GermanSkript

GermanSkript ist eine interpretierte, objektorientierte, statisch typisierte Programmiersprache, in die eine Teilmenge der deutschen Sprache eingewoben ist (vgl. [Gob21a]). GermanSkript versucht das Problem der Sprachbarriere in der Programmierung für Deutschsprachige zu beheben.

Das Programm „Hallo Nutzer“ sieht in GermanSkript folgendermaßen aus:

```
schreibe die Zeichenfolge "Hallo, wie heißt du?:"  
der Name ist lese  
schreibe die Zeile "Hallo #{Name}!"
```

Listing 5.1. GermanSkript: „Hallo Nutzer“

Und die rekursive *Fibonacci*-Funktion kann in GermanSkript folgendermaßen erstellt werden:

```
Verb(Zahl) fibonacci von der Zahl:  
    wenn die Zahl <= 0 ist:  
        werfe einen Fehler  
        mit der Fehlermeldung "Die Zahl darf nicht kleiner gleich 0 sein"
```

```
wenn die Zahl kleiner 2 ist: gebe 1 zurück.

gebe (fibonacci von der Zahl - 1) + (fibonacci von der Zahl - 2) zurück

.

// Ausgabe und Funktionsaufruf
schreibe die Zahl (fibonacci von der Zahl 5)
```

Listing 5.2. GermanSkript: rekursive *Fibonacci*-Funktion

Der rekursive Aufruf der *Fibonacci*-Funktion ist problematisch, da wenn der Leerraum zwischen - und der Zahl vergessen wird (also so: fibonacci von der Zahl -1) dann ist dies äquivalent zu dem Aufruf fibonacci(-1) und nicht fibonacci(Zahl - 1). Parameternamen werden in GermanSkript bei einem Funktionsaufruf nämlich immer genannt, doch es gibt die Möglichkeit einen Variablenamen mit diesen zu verschmelzen. Ohne Verschmelzung würde der rekursive Aufruf folgendermaßen geschrieben werden: fibonacci von der Zahl (Zahl-1).

Insgesamt hat GermanSkript viele Funktionen, wobei diese von den Allzweckprogrammiersprachen wie Kotlin und Rust stark inspiriert wurden sind. So unterstützt GermanSkript Klassen, Schnittstellen, *Generics* und auch Lambdas und funktionale Programmierung mit Operationen wie `map`, `filter` und `reduce`.

Ich denke genau diese Fülle an Funktionen macht GermanSkript zu einer schlechteren *Teaching Language*, da das Kriterium *Feature Uniformity* und ein minimaler Funktionssatz außer Acht gelassen wurde. Mein Anspruch war es eine „deutsche“ Allzweckprogrammiersprache zu erstellen. GermanSkript als *Teaching Language* war erst ein Nachgedanke. Der einzige Vorteil von GermanSkript als *Teaching Language* ist die Natürlichsprachlichkeit. Aber genau diese macht GermanSkript zu keiner guten Allzweckprogrammiersprache, da GermanSkript-Code sehr verbos und wie an dem Beispiel der *Fibonacci*-Funktion gezeigt werden kann, auch schnell mehrdeutig sein kann, was nicht den Anspruch einer professionellen Programmiersprache erfüllt.

Aus dem gescheiterten Versuch eine „deutsche“ Allzweckprogrammiersprache mit hoher Natürlichsprachlichkeit zu erstellen, kann die Einsicht gewonnen werden, dass natürlichsprachliche Programmiersprachen nicht gut als Allzweckprogrammiersprachen geeignet sind. Schon Dijkstra formulierte 1979 in seinem Aufsatz „On the foolishness of natural language programming“, dass natürliche Sprachen unpräzise sind und formale Sprachen einen hohen Wert haben:

„The virtue of formal texts is that their manipulations, in order to be legitimate, need to satisfy only a few simple rules; they are, when you come to think of it, an amazingly effective tool for ruling out all sorts of nonsense that, when we use our native tongues, are almost impossible to avoid.“ [Dij79]

Projekte wie Game-Changineer oder z.B. auch *Inform* (s. [Nel20]) – eine auf Natürlichsprachlichkeit basierende Programmiersprache, zum Erstellen von *Textadventures* – zeigen, dass natürlichsprachige Programmiersprachen auf jeden Fall einen wichtigen Platz in der Sparte von DSLs einnehmen und sehr effektiv sein können.

Zudem hat sich die Technologie seit Dijkstra's Aufsatz weiterentwickelt und neue Entwicklungen in *Machine Learning* und *Natural Language Processing* können neuartige Anwendungen in diesem Bereich ermöglichen. Ein Beispiel dafür ist GPT-3, ein mächtiges *Machine Learning*-Modell für Sprachen von *OpenAI*, welches 2020 veröffentlicht wurde und u. a. ermöglicht, Code aus sprachlichen Beschreibungen zu generieren (vgl. [Dal21]).

Eine *Teaching Language* zu erstellen, die lokalisiert und natürlichsprachlich ist, halte ich nach wie vor für eine sinnvolle Idee und ich habe schon einige Ideen, die von den in dieser Arbeit untersuchten *Teaching Languages* inspiriert sind.

5.3. Diskussion & Fazit

Die aufgestellten Bewertungsbereiche und Bewertungskriterien könnten möglicherweise noch besser strukturiert sein und ergänzt werden. Es fällt auf, dass einige Bewertungsbereiche sehr überlappen: Wie z.B. der Bewertungsbereich „Konstruktionismus“ und „Anwendungsbereiche und Interoperabilität“. Beim ersten Bereich wird geschaut, wie viele Mikrokosmen es gibt und beim Zweiten, was es für Anwendungsbereiche gibt.

Zudem sind einige Bereiche sehr klein wie „Inkrementelle Einführung“ oder „hohe Abstraktionsniveau für Datentypen“ und andere größer, wie z.B. die Bewertung der Programmierumgebung. Das Bewertungskriterium der Interaktivität bei der Programmierumgebung ist ein ziemlich Wichtiges bei *Teaching Languages* und könnte sogar als alleinstehender Bewertungsbereich extrahiert werden.

Weiterhin könnten noch Bewertungsbereiche, wie z.B. die Einfachheit des Übergangs in industrielle Programmiersprachen, hinzugefügt werden. Bei diesem Bereich würde auch das Programmierparadigma eine große Rollen spielen. So würden objektorientierte Sprachen bevorzugt werden, da dieses Paradigma in der Industrie großflächig eingesetzt wird.

Außerdem erfolgte die Gewichtung der einzelnen Bewertungskriterien in dem Bewertungsbogen nach eigener Intuition. Kleinere Änderungen der Gewichtungen, könnten schon eine andere Rangfolge erzeugen. Zudem sind einige Bewertungsbereiche sehr abstrakt (wie z.B. „Erlernbarkeit nach Konstruktionismus“) und somit schwer bewertbar. So kann nicht mit Sicherheit gesagt werden, dass die eine *Teaching Language* wirklich besser als die andere *Teaching Language* in diesem Bereich ist.

Die Bewertungsbereiche wurden so gewählt, um ein möglichst allumfassendes Bild zu erzeugen. Zwischen der verschiedenen Bewertungsbereichen gibt es jedoch Zielkonflikte. Es zeigt sich z.B., dass *Teaching Languages*, die sich auf das Ziel des Konstruktionismus konzentrieren, *Clean Code* vernachlässigen. So schneidet Scratch am besten im Bewertungsbereich „Erlernbarkeit nach Konstruktionismus“ ab und am schlechtesten im Bewertungsbereich *Clean Code*. Die Ziele des Konstruktionismus und *Clean Code* schließen sich jedoch nicht voneinander aus. Es kann ein Mittelweg gewählt werden, wie z.B. ToonTalk zeigt.

Die systematische Untersuchung und Bewertung von *Teaching Languages*, hat viel Wissen aufdecken können. Es zeigt sich, dass jede Designentscheidung eine wichtige Rolle spielt und Zielkonflikte berücksichtigt werden müssen. Diese Arbeit kann zur Orientierung und als Inspirationsquelle verwendet werden, um neuartige *Teaching Languages* zu entwickeln.

Akronyme

4C-ID Four Component Instructional Design. 12, 19, 29, 33

API Application Programming Interface. 96

CCLP Concurrent Constraint Logic Programming. 18, 125, 134, 156

CLT Cognitive Load Theory. 8, 11, 12, 14, 50

CNL Constrained Natural Language. 142

DBC Design by Contract. 46

DSL Domain Specific Language. 38, 158

FPL First Programming Language. 23, 38

FPS Frames pro Sekunde. 15, 59, 71, 86

GUI Graphical User Interface (zu dt. grafische Benutzerschnittstelle). 26, 27, 41, 64, 70, 85, 90, 110, 135

IDE Integrierte Entwicklungsumgebung (engl. integrated development environment). 4, 19, 23, 24, 34, 40–42, 44, 45, 50, 51, 72, 75, 76, 80, 81, 92, 96–98, 115, 139, 152, 155, 156

IoT Internet of things. 54

IR Immediate Representation. 142

LSP Language Server Protocol. 97, 98

NLN Natural Language Neutrality. 34

NLP Natural Language Programming. 142

OONL Object Oriented Natural Language. 142

PbD Programming by Demonstration. 121, 156

REPL Read Eval Print Loop. 42–45, 53, 58, 64, 71–73, 76, 97, 155, 162

TAM Token Accuracy Map. 32

Tabellenverzeichnis

| | | |
|------|---|-----|
| 3.1. | Zielgruppen | 17 |
| 3.2. | Einordnung der ausgewählten <i>Teaching Languages</i> | 18 |
| 3.3. | Einordnung der weiteren <i>Teaching Languages</i> | 18 |
| 4.1. | Quorum: Klassen, die eine Entdeckung eines bestimmten Bereich ermöglichen (s. https://quorumlanguag.com/libraries.html) | 92 |
| 4.2. | ToonTalk: Zuordnung von Computer-Abstraktionen auf Objekte | 124 |
| 4.3. | ToonTalk: Werkzeuge | 125 |

Abbildungsverzeichnis

| | |
|--|-----|
| 2.1. Game Loop | 15 |
| 3.1. Syntax-Hilfe in DrRacket | 43 |
| 4.1. Programmierumgebung in Pyret | 59 |
| 4.2. Beispielschaltung | 61 |
| 4.3. Code für das X-Symbol in der Konsole | 63 |
| 4.4. Pyret: Ausführung des Button-Codes in der REPL | 64 |
| 4.5. Pyret: Unentschieden in Tic-Tac-Toe | 66 |
| 4.6. Pyret: verwirrende Fehlermeldung | 78 |
| 4.7. Pyret: schlechte Fehlermeldung | 78 |
| 4.8. Pyret: gute Fehlermeldung | 79 |
| 4.9. Sprite-Animation des Vogels | 87 |
| 4.10. Quorum's Web-Editor | 96 |
| 4.11. Scratch: Blockarten | 103 |
| 4.12. Scratch: Bouncing Ball | 104 |
| 4.13. Scratch: Tamagochi | 105 |
| 4.15. Scratch: Tamagochi – Objekte | 105 |
| 4.14. Scratch: Tamagochi – Start-Ereignis des Haustiers | 106 |
| 4.16. Scratch: Tamagochi – Definition des <i>Clamp-Blocks</i> | 106 |
| 4.17. Scratch: Tamagochi – Ändere Energie und Debuff | 108 |
| 4.18. Scratch: Tamagochi – Blöcke für das Kuchen-Symbol | 109 |
| 4.19. Scratch: „Hallo Nutzer“ | 111 |
| 4.20. Scratch: rekursive <i>Fibonacci</i> -Funktion | 114 |
| 4.21. Scratch: Ungültige Operation, die jedoch <i>soft failed</i> | 117 |
| 4.22. Scratch: Unit-Tests | 117 |
| 4.23. ToonTalk: Kollisionssensor | 126 |
| 4.24. ToonTalk: Bouncing-Ball Eingaben | 126 |
| 4.26. ToonTalk: Tic-Tac-Toe – Sequenzdiagramm | 128 |
| 4.27. ToonTalk: Maus für Tic-Tac-Toe | 129 |
| 4.28. ToonTalk: Tic-Tic-Toe - Eingabe-Box des UI-Felds | 130 |
| 4.29. ToonTalk: Tic-Tac-Toe – Eingabe für den Feld-Konstruktor-Roboter | 130 |
| 4.30. ToonTalk: Tic-Tac-Toe - UI-Feld-Konstruktor | 131 |
| 4.31. ToonTalk: Tic-Tac-Toe - Turn-Objekt | 131 |
| 4.32. ToonTalk: Tic-Tac-Toe - Die Roboter des Turn-Objekts | 132 |
| 4.33. ToonTalk: Tic-Tac-Toe - Anzeige | 132 |
| 4.34. ToonTalk: Tic-Tac-Toe - Felder | 133 |
| 4.36. ToonTalk: fertiges Tic-Tac-Toe | 134 |

| | |
|--|-----|
| 4.37. ToonTalk: Roboter für die <i>Fibonacci</i> -Funktion | 138 |
| 4.38. ToonTalk: Anzeige von langen Zahlen | 141 |
| 4.39. Game-Changineer-Webseite | 143 |
| 4.40. Postionierung durch Anweisungen | 144 |
| 4.42. Game-Changineer: Game-Over | 148 |
| 4.43. Game-Changineer: <i>Turtle</i> -Grafiken | 150 |

Quellenverzeichnis

Literatur

- [Agg18] Sanchit Aggarwal. „Modern web-development using reactjs“. In: *International Journal of Recent Research Aspects* 5.1 (2018), S. 2349–7688.
- [Ahm+16] Nova Ahmed u. a. „My code in my native tone: Cha Script“. In: *Proceedings of the Eighth International Conference on Information and Communication Technologies and Development*. 2016, S. 1–4.
- [AW12] Joel C Adams und Andrew R Webster. „What do students learn about programming from game, music video, and storytelling projects?“ In: *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 2012, S. 643–648.
- [Bac78] John Backus. „The history of Fortran I, II, and III“. In: *ACM Sigplan Notices* 13.8 (1978), S. 165–180.
- [Bai+13] Engineer Bainomugisha u. a. „A Survey on Reactive Programming“. In: *ACM Comput. Surv.* 45.4 (Aug. 2013). ISSN: 0360-0300. DOI: 10.1145/2501654.2501666. URL: <https://doi.org/10.1145/2501654.2501666>.
- [Ban10] Ashok Banerji. *Multimedia technologies*. New Delhi, India: Tata McGraw Hill, 2010. ISBN: 9780070669239.
- [BBN10] Andrew Black, Kim B Bruce und James Noble. „Designing the Next Educational Programming Language“. In: (2010).
- [Bec+19] Brett A Becker u. a. „Compiler error messages considered unhelpful: The landscape of text-based programming error message research“. In: *Proceedings of the Working Group Reports on Innovation and Technology in Computer Science Education*. 2019, S. 177–210.
- [Bla+13] Andrew P Black u. a. „Seeking Grace: a new object-oriented language for novices“. In: *Proceeding of the 44th ACM technical symposium on Computer science education*. 2013, S. 129–134.
- [BMK20] Marini Abu Bakar, Muriati Mukhtar und Fariza Khalid. „The Effect of Turtle Graphics Approach on Students’ Motivation to Learn Programming: A Case Study in a Malaysian University“. In: *International Journal of Information and Education Technology* 10.4 (2020).
- [BR+12] Karen Brennan, Mitchel Resnick u. a. „Using artifact-based interviews to study the development of computational thinking in interactive media design“. In: *annual American Educational Research Association meeting, Vancouver, BC, Canada*. 2012, S. 1–25.
- [Cow01] Nelson Cowan. „The magical number 4 in short-term memory: A reconsideration of mental storage capacity“. In: *Behavioral and brain sciences* 24.1 (2001), S. 87–114.
- [Dal21] Robert Dale. „GPT-3: What’s it good for?“ In: *Natural Language Engineering* 27.1 (2021), S. 113–118.

- [Das+15] Sayamindu Dasgupta u. a. „Extending Scratch: New pathways into programming“. In: *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE. 2015, S. 165–169.
- [Den+11] Paul Denny u. a. „Understanding the syntax barrier for novices“. In: *Proceedings of the 16th annual joint conference on Innovation and technology in computer science education*. 2011, S. 208–212.
- [DH17] Sayamindu Dasgupta und Benjamin Mako Hill. „Learning to code in localized programming languages“. In: *Proceedings of the fourth (2017) ACM conference on learning@ scale*. 2017, S. 33–39.
- [Dij68] Edsger W Dijkstra. „Letters to the editor: go to statement considered harmful“. In: *Communications of the ACM* 11.3 (1968), S. 147–148.
- [Dij79] Edsger W Dijkstra. „On the foolishness of "natural language programming"“. In: *Program construction*. Springer, 1979, S. 51–53.
- [Eze18] Onyeka Ezenwoye. „What language?-The choice of an introductory programming language“. In: *2018 IEEE Frontiers in Education Conference (FIE)*. IEEE. 2018, S. 1–8.
- [Far+14] Muhammad Shoaib Farooq u. a. „An evaluation framework and comparative analysis of the widely used first programming languages“. In: *PloS one* 9.2 (2014), e88941.
- [Fel+18] Matthias Felleisen u. a. *How to design programs: an introduction to programming and computing*. MIT Press, 2018.
- [Fel91] Matthias Felleisen. „On the expressive power of programming languages“. In: *Science of computer programming* 17.1-3 (1991), S. 35–75.
- [Guo18] Philip J Guo. „Non-native english speakers learning computer programming: Barriers, desires, and design opportunities“. In: *Proceedings of the 2018 CHI conference on human factors in computing systems*. 2018, S. 1–14.
- [Gup04] Diwaker Gupta. „What is a good first programming language?“ In: *Crossroads* 10.4 (2004), S. 7–7.
- [Her20] Felienne Hermans. „Hedy: A Gradual Language for Programming Education“. In: *Proceedings of the 2020 ACM Conference on International Computing Education Research*. 2020, S. 259–270.
- [HP91] Idit Ed Harel und Seymour Ed Papert. *Constructionism*. Ablex Publishing, 1991.
- [Hsi18] Michael S Hsiao. „Automated Program Synthesis from Object-Oriented Natural Language for Computer Games.“ In: *CNL*. 2018, S. 71–74.
- [Jon10] David H Jonassen. *Learning to solve problems: A handbook for designing problem-solving learning environments*. Routledge, 2010.
- [Kah04a] Ken Kahn. „The child-engineering of arithmetic in ToonTalk“. In: *Proceedings of the 2004 conference on Interaction design and children: building a community*. 2004, S. 141–142.
- [Kah04b] Ken Kahn. „Toontalk-steps towards ideal computer-based learning environments“. In: *A Learning Zone of One's Own: Sharing Representations and Flow in Collaborative Learning Environments* (2004), S. 253–270.
- [Kah14] Ken Kahn. „TOONTALK REBORN-Re-implementing and re-conceptualising ToonTalk for the Web“. In: *Constructionism, Vienna, Austria* (2014).

- [Köl+03] Michael Kölling u. a. „The BlueJ system and its pedagogy“. In: *Computer Science Education* 13.4 (2003), S. 249–268.
- [Köl99] Michael Kölling. „The problem of teaching object-oriented programming, Part 1: Languages“. In: *Journal of Object-oriented programming* 11.8 (1999), S. 8–15.
- [KS07] Herbert Klaeren und Michael Sperber. *Die Macht der Abstraktion: Einführung in die Programmierung*. 1. Aufl. Leitfäden der Informatik; Lehrbuch Informatik. Teubner, 2007. ISBN: 978-3-8351-0155-5. URL: %5Curl%7Bhttp://digitale-objekte.hbz-nrw.de/storage/2007/06/06/file_103/1986581.pdf%7D.
- [Mal+10] John Maloney u. a. „The Scratch Programming Language and Environment“. In: *ACM Trans. Comput. Educ.* 10.4 (Nov. 2010). DOI: 10.1145/1868358.1868363. URL: <https://doi.org/10.1145/1868358.1868363>.
- [MC96] Linda McIver und Damian Conway. „Seven deadly sins of introductory programming language design“. In: *Proceedings 1996 International Conference Software Engineering: Education and Practice*. IEEE. 1996, S. 309–316.
- [McC+08] Renee McCauley u. a. „Debugging: a review of the literature from an educational perspective“. In: *Computer Science Education* 18.2 (2008), S. 67–92.
- [MFK11] Guillaume Marceau, Kathi Fisler und Shriram Krishnamurthi. „Mind your language: on novices' interactions with error messages“. In: *Proceedings of the 10th SIGPLAN symposium on New ideas, new paradigms, and reflections on programming and software*. 2011, S. 3–18.
- [MG11] John T Minor und Laxmi P Gewali. „Design Principles for a Beginning Programming Language“. In: *Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)*. Citeseer. 2011, S. 1.
- [Mic06] David Michael. *Serious Games : Games That Educate, Train and Inform*. Course PTR, 2006. ISBN: 9781592006229. URL: <http://search.ebscohost.com/login.aspx?direct=true&db=nlebk&AN=144858&site=ehost-live>.
- [Pol+18] Joe Gibbs Politz u. a. „From Spreadsheets to Programs: Data Science and CS1 in Pyret“. In: *Proceedings of the 49th ACM Technical Symposium on Computer Science Education*. 2018, S. 1058–1058.
- [RD16] Ivan Ruby und Salomão David. „Natural-Language Neutrality in Programming Languages: Bridging the Knowledge Divide in Software Engineering“. In: *International Conference on Learning and Collaboration Technologies*. Springer. 2016, S. 628–638.
- [Ruk18] Jason Rukman. *The everything kids' Scratch coding book : learn to code and create your own cool games*. New York: Adams Media, 2018. ISBN: 978-1507207970.
- [SMP19] John Sweller, Jeroen JG van Merriënboer und Fred Paas. „Cognitive architecture and instructional design: 20 years later“. In: *Educational Psychology Review* 31.2 (2019), S. 261–292.
- [SR89] Vijay A Saraswat und Martin Rinard. „Concurrent constraint programming“. In: *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 1989, S. 232–245.
- [SS13] Andreas Steffik und Susanna Siebert. „An empirical investigation into programming language syntax“. In: *ACM Transactions on Computing Education (TOCE)* 13.4 (2013), S. 1–40.
- [Swe16] Al Sweigart. *Scratch Programming Playground: Learn to Program by Making Cool Games*. No Starch Press, 2016.

- [Wik21d] Wikipedia. *Sprite (Computergrafik)* — Wikipedia, Die freie Enzyklopädie. [Online; Stand 16. Juli 2021]. 2021. URL: %5Curl%7Bhttps://de.wikipedia.org/w/index.php?title=Sprite_(Computergrafik)&oldid=207123321%7D.
- [Win06] Jeannette M Wing. „Computational thinking“. In: *Communications of the ACM* 49.3 (2006), S. 33–35.
- [Win11] Jeanette Wing. „Research notebook: Computational thinking—What and why“. In: *The link magazine* 6 (2011), S. 20–23.
- [WLG12] Christopher Watson, Frederick WB Li und Jamie L Godwin. „Bluefix: Using crowd-sourced feedback to support programming students in error diagnosis and repair“. In: *International Conference on Web-Based Learning*. Springer. 2012, S. 228–239.

Internetquellen

- [blu12] blueshift. *Bootstrapping an interpreter?* [Online; Stand 23. Juni 2021]. 2012. URL: %5Curl%7Bhttps://stackoverflow.com/questions/9853541/bootstrapping-an-interpreter%7D.
- [Cod15] Code.org. *Hour of Code*. [Online; Stand 30. Juni 2021]. 2015. URL: %5Curl%7Bhttps://hourofcode.com/de%7D.
- [Col14] Dartmouth College. *Birth of BASIC*. [Online; Stand 9. Juli 2021]. 2014. URL: %5Curl%7Bhttps://www.youtube.com/watch?v=WYPNjSoDrqw%7D.
- [Con21] Scratch Wiki Contributors. *Block Plugin*. [Online; Stand 6. Juli 2021]. 2021. URL: %5Curl%7Bhttps://en.scratch-wiki.info/wiki/Block_Plugin%7D.
- [Fin21] Robert Findler. *How to Design Programs Teaching Languages*. [Online; Stand 16. Juli 2021]. 2021. URL: %5Curl%7Bhttps://docs.racket-lang.org/drracket/htdp-langs.html%7D.
- [Fou15] Logo Foundation. *Logo History*. [Online; Stand 16. Mai 2021]. 2015. URL: %5Curl%7Bhttps://el.media.mit.edu/logo-foundation/what_is_logo/history.html%7D.
- [fou20] kogics foundation. *Kojo*. [Online; Stand 11. Juli 2021]. 2020. URL: %5Curl%7Bhttp://www.kogics.net/%7D.
- [Gam21] GameChangineer. *Tutorials*. [Online; Stand 5. Juli 2021]. 2021. URL: %5Curl%7Bhttps://gc.ece.vt.edu/trial/tutorials/%7D.
- [Ger11] Andrew Gerrand. *Error handling and Go*. [Online; 21. Juni 2021]. 2011. URL: %5Curl%7Bhttps://blog.golang.org/error-handling-and-go%7D.
- [Gob21a] Lukas Gobelet. *GermanSkript*. [Online; Stand 11. Juli 2021]. 2021. URL: %5Curl%7Bhttps://github.com/lulugo19/GermanSkript%7D.
- [Gob21b] Lukas Gobelet. *Implementierungen der Computerspiele in verschiedenen Teaching Languages*. [Online; Stand 11. Juli 2021]. 2021. URL: %5Curl%7Bhttps://github.com/lulugo19/teaching-languages%7D.
- [Inc19] Logo Computer Systems Inc. *MicroWorlds Homepage*. [Online; 17. Juni 2021]. 2019. URL: %5Curl%7Bhttp://www.microworlds.com/%7D.
- [Iu21a] Dr. Ming-Yee Iu. *Babylscript*. [Online; Stand 10. Juni 2021]. 2021. URL: %5Curl%7Bhttp://www.babylscript.com/%7D.
- [Iu21b] Dr. Ming-Yee Iu. *Programming Basics*. [Online; Stand 10. Juni 2021]. 2021. URL: %5Curl%7Bhttp://www.programmingbasics.org/de/%7D.

- [Joh19] Johanna. *5 Scratch Bücher für Kinder, die programmieren möchten – Level Anfänger*. [Online; 2. Juli 2021]. 2019. URL: %5Curl%7Bhttps://kinderprogrammieren.de/buch/5-buecher-fuer-kinder-scratch-programmierung/%7D.
- [jsw20] jswrenn. *Pyret is not fully Unicode-aware*. [Online; Stand 28. Juni 2021]. 2020. URL: %5Curl%7Bhttps://github.com/brownplt/pyret-lang/issues/1534%7D.
- [Kah] Ken Kahn. *ToonTalk and Java*. [Online; Stand 3. Juli 2021]. URL: %5Curl%7Bhttp://www.toontalk.com/English/java.htm%7D.
- [Kah16] Ken Kahn. *A Players Guide to ToonTalk*. [Online; Stand 2. Juli 2021]. 2016. URL: %5Curl%7Bhttp://www.toontalk.com/English/doc.htm%7D.
- [KBA] Michael Kölling, Neil Brown und Amjad Altadmri. *Frame-Based Editing*. [Online; Stand 16. Juli 2021]. URL: %5Curl%7Bhttps://www.greenfoot.org/frames/%7D.
- [Köl+20] Michael Kölling u.a. *Stride*. [Online; Stand 16. Juli 2021]. 2020. URL: %5Curl%7Bhttps://stride-lang.net/%7D.
- [Kri+20] Shiriam Krishnamurthi u.a. *Programming and Programming Languages*. [Online; Stand 27. Juni 2021]. 2020. URL: https://papl.cs.brown.edu/2020/.
- [Kri20] Shriram Krishnamurthi. *What is a Pedagogic IDE?* [Online; Stand 14. Juni 2021]. 2020. URL: %7Bhttps://parentheticallyspeaking.org/articles/pedagogic-ide/%7D.
- [Lan19a] The Quorum Programming Language. *Lab 5.2: Class Variables and Modifiers*. [Online; Stand 30. Juni 2021]. 2019. URL: %5Curl%7Bhttps://quorumlanguage.com/lessons/chapter5/lab5_2.html%7D.
- [Lan19b] The Quorum Programming Language. *Quorum Coding Conventions*. [Online; Stand 30. Juni 2021]. 2019. URL: %5Curl%7Bhttps://quorumlanguage.com/tutorials/language/autoboxing.html%7D.
- [Lan19c] The Quorum Programming Language. *Quorum Coding Conventions*. [Online; Stand 30. Juni 2021]. 2019. URL: %5Curl%7Bhttps://quorumlanguage.com/tutorials/language/codingGuidelines.html%7D.
- [Lan19d] The Quorum Programming Language. *Tutorial: Writing Quorum Plugins*. [Online; Stand 30. Juni 2021]. 2019. URL: %5Curl%7Bhttps://quorumlanguage.com/libraries.html%7D.
- [Lan20] The Quorum Programming Language. *Quorum Reference Pages*. [Online; Stand 28. Juni 2021]. 2020. URL: %5Curl%7Bhttps://quorumlanguage.com/reference.html%7D.
- [Ltd21] MacroMates Ltd. *Language Grammars*. [Online; Stand 29. Juni 2021]. 2021. URL: %5Curl%7Bhttps://macromates.com/manual/en/language_grammars%7D.
- [Man06] Walker Mangum. *Pirate Speak*. [Online; Stand 27. Juni 2021]. 2006. URL: %5Curl%7Bhttp://bvipirate.com/piratespeak/index.html%7D.
- [Mic21a] Microsoft. *Language Server Protocol*. [Online; Stand 29. Juni 2021]. 2021. URL: %5Curl%7Bhttps://microsoft.github.io/language-server-protocol/%7D.
- [Mic21b] Microsoft. *Monaco Editor*. [Online; Stand 30. Juni 2021]. 2021. URL: %5Curl%7Bhttps://microsoft.github.io/monaco-editor/%7D.
- [Nel20] Graham Nelson. *Inform 7 - A Design System for Interactive Fiction*. [Online; Stand 12. Juli 2021]. 2020. URL: %5Curl%7Bhttp://inform7.com/%7D.

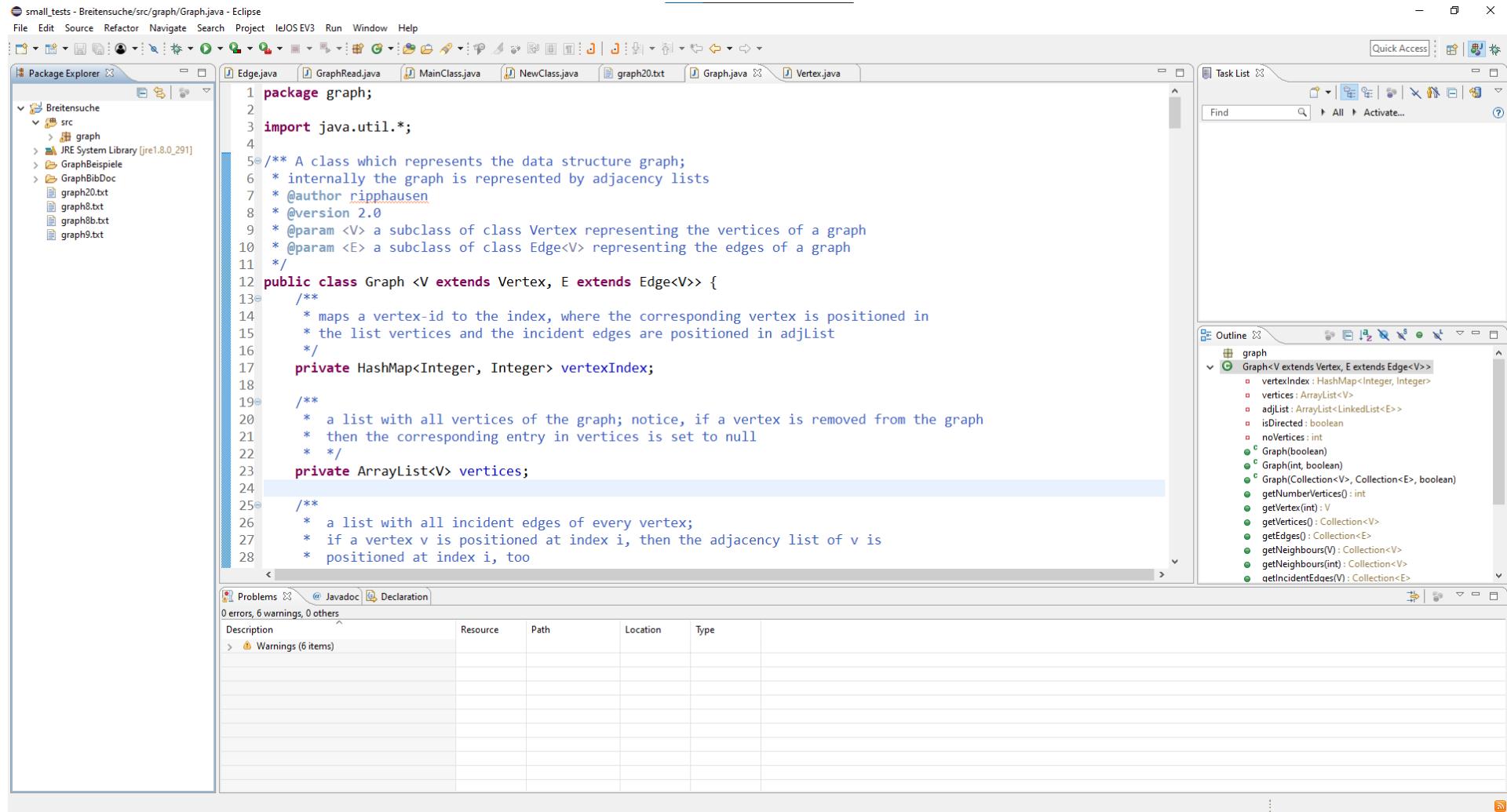
- [Nys14] Robert Nystrom. *Game Programming Patterns - Game Loop*. [Online; Stand 12. Mai 2021]. 2014. URL: %5Curl%7Bhttp://gameprogrammingpatterns.com/game-loop.html%7D.
- [Pap80] Seymour Papert. *Constructionism vs. Instructionism*. [Online; Stand 19. Mai 2021]. 1980. URL: %5Curl%7Bhttp://www.papert.org/articles/const_inst/const_inst1.html%7D.
- [Pap99] Papert, Seymour and Caperton, Gaston. *Vision for Education: The Caperton-Papert Platform*. [Online; Stand 9. Juli 2021]. 1999. URL: %5Curl%7Bhttp://www.papert.org/articles/Vision_for_education.html%7D.
- [Pas19] Erik Pasternak. *Scratch 3.0's new programming blocks, built on Blockly*. [Online; Stand 21. Juli 2021]. 2019. URL: %5Curl%7Bhttps://developers.googleblog.com/2019/01/scratch-30s-new-programming-blocks.html%7D.
- [Pro20] Leopard Project. *Leopard*. [Online; Stand 2. Juli 2021]. 2020. URL: %5Curl%7Bhttps://leopardjs.vercel.app/%7D.
- [Pro21] Processing. *Processing Foundation*. [Online; Stand 25. Juni 2021]. 2021. URL: %5Curl%7Bhttps://processing.org/copyright.html%7D.
- [Pyr21a] Pyret. *Pyret - Home*. [Online; Stand 14. April 2021]. 2021. URL: %5Curl%7Bhttps://www.pyret.org/%7D.
- [Pyr21b] Pyret. *Pyret Documentation*. [Online; Stand 27. Juni 2021]. 2021. URL: %5Curl%7Bhttps://www.pyret.org/docs/latest/%7D.
- [Pyr21c] Pyret. *Pyret Style Guide*. [Online; Stand 28. Juni 2021]. 2021. URL: %5Curl%7Bhttps://www.pyret.org/docs/latest/Pyret_Style_Guide.html#%28part_._.Annotations%29%7D.
- [Pyr21d] Pyret. *Tables*. [Online; Stand 27. Juni 2021]. 2021. URL: %5Curl%7Bhttps://www.pyret.org/docs/latest/tables.html%7D.
- [Pyr21e] Pyret. *The Pyret Code; or A Rationale for the Pyret Programming Language*. [Online; Stand 28. Juni 2021]. 2021. URL: %5Curl%7Bhttps://www.pyret.org/%7D.
- [Quo] Quorum. *A quick primer on human-factors evidence in programming language design*. [Online; Stand 10. Mai 2021]. URL: %5Curl%7Bhttps://quorumlanguage.com/evidence.html%7D.
- [Quo18] Quorum. *Quorum 9.0 Standard Library*. [Online; Stand 28. Juni 2021]. 2018. URL: %5Curl%7Bhttps://quorumlanguage.com/libraries.html%7D.
- [Scr21a] Scratch. *About Scratch*. [Online; Stand 1. Juli 2021]. 2021. URL: %5Curl%7Bhttps://scratch.mit.edu/about%7D.
- [Scr21b] ScratchX. *Gallery of Experimental Extensions*. [Online; Stand 1. Juli 2021]. 2021. URL: %5Curl%7Bhttps://scratchx.org/#extensions%7D.
- [Sna21] Snap. *Snap!* [Online; Stand 16. Juli 2021]. 2021. URL: %5Curl%7Bhttps://snap.berkeley.edu/%7D.
- [Sul16] Sulfurous-Project. *Sulfurous*. [Online; Stand 2. Juli 2021]. 2016. URL: %5Curl%7Bhttps://sulfurous.aau.at/%7D.
- [Tea21] Scratch Team. *Scratch Link*. [Online; Stand 2. Juli 2021]. 2021.
- [TIO20] TIOBE. *TIOBE Index for April 2020*. [Online; 2. Juli 2021]. 2020. URL: %5Curl%7Bhttps://web.archive.org/web/20200414203423/https://www.tiobe.com/tiobe-index/%7D.

- [Tom21] Context Free (Tom). *Bookends and curly braces*. [Online; Stand 12. Juni 2021]. 2021. URL: %5Curl1%7Bhttps://www.youtube.com/watch?v=FnyB7H42jIc%7D.
- [Web] WebLabsGames. *ToonTalk RCX extension*. [Online; Stand 2. Juli 2021]. URL: %5Curl1%7Bhttp://www.toontalk.com/Tools/RCX/doc/toontalk_rcx_extension.htm%7D.
- [Wik19] Wikipedia. *Literate programming — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 16. Juni 2021]. 2019. URL: %5Curl1%7Bhttps://de.wikipedia.org/w/index.php?title=Literate_programming&oldid=194720083%7D.
- [Wik20a] Scratch Wiki. *Scratch Wiki*. [Online; Stand 22. Mai 2021]. 2020. URL: %5Curl1%7Bhttps://en.scratch-wiki.info/wiki%7D.
- [Wik20b] Wikipedia. *Algebraischer Datentyp — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 6. Mai 2021]. 2020. URL: %5Curl1%7Bhttps://de.wikipedia.org/w/index.php?title=Algebraischer_Datentyp&oldid=207100845%7D.
- [Wik20c] Wikipedia. *Bootstrapping (Programmierung) — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 23. Juni 2021]. 2020. URL: %5Curl1%7Bhttps://de.wikipedia.org/w/index.php?title=Bootstrapping_(Programmierung)&oldid=198644005%7D.
- [Wik20d] Wikipedia contributors. *Incremental compiler — Wikipedia, The Free Encyclopedia*. [Online; 29. Juni 2021]. 2020. URL: %5Curl1%7Bhttps://en.wikipedia.org/w/index.php?title=Incremental_compiler&oldid=968505347%7D.
- [Wik20e] Wikipedia contributors. *Interactive programming — Wikipedia, The Free Encyclopedia*. [Online; 15. Juni 2021]. 2020. URL: %5Curl1%7Bhttps://en.wikipedia.org/w/index.php?title=Interactive_programming&oldid=937341336%7D.
- [Wik21a] Wikipedia. *Drehrichtung — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 27. Juni 2021]. 2021. URL: %5Curl1%7Bhttps://de.wikipedia.org/w/index.php?title=Drehrichtung&oldid=212600695%7D.
- [Wik21b] Wikipedia. *Flappy Bird — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 30. März 2021]. 2021. URL: %5Curl1%7Bhttps://de.wikipedia.org/w/index.php?title=Flappy_Bird&oldid=209768791%7D.
- [Wik21c] Wikipedia. *K-12 — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 10. Mai 2021]. 2021. URL: %5Curl1%7Bhttps://de.wikipedia.org/w/index.php?title=K-12&oldid=211067357%7D.
- [Wik21e] Wikipedia. *Tamagotchi — Wikipedia, Die freie Enzyklopädie*. [Online; Stand 31. März 2021]. 2021. URL: %5Curl1%7Bhttps://de.wikipedia.org/w/index.php?title=Tamagotchi&oldid=208910983%7D.
- [Wik21f] Wikipedia contributors. *Controlled natural language — Wikipedia, The Free Encyclopedia*. [Online; accessed 26-May-2021]. 2021. URL: %5Curl1%7Bhttps://en.wikipedia.org/w/index.php?title=Controlled_natural_language&oldid=1019278276%7D.
- [Wik21g] Wikipedia contributors. *Hot swapping — Wikipedia, The Free Encyclopedia*. [Online; 15. Juni 2021]. 2021. URL: %5Curl1%7Bhttps://en.wikipedia.org/w/index.php?title=Hot_swapping&oldid=1018488972%7D.
- [Wik21h] Wikipedia contributors. *Invariant (mathematics) — Wikipedia, The Free Encyclopedia*. [Online; 21. Juni 2021]. 2021. URL: %5Curl1%7Bhttps://en.wikipedia.org/w/index.php?title=Invariant_(mathematics)&oldid=1028569025%7D.

- [Wik21i] Wikipedia contributors. *Non-English-based programming languages* — Wikipedia, The Free Encyclopedia. [Online; Stand 10. Juni 2021]. 2021. URL: %5Curl%7Bhttps://en.wikipedia.org/w/index.php?title=Non-English-based_programming_languages&oldid=1027490547%7D.
- [Wik21j] Wikipedia contributors. *Notebook interface* — Wikipedia, The Free Encyclopedia. [Online; Stand 16. Juni 2021]. 2021. URL: %5Curl%7Bhttps://en.wikipedia.org/w/index.php?title=Notebook_interface&oldid=1027302000%7D.
- [Wik21k] Wikipedia contributors. *Read–eval–print loop* — Wikipedia, The Free Encyclopedia. [Online; 15. Juni 2021]. 2021. URL: %5Curl%7Bhttps://en.wikipedia.org/w/index.php?title=Read%20%93eval%20%93print_loop&oldid=1024927937%7D.
- [Wik21l] Wikipedia contributors. *Syntactic sugar* — Wikipedia, The Free Encyclopedia. [Online; Stand 24. Juni 2021]. 2021. URL: %5Curl%7Bhttps://en.wikipedia.org/w/index.php?title=Syntactic_sugar&oldid=1027587984%7D.
- [Wik21m] Wikipedia contributors. *Text (literary theory)* — Wikipedia, The Free Encyclopedia. [Online; Stand 9. Juni 2021]. 2021. URL: %5Curl%7Bhttps://en.wikipedia.org/w/index.php?title=Text_(literary_theory)&oldid=1004252423%7D.

A. Diverses

A.1. Bildschirmaufnahmen IDEs



Standardlayout in Eclipse

The screenshot shows the DrRacket interface with a file named "robot-name.rkt" open. The code defines a language named "racket" which provides a "make-robot" procedure. This procedure takes three arguments: "name", "reset!", and "reset-name-cache!". It also saves the robots in a global mutable set named "robot-set". The code includes helper procedures for generating random names based on ASCII ranges and random numbers.

```

#lang racket

(provide make-robot
         name
         reset!
         reset-name-cache!)

; save the robots in a global mutable set
(define robot-set (mutable-set))

(define (make-robot)
  (let ([name (random-name)])
    (if (set-member? robot-set name) (make-robot) (begin (set-add! robot-set name) name)))))

(define (name robot) robot)

(define (reset! robot) (set-remove! robot-set robot) (make-robot))

(define (reset-name-cache!) (set-clear! robot-set))

; helper procedures for generating the robot name
(define (random-name) (string (random-letter) (random-letter) (random-number) (random-number) (random-number)))
(define (random-char ascii-start ascii-end) (integer->char (random ascii-start ascii-end)))
(define (random-letter) (random-char 65 91))
(define (random-number) (random-char 48 58))

```

A tooltip window is displayed over the "define" keyword, showing its syntax definition:

```

(define id expr)           syntax ^
(define (head args) body ...+)
head = id
      | (head args)

args = arg ...
      | arg ... . rest-id

arg = arg-id
      | [arg-id default-expr]
      | keyword arg-id
      | keyword [arg-id default-expr]

```

[read more...](#)

DrRacket

The screenshot shows the Quorum Studio 2.5.1 interface. The top menu bar includes File, Edit, View, Navigate, Run, Team, Windows, and Help. Below the menu is a toolbar with various icons for file operations like Open, Save, and Run. The left sidebar has tabs for Projects, Scene, and Palette, with the Projects tab currently selected. Under Projects, there is a FlappyBird folder containing Assets and SourceCode. Inside SourceCode, there are files named Main.quorum and SpriteAnimation.quorum. The main workspace is a code editor showing the content of Main.quorum. The code defines a class Main with various properties and methods, including an action Main that sets up Android configuration. The bottom part of the interface is a console window displaying build logs and user input.

```
18 use Libraries.Interface.Events.TouchListener
19 use Libraries.Interface.Events.TouchEvent
20 use Libraries.Game.AndroidConfiguration
21
22 class Main is Game, TouchListener, CollisionListener2D, Behavior
23     constant integer GRAVITY = 300
24     constant integer GAP_HEIGHT = 100
25     constant Vector2 FLY_VELOCITY
26     constant Vector2 FALL_VELOCITY
27
28     Random random
29     Math math
30     integer WIDTH = 280
31     integer HEIGHT = 500
32
33     SpriteAnimation bird
34     Drawable topPipe
35     Drawable bottomPipe
36     Label scoreLabel
37     Label gameOverLabel
38     Button playAgainButton
39     integer score = 0
40     boolean gameOver = false
41
42     Audio pointAudio
43     Audio hitAudio
44     Audio wingAudio
45     Audio gameOverAudio
46
47     action Main
48         AndroidConfiguration conf
49         conf:touchSleepTime = 16
50         conf:hideStatusBar = true
51         conf:useAspectRatio = false
52         conf:targetWidth = 0
53         conf:targetHeight = 0
54         conf:multipleTapTimer = 0.25
55
```

Console Errors Variables

Building HelloUser
Running HelloUser
Build Successful in 0.02 seconds
Hallo, wie heisst du?

Send

Quorum-Studio

A.2. Emails

A.2.1. Ken Kahn (ToonTalk)

Listing A.1. Email: Ken Kahn

Hi

I'm glad to hear of your interest in ToonTalk. It was the focus of my research from 1992 to about 2007. Then after becoming involved in other research projects I returned to ToonTalk several years later to create ToonTalk Reborn . Unfortunately for ToonTalk, as I was completely it I joined a series of other research projects and haven't had a chance to get back to it. The original ToonTalk might be more appropriate since it went through 3 major releases and had thousands of users. And many more game applications. ToonTalk Reborn can import browser elements and give them behaviours so in theory it should support Tic Tac Toe but it would not surprise me if several bugs would be encountered making it difficult.

Classic ToonTalk is free but only works in Microsoft Windows. Visit toontalk.com if you haven't already. You'll find many sample games, though I don't think any you mentioned.

Good luck and I'm happy to answer any questions if they come up.

Best,

-ken

On Fri, 30 Apr 2021 at 09:18, Lukas Gobelet <lukas.gobelet@smail.th-koeln.de> wrote:

Dear Mr. Kahn,

I am a CS student and I am currently working on my bachelor thesis with the topic "Teaching Languages", that covers programming languages that are explicitly designed for learning and teaching programming.

During my research, I discovered ToonTalk and I really love the concepts and the idea of "Learning through play" and "Programming by demonstration" behind it.

In my bachelor thesis I describe various teaching languages and in each language I program 3 small games:

- 1.) Tic-Tac-Toe
- 2.) Flappy Bird (Where you have to fly a bird between pipes)
- 3.) Tamagochi (a reduced form of this, where you have to keep a pet alive)

I tried to program the first game "Tic-Tac-Toe" in Toontalk Reborn, but I couldn't manage to do it.

The usability is kind of hard and often the program didn't do, what I expected it to.

Nevertheless I want to include Toontalk Reborn in my thesis, writing about the concepts and the idea behind it.

I think programming these games in Toontalk is possible, I just need to get myself a bit more into it.

Maybe, you could provide me an example for programming Tic-Tac-Toe in ToonTalk Reborn.

Sincerely,

Lukas Gobelet

```
--  
*****  
** Technische Hochschule Koeln / University of Applied Sciences  
**  
** Lukas Gobelet  
** E-Mail: lukas.gobelet@smail.th-koeln.de  
*****
```

A.2.2. Game-Changineer

Listing A.2. Email: Game-Changineer

Lukas,

It's wonderful that you have created a great game. To answer your questions:

1. Time control - using moving objects is a great way to control time. But there are other ways, using Boolean attributes, such as "When the yellow puppy is not empowered, it ...". You can find out more in the Tutorials.

Also, you can make the moving timer invisible. For example: "The turtle is invisible. The dog is invisible."

2. The cat and the dog should be able to insert bricks at the same time now.
3. The system can handle many instances of characters. So you may not need to make the bricks disappear -- just let them continue to move left.
4. How about sentences such as:

When space is pressed, the bird jumps 5 pixels and the angle of the bird equals 0.

When the direction_y of the bird is less than 0, the angle of the bird decreases by 1.5.

Otherwise, the angle of the bird increases.

When the angle of the bird is greater than 30, the angle of the bird equals 30.

Hope you find the above helpful!

On Tue, Mar 30, 2021 at 1:34 PM Lukas Gobelet <lukas.gobelet@smail.th-koeln.de> wrote:

Thank you for your quick responses.

So I have created a simple flappy bird game and I have some questions.

Here is the code:

// Setting up the turtle timer.

The turtle moves right.

The speed of the turtle is 4 pixels per frame.

When the turtle reaches the right border, it wraps around and it becomes yellow for 0.1 second.

When the turtle is yellow, the score increases by 1.

When the turtle is yellow, the dog becomes notified for 0.1 second.

When the turtle is yellow, the cat becomes notified for 0.8 second.

// The cat and the dog are the brick spawner.

When the dog is notified, the dog inserts[4, 600, 600, 0, 130] a brick.

When the cat is notified, the cat inserts[4, 600, 600, 470, 600] a brick.

// Defining the bricks.

There are 0 bricks.

The size of the bricks is 300.

The angle of the bricks is 90.

The bricks are moving left at 200 pixels per frame.

When the position_x of the brick is less than 0, it disappears.

// Setting up the bird.

The player controls the bird.

The bird falls.

The bird is fit.

When the space bar is pressed, the bird becomes energized for 0.1 second.

When the fit bird is energized, it jumps 5 pixels per frame.

When the bird whose angle is greater than -45 is energized, the angle of the bird decreases by 5.

When the bird whose angle is less than 45 is not energized, the angle of the bird increases by 2.

When the bird touches the bottom border, game over.

When the bird collides with a brick, it is not fit.

When the bird is not fit, it moves down at 600 pixels per frame.
 When the bird is not fit, the angle of the bird increases by 1.

My question are:

1.) I used a running turtle as a timer for spawning the obstacles. Are there any other possibilties for a timer that don't involve moving an object?

2.) I want the upper and lower obstacles to spawn simultaneously, but it didn't work.

In the notifications it said that a single instance can only insert one random object in 0.5 seconds to avoid cluttering.

But I have two spawners (the cat and the dog) so a simultaenous spawn should be possible. I solved it for now so that the cat obstacle is spawned 0.5 seconds later.

3.) I want the obstacles to disappear when they are out of the screen. So I tried 'When the position_x of a brick is less than -100, the brick disappears'.

But this didn't work. It seems that the -100 is interpreted as a positive number.

I then settled with 'When the position_x of a brick is less than 0' which is good enough.

4.) I want the bird to turn up a bit when flying and turn down when falling, but I got confused with the angles. How are angles compared. How do they wrap? Is -45 deg same as 315 deg?

Thank you in advance,

Lukas

```
*****
** Technische Hochschule Koeln / University of Applied Sciences
**
** Lukas Gobelet
** E-Mail: lukas.gobelet@smail.th-koeln.de
*****
```

A.3. Quorum: Auflistung der Packages der Standardbibliothek

```
Libraries.Compute
Libraries.Compute.MatrixTransform
Libraries.Compute.Statistics
Libraries.Compute.Statistics.Analysis
Libraries.Compute.Statistics.Calculations
Libraries.Compute.Statistics.Charts
Libraries.Compute.Statistics.Columns
Libraries.Compute.Statistics.Distributions
```

Libraries.Compute.Statistics.Inputs
Libraries.Compute.Statistics.Loaders
Libraries.Compute.Statistics.Reporting
Libraries.Compute.Statistics.Tests
Libraries.Compute.Statistics.Transforms
Libraries.Compute.Statistics.WindowingActions
Libraries.Containers
Libraries.Containers.Support
Libraries.Controls.Charts
Libraries.Curriculum.AudioGame
Libraries.Curriculum.TurtleProgram
Libraries.Data.Compression
Libraries.Data.Formats
Libraries.Data.Formats.ScalableVectorGraphics
Libraries.Game
Libraries.Game.Collision
Libraries.Game.Collision.Narrowphase
Libraries.Game.Collision.Shapes
Libraries.Game.Graphics
Libraries.Game.Graphics.Fonts
Libraries.Game.Graphics.ModelData
Libraries.Game.Graphics.ModelLoaders
Libraries.Game.Graphics.ModelLoaders.WavefrontObject
Libraries.Game.Physics
Libraries.Game.Physics.Joints
Libraries.Game.Scenes
Libraries.Game.Shapes
Libraries.Interface
Libraries.Interface.Behaviors
Libraries.Interface.Behaviors.Controls
Libraries.Interface.Controls
Libraries.Interface.Controls.Charts
Libraries.Interface.Controls.TextStyles
Libraries.Interface.Events
Libraries.Interface.Layouts
Libraries.Interface.Mobile
Libraries.Interface.Selections
Libraries.Interface.Undo
Libraries.Interface.Vibration
Libraries.Interface.Views
Libraries.Language
Libraries.Language.Compile
Libraries.Language.Compile.Context
Libraries.Language.Compile.Documentation
Libraries.Language.Compile.Hints
Libraries.Language.Compile.Interpreter
Libraries.Language.Compile.Parsing
Libraries.Language.Compile.Symbol
Libraries.Language.Compile.Translate
Libraries.Language.Errors
Libraries.Language.Support
Libraries.Language.Types
Libraries.Network
Libraries.Robots.Lego

Libraries.Science.Astronomy
Libraries.Sound
Libraries.System
Libraries.System.Blueprints
Libraries.Testing
Libraries.Web
Libraries.Web.Page

B. Bewertungen

B.1. Pyret

| Kriterium | Gewicht | Bewertung | Begründung |
|---|---------|------------|---|
| Einfachheit der Implementierung der Computerspiele | | -0.275 | |
| Alle Anforderungen erfüllt | 0.15 | 1 | alles erfüllt außer Sound-Effekte |
| Aufwand der Implementierung | 0.15 | 0 | subjektiv geringer als ToonTalk |
| Komplexität der Implementierung (Hacks) | 0.25 | 0 | - funktionale reaktive Programmierung - nur globale Mausklick-Erkennung - zeitliche Veränderung der Werte bei Tamagochi mit FPS |
| Grafischer Szenen-Editor | 0.15 | -1 | kein Editor, aber REPL |
| Koordinatensystem und Drehrichtung nach mathematischer Konvention | 0.025 | -2 | rechthändiges Koordinatensystem mit falscher Drehrichtung |
| Ankerpunkt gut gesetzt? | 0.025 | 1 | mittig gesetzt, gute Wahl, da es keinen Editor gibt |
| GUI-Bibliothek? | 0.05 | -2 | Nein |
| Abspielen von Audio? | 0.05 | -2 | Nein |
| Animation von Sprites? | 0.05 | -1 | nicht unterstützt, Implementierung kompliziert |
| Abstraktion von <i>Game Loop</i> ? | 0.1 | 0 | über funktionale reaktive Programmierung |
| Erlernbarkeit nach Konstruktionismus | | -5.551E-17 | |
| Anzahl und Art der Mikrokosmen | 0.2 | -1 | Computerspiele, Simulationen Data Science: Tabellen, Diagramme, Plots, Statistik |
| Unterstützung von <i>Turtle</i> -Grafiken | 0.1 | -2 | Nein |
| Einbindung von unterschiedlichen Medien und Technologien | 0.2 | 0 | Bilder, Tabellen (von Google-Spreadsheets), Diagramme, Plots |
| Selbstoffenbarung der Umgebung | 0.2 | -1 | nur REPL |
| Sicherheit der Umgebung | 0.1 | 2 | sehr sicher |
| Dynamische Typisierung | 0.2 | 2 | graduelle Typisierung => Ja |
| Inkrementelle Einführung | | 1 | |
| Einfachheit des Programms "Hallo Nutzer" | 0.5 | 0 | Ausgabe: print("Hallo") Eingabe: über REPL |
| Inkrementelle Einführung der Programmierkonzepte | 0.5 | 2 | Ja, siehe Textbuch Programming and Programming Languages |
| Intuitive Syntax und Natürlichsprachlichkeit | | -0.35 | |

| | | | |
|--|-------|-------|--|
| Intuitivität der Syntax | 0.6 | 0 | Intuitive Operatoren sehr viel syntaktischer Zucker |
| Internationalisierung der Syntax | 0.25 | -2 | keine Internationalisierung |
| syntaktische Feinheiten | 0.15 | 1 | Blockstrukturen: keine spezifischen Anfangs- und Endtags dafür konsistent Trailing Return Type: ja |
| Feature Uniformity und Orthogonalität | | 0.8 | |
| wenige syntaktische Synonyme | 0.2 | -1 | Nein, sehr viel syntaktischer Zucker |
| wenige syntaktische Homonyme | 0.2 | 2 | Ja |
| kleine, simple Funktionsmenge | 0.3 | 0 | Nein, große Funktionsmenge |
| konsistente Syntax (Orthogonalität) | 0.3 | 2 | Ja, sehr konsistent |
| Programmierkonzepte & Datenstrukturen | | 1.4 | |
| strukturierte Programmierung | 0.25 | 2 | Schleife: Ja Bedingungen: Ja |
| prozedurale Programmierung | 0.25 | 2 | Funktionen: Ja lokale Variablen: Ja Rekursion: Ja |
| Datenstrukturen | 0.25 | 0 | Liste: Ja Queue: Nein Stack: Nein HashMap: Ja aber nur Strings als Key Set: Ja |
| Implementierung der Fibonacci-Funktion | 0.15 | 2 | sehr einfach |
| self-hosting Compiler? | 0.05 | 2 | Ja |
| Programmierumgebung (IDE) | | 1.475 | |
| Einfachheit der Umgebung | 0.25 | 2 | sehr übersichtlich und einfach |
| Interaktivität | 0.25 | 2 | Turnaround Time: gering Interaktive Programmierung (REPL): Ja Live Coding: Nein |
| Debugging Support | 0.2 | 2 | Unit-Tests, REPL |
| Einfachheit der Installation | 0.075 | 2 | webbasiert |
| Einfachheit der Kollaboration | 0.075 | 1 | Programmieren im Team: Nein Teilen von Programmen: Ja |
| Moderne Features | 0.15 | -1 | Syntax Highlighting: Ja Code-Completion: Nein Navigation: Nein Formatierer: Ja Refactoring: Nein |
| Clean Code | | 1.6 | |
| Organisation in Modulen | 0.15 | 2 | Ja |
| Information Hiding | 0.15 | 2 | Ja |
| Dokumentations-Kommentare | 0.03 | 2 | Ja |
| statische Typisierung | 0.2 | 1 | graduelle Typisierung => Ja, aber nicht vollständig |

| | | | |
|---|------|------|---|
| explizite Casts und Typumwandlungen | 0.1 | 2 | (keine Casts) Ja |
| Exception-Handling | 0.1 | 0 | Ja, jedoch kein catch und finally |
| Unit-Tests | 0.1 | 2 | ausgezeichnete Unterstützung, dedizierte Syntax, exzellente Ausgabe |
| Style-Guide | 0.1 | 2 | Ja |
| Weitere Eigenschaften | 0.07 | 2 | + keine Nebeneffekte, nur durch zusätzliche Syntax + ebenso bei Variable Shadowing + automatische defensive Programmierung + keine Operator-Rangfolge - Operator-Überladung |
| Fehlermeldungen | | 1.35 | |
| vereinfachtes Vokabular | 0.1 | 1 | Nein, dafür farbliche Assoziation zwischen Vokabular und Code |
| FEhlerposition wird im Code angezeigt? | 0.2 | 2 | Ja |
| unterstützende Elemente | 0.1 | 2 | Ja, sehr viele |
| keine störenden Elemente | 0.1 | 0 | selten, bei Bugs |
| keine falsche Lösung vorschlagen | 0.15 | 2 | Ja |
| positiver Ton | 0.15 | 1 | neutraler Ton |
| Kontext und Informationen anzeigen, um Fehler zu beheben | 0.15 | 2 | Ja, sehr viel Kontext |
| Fehlermeldung lokalisiert | 0.05 | -2 | Nein |
| hohes Abstraktionsniveau für Datentypen | | 1.5 | |
| hohe String-Abstraktion im Unicode-Standard | 0.5 | 1 | Ja, nur kleiner (seltener) Bug |
| einiger Zahlentyp Langzahlarithmetik | 0.5 | 2 | standardmäßig Langzahlarithmetik, bei Bedarf Fixzahl |
| Anwendungsbereiche und Interoperabilität | | 0.3 | |
| Anwendungsbereiche der Programmiersprache | 0.5 | 0 | - 2D-Grafik - Simulationen, interaktive Programme und Computerspiele - Data Science - Tabellen (Import von Google-Sheets) - Statistik - Plots und Diagramm |
| Plattformen | 0.3 | 1 | Web Konsole (ohne Grafik-Bibliothek) |
| Interoperabilität mit oder Transpilierung in andere Programmiersprachen | 0.2 | 0 | Nein |
| Dokumentation & Community Support | | 1.3 | |
| Qualität der Dokumentation | 0.3 | 2 | Hervorragend |
| Größe und Hilfsbereitschaft der Community | 0.1 | 1 | Kleine Community, große Hilfsbereitschaft |
| offizielle/inoffizielle Foren und/oder Gruppen | 0.2 | 1 | Mailing-List, Google-Gruppe, Github |
| Beispiele, Tutorials, Bücher | 0.3 | 2 | Getting Started A Flight Lander Game |

| | | | |
|--|-----|----|------|
| Dokumentation, Tutorials, Beispiele, Lehrmaterial internationalisiert? | 0.1 | -2 | Nein |
|--|-----|----|------|

B.2. Quorum

| Kriterium | Gewicht | Bewertung | Begründung |
|---|---------|-----------|--|
| Einfachheit der Implementierung der Computerspiele | | 1.35 | |
| Alle Anforderungen erfüllt | 0.15 | 2 | Ja |
| Aufwand der Implementierung | 0.15 | 2 | subjektiv geriger als Pyret (besonders bei Flappy-Bird wegen Physics-Engine) |
| Komplexität der Implementierung (Hacks) | 0.25 | 1 | Tic-Tac-Toe: überschreibe Eventquelle in Field mit event:setSource(me) |
| Grafischer Szenen-Editor | 0.15 | 0 | Ja, aber keine gute Usability |
| Koordinatensystem und Drehrichtung nach mathematischer Konvention | 0.025 | 2 | rechtshändig gegen den Uhrzeigersinn |
| Ankerpunkt gut gesetzt? | 0.025 | 2 | unten links => Ja |
| GUI-Bibliothek? | 0.05 | 2 | Ja |
| Abspielen von Audio? | 0.05 | 2 | Ja |
| Animation von Sprites? | 0.05 | 0 | Nein, doch Implementierung ist einfach und macht Programm nicht komplexer |
| Abstraktion von <i>Game Loop</i> ? | 0.1 | 2 | Über Vererbung, zusätzlich Physics-Engine |
| Erlernbarkeit nach Konstruktionismus | | 0.4 | |
| Anzahl und Art der Mikrokosmen | 0.2 | 2 | Skynet, Musik, Lego Mindstorms, SVG, <i>Turtle</i> -Grafiken, 3D-Physik, Computerspiele, Webentwicklung, ... |
| Unterstützung von <i>Turtle</i> -Grafiken | 0.1 | 0 | Ja, aber schlecht dokumentiert |
| Einbindung von unterschiedlichen Medien und Technologien | 0.2 | 2 | 2D-Grafiken, 3D-Modelle, Text-To-Speech, Audioverarbeitung und -synthesierung, Musik, Lego Mindstorms, Web-Entwicklung, Datenformate: SVG, JSON, XML, MIDI |
| Selbstoffenbarung der Umgebung | 0.2 | -1 | Autovervollständigung in IDE |
| Sicherheit der Umgebung | 0.1 | 0 | Ausführung im Web: Ja Lokale Ausführung: nicht ganz so sicher |
| Dynamische Typisierung | 0.2 | -1 | statische Typisierung mit Type Inference |
| Inkrementelle Einführung | | 2 | |
| Einfachheit des Programms "Hallo Nutzer" | 0.5 | 2 | sehr einfach Ausgabe: über output oder say Eingabe: über input |
| Inkrementelle Einführung der Programmierkonzepte | 0.5 | 2 | Variablen -> Datentypen -> Ausgabe und Eingabe -> Bedingungen -> Schleifen -> prozedurale und objektorientierte Programmierung |
| Intuitive Syntax und Natürlichsprachlichkeit | | 0.85 | |
| Intuitivität der Syntax | 0.6 | 2 | Ja, sehr intuitiv |

| | | | |
|--|-------|------|---|
| Internationalisierung der Syntax | 0.25 | -2 | Nein |
| syntaktische Feinheiten | 0.15 | 1 | Blockstrukturen: kein Anfangstoken für Block Trailing Return Type: Ja |
| Feature Uniformity und Orthogonalität | | 1.4 | |
| wenige syntaktische Synonyme | 0.2 | 2 | Ja |
| wenige syntaktische Homonyme | 0.2 | 2 | Ja |
| kleine, simple Funktionsmenge | 0.3 | 2 | Ja |
| konsistente Syntax (Orthogonalität) | 0.3 | 0 | nicht ganz (Kommandos) |
| Programmierkonzepte & Datenstrukturen | | 1.65 | |
| strukturierte Programmierung | 0.25 | 2 | Schleife: Ja Bedingungen: Ja |
| prozedurale Programmierung | 0.25 | 2 | Funktionen: Ja lokale Variablen: Ja Rekursion: Ja |
| Datenstrukturen | 0.25 | 1 | Liste: Ja Queue: Ja Stack: Ja HashMap: Ja Set: Nein |
| Implementierung der Fibonacci-Funktion | 0.15 | 2 | sehr einfach Ausgabe: über output oder say Eingabe: über input |
| self-hosting Compiler? | 0.05 | 2 | Ja |
| Programmierumgebung (IDE) | | 0.5 | |
| Einfachheit der Umgebung | 0.3 | 1 | einfach, aber keine gute Usability |
| Interaktivität | 0.3 | -1 | Turnaround Time: ca. 1 Sekunde Interaktive Programmierung (REPL): Nein Live Coding: Nein |
| Debugging Support | 0.2 | 1 | Quorum Studio hat visuellen Debugger |
| Einfachheit der Installation | 0.075 | 2 | Einfach und kann auch im Web ausgeführt werden |
| Einfachheit der Kollaboration | 0.075 | 2 | Programmieren im Team: Git-Integration von Quorum Studio Teilen von Programmen: Programme können in Webseiten eingebettet werden |
| Moderne Features | 0.06 | 0 | Syntax Highlighting: Ja in Quorum Studio, Nein im Web-Editor Code-Completion: Ja in Quorum Studio, Nein im Web-Editor Navigation: Nein Formatierer: Nein Refactoring: Ja im Quorum Studio |
| Clean Code | | 1.67 | |
| Organisation in Modulen | 0.15 | 2 | Pakete und Klassen |
| Information Hiding | 0.15 | 2 | Zugriffsmodifizierer: public und private private ist der Standard |

| | | | |
|---|------|-----|--|
| Dokumentations-Kommentare | 0.03 | 0 | Ja, aber keine dedizierte Syntax und schlecht dokumentiert |
| statische Typisierung | 0.2 | 2 | Ja |
| explizite Casts und Typumwandlungen | 0.1 | 1 | explizite Casts: Ja implizite Typumwandlungen bei der String-Konkatenation und bei Operationen mit Ganz- und Fließkommazahlen |
| Exception-Handling | 0.1 | 2 | Ja: check (try), detect (catch), always (finally) und alert (throw) |
| Unit-Tests | 0.1 | 1 | Ja, aber keine IDE-Unterstützung |
| Style-Guide | 0.1 | 2 | Ja |
| Weitere Eigenschaften | 0.07 | 1 | Operatoren ohne Nebeneffekte: Ja Keine Überladung von Operatoren: Ja Keine Variablenüberschattung: Ja |
| Fehlermeldungen | | 0.5 | |
| vereinfachtes Vokabular | 0.1 | 2 | Ja, angepasst an die Syntax und Vokabular von Quorum |
| Fehlerposition wird im Code angezeigt? | 0.2 | 1 | Ja, in Quorum Studio über rote Kringel |
| unterstützende Elemente | 0.1 | -2 | Nein |
| keine störenden Elemente | 0.1 | -1 | Ausgabe von Stack-Trace ist störend |
| keine falsche Lösung vorschlagen | 0.15 | 2 | Ja |
| positiver Ton | 0.15 | 2 | Ja |
| Kontext und Informationen anzeigen, um Fehler zu beheben | 0.1 | 0 | Teilweise, könnte besser sein |
| Fehlermeldung lokalisiert | 0.1 | -2 | Nein |
| hohes Abstraktionsniveau für Datentypen | | -1 | |
| hohe String-Abstraktion im Unicode-Standard | 0.5 | 0 | Ja, aber keine Unterstützung in der Umgebung |
| einzigster Zahlentyp Langzahlarithmetik | 0.5 | -2 | Nein, 32-Bit-Ganzzahl und 64-Bit-Fließkommazahl |
| Anwendungsbereiche und Interoperabilität | | 2 | |
| Anwendungsbereiche der Programmiersprache | 0.5 | 2 | große Standardbibliothek (63 Pakete) |
| Plattformen | 0.3 | 2 | Konsole, grafische Desktop-Anwendungen, Android Apps, LEGO Roboter, Web |
| Interoperabilität mit oder Transpilierung in andere Programmiersprachen | 0.2 | 2 | Java-Interoperabilität Transpilierung in Javascript |
| Dokumentation & Community Support | | 0.7 | |
| Qualität der Dokumentation | 0.3 | 0 | Gut dokumentiert, doch schlecht navigierbar |
| Größe und Hilfsbereitschaft der Community | 0.1 | 1 | nicht identifiziert |
| offizielle/inoffizielle Foren und/oder Gruppen | 0.2 | 1 | Google-, Facebook-Gruppe, Mailing-Liste |
| Beispiele, Tutorials, Bücher | 0.3 | 2 | - Referenzseite mit 12 Kapiteln - 6 verschiedene Tracks zum Lernen von Quorum innerhalb verschiedener Anwendungsbereichen - partizipiert in Hour of Code |

| | | | |
|--|-----|----|------|
| Dokumentation, Tutorials, Beispiele, Lehrmaterial internationalisiert? | 0.1 | -2 | Nein |
|--|-----|----|------|

B.3. Scratch

| Kriterium | Gewicht | Bewertung | Begründung |
|---|---------|-----------|---|
| Einfachheit der Implementierung der Computerspiele | | 0.525 | |
| Alle Anforderungen erfüllt | 0.15 | | Ja |
| Aufwand der Implementierung | 0.15 | 1 | nicht höher als Quorum |
| Komplexität der Implementierung (Hacks) | 0.25 | 0 | Tic-Tac-Toe: für jedes Feld Code kopieren und modifizieren |
| Grafischer Szenen-Editor | 0.15 | 1 | Ja, nur Positionierung Rotation und Größe über Objekt-Eigenschafts-Panel |
| Koordinatensystem und Drehrichtung nach mathematischer Konvention | 0.025 | -2 | rechtshändiges Koordinatensystem in der Mitte, doch Drehrichtung im Uhrzeigersinn |
| Ankerpunkt gut gesetzt? | 0.025 | -1 | Ankerpunkt in der Mitte, besser wäre in der Ecke |
| GUI-Bibliothek? | 0.05 | 0 | Nein, jedoch können Elemente über Objekte erstellt werden |
| Abspielen von Audio? | 0.05 | 2 | Ja |
| Animation von Sprites? | 0.05 | 0 | Nein, Implementierung ist jedoch nicht kompliziert |
| Abstraktion von <i>Game Loop</i> ? | 0.1 | 2 | Update-Loop muss selbst implementiert werden Eingaben über Ereignis-Blöcke |
| Erlernbarkeit nach Konstruktionismus | | 1.8 | |
| Anzahl und Art der Mikrokosmen | 0.2 | 2 | über Scratch-Erweiterungen: 11 offizielle Erweiterungen zahlreiche inoffizielle Erweiterungen |
| Unterstützung von <i>Turtle</i> -Grafiken | 0.1 | 2 | Ja über Malstift-Erweiterung |
| Einbindung von unterschiedlichen Medien und Technologien | 0.2 | 2 | sehr hoch, Grafiken, Audio, Musik, Roboter und physische Gadgets |
| Selbstoffenbarung der Umgebung | 0.2 | 1 | Ja relativ selbstoffenbarend |
| Sicherheit der Umgebung | 0.1 | 2 | Ziemlich sicher |
| Dynamische Typisierung | 0.2 | 2 | Ja |
| Inkrementelle Einführung | | -0.5 | |
| Einfachheit des Programms "Hallo Nutzer" | 0.5 | -1 | komplexer, da Blöcke aus 4 verschiedenen Bereichen verwendet werden |
| Inkrementelle Einführung der Programmierkonzepte | 0.5 | 0 | Ja, es können Programme mit einer kleinen Menge von Blockarten geschrieben werden |
| Intuitive Syntax und Natürlichsprachlichkeit | | 2 | |

| | | | |
|--|-------|-------|---|
| Intuitivität der Syntax | 0.6 | 2 | Sehr intuitiv, Programme lassen sich wie Puzzle erstellen |
| Internationalisierung der Syntax | 0.25 | 2 | Ja, in über 60 Sprachen internationalisiert |
| syntaktische Feinheiten | 0.15 | 2 | Blöcke haben verschiedene Formen (je nach Art) und Farben (je nach Kategorie) |
| Feature Uniformity und Orthogonalität | | 1 | |
| wenige syntaktische Synonyme | 0.2 | 0 | es gibt z.B. 3 Schleifen-Arten |
| wenige syntaktische Homonyme | 0.2 | 2 | Keine |
| kleine, simple Funktionsmenge | 0.3 | 0 | Die Anzahl der Blöcke wird möglichst reduziert, doch für die Usability gibt es mehr Blöcke als nötig wären |
| konsistente Syntax (Orthogonalität) | 0.3 | 2 | Durch Blöcke kein Raum für inkonsistente Syntax |
| Programmierkonzepte & Datenstrukturen | | -0.25 | |
| strukturierte Programmierung | 0.25 | 2 | Schleife: Js Bedingungen: Ja |
| prozedurale Programmierung | 0.25 | -1 | Funktionen: ohne Rückgabewert lokale Variablen: Nein Rekursion Ja |
| Datenstrukturen | 0.25 | -1 | Liste: Ja Queue: nur über Liste Stack: nur über Liste HashMap: Nein Set: Nein |
| Implementierung der Fibonacci-Funktion | 0.15 | -1 | Sehr aufwendig und nur über Hack |
| self-hosting Compiler? | 0.05 | -2 | Nein |
| Programmierumgebung (IDE) | | 1.35 | |
| Einfachheit der Umgebung | 0.25 | 2 | Sehr einfach und sauber |
| Interaktivität | 0.25 | 2 | Turnaround Time: keine, direkt ausführbar Interaktive Programmierung (REPL): Nein Live Coding: Ja |
| Debugging Support | 0.2 | 1 | Ja, aber keinen Schritt-Für-Schritt-Debugger |
| Einfachheit der Installation | 0.075 | 2 | Web-IDE => sehr einfach |
| Einfachheit der Kollaboration | 0.075 | 2 | Programmieren im Team: Studios Teilen von Programmen: Veröffentlichung, Remixing, Rucksack (Copy-Paste von Skripts) |
| Moderne Features | 0.15 | -1 | Syntax Highlighting: Ja, Farben und Formen von Blöcken Code-Completion: Nein Navigation: Nein Formatierer: Ja Refactoring: Nein |
| Clean Code | | -1.65 | |
| Organisation in Modulen | 0.15 | -2 | Scratch-Scripts separat pro Objekt, kein Code-Sharing zwischen Objekten |

| | | | |
|---|------|-----|---|
| Information Hiding | 0.15 | -1 | objektlokale Variablen |
| Dokumentations-Kommentare | 0.03 | -2 | Nein |
| statische Typisierung | 0.2 | -2 | Nein |
| explizite Casts und Typumwandlungen | 0.1 | -2 | implizite Casts und Typumwandlungen (Fail-Soft-Strategie) |
| Exception-Handling | 0.1 | -2 | Nein |
| Unit-Tests | 0.1 | 0 | kein First-Support, kann über Blöcke repräsentiert werden |
| Style-Guide | 0.1 | -2 | Nein |
| Weitere Eigenschaften | 0.07 | -2 | Nein |
| Fehlermeldungen | | 2 | Durch das Design von Scratch wird die Notwendigkeit von Fehlermeldungen vollkommen eliminiert |
| vereinfachtes Vokabular | 0.1 | | |
| Fehlerposition wird im Code angezeigt? | 0.2 | | |
| unterstützende Elemente | 0.1 | | |
| keine störenden Elemente | 0.1 | | |
| keine falsche Lösung vorschlagen | 0.15 | | |
| positiver Ton | 0.15 | | |
| Kontext und Informationen anzeigen, um Fehler zu beheben | 0.1 | | |
| Fehlermeldung lokalisiert | 0.1 | | |
| hohes Abstraktionsniveau für Datentypen | | 0 | |
| hohe String-Abstraktion im Unicode-Standard | 0.5 | 2 | Ja |
| einiger Zahlentyp Langzahlarithmetik | 0.5 | -2 | Nein |
| Anwendungsbereiche und Interoperabilität | | 1.3 | |
| Anwendungsbereiche der Programmiersprache | 0.5 | 2 | 11 offizielle Erweiterungen und viele weitere inoffizielle |
| Plattformen | 0.3 | 1 | Web, Hardware-Verbindung über Scratch Link |
| Interoperabilität mit oder Transpilierung in andere Programmiersprachen | 0.2 | 0 | Projekte wie Sulfurous-Player oder das Leopard-Projekt erlauben Transpilierung in Javascript |
| Dokumentation & Community Support | | 2 | |
| Qualität der Dokumentation | 0.3 | 2 | Scratch-Wiki |
| Größe und Hilfsbereitschaft der Community | 0.1 | 2 | größte Teaching-Language-Community über 50 Millionen Projekte |
| offizielle/inoffizielle Foren und/oder Gruppen | 0.2 | 2 | Scratch Forum sowie zahlreiche andere Gruppen auf verschiedenen Social Media-Kanälen |
| Beispiele, Tutorials, Bücher | 0.3 | 2 | sehr viele |
| Dokumentation, Tutorials, Beispiele, Lehrmaterial internationalisiert? | 0.1 | 2 | Ja |

B.4. ToonTalk

| Kriterium | Gewicht | Bewertung | Begründung |
|---|---------|-----------|--|
| Einfachheit der Implementierung der Computerspiele | | -0.375 | |
| Alle Anforderungen erfüllt | 0.15 | -1 | Tic-Tac-Toe kann nicht neu gestartet werden. Die anderen Spiele wurden nicht implementiert. |
| Aufwand der Implementierung | 0.15 | -2 | sehr aufwendig |
| Komplexität der Implementierung (Hacks) | 0.25 | -2 | Komplex, die Maus musste bspw. selbst implementiert werden |
| Grafischer Szenen-Editor | 0.15 | 2 | ToonTalk (das Spiel) an sich ist der Editor |
| Koordinatensystem und Drehrichtung nach mathematischer Konvention | 0.025 | -1 | rechtshändig, keine Drehung möglich |
| Ankerpunkt gut gesetzt? | 0.025 | 0 | konnte nicht ermittelt werden |
| GUI-Bibliothek? | 0.05 | -2 | Nein |
| Abspielen von Audio? | 0.05 | 2 | Ja |
| Animation von Sprites? | 0.05 | 2 | Ja |
| Abstraktion von <i>Game Loop</i> ? | 0.1 | 2 | ToonTalk ist ein Computerspiel und man befindet sich somit schon im <i>Game Loop</i> |
| Erlernbarkeit nach Konstruktionismus | | 0.7 | |
| Anzahl und Art der Mikrokosmen | 0.2 | 0 | Grafik, Sound & Musik, Computerspiele, LEGO Mindstorms |
| Unterstützung von <i>Turtle</i> -Grafiken | 0.1 | -2 | Nein |
| Einbindung von unterschiedlichen Medien und Technologien | 0.2 | 1 | Bilder, Sound, Musik, LEGO Mindstorms |
| Selbstoffenbarung der Umgebung | 0.2 | 2 | ToonTalk ist ein Computerspiel, Soundeffekte und Animationen |
| Sicherheit der Umgebung | 0.1 | -1 | Aktionen können rückgängig gemacht werden, ToonTalk kann crashen |
| Dynamische Typisierung | 0.2 | 2 | Ja |
| Inkrementelle Einführung | | 1.5 | |
| Einfachheit des Programms "Hallo Nutzer" | 0.5 | 1 | Roboter trainieren |
| Inkrementelle Einführung der Programmierkonzepte | 0.5 | 2 | Puzzle-Game in ToonTalk |
| Intuitive Syntax und Natürlichsprachlichkeit | | 2 | Es wird gar keine Syntax benötigt! |
| Intuitivität der Syntax | 0.6 | | 2 |
| Internationalisierung der Syntax | 0.25 | | |
| syntaktische Feinheiten | 0.15 | | |
| Feature Uniformity und Orthogonalität | | 2 | |
| wenige syntaktische Synonyme | 0.2 | 2 | keine Überschneidung von Funktionen |
| wenige syntaktische Homonyme | 0.2 | 2 | |
| kleine, simple Funktionsmenge | 0.3 | 2 | gute Integration von Funktionen |
| konsistente Syntax (Orthogonalität) | 0.3 | 2 | Werkzeuge sind konsistent in ihrer Verwendung |
| Programmierkonzepte & Datenstrukturen | | 0.65 | |

| | | | |
|--|-------|-------|---|
| strukturierte Programmierung | 0.25 | 2 | Schleife: Roboter führen wiederholt Aktionen aus Bedingungen: über Bedingungen der Roboter |
| prozedurale Programmierung | 0.25 | 1 | Funktionen: Ja über Truck, Roboter, und Nest lokale Variablen: teilweise, Variablen sind im Prozess lokal Rekursion: Ja |
| Datenstrukturen | 0.25 | 0 | Liste: Ja über Boxen Queue: Ja über Nest Stack: Nein HashMap: Ja über Notizbuch aber nur String als Schlüssel Set: Nein |
| Implementierung der Fibonacci-Funktion | 0.15 | | Ja, aber wenig performant |
| self-hosting Compiler? | 0.05 | -2 | Nein |
| Programmierumgebung (IDE) | | 1.225 | |
| Einfachheit der Umgebung | 0.25 | 1 | einfach gestaltet |
| Interaktivität | 0.25 | 2 | hohe Interaktivität |
| Debugging Support | 0.2 | 2 | hohe Interaktivität, Animation, Zeitreisefunktion |
| Einfachheit der Installation | 0.075 | 1 | über einen Installer |
| Einfachheit der Kollaboration | 0.075 | 0 | Programmieren im Team: Nein Teilen von Programmen: Ja über Copy-Paste-Funktion |
| Moderne Features | 0.15 | | moderne Features einer IDE kann für ToonTalk nicht angewandt werden |
| Clean Code | | -0.28 | |
| Organisation in Modulen | 0.15 | 1 | Programme können in Notzbücher gespeichert und wiederverwendet werden. Elemente können auf die Rückseite von Objekten platziert werden. |
| Information Hiding | 0.15 | 0 | Ja über die Rückseite von Objekten. Wenn Objekte jedoch miteinander verbunden werden sollen, müssen sie umgedreht werden. |
| Dokumentations-Kommentare | 0.03 | 0 | Kommentare schon, jedoch nicht zur Generierung von Dokumentation |
| statische Typisierung | 0.2 | | Nein, jedoch strenge Typisierung |
| explizite Casts und Typumwandlungen | 0.1 | 0 | es gibt überhaupt keine Casts oder Typumwandlungen |
| Exception-Handling | 0.1 | -2 | Nein |
| Unit-Tests | 0.1 | -1 | Unit-Tests können nachgebaut werden |
| Style-Guide | 0.1 | -2 | Nein |
| Weitere Eigenschaften | 0.07 | 1 | Operatoren ohne Nebeneffekte: Ja Keine Überladung von Operatoren: Ja |
| Fehlermeldungen | | 1 | |
| vereinfachtes Vokabular | 0.1 | 2 | Ja |

| | | | |
|---|------|------|--|
| Fehlerposition wird im Code angezeigt? | 0.2 | 0 | rot blinkende Bedingung bei Robotern |
| unterstützende Elemente | 0.1 | 2 | Ja, z.B. rot blinkende Bedingung |
| keine störenden Elemente | 0.1 | 2 | Ja |
| keine falsche Lösung vorschlagen | 0.15 | 2 | Ja |
| positiver Ton | 0.15 | 2 | Ja |
| Kontext und Informationen anzeigen, um Fehler zu beheben | 0.1 | 0 | nur Marty |
| Fehlermeldung lokalisiert | 0.1 | -2 | Nein |
| hohes Abstraktionsniveau für Datentypen | | 2 | |
| hohe String-Abstraktion im Unicode-Standard | 0.5 | 2 | Ja |
| einiger Zahlentyp Langzahlarithmetik | 0.5 | 2 | Ja, sogar besondere Anzeige-Lösung für lange Zahlen |
| Anwendungsbereiche und Interoperabilität | | 0.7 | |
| Anwendungsbereiche der Programmiersprache | 0.5 | 1 | 2D-Grafiken, Computerspiele, Musik und Sound, LEGO Mindstorm Roboter |
| Plattformen | 0.3 | 0 | Windows und mit ToonTalk Reborn im Web |
| Interoperabilität mit oder Transpilierung in andere Programmiersprachen | 0.2 | 1 | ToonTalk-Programme können in Java-Applets umgewandelt werden |
| Dokumentation & Community Support | | -0.3 | |
| Qualität der Dokumentation | 0.3 | 0 | ausführlich, doch schlecht lesbar und navigierbar |
| Größe und Hilfsbereitschaft der Community | 0.1 | 0 | kleine Community, große Hilfsbereitschaft |
| offizielle/inoffizielle Foren und/oder Gruppen | 0.2 | -2 | Keine |
| Beispiele, Tutorials, Bücher | 0.3 | 1 | Puzzle-Game, Beispiele und Demos sind in ToonTalk selbst eingebettet |
| Dokumentation, Tutorials, Beispiele, Lehrmaterial internationalisiert? | 0.1 | -2 | Nein |

B.5. Game-Changineer

| Kriterium | Gewicht | Bewertung | Begründung |
|---|---------|-----------|---|
| Einfachheit der Implementierung der Computerspiele | | -0.375 | |
| Alle Anforderungen erfüllt | 0.15 | -1 | Tic-Tac-Toe: Spielendbedingungen konnten nicht umgesetzt werden, Gewinner oder Unentschieden wird nicht angezeigt Flappy Bird: keine Sound-Effekte |
| Aufwand der Implementierung | 0.15 | 0 | aufwendig, Unschärfe der Natürlichen Sprache und Bugs |
| Komplexität der Implementierung (Hacks) | 0.25 | -1 | viele Hacks |
| Grafischer Szenen-Editor | 0.15 | 0 | textueller Map-Editor |
| Koordinatensystem und Drehrichtung nach mathematischer Konvention | 0.025 | 1 | linkshändig, Drehrichtung richtig |
| Ankerpunkt gut gesetzt? | 0.025 | 2 | Ja |

| | | | |
|--|------|--------|--|
| GUI-Bibliothek? | 0.05 | -2 | Nein |
| Abspielen von Audio? | 0.05 | -2 | Nein |
| Animation von Sprites? | 0.05 | -1 | nur vorgebene Animationen |
| Abstraktion von <i>Game Loop</i> ? | 0.1 | 2 | sehr hoch |
| Erlernbarkeit nach Konstruktionismus | | 0.6 | |
| Anzahl und Art der Mikrokosmen | 0.2 | -1 | Computerspiele, Simulationen, Kunst und Animationen |
| Unterstützung von <i>Turtle</i> -Grafiken | 0.1 | 2 | Ja |
| Einbindung von unterschiedlichen Medien und Technologien | 0.2 | -2 | Nein |
| Selbstoffenbarung der Umgebung | 0.2 | 2 | selbstoffenbarend durch Natürliche Sprache und Beispielvokabeln |
| Sicherheit der Umgebung | 0.1 | 2 | Webanwendung, sicher, Projekte müssen manuell gespeichert werden |
| Dynamische Typisierung | 0.2 | 2 | Ja |
| Inkrementelle Einführung | | -0.5 | |
| Einfachheit des Programms "Hallo Nutzer" | 0.5 | -2 | ist nicht möglich |
| Inkrementelle Einführung der Programmierkonzepte | 0.5 | 1 | Ja: Beginner-, Mittel- und Fortgeschrittenen-Stufe |
| Intuitive Syntax und Natürlichsprachlichkeit | | 1 | |
| Intuitivität der Syntax | 0.6 | 2 | Natürliche Sprache, lässt sich wie eine Spielbeschreibung lesen |
| Internationalisierung der Syntax | 0.25 | -2 | Nein |
| syntaktische Feinheiten | 0.15 | 2 | |
| Feature Uniformity und Orthogonalität | | -1.1 | |
| wenige syntaktische Synonyme | 0.2 | -2 | sehr viele Synonyme und Arten sich auszudrücken |
| wenige syntaktische Homonyme | 0.2 | 1 | konnten nicht identifiziert werden |
| kleine, simple Funktionsmenge | 0.3 | -1 | eher groß |
| konsistente Syntax (Orthogonalität) | 0.3 | -2 | inserts[] passt nicht in die Syntax |
| Programmierkonzepte & Datenstrukturen | | -1.25 | |
| strukturierte Programmierung | 0.25 | 0 | Schleife: nicht explizit, nur über Game Loop Bedingungen: Ja |
| prozedurale Programmierung | 0.25 | -2 | Nein |
| Datenstrukturen | 0.25 | -2 | nur Liste (eventuell) |
| Implementierung der Fibonacci-Funktion | 0.15 | -1 | Nein, nur iterativ möglich |
| self-hosting Compiler? | 0.05 | -2 | Nein |
| Programmierumgebung (IDE) | | -0.725 | |
| Einfachheit der Umgebung | 0.25 | 0 | Usability könnte besser sein |
| Interaktivität | 0.25 | -1 | Turnaround Time: ca. 1 Sekunde Interaktive Programmierung (REPL): Nein Live Coding: Nein |

| | | | |
|---|-------|-------|--|
| Debugging Support | 0.2 | -2 | kein Debugging Support |
| Einfachheit der Installation | 0.075 | 2 | |
| Einfachheit der Kollaboration | 0.075 | -1 | Programmieren im Team: Nein Teilen von Programmen: Copy Paste, Community-Showcase |
| Moderne Features | 0.15 | -1 | Syntax Highlighting: Nein Code-Completion: Ja Navigation: Nein Formatierer: Nein Refactoring: Nein |
| Clean Code | | -1.67 | keine Möglichkeit Code zu organisieren |
| Organisation in Modulen | 0.15 | -2 | |
| Information Hiding | 0.15 | -2 | |
| Dokumentations-Kommentare | 0.03 | -1 | |
| statische Typisierung | 0.2 | -2 | |
| explizite Casts und Typumwandlungen | 0.1 | -2 | |
| Exception-Handling | 0.1 | -2 | |
| Unit-Tests | 0.1 | -2 | |
| Style-Guide | 0.1 | 1 | |
| Weitere Eigenschaften | 0.07 | -2 | |
| Fehlermeldungen | | 0.6 | |
| vereinfachtes Vokabular | 0.1 | -1 | Nein |
| Fehlerposition wird im Code angezeigt? | 0.2 | 2 | Ja |
| unterstützende Elemente | 0.1 | 2 | Beispiele, Anzeige im Code, Text-To-Speech, Notes und Warnungen |
| keine störenden Elemente | 0.1 | -2 | Nein: Text-To-Speech, sehr verbos |
| keine falsche Lösung vorschlagen | 0.15 | 2 | Ja |
| positiver Ton | 0.15 | 0 | neutraler Ton |
| Kontext und Informationen anzeigen, um Fehler zu beheben | 0.1 | 2 | Ja, sehr viele Informationen |
| Fehlermeldung lokalisiert | 0.1 | -2 | Nein |
| hohes Abstraktionsniveau für Datentypen | | -2 | |
| hohe String-Abstraktion im Unicode-Standard | 0.5 | -2 | Nein |
| einzigster Zahlentyp Langzahlarithmetik | 0.5 | -2 | Javascript-Zahlen (64-Bit-Fließkommazahl) |
| Anwendungsbereiche und Interoperabilität | | -1.2 | |
| Anwendungsbereiche der Programmiersprache | 0.5 | -1 | Computerspiele, Simulationen, Kunst und Animationen |
| Plattformen | 0.3 | -1 | Web |
| Interoperabilität mit oder Transpilierung in andere Programmiersprachen | 0.2 | -2 | Nein |
| Dokumentation & Community Support | | -0.3 | |

| | | | |
|--|-----|----|--|
| Qualität der Dokumentation | 0.3 | 0 | ausführlich, aber nicht so übersichtlich und navigierbar |
| Größe und Hilfsbereitschaft der Community | 0.1 | 0 | klein, aber das Team dahinter ist sehr hilfsbereit |
| offizielle/inoffizielle Foren und/oder Gruppen | 0.2 | -2 | Nein |
| Beispiele, Tutorials, Bücher | 0.3 | 1 | (Video-)Tutorials, Beispiele, Hour Of Code |
| Dokumentation, Tutorials, Beispiele, Lehrmaterial internationalisiert? | 0.1 | -2 | Nein |

B.6. Rangliste pro Bewertungsbereich

| Bewertungsbereich | Ranking |
|---|---|
| Einfachheit der Implementierung von Computerspielen | 1. Quorum (1.4) 2. Scratch (0.5) 3. Pyret (-0.3) 4. Game-Changineer (-0.4) 5. ToonTalk (-0.4) |
| Erlernbarkeit nach Konstruktionismus | 1. Scratch (1.8) 2. ToonTalk (0.7) 3. Game-Changineer (0.6) 4. Quorum (0.4) 5. Pyret (0.0) |
| Inkrementelle Einführung | 1. Quorum (2.0) 2. ToonTalk (1.5) 3. Pyret (1.0) 4. Game-Changineer (-0.5) 5. Scratch (-0.5) |
| Intuitive Syntax und Natürlichsprachlichkeit | 1. ToonTalk (2.0) 2. Scratch (2.0) 3. Game-Changineer (1.0) 4. Quorum (0.9) 5. Pyret (-0.4) |
| Feature Uniformity und Orthogonalität | 1. ToonTalk (2.0) 2. Quorum (1.4) 3. Scratch (1) 4. Pyret (0.8) 5. Game-Changineer (-1.1) |
| Unterstützung der wichtigsten Programmierkonzepte & Datenstrukturen | 1. Quorum (1.7) 2. Pyret (1.4) 3. ToonTalk (0.7) 4. Scratch (-0.3) 5. Game-Changineer (-1.3) |
| Programmierumgebung (IDE) | 1. Pyret (1.5) 2. Scratch (1.4) 3. ToonTalk (1.2) 4. Quorum (0.5) 5. Game-Changineer (-0.8) |

| | |
|---|---|
| Das Erzwingen und Erlauben guter Programmierpraktiken und -disziplin (Clean Code) | 1. Quorum (1.7) 2. Pyret (1.6) 3. ToonTalk (-0.3) 4. Scratch (-1.7) 5. Game-Changineer (-1.7) |
| Fehlermeldungen | 1. Scratch (2.0) 2. Pyret (1.4) 3. ToonTalk (1.0) 4. Game-Changineer (0.6) 5. Quorum (0.5) |
| hohes Abstraktionsniveau für Datentypen | 1. ToonTalk (2.0) 2. Pyret (1.5) 3. Scratch (0.0) 4. Quorum (-1) 5. Game-Changineer (-2) |
| Anwendungsbereiche und Interoperabilität mit Technologien und anderen Programmiersprachen | 1. Quorum (2.0) 2. Scratch (1.3) 3. ToonTalk (0.7) 4. Pyret (0.3) 5. Game-Changineer (-1.2) |
| Dokumentation & Community Support | 1. Scratch (2.0) 2. Pyret (1.3) 3. Quorum (0.7) 4. Game-Changineer (-0.3) 5. ToonTalk (-0.3) |

C. Implementierungen

C.1. Implementierungen in Pyret

C.1.1. Fibonacci

Listing C.1. fibonacci.arr

```
fun greater-zero(n :: Number) -> Boolean:  
    n > 0  
end  
  
fun fib(n :: Number%{greater-zero}):  
    if (n == 1) or (n == 2):  
        1  
    else:  
        fib(n - 2) + fib(n - 1)  
    end  
where:  
    fib(1) is 1  
    fib(2) is 1  
    fib(3) is 2  
    fib(4) is 3  
    fib(5) is 5  
end
```

C.1.2. Bouncing Ball

Listing C.2. bouncing-ball.arr

```
import world as W  
import image as I  
  
# 1. Konstanten  
WIDTH = 500  
HEIGHT = 300  
SCENE = I.empty-scene(WIDTH, HEIGHT)  
RADIUS = 10  
BALL = I.circle(RADIUS, "solid", "red")  
SPEED-X = 10  
SPEED-Y = 10  
  
# 2. Datentypen  
data Ball:  
    ball(x :: Number, y :: Number, vx :: Number, vy :: Number)  
end
```

```

# 3. Initialer Weltzustand
INIT_BALL = ball(WIDTH / 2, HEIGHT / 2, SPEED-X, SPEED-Y)

# 4. Hilfefunktionen
fun clamp(n :: Number, min :: Number, max :: Number) -> Number:
    num-max(min, num-min(max, n))
end

# 5. Zeichenfunktion
fun draw(b :: Ball) -> I.Image:
    I.place-image(BALL, b.x, b.y, SCENE)
end

# 6. Updatefunktion
fun update(b :: Ball) -> Ball:
    vx = if (b.x <= RADIUS) or (b.x >= (WIDTH - RADIUS)):
        b.vx * -1 else: b.vx end

    vy = if (b.y <= RADIUS) or (b.y >= (HEIGHT - RADIUS)):
        b.vy * -1 else: b.vy end

    x = clamp(b.x + vx, RADIUS, WIDTH - RADIUS)
    y = clamp(b.y + vy, RADIUS, HEIGHT - RADIUS)

    ball(x, y, vx, vy)
end

# 7. Event-Handlers
# keine Event-Handler

# 8. Welten-Erstellung
W.big-bang(INIT_BALL, [list:
    W.on-tick(update),
    W.to-draw(draw)
])

```

C.1.3. Tic-Tac-Toe

Listing C.3. tictactoe.arr

```

import image as I
import world as W
import lists as L

# CONSTANTS
# -----
WIDTH = 500
HEIGHT = 500
CENTER = WIDTH / 2
FIELD-SIZE = 100
FIELD-SIZE-HALF = FIELD-SIZE / 2

FIELD-SYMBOL-BORDER-SIZE = 20

```

```

FIELD-SYMBOL-SIZE = FIELD-SIZE - FIELD-SYMBOL-BORDER-SIZE

EMPTY-SCENE = I.empty-scene(WIDTH, HEIGHT)

X-COLOR = "red"
O-COLOR = "blue"

#|
X-SYMBOL = I.add-line(I.add-line(
    I.empty-image,
    0, 0, FIELD-SYMBOL-SIZE, FIELD-SYMBOL-SIZE, X-COLOR),
    0, FIELD-SYMBOL-SIZE, FIELD-SYMBOL-SIZE, 0, X-COLOR)
|#

X-SYMBOL = I.empty-image ^
I.add-line(_, 0, 0, FIELD-SYMBOL-SIZE, FIELD-SYMBOL-SIZE, X-COLOR) ^
I.add-line(_, FIELD-SYMBOL-SIZE, 0, 0, FIELD-SYMBOL-SIZE, X-COLOR)

O-SYMBOL = I.circle(FIELD-SYMBOL-SIZE / 2, "outline", O-COLOR)

FIELD = I.square(FIELD-SIZE, "outline", "light-steel-blue")

PLAY-AGAIN-BTN-WIDTH = 200
PLAY-AGAIN-BTN-HEIGHT = 50

PLAY-AGAIN-BUTTON =
I.rectangle(200, PLAY-AGAIN-BTN-HEIGHT, "solid", "light-grey") ^
I.overlay-align("center", "center", I.text("Nochmal spielen", 28, "black"), _) ^
I.overlay-align("center", "center", I.empty-scene(PLAY-AGAIN-BTN-WIDTH, PLAY-AGAIN-BTN-HEIGHT), _)

PLAY-AGAIN-BUTTON-BOT-OFFSET = 50
PLAY-AGAIN-BUTTON-POS = I.point(CENTER, HEIGHT - PLAY-AGAIN-BUTTON-BOT-OFFSET)

TOP-TEXT-OFFSET = 50
GAME-OVER-FONT-SIZE = 54
PLAYERS-TURN-FONT-SIZE = 38

# DATA DEFINITIONS
# -----
data Player:
| x-player
| o-player
end

data Field:
| field(player :: Option<Player>, index :: Number, pos :: I.Point)
end

data Tic-Tac-Toe:
| tic-tac-toe(fields :: List<Field>, player :: Option<Player>, game-over :: Boolean)
end

# INITIALIZE GAME STATE

```

```

# -----
# Berechne die Positionen für die Tic-Tac-Toe-Felder
# die in einem 3x3-Gitter angeordnet sind.
FIELD-POSITIONS = for pos from [list:
    I.point(-1,-1), I.point(0,-1), I.point(1,-1),
    I.point(-1, 0), I.point(0, 0), I.point(1, 0),
    I.point(-1, 1), I.point(0, 1), I.point(1, 1)
    ]):

    I.point(
        CENTER + (pos.x * FIELD-SIZE),
        CENTER + (pos.y * FIELD-SIZE)
    )
end

# Initialisiere die Felder aus den Positionen.
# Weise ihnen ihren Index und ihre Position zu.
INIT-FIELDS = for map_n(index from 0, pos from FIELD-POSITIONS):
    field(None, index, pos)
end

# Initialisiere den initialen Game-State. Der X-Spieler beginnt.
INIT-TIC-TAC-TOE = tic-tac-toe(INIT-FIELDS, some(x-player), false)

# HELPER-FUNCTIONS
# -----
fun is-game-over(t :: Tic-Tac-Toe) -> Boolean:
    t.game-over
end

#|
  Gibt zurück ob ein Punkt mit den Koordinaten 'px' und 'py',
  sich innerhalb (bzw. auf der Kante) eines Rechtecks befindet, welches
  den Mittelpunkt auf den Koordinaten 'rx' und 'ry' hat
  und eine Weite von 'rw' und eine Höhe von 'rh' hat.
|#
fun is-point-inside-rect(
    px :: Number, py :: Number,
    rx :: Number, ry :: Number,
    rw :: Number, rh :: Number
) -> Boolean:
    w-half = rw / 2
    h-half = rh / 2

    l = rx - w-half
    r = rx + w-half
    t = ry - h-half
    b = ry + h-half

    (px >= l) and (px <= r) and (py >= t) and (py <= b)
where:
    is-point-inside-rect(10, 10, 10, 10, 20, 20) is true
    is-point-inside-rect(20, 2, 10, 2, 19, 1) is false

```

```

    is-point-inside-rect(20, 20, 0, 0, 20, 20) is true
end

# DRAW FUNCTIONS
-----
fun draw-player-symbol(p :: Player) -> I.Image:
  cases(Player) p:
    | x-player => X-SYMBOL
    | o-player => O-SYMBOL
  end
end

fun draw-field(f :: Field) -> I.Image:
  cases(Option<Player>) f.player:
    | some(p) => I.overlay(FIELD, draw-player-symbol(p))
    | none => FIELD
  end
end

fun draw-tic-tac-toe-fields(fields :: List<Field>) -> I.Image:
  for fold(scene from EMPTY-SCENE, f from fields):
    I.place-image(draw-field(f), f.pos.x, f.pos.y, scene)
  end
end

fun draw-game-over-text(t :: Tic-Tac-Toe%(is-game-over)) -> I.Image:
  cases(Option<Player>) t.player:
    | some(p) => cases(Player) p:
        | x-player => I.text("X gewinnt!", GAME-OVER-FONT-SIZE, X-COLOR)
        | o-player => I.text("O gewinnt!", GAME-OVER-FONT-SIZE, O-COLOR)
      end
    | none => I.text("Unentschieden", GAME-OVER-FONT-SIZE, "black")
  end
end

fun draw-players-turn-text(t :: Tic-Tac-Toe) -> I.Image:
  cases(Option<Player>) t.player:
    | some(p) => cases(Player) p:
        | x-player => I.text("X ist dran!", PLAYERS-TURN-FONT-SIZE, X-COLOR)
        | o-player => I.text("O ist dran!", PLAYERS-TURN-FONT-SIZE, O-COLOR)
      end
    | none => raise("Es muss ein Spieler dran sein")
  end
end

fun draw(t :: Tic-Tac-Toe) -> I.Image:
  background = draw-tic-tac-toe-fields(t.fields)
  if t.game-over:
    background ^
      I.place-image(draw-game-over-text(t), CENTER, TOP-TEXT-OFFSET, _)
      I.place-image(PLAY AGAIN BUTTON, PLAY-AGAIN-BUTTON-POS.x, PLAY-AGAIN-BUTTON-POS.y, _)
  else:
    background ^

```

```

    I.place-image(draw-players-turn-text(t),CENTER, TOP-TEXT-OFFSET, _)
end
end

# UPDATE FUNCTIONS
#-----

#|
Wird aufgerufen wenn ein Tic-Tac-Toe-Feld gesetzt wird.
Erstellt den neuen Tic-Tac-Toe-State mit dem gesetzten Feld.
Evaluiert ob der Spieler, der das Feld klickt gewonnen hat oder,
ob Unentschieden ist.
|#
fun set-field(t :: Tic-Tac-Toe, f :: Field) -> Tic-Tac-Toe:
  # Setze den Spieler, der gerade dran ist auf das Feld
  new-f = field(t.player, f.index, f.pos)
  # Update die Felder
  new-fields = L.set(t.fields, f.index, new-f)

  # Überprüfe ob das Spiel vorbei ist und updatet den Spieler.
  {game-over; next-p} = if check-player-won(new-fields, t.player):
    # Der Spieler der gewonnen hat, wird zurückgegeben
    {true; t.player}
  else if check-full(new-fields):
    # Unentschieden, deshalb wird 'none' als Spieler zurückgegeben
    {true; none}
  else:
    # Der nächste Spieler ist dran
    {false; alternate-player(t.player)}
  end

  tic-tac-toe(new-fields, next-p, game-over)
end

fun check-player-won(fields :: List<Field>, player :: Option<Player>) -> Boolean:
  WINNING-INDICES = [list:
    # Horizontalen
    [list: 0, 1, 2],
    [list: 3, 4, 5],
    [list: 6, 7, 8],
    # Vertikalen
    [list: 0, 3, 6],
    [list: 1, 4, 7],
    [list: 2, 5, 8],
    # Diagonalen
    [list: 0, 4, 8],
    [list: 2, 4, 6]
  ]

  for any(indices from WINNING-INDICES):
    for L.all(index from indices):
      L.get(fields, index).player == player
    end
  end

```

```

end

fun check-full(fields :: List<Field>) -> Boolean:
    for L.all(f from fields):
        is-some(f.player)
    end
end

fun alternate-player(o :: Option<Player>) -> Option<Player>:
    cases(Option<Player>) o:
        | some(p) => cases(Player) p:
            | x-player => some(o-player)
            | o-player => some(x-player)
        end
        | none => none
    end
end

# EVENT-HANDLERS
# -----
#| Es wird überprüft, ob der Replay-Button gedrückt wurde.
# Wenn er gedrückt wurde wird das Spiel neu gestartet,
# indem das initiale Tic-Tac-Toe-Spiel zurückgegeben wird.
|##
fun on-click-replay-button(t :: Tic-Tac-Toe, mx :: Number, my :: Number) -> Tic-Tac-Toe:
    if is-point-inside-rect(
        mx, my,
        PLAY AGAIN BUTTON POS.x,
        PLAY AGAIN BUTTON POS.y,
        I.image-width(PLAY AGAIN BUTTON),
        I.image-height(PLAY AGAIN BUTTON)):

        INIT-TIC-TAC-TOE
    else:
        t
    end
end

fun on-click-field(t :: Tic-Tac-Toe, mx :: Number, my :: Number) -> Tic-Tac-Toe:
    # Finde ein leeres Feld, auf das mit der Maus geklickt wurde
    clicked-field = for find(f from t.fields):
        is-none(f.player) and
        is-point-inside-rect(mx, my, f.pos.x, f.pos.y, FIELD-SIZE, FIELD-SIZE)
    end

    cases(Option<Field>) clicked-field:
        | some(f) => set-field(t, f)
        | none => t
    end
end

```

```

fun on-mouse(t :: Tic-Tac-Toe, mx :: Number, my :: Number, event :: String) -> Tic-Tac-Toe:
  if event == "button-down":
    if t.game-over:
      on-click-replay-button(t, mx, my)
    else:
      on-click-field(t, mx, my)
    end
  else:
    t
  end
end

# create world
W.big-bang(INIT-TIC-TAC-TOE, [list:
  W.to-draw(draw), # Installiere den Draw-Handler
  W.on-mouse(on-mouse)]) # Installiere den Mouse-Update-Handler

```

C.1.4. Flappy-Bird

Listing C.4. flappy-bird.arr

```

import world as W
import image as I

# CONSTANTS
#-----
WIDTH = 280
HEIGHT = 500

EMPTY-SCENE = I.empty-scene(WIDTH, HEIGHT)

BIRD = I.image-url("https://raw.githubusercontent.com/samuelcust/flappy-bird-assets/master/sprites/
  bluebird-midflap.png")
BIRD-HALF-HEIGHT = I.image-height(BIRD) / 2
BIRD-HALF-WIDTH = I.image-width(BIRD) / 2
BIRD-X-POS = 50
BIRD-X-POS-LEFT = BIRD-X-POS - BIRD-HALF-WIDTH
BIRD-X-POS-RIGHT = BIRD-X-POS + BIRD-HALF-WIDTH

PIPE-BOTTOM = I.image-url("https://raw.githubusercontent.com/samuelcust/flappy-bird-assets/master/
  sprites/pipe-green.png")
PIPE-TOP = I.flip-vertical(PIPE-BOTTOM)
PIPE-HEIGHT = I.image-height(PIPE-BOTTOM)
PIPE-WIDTH = I.image-width(PIPE-BOTTOM)
PIPE-HALF-WIDTH = PIPE-WIDTH / 2

BACKGROUND-IMAGE = I.image-url("https://raw.githubusercontent.com/samuelcust/flappy-bird-assets/master
  /sprites/background-day.png")
BACKGROUND = I.place-image(BACKGROUND-IMAGE, WIDTH / 2, HEIGHT / 2, EMPTY-SCENE)

GAP-HEIGHT = 110
GAP-HALF-HEIGHT = GAP-HEIGHT / 2

```

```

PIPE-SPEED = -7.5
GRAVITY = 0.6

SCORE-TEXT-SIZE = 32
SCORE-Y-POS = 30
END-SCORE-TEXT-SIZE = 45
END-SCORE-Y-POS = 150

PLAY-AGAIN-BUTTON = I.overlay-align(
    "center",
    "center",
    I.text("Nochmal spielen", 28, "black"),
    I.rectangle(200, 50, "solid", "light-grey")
) ^
I.overlay-align("center", "center", I.empty-scene(200, 50), _)

PLAY-AGAIN-BUTTON-POS = I.point(WIDTH / 2, 250)

# DATA DEFINITIONS
-----
data Bird:
    | flappy-bird(y :: Number, v :: Number, collided :: Boolean)
end

data Obstacle:
    | pipe-obstacle(x :: Number, y :: Number)
end

data World:
    | world-playing(bird :: Bird, pipe :: Obstacle, score :: Number)
    | world-game-over(pipe :: Obstacle, end-score :: Number)
end

fun check-bird-pipe-collision(bird :: Bird, pipe :: Obstacle) -> Boolean:
    fun is-bird-inside-gap() -> Boolean:
        bird-top = bird.y - BIRD-HALF-HEIGHT
        bird-bot = bird.y + BIRD-HALF-HEIGHT
        gap-top = pipe.y - GAP-HALF-HEIGHT
        gap-bot = pipe.y + GAP-HALF-HEIGHT
        (bird-top > gap-top) and (bird-bot < gap-bot)
    end

    fun is-bird-between-pipes() -> Boolean:
        pipe-left = pipe.x - PIPE-HALF-WIDTH
        pipe-right = pipe.x + PIPE-HALF-WIDTH

        (BIRD-X-POS-RIGHT >= pipe-left) and (BIRD-X-POS-LEFT <= pipe-right)
    end

    is-bird-between-pipes() and not(is-bird-inside-gap())
where:
    bird-a = flappy-bird(30, 0, false)
    pipe-a = pipe-obstacle(BIRD-X-POS, 30)
    check-bird-pipe-collision(bird-a, pipe-a) is false

```

```

bird-b = flappy-bird(30, 0, false)
pipe-b = pipe-obstacle(BIRD-X-POS, 200)
check-bird-pipe-collision(bird-b, pipe-b) is true
end

# INITIAL GAME STATE
-----
fun new-random-pipe() -> Obstacle:
    half-height = HEIGHT / 2
    quarter-height = HEIGHT / 4
    y = half-height + (num-random(half-height) - quarter-height)
    pipe-obstacle(WIDTH + PIPE-WIDTH, y)
end

INIT-BIRD = flappy-bird(HEIGHT / 2, 0, false)
INIT-PIPE = new-random-pipe()

INIT-WORLD = world-playing(INIT-BIRD, INIT-PIPE, 0)

# HELPER FUNCTIONS
-----
fun clamp(n :: Number, min :: Number, max :: Number) -> Number:
    num-max(min, num-min(max, n))
end

#|
Gibt zurück ob ein Punkt mit den Koordinaten 'px' und 'py',
sich innerhalb (bzw. auf der Kante) eines Rechtecks befindet, welches
den Mittelpunkt auf den Koordinaten 'rx' und 'ry' hat
und eine Weite von 'rw' und eine Höhe von 'rh' hat.
|#
fun is-point-inside-rect(
    px :: Number, py :: Number,
    rx :: Number, ry :: Number,
    rw :: Number, rh :: Number
) -> Boolean:
    w-half = rw / 2
    h-half = rh / 2

    l = rx - w-half
    r = rx + w-half
    t = ry - h-half
    b = ry + h-half

    (px >= l) and (px <= r) and (py >= t) and (py <= b)
where:
    is-point-inside-rect(10, 10, 10, 10, 20, 20) is true
    is-point-inside-rect(20, 2, 10, 2, 19, 1) is false
    is-point-inside-rect(20, 20, 0, 0, 20, 20) is true
end

# UPDATE FUNCTIONS

```

```

#-----
fun update-bird(bird :: Bird, collision-detected :: Boolean) -> Bird:
    y = num-max(BIRD-HALF-HEIGHT, bird.y + bird.v)
    block:
        var v = bird.v + GRAVITY
        when bird.collided:
            v := num-max(15, v)
        end
        flappy-bird(y, v, bird.collided or collision-detected)
    end
end

fun update-playing-world(world :: World) -> World:
    collision-detected = check-bird-pipe-collision(world.bird, world.pipe)
    bird = update-bird(world.bird, collision-detected)
    pipe-x = world.pipe.x + PIPE-SPEED
    is-pipe-outside = pipe-x < (PIPE-HALF-WIDTH * -1)
    pipe = if is-pipe-outside:
        new-random-pipe()
    else:
        pipe-obstacle(pipe-x, world.pipe.y)
    end

    score = if is-pipe-outside and not(bird.collided):
        world.score + 1
    else:
        world.score
    end

    game-over = (bird.y + BIRD-HALF-HEIGHT) > HEIGHT
    if game-over:
        world-game-over(pipe, score)
    else:
        world-playing(bird, pipe, score)
    end
end

fun update-world(world :: World) -> World:
    if is-world-playing(world):
        update-playing-world(world)
    else:
        world
    end
end

# DRAW FUNCTIONS
#-----
fun draw-bird(scene :: I.Image, bird :: Bird) -> I.Image:
    angle = -90 * clamp(bird.v / 20, -1, 1)
    rotated-bird = I.rotate(angle, BIRD)
    I.place-image(rotated-bird, BIRD-X-POS, bird.y, scene)
end

fun draw-pipe(scene :: I.Image, pipe :: Obstacle) -> I.Image:

```

```

scene ^
I.place-image-align(PIPE-TOP, pipe.x, pipe.y - GAP-HALF-HEIGHT, "center", "bottom", _)^
I.place-image-align(PIPE-BOTTOM, pipe.x, pipe.y + GAP-HALF-HEIGHT, "center", "top", _)^
end

fun draw-score(scene :: I.Image, score :: Number) -> I.Image:
  score-str = "Score: " + num-to-string(score)
  score-text = I.text(score-str, SCORE-TEXT-SIZE, "black")
  I.place-image(score-text, WIDTH / 2, SCORE-Y-POS, scene)
end

fun draw-playing-world(world :: World) -> I.Image:
  BACKGROUND ^
  draw-pipe(_, world.pipe) ^
  draw-bird(_, world.bird) ^
  draw-score(_, world.score)
end

fun draw-end-score(scene :: I.Image, score :: Number) -> I.Image:
  score-str = "Score: " + num-to-string(score)
  score-text = I.text(score-str, END-SCORE-TEXT-SIZE, "black")
  I.place-image(score-text, WIDTH / 2, END-SCORE-Y-POS, scene)
end

fun draw-play-again-button(scene :: I.Image) -> I.Image:
  I.place-image(PLAY-AGAIN-BUTTON, PLAY-AGAIN-BUTTON-POS.x, PLAY-AGAIN-BUTTON-POS.y, scene)
end

fun draw-game-over-world(world :: World%(is-world-game-over)) -> I.Image:
  BACKGROUND ^
  draw-pipe(_, world.pipe) ^
  draw-end-score(_, world.end-score) ^
  draw-play-again-button
end

fun draw-world(world :: World) -> I.Image:
  if is-world-playing(world):
    draw-playing-world(world)
  else:
    draw-game-over-world(world)
  end
end

# EVENT HANDLERS
# -----
fun on-key(world :: World, key :: String) -> World:
  if (key == " ") and not(world.bird.collided):
    bird = flappy-bird(world.bird.y, -7, false)
    world-playing(bird, world.pipe, world.score)
  else:
    world
  end
end

```

```

fun on-click-replay(world :: World, mx :: Number, my :: Number) -> World:
  if is-point-inside-rect(mx, my,
    PLAY-AGAIN-BUTTON-POS.x,
    PLAY-AGAIN-BUTTON-POS.y,
    I.image-width(PLAY-AGAIN-BUTTON),
    I.image-height(PLAY-AGAIN-BUTTON)):

    INIT-WORLD
  else:
    world
  end
end

fun on-mouse(world :: World, mx :: Number, my :: Number, event :: String) -> World:
  if (event == "button-down") and is-world-game-over(world):
    on-click-replay(world, mx, my)
  else:
    world
  end
end

# create world
W.big-bang(INIT-WORLD, [list:
  W.to-draw(draw-world),
  W.on-tick(update-world),
  W.on-key(on-key),
  W.on-mouse(on-mouse)
]
)

```

C.1.5. Tamagochi

Listing C.5. positioned-image.arr

```

include image
include world

provide:
  type Positioned-Image,
  pos-image,
  pos-image-contains-point,
  pos-image-draw
end

data Positioned-Image:
| _pos-image(
  pos :: Point,
  img :: Image,
  contains-point :: (Positioned-Image -> Boolean),
  draw :: (Positioned-Image, Image -> Image))
end

fun pos-image-contains-point(pos-img :: Positioned-Image, p :: Point) -> Boolean:

```

```



```

Listing C.6. tamagochi.arr

```

import world as W
import image as I
import shared-gdrive("positioned-image.arr", "1bnu9qW1_8qxHkxt0cy5w7A7DlXk-4Zuj") as P-IMG

# CONSTANTS
#-----
WIDTH = 400
HEIGHT = 600
CENTER = WIDTH / 2

EMPTY-SCENE = I.empty-scene(WIDTH, HEIGHT)

HEART-IMAGE = I.image-url("https://raw.githubusercontent.com/lulugo19/teaching-languages/main/assets/

```

```

    tamagochi/heart.png")
BROCCOLI-IMAGE = I.image-url("https://raw.githubusercontent.com/lulugo19/teaching-languages/main/
    assets/tamagochi/broccoli.png")
CAKE-IMAGE = I.image-url("https://raw.githubusercontent.com/lulugo19/teaching-languages/main/assets/
    tamagochi/cake.png")
PET-IMAGE = I.image-url("https://raw.githubusercontent.com/lulugo19/teaching-languages/main/assets/
    tamagochi/pet.png")
PET-SICK-IMAGE = I.image-url("https://raw.githubusercontent.com/lulugo19/teaching-languages/main/
    assets/tamagochi/pet-sick.png")
PET-SLEEPY-IMAGE = I.image-url("https://raw.githubusercontent.com/lulugo19/teaching-languages/main/
    assets/tamagochi/pet-asleep.png")
TOMBSTONE-IMAGE = I.image-url("https://raw.githubusercontent.com/lulugo19/teaching-languages/main/
    assets/tamagochi/pet-dead.png")
CROSS-IMAGE = I.image-url("https://raw.githubusercontent.com/lulugo19/teaching-languages/main/assets/
    tamagochi/cross.png")

SYMBOL-Y-POS = 150
HEART = P-IMG.pos-image(I.point(75, SYMBOL-Y-POS), HEART-IMAGE)
BROCCOLI = P-IMG.pos-image(I.point(200, SYMBOL-Y-POS), BROCCOLI-IMAGE)
CAKE = P-IMG.pos-image(I.point(325, SYMBOL-Y-POS), CAKE-IMAGE)

PET-POS = I.point(WIDTH / 2, 400)
PET-HEALTHY = P-IMG.pos-image(PET-POS, PET-IMAGE)
PET-SICK = P-IMG.pos-image(PET-POS, PET-SICK-IMAGE)
PET-ASLEEP = P-IMG.pos-image(PET-POS, PET-SLEEPY-IMAGE)
TOMBSTONE = P-IMG.pos-image(PET-POS, TOMBSTONE-IMAGE)

PLAY-AGAIN-BUTTON = I.overlay-align(
    "center",
    "center",
    I.text("Nochmal spielen", 28, "black"),
    I.rectangle(200, 50, "solid", "light-grey")
) ^
I.overlay-align("center", "center", I.empty-scene(200, 50), _) ^
P-IMG.pos-image(I.point(CENTER, HEIGHT - 30), _)

MAX-BAR-LENGTH = 350
BAR-HEIGHT = 25
BAR-LEFT-X-POS = (WIDTH - MAX-BAR-LENGTH) / 2
BAR-TEXT-SIZE = 20

LOVE-BAR-Y-POS = 20
LOVE-BAR-COLOR = "red"
ENERGY-BAR-Y-POS = 50
ENERGY-BAR-COLOR = "yellow"

FPS = 21
ONE-PER-SECOND = 1 / FPS

NORMAL-ENERGY-CHANGE = -1 * ONE-PER-SECOND
NORMAL-LOVE-CHANGE = -1 * ONE-PER-SECOND

ASLEEP-DURATION = FPS * 10
ASLEEP-ENERGY-CHANGE = 2 * ONE-PER-SECOND

```

```

SICK-DURATION = FPS * 15
SICK-ENERGY-CHANGE = -3 * ONE-PER-SECOND

START-LOVE = 75
START-ENERGY = 75

HEART-LOVE-CHANGE = 10
HEART-ENERGY-CHANGE = -15

BROCCOLI-LOVE-CHANGE = -10
BROCCOLI-ENERGY-CHANGE = 15

CAKE-ENERGY-CHANGE = 15
CAKE-SICK-PROBABILITY = 30 # in Prozent

# DATA-DEFINITIONS
#-----
data Tamagochi-Debuff:
| debuff-sick
| debuff-asleep
end

data Tamagochi-Status:
| status-normal
| status-dead
| status-debuff(debuff :: Tamagochi-Debuff, remaining-duration :: Number)
end

NEW-SICK-STATUS = status-debuff(debuff-sick, SICK-DURATION)
NEW-SLEEP-STATUS = status-debuff(debuff-asleep, ASLEEP-DURATION)

fun identity(x :: Any) -> Any: x end

data Tamagochi-Symbol:
| tama-symbol(
  img :: P-IMG.Positioned-Image,
  love-change :: Number,
  energy-change :: Number,
  disabled-debuffs :: List<Tamagochi-Debuff>,
  status-update :: (Tamagochi-Status -> Tamagochi-Status))
end

HEART-SYMBOL = tama-symbol(HEART, 20, -10, [list: debuff-sick, debuff-asleep], identity)
BROCCOLI-SYMBOL = tama-symbol(BROCCOLI, -10, 10, [list: debuff-asleep], identity)
CAKE-SYMBOL = tama-symbol(CAKE, 0, 20, [list: debuff-sick, debuff-asleep],
  lam(s):
    if num-random(100) < CAKE-SICK-PROBABILITY: NEW-SICK-STATUS
    else: s end
  end)

SYMBOLS = [list: HEART-SYMBOL, BROCCOLI-SYMBOL, CAKE-SYMBOL]

data Tamagochi:

```

```

| tama-pet(love :: Number, energy :: Number, status :: Tamagochi-Status, life-time :: Number)
end

INIT-PET = tama-pet(START-LOVE, START-ENERGY, status-normal, 0)

# HELPER FUNCTIONS
-----
fun clamp(n :: Number, min :: Number, max :: Number) -> Number:
    num-max(min, num-min(max, n))
end

fun has-pet-debuff(pet, debuff):
    status = pet.status
    is-status-debuff(status) and (status.debuff == debuff)
end

# DRAW FUNCTIONS
-----
fun draw-symbols(scene :: I.Image, pet :: Tamagochi):
    for fold(s from scene, sym from SYMBOLS):
        symbol = P-IMG.pos-image-draw(sym.img, s)

        disabled = is-status-dead(pet.status) or
        sym.disabled-debuffs.any(lam(d): has-pet-debuff(pet, d) end)

        if disabled:
            I.place-image(CROSS-IMAGE, sym.img.pos.x, sym.img.pos.y, symbol)
        else:
            symbol
        end
    end
end

fun draw-bar(scene :: I.Image, y :: Number, value :: Number, value-desc :: String, color :: String)
-> I.Image:

background = I.empty-scene(MAX-BAR-LENGTH, BAR-HEIGHT)

width = (value / 100) * MAX-BAR-LENGTH
fill = I.rectangle(width, BAR-HEIGHT, "solid", color)

text-str = num-to-string(num-round(value)) + "% " + value-desc
text = I.text(text-str, BAR-TEXT-SIZE, "black")

bar = background ^
I.overlay-align("left", "center", fill, _) ^
I.overlay-align("center", "center", text, _)

I.place-image-align(bar, BAR-LEFT-X-POS, y, "left", "center", scene)
end

fun draw-bars(scene :: I.Image, pet :: Tamagochi) -> I.Image:
    scene ^

```

```

draw-bar(_, LOVE-BAR-Y-POS, pet.love, "Liebe", LOVE-BAR-COLOR) ^
draw-bar(_, ENERGY-BAR-Y-POS, pet.energy, "Energie", ENERGY-BAR-COLOR)
end

fun draw-tombstone-with-life-time(scene :: I.Image, life-time :: Number) -> I.Image:
    life-time-in-seconds = num-round(life-time / FPS)
    tomb-text = I.text(num-to-string(life-time-in-seconds) + " s", 33, "black")
    TOMBSTONE.draw(scene) ^
    I.overlay-onto-offset(tomb-text, "center", "center", 0, -30, _, "center", "center")
end

fun draw-pet(scene :: I.Image, pet :: Tamagochi) -> I.Image:
    if is-status-dead(pet.status):
        draw-tombstone-with-life-time(scene, pet.life-time)
    else:
        pet-img = cases(Tamagochi>Status) pet.status:
            | status-normal => PET-HEALTHY
            | status-debuff(d, _) => cases(Tamagochi-Debuff) d:
                | debuff-sick => PET-SICK
                | debuff-asleep => PET-ASLEEP
            end
        end
        pet-img.draw(scene)
    end
end

fun draw-game-over-when-pet-is-dead(scene :: I.Image, pet :: Tamagochi) -> I.Image:
    if is-status-dead(pet.status):
        PLAY-AGAIN-BUTTON.draw(scene)
    else:
        scene
    end
end

fun draw(pet :: Tamagochi) -> I.Image:
    EMPTY-SCENE ^
    draw-symbols(_, pet) ^
    draw-bars(_, pet) ^
    draw-pet(_, pet) ^
    draw-game-over-when-pet-is-dead(_, pet)
end

# UPDATE FUNCTIONS
-----
fun update-pet-status(pet :: Tamagochi) -> Tamagochi>Status:
    status = pet.status
    if pet.love == 0:
        status-dead
    else:
        pet-is-asleep = has-pet-debuff(pet, debuff-asleep)

        if not(pet-is-asleep) and (pet.energy <= 10):
            NEW-SLEEP-STATUS
        else:

```

```

cases(Tamagochi-Status) status:
| status-debuff(debuff, rem-dur) =>
  if rem-dur == 0: status-normal
  else: status-debuff(debuff, rem-dur - 1) end
| else => status
end
end
end

fun update-pet-love(pet :: Tamagochi) -> Number:
  clamp(pet.love + NORMAL-LOVE-CHANGE, 0, 100)
end

fun update-pet-energy(pet :: Tamagochi) -> Number:
  energy-change = cases(Tamagochi-Status) pet.status:
  | status-normal => NORMAL-ENERGY-CHANGE
  | status-dead => 0
  | status-debuff(d, _) => cases(Tamagochi-Debuff) d:
    | debuff-sick => SICK-ENERGY-CHANGE
    | debuff-asleep => ASLEEP-ENERGY-CHANGE
  end
end

pet.energy + energy-change
end

fun update-pet(pet :: Tamagochi) -> Tamagochi:
  if is-status-dead(pet.status):
    pet
  else:
    updated-status = update-pet-status(pet)
    updated-love = update-pet-love(pet)
    updated-energy = update-pet-energy(pet)
    updated-life-time = pet.life-time + 1
    tama-pet(updated-love, updated-energy, updated-status, updated-life-time)
  end
end

# EVENT HANDLERS
-----
fun on-mouse(pet :: Tamagochi, mx :: Number, my :: Number, event :: String):
  if event == "button-down":
    mouse-pos = I.point(mx, my)

  if is-status-dead(pet.status):
    if PLAY-AGAIN-BUTTON.contains-point(mouse-pos):
      INIT-PET
    else:
      pet
    end
  else:
    clicked-symbol = SYMBOLS.find(
      lam(sym):

```

```
sym.img.contains-point(mouse-pos) and
not(sym.disabled-debuffs.any(has-pet-debuff(pet, _)))
end)

cases (Option<Tamagochi-Symbol>) clicked-symbol:
| some(symbol) =>
  block:
    new-health = clamp(pet.love + symbol.love-change, 0, 100)
    new-energy = clamp(pet.energy + symbol.energy-change, 0, 100)
    new-status = symbol.status-update(pet.status)

    tama-pet(new-health, new-energy, new-status, pet.life-time)
  end
| none => pet
end
else:
  pet
end
end

# create world
W.big-bang(INIT-PET, [list:
  W.to-draw(draw),
  W.on-tick(update-pet),
  W.on-mouse(on-mouse)
])
```

C.2. Implementierungen in Quorum

C.2.1. Fibonacci

Listing C.7. Main.quorum

```

class Main
    action Main
        integer n = cast(integer, input("Welche Fibonacci-Zahl soll berechnet werden: "))
        integer result = Fib(n)
        output "Die " + n + "." + " Fibonacci-Zahl ist: " + result
    end

    action Fib(integer n)
        if n <= 0
            alert(n + " muss größer 0 sein!")
        elseif n = 1 or n = 2
            return 1
        else
            return Fib(n-1) + Fib(n-2)
        end
    end
end

```

C.2.2. Bouncing Ball

Listing C.8. Main.quorum

```

use Libraries.Game.Game
use Libraries.Game.Graphics.Drawable
use Libraries.Game.Graphics.Color
use Libraries.Compute.Vector2
use Libraries.Sound.Audio

class Main is Game
    constant integer WIDTH = 500
    constant integer HEIGHT = 300
    constant integer BALL_RADIUS = 10
    constant integer BALL_DIAMETER = BALL_RADIUS * 2

    integer xSpeed = 100
    integer ySpeed = 100
    Audio bleepAudio
    Drawable ball

    action Main
        SetScreenSize(WIDTH, HEIGHT)
        StartGame()
    end

    action CreateGame
        // Erstelle einen roten Kreis und platziere in in der Mitte des Fensters
        Color color
        ball:LoadFilledCircle(BALL_RADIUS, color:Red())
        ball:SetPosition(WIDTH / 2, HEIGHT / 2)
    end

```

```

// Füge den Ball zum Spiel hinzu
Add(ball)

// Lade den Audio-Clip
bleepAudio:Load("bleep.wav")

action Update(number seconds)
    x = ball:GetX()
    y = ball:GetY()

    // Pralle vom linken Rand ab
    if x < 0
        ball:SetX(0)
        xSpeed = xSpeed * -1
        bleepAudio:Play()
    // Pralle vom rechten Rand ab
    elseif (x + BALL_DIAMETER) > WIDTH
        ball:SetX(WIDTH - BALL_DIAMETER)
        xSpeed = xSpeed * -1
        bleepAudio:Play()
    // Pralle vom unteren Rand ab
    elseif y < 0
        ball:SetY(0)
        ySpeed = ySpeed * -1
        bleepAudio:Play()
    // Pralle vom oberen Rand ab
    elseif (y + BALL_DIAMETER) > HEIGHT
        ball:SetY(HEIGHT - BALL_DIAMETER)
        ySpeed = ySpeed * -1
        bleepAudio:Play()
    end

    // Skaliere die Geschwindigkeit mit den vergangen Sekunden
    // und bewege den Ball
    Vector2 velocity
    velocity:Set(xSpeed, ySpeed)
    velocity:Scale(seconds)
    ball:Move(velocity)
end

```

C.2.3. Tic-Tac-Toe

Listing C.9. Main.quorum

```

use Libraries.Game.Game
use Libraries.Game.Graphics.Drawable
use Libraries.Game.Graphics.Color
use Libraries.Game.Graphics.Label
use Libraries.Containers.Array
use Libraries.Interface.Controls.Button
use Libraries.Interface.Events.MouseEvent
use Libraries.Interface.Events.MouseListener
use Libraries.Interface.Behaviors.Behavior

```

```
use Libraries.Interface.Events.BehaviorEvent
use Libraries.Interface.Item
use Libraries.Interface.Item2D
use Libraries.Language.Errors.CastError

class Main is Game, MouseListener, Behavior
/*
   This action, Main, starts our computer program. In Quorum, programs always
   begin from Main.
*/
constant integer WIDTH = 500
constant integer HEIGHT = 500

constant integer FIELD_SIZE = 90
constant integer FIELD_BORDER = 5
constant integer OFFSET = FIELD_SIZE + 3 * FIELD_BORDER
constant integer SYMBOL_SIZE = 80
constant integer GAME_OVER_FONT_SIZE = 30
constant Color FIELD_COLOR

constant Color DRAW_COLOR
constant Color PLAYER_X_COLOR
constant Color PLAYER_O_COLOR

text player = "X"
Array<Array<Field>> fields
boolean gameOver = false
Label gameLabel
Button replayButton

action Main
    SetScreenSize(WIDTH, HEIGHT)
    SetGameName("Tic Tac Toe")
    StartGame()
end

action CreateGame
    CreateColors()
    CreateGameLabel()
    CreateReplayButton()
    CreateTicTacToe()
end

action CreateColors
    FIELD_COLOR:SetColor(0.4, 0.8, 1.0, 1.0)
    PLAYER_X_COLOR:SetColor(1.0, 0.0, 0.0, 1.0)
    PLAYER_O_COLOR:SetColor(0.0, 0.0, 1.0, 1.0)
    DRAW_COLOR:SetColor(0.0, 0.0, 0.0, 1.0)
end

action CreateGameLabel
```

```

gameLabel:SetFontSize(GAME_OVER_FONT_SIZE)
gameLabel:SetY(HEIGHT - 75)
SetPlayerTurnText()
Add(gameLabel)

end

action CenterItem2DHorizontally(Item2D item)
    number w = item:GetWidth()
    item:SetX(WIDTH / 2 - w / 2)
end

action CreateReplayButton
    replayButton:SetName("Nochmal Spielen")
    replayButton:SetWidth(200)
    replayButton:SetHeight(100)
    replayButton:SetY(50)
    CenterItem2DHorizontally(replayButton)
    replayButton:SetBehavior(me)
    replayButton:Hide()
    Add(replayButton)
end

action CreateTicTacToe
    integer y = 0
    repeat until y = 3
        integer x = 0
        Array<Field> row
        repeat until x = 3
            Field field
            field:Initialize(FIELD_SIZE, SYMBOL_SIZE, FIELD_COLOR, PLAYER_X_COLOR, PLAYER_O_COLOR)
            integer px = OFFSET + x * (FIELD_SIZE + FIELD_BORDER)
            integer py = OFFSET + y * (FIELD_SIZE + FIELD_BORDER)
            field:SetPosition(px, py)
            field:AddMouseListener(me)
            Add(field)
            row:Add(field)
            x = x + 1
        end
        fields:Add(row)
        y = y + 1
    end
end

action ClickedMouse(MouseEvent event)
    if not gameOver and event:IsClicked() and event:IsLeftButtonEvent()
        Item source = event:GetSource()
        Field field = cast(Field, source)
        if field:GetSymbol() = ""
            field:SetSymbol(player)
            endCondition = CheckEndCondition()
            if endCondition not= undefined
                gameOver = true
                player = endCondition
                GameOver()
            end
    end
end

```

```

        else
            AlternatePlayer()
        end
    end
end

action Run(BehaviorEvent event)
    Reset()
end

action Reset
    integer y = 0
    repeat until y = 3
        integer x = 0
        repeat until x = 3
            Field f = fields:Get(y):Get(x)
            f:Clear()
            x = x + 1
        end
        y = y + 1
    end
    player = "X"
    gameOver = false
    replayButton:Hide()
    SetPlayerTurnText()
end

/*
Überprüft die Tic-Tac-Toe Felder auf eine Endbedingung.
Rückgaben:
    undefined => Das Spiel ist noch nicht vorbei
    "X" => Spieler X hat gewonnen
    "O" => Spieler O hat gewonnen
    ""  => Unentschieden
*/
action CheckEndCondition returns text
    boolean hasNoEmptyField = true
    text playerWonDiag1 = fields:Get(0):Get(0):GetSymbol()
    text playerWonDiag2 = fields:Get(0):Get(2):GetSymbol()
    integer i = 0
    repeat until i = 3
        text symbolDiag1 = fields:Get(i):Get(i):GetSymbol()
        if playerWonDiag1 = symbolDiag1
            playerWonDiag1 = symbolDiag1
        else
            playerWonDiag1 = ""
        end

        text symbolDiag2 = fields:Get(i):Get(2-i):GetSymbol()
        if playerWonDiag2 = symbolDiag2
            playerWonDiag2 = symbolDiag2
        else
            playerWonDiag2 = ""
        end
    end

```

```
end

text playerWonRow = fields:Get(i):Get(0):GetSymbol()
text playerWonCol = fields:Get(0):Get(i):GetSymbol()

integer j = 1

repeat until j = 3
    text symbolRow = fields:Get(i):Get(j):GetSymbol()
    if hasNoEmptyField and symbolRow = ""
        hasNoEmptyField = false
    end

    if playerWonRow = symbolRow
        playerWonRow = symbolRow
    else
        playerWonRow = ""
    end

    text symbolCol = fields:Get(j):Get(i):GetSymbol()
    if playerWonCol = symbolCol
        playerWonCol = symbolCol
    else
        playerWonCol = ""
    end

    j = j + 1
end

if playerWonRow not= ""
    return playerWonRow
end

if playerWonCol not= ""
    return playerWonCol
end

i = i + 1
end

if playerWonDiag1 not= ""
    return playerWonDiag1
end

if playerWonDiag2 not= ""
    return playerWonDiag2
end

if hasNoEmptyField
    return ""
end

return undefined
```

```

end

action SetGameLabelText(text str)
    gameLabel:SetText(str)
    CenterItem2DHorizontally(gameLabel)
end

action SetPlayerTurnText
    SetGameLabelText("Spieler " + player + " ist dran!")
    if player = "X"
        gameLabel:SetColor(PLAYER_X_COLOR)
    else
        gameLabel:SetColor(PLAYER_O_COLOR)
    end
end

action AlternatePlayer
    if player = "X"
        player = "O"
    else
        player = "X"
    end
    SetPlayerTurnText()
end

action GameOver
    if player = ""
        SetGameLabelText("Unentschieden!")
        gameLabel:SetColor(DRAW_COLOR)
    else
        SetGameLabelText("Spieler " + player + " hat gewonnen!")
        if player = "X"
            gameLabel:SetColor(PLAYER_X_COLOR)
        else
            gameLabel:SetColor(PLAYER_O_COLOR)
        end
    end
    replayButton>Show()
end

action Update(number time)

end
end

```

Listing C.10. Field.quorum

```

use Libraries.Game.Graphics.Drawable
use Libraries.Game.Graphics.Color
use Libraries.Interface.Events.MouseListener
use Libraries.Interface.Events.MouseEvent

class Field is Drawable, MouseListener
    private integer size = 90

```

```
private integer symbolSize = 80

private text symbol = ""

private Color playerXColor
private Color playerOCOLOR

action Initialize(
    integer size,
    integer symbolSize,
    Color backgroundColor,
    Color playerXColor,
    Color playerOCOLOR)

    me:playerXColor = playerXColor
    me:playerOCOLOR = playerOCOLOR
    LoadFilledRectangle(size, size, backgroundColor)
    AddMouseListener(me)
end

action SetSymbol(text symbol)
    if GetItemCount() > 0
        alert("There is already a symbol on this field")
    end
    me:symbol = symbol
    if symbol = "X"
        Add(CreateCross())
    elseif symbol = "O"
        Add(CreateCircle())
    else
        alert("Invalid symbol!")
    end
end

action GetSymbol returns text
    return symbol
end

action Clear()
    symbol = ""
    if GetItemCount() > 0
        Remove(0)
    end
end

private action ClickedMouse(MouseEvent event)
    event:SetSource(me)
end

private action CreateCross returns Drawable
    Drawable cross

    integer symbolPadding = (size - symbolSize) / 2
```

```

cross:SetWidth(symbolSize)
cross:SetHeight(symbolSize)
cross:SetPosition(symbolPadding, symbolPadding)

Drawable line1
line1:LoadLine(symbolSize, symbolSize, playerXColor)

Drawable line2
line2:Load(line1)
line2:FlipY()
line2:SetColor(playerXColor)

cross:Add(line1)
cross:Add(line2)

return cross
end

private action CreateCircle returns Drawable
Drawable circle

integer symbolPadding = (size - symbolSize) / 2

circle:LoadCircle(symbolSize / 2, player0Color)
circle:SetPosition(symbolPadding, symbolPadding)

return circle
end
end

```

C.2.4. Flappy-Bird

Listing C.11. Main.quorum

```

use Libraries.Containers.Array
use Libraries.Game.Game
use Libraries.Game.Graphics.Drawable
use Libraries.Interface.Events.KeyboardListener
use Libraries.Interface.Events.KeyboardEvent
use Libraries.Interface.Events.CollisionListener2D
use Libraries.Interface.Events.CollisionEvent2D
use Libraries.Game.Graphics.Label
use Libraries.Interface.Controls.Button
use Libraries.Interface.Behaviors.Behavior
use Libraries.Interface.Events.BehaviorEvent
use Libraries.Compute.Vector2
use Libraries.Compute.Random
use Libraries.Compute.Math
use Libraries.Game/DesktopConfiguration
use Libraries.Sound.Audio
use Libraries.System.File

class Main is Game, KeyboardListener, CollisionListener2D, Behavior
constant integer WIDTH = 280
constant integer HEIGHT = 500

```

```
constant integer GRAVITY = 300
constant integer GAP_HEIGHT = 100
constant Vector2 FLY_VELOCITY
constant Vector2 FALL_VELOCITY

Random random
Math math

SpriteAnimation bird
Drawable topPipe
Drawable bottomPipe
Label scoreLabel
Label gameOverLabel
Button playAgainButton
integer score = 0
boolean gameOver = false

Audio pointAudio
Audio hitAudio
Audio wingAudio
Audio gameOverAudio

action Main
    DesktopConfiguration conf
    conf:targetFramesPerSecond = 30
    conf:limitFramesPerSecond = true
    conf:title = "Flappy Bird"
    conf:width = WIDTH
    conf:height = HEIGHT
    conf:multipleKeyPressTimer = 0.5
    conf:vSyncEnabled = true
    conf:minimumFrameDelay = 0.0

    SetConfiguration(conf)
    StartGame()
end

action CreateGame
    // set Icon
    File iconFile
    iconFile:SetPath("../assets/flappy-bird/sprites/bluebird-midflap.png")
    SetApplicationIcon(iconFile)

    // Enable Gravity
    EnablePhysics2D(true)
    SetGravity2D(0, -GRAVITY)

    LoadAudio()

    CreateBackground()
    CreatePipes()
    CreateBird()
    CreateScoreLabel()
    CreateGameOverLabel()
```

```
CreatePlayAgainButton()

// listen to keyboard input
AddKeyboardListener(me)

// check bird and pipe collision
AddCollisionListener(me)
end

action LoadAudio
    wingAudio:Load("../assets/flappy-bird/audio/wing.ogg")
    pointAudio:Load("../assets/flappy-bird/audio/point.ogg")
    hitAudio:Load("../assets/flappy-bird/audio/hit.ogg")
    gameOverAudio:Load("../assets/flappy-bird/audio/die.ogg")
end

action CreateBackground
    Drawable background
    background:Load("../assets/flappy-bird/sprites/background-day.png")
    Add(background)
end

action CreateScoreLabel
    scoreLabel:SetFontSize(20)
    scoreLabel:SetText("Score: 0")
    scoreLabel:SetPosition(100, HEIGHT - 30)
    Add(scoreLabel)
end

action CreateGameOverLabel
    gameOverLabel:SetText("Score: 0")
    gameOverLabel:SetFontSize(30)
    gameOverLabel:SetPosition(80, 250)
    gameOverLabel:Hide()
    Add(gameOverLabel)
end

action CreatePlayAgainButton
    playAgainButton:SetName("Nochmal Spielen!")
    playAgainButton:SetSize(200, 50)
    playAgainButton:SetPosition(70, 180)
    playAgainButton:SetBehavior(me)
    playAgainButton:Hide()
    playAgainButton:SetFocusable(false)
    Add(playAgainButton)
end

action CreateBird
    Drawable midFlap
    midFlap:Load("../assets/flappy-bird/sprites/bluebird-midflap.png")
    Drawable upFlap
    upFlap:Load("../assets/flappy-bird/sprites/bluebird-upflap.png")
    Drawable downFlap
    downFlap:Load("../assets/flappy-bird/sprites/bluebird-upflap.png")
```

```

Array<Drawable> sprites
sprites:Add(midFlap)
sprites:Add(upFlap)
sprites:Add(midFlap)
sprites:Add(downFlap)

bird:Create(sprites, 3)
bird:SetPosition(50, HEIGHT / 2)

// the bird reacts to gravity
bird:EnablePhysics(true)
bird:SetResponsive()
FLY_VELOCITY:Set(0, 200)
FALL_VELOCITY:Set(20, -400)
bird:SetAngularVelocity(-1.5)

Add(bird)
end

action CreatePipes
bottomPipe:Load("../assets/flappy-bird/sprites/pipe-green.png")
topPipe:Load(bottomPipe)
topPipe:FlipY()

bottomPipe:EnablePhysics(true)
bottomPipe:SetNonResponsive()
topPipe:EnablePhysics(true)
topPipe:SetNonResponsive()

Vector2 velocity
velocity:Set(-120, 0)
topPipe:SetLinearVelocity(velocity)
bottomPipe:SetLinearVelocity(velocity)

Add(topPipe)
Add(bottomPipe)

ResetPipes()
end

action ResetPipes
integer offset = random:RandomInteger(HEIGHT / 2)
bottomPipe:SetPosition(WIDTH, -offset)
topPipe:SetPosition(WIDTH, -offset + bottomPipe:GetHeight() + GAP_HEIGHT)
end

action Update(number time)
if not gameOver
    // Animations Update
    bird:Update(time)

    // Der Vogel kann nicht aus dem oberen Rand rausfliegen
    if bird:GetY() + bird:GetHeight() > HEIGHT

```

```

        bird:SetY(HEIGHT - bird:GetHeight())
    // Das Spiel ist vorbei, wenn der Vogel unten aus dem Bildschirm fällt
    elseif bird:GetY() < -bird:GetHeight()
        GameOver()
    end

    bird:SetRotation(math:MinimumOf(bird:GetRotation(), 80))

    // Die Röhren werden nach rechts neu versetzt und Spieler bekommt dann einen Punkt
    if topPipe:GetX() < -topPipe:GetWidth() and bird:IsResponsive()
        pointAudio:Play()
        score = score + 1
        scoreLabel:SetText("Score: " + score:GetText())
        ResetPipes()
    end
end

action GameOver
    gameOverAudio:Play()
    gameOver = true
    gameOverLabel:SetText(scoreLabel:GetText())
    gameOverLabel:Show()
    playAgainButton:Show()
    scoreLabel:Hide()
    bird:SetUnmovable()
    topPipe:SetUnmovable()
    bottomPipe:SetUnmovable()
end

// Das hier wird ausgeführt, wenn der "Nochmal Spielen"-Button gedrückt wird
action Run(BehaviorEvent event)
    gameOver = false
    gameOverLabel:Hide()
    playAgainButton:Hide()
    score = 0
    scoreLabel:SetText("Score: 0")
    scoreLabel:Show()
    bird:SetPosition(50, HEIGHT / 2)
    bird:GetLinearVelocity():SetZero()
    bird:SetRotation(0)
    bird:SetResponsive()
    bottomPipe:SetNonResponsive()
    topPipe:SetNonResponsive()
    ResetPipes()
end

action BeginCollision(CollisionEvent2D event)
    if bird:IsResponsive()
        hitAudio:Play()
        bird:SetNonResponsive()
        bird:SetLinearVelocity(FALL_VELOCITY)
        bird:SetRotation(80)
    end

```

```

end

action PressedKey(KeyboardEvent event)
    if bird:IsResponsive() and event:keyCode = event:SPACE
        wingAudio:Play()
        bird:SetLinearVelocity(FLY_VELOCITY)
        bird:SetRotation(-70)
    end
end
end

```

Listing C.12. SpriteAnimation.quorum

```

use Libraries.Containers.Array
use Libraries.Game.Graphics.Drawable

class SpriteAnimation is Drawable
    private Array<Drawable> sprites
    private number fps = 4

    private number elapsedTime = 0
    private integer spriteIndex = 0

    action SetFPS(number fps)
        me:fps = fps
    end

    action GetFPS returns number
        return fps
    end

    action GetSprites returns Array<Drawable>
        return sprites
    end

    action Create(Array<Drawable> sprites, number fps)
        me:sprites = sprites
        me:fps = fps
        if sprites:GetSize() = 0
            alert("There must be at least one sprite!")
        end
        Load(sprites:Get(spriteIndex))
    end

    action Update(number seconds)
        elapsedTime = elapsedTime + seconds
        if elapsedTime >= 1/fps
            spriteIndex = (spriteIndex + 1) mod sprites:GetSize()
            Load(sprites:Get(spriteIndex))
            elapsedTime = elapsedTime - 1/fps
        end
    end
end

```

C.2.5. Tamagochi

Listing C.13. Main.quorum

```

use Libraries.Containers.Array
use Libraries.Game.Game
use Libraries.Game.Graphics.Drawable
use Libraries.Game.Graphics.Label
use Libraries.Game.Graphics.Color
use Libraries.Interface.Controls.Button
use Libraries.Interface.Behaviors.Behavior
use Libraries.Interface.Events.BehaviorEvent

class Main is Game, PetListener, Behavior

constant integer HEIGHT = 600
constant integer WIDTH = 400

Color color

Pet pet
Bar loveBar
Bar energyBar
Label secondsAliveLabel
Symbol heart
Symbol broccoli
CakeSymbol cake
Button playAgainButton

action Main
    SetGameName("Tamagochi")
    SetScreenSize(WIDTH, HEIGHT)
    StartGame()
end

action CreateGame
    CreateBars()
    CreatePet()
    CreateSymbols()
    CreateSecondsAliveLabel()
    CreatePlayAgainButton()
end

action CreatePet
    pet:Create()
    pet:SetCenter(WIDTH / 2, 200)
    pet:AddPetListener(me)
    Add(pet)
end

action CreateBars
    loveBar:Create(300, 30, color:White(), color:Red(), color:Black(), "Love")
    energyBar:Create(300, 30, color:White(), color:Orange(), color:Black(), "Energy")

```

```
loveBar:SetCenter(WIDTH / 2, HEIGHT - 30)
energyBar:SetCenter(WIDTH / 2, HEIGHT - 70)

loveBar:SetPercent(50)
energyBar:SetPercent(50)

Add(loveBar)
Add(energyBar)
end

action CreateSymbols
    heart:Load("../assets/tamagochi/heart.png")
    broccoli:Load("../assets/tamagochi/broccoli.png")
    cake:Load("../assets/tamagochi/cake.png")

    heart>Create(pet, 10, -15)
    broccoli>Create(pet, -10, 15)
    cake>Create(pet, 0, 15)

    heart:SetPosition(40, 400)
    broccoli:SetPosition(150, 400)
    cake:SetPosition(260, 400)

    Add(heart)
    Add(broccoli)
    Add(cake)
end

action CreatePlayAgainButton
    playAgainButton:SetName("Nochmal Spielen")
    playAgainButton:SetSize(200, 50)
    playAgainButton:SetCenter(WIDTH / 2, 30)
    playAgainButton:Hide()
    playAgainButton:SetBehavior(me)
    Add(playAgainButton)
end

action CreateSecondsAliveLabel
    secondsAliveLabel:SetFontSize(30)
    secondsAliveLabel:SetText("10 s")
    secondsAliveLabel:SetPosition(WIDTH / 2 - secondsAliveLabel:GetWidth() / 2, 250)
    secondsAliveLabel:Hide()
    Add(secondsAliveLabel)
end

action Update(number seconds)
    pet:Update(seconds)
end

// Reset Game
action Run(BehaviorEvent event)
    pet:Reset()
    playAgainButton:Hide()
    secondsAliveLabel:Hide()
```

```

end

action OnPetLoveChanged(number newLove)
    loveBar:SetPercent(newLove)
end

action OnPetEnergyChanged(number newEnergy)
    energyBar:SetPercent(newEnergy)
end

action OnPetStatusChanged(integer newStatus)
    if newStatus = pet:STATUS_ASLEEP or newStatus = pet:STATUS_DEAD
        cake:Disable()
        broccoli:Disable()
        heart:Disable()
    elseif newStatus = pet:STATUS_SICK
        cake:Disable()
        heart:Disable()
    elseif newStatus = pet:STATUS_NORMAL
        cake:Enable()
        broccoli:Enable()
        heart:Enable()
    end

    if newStatus = pet:STATUS_DEAD
        secondsAliveLabel:SetText(pet:GetSecondsAlive():GetText() + " s")
        secondsAliveLabel>Show()
        playAgainButton>Show()
    end
end
end

```

Listing C.14. Pet.quorum

```

use Libraries.Game.Graphics.Drawable
use Libraries.Compute.Math
use Libraries.Containers.Array
use Libraries.Containers.Iterator
use Libraries.Game.Graphics.Label

class Pet is Drawable
    public constant integer STATUS_NORMAL = 0
    public constant integer STATUS_ASLEEP = 1
    public constant integer STATUS_SICK = 2
    public constant integer STATUS_DEAD = 3

    private Math math

    constant Array<Drawable> STATUS_IMAGES
    private integer status = STATUS_NORMAL
    private number remainingDebuffTime = 0
    private number secondsAlive = 0

    private Array<PetListener> listeners

```

```
private number love = 75
private number energy = 75
private Label secondsAliveLabel

action Create
    LoadStatusImages()
    Reset()
end

action LoadStatusImages
    Drawable normal
    normal:Load("../assets/tamagochi/pet.png")
    Drawable asleep
    asleep:Load("../assets/tamagochi/pet-asleep.png")
    Drawable sick
    sick:Load("../assets/tamagochi/pet-sick.png")

    Drawable dead
    dead:Load("../assets/tamagochi/pet-dead.png")
    dead:Add(secondsAliveLabel)

    STATUS_IMAGES:Add(normal)
    STATUS_IMAGES:Add(asleep)
    STATUS_IMAGES:Add(sick)
    STATUS_IMAGES:Add(dead)
end

action AddPetListener(PetListener listener)
    listeners:Add(listener)
end

action RemovePetListener(PetListener listener)
    listeners:Remove(listener)
end

action SetStatus(integer status, number debuffTime)
    me:status = status
    remainingDebuffTime = debuffTime
    Load(STATUS_IMAGES:Get(status))

    Iterator<PetListener> it = listeners:GetIterator()
    repeat while it:HasNext()
        it:Next():OnPetStatusChanged(status)
    end
end

action GetEnergy returns integer
    return math:Round(energy):GetInteger()
end

private action SetEnergy(number energy)
    me:energy = Clamp(energy, 0, 100)
    Iterator<PetListener> it = listeners:GetIterator()
```

```

repeat while it:HasNext()
    it:Next():OnPetEnergyChanged(energy)
end

if energy <= 10
    SetStatus(STATUS_ASLEEP, 10)
end
end

action ChangeEnergy(number change)
    SetEnergy(energy + change)
end

action GetLove returns integer
    return math:Round(love):GetInteger()
end

private action SetLove(number love)
    me:love = Clamp(love, 0, 100)

Iterator<PetListener> it = listeners:GetIterator()
repeat while it:HasNext()
    it:Next():OnPetLoveChanged(love)
end

if love <= 0
    secondsAliveLabel:SetText(secondsAlive:GetText() + " s")
    SetStatus(STATUS_DEAD, 0)
end
end

action ChangeLove(number change)
    SetLove(love + change)
end

action GetSecondsAlive returns integer
    return math:Round(secondsAlive):GetInteger()
end

action Update(number seconds)
    if status not= STATUS_DEAD
        secondsAlive = secondsAlive + seconds

        number energyFactor = -1
        if status = STATUS_ASLEEP
            energyFactor = 2
        elseif status = STATUS_SICK
            energyFactor = -3
        end

        if status not= STATUS_NORMAL
            remainingDebuffTime = remainingDebuffTime - seconds
            if remainingDebuffTime <= 0
                SetStatus(STATUS_NORMAL, 0)
            end
        end
    end
end

```

```

        end
    end

    ChangeLove(-seconds)
    ChangeEnergy(seconds * energyFactor)
end
end

private action Clamp(number n, number min, number max) returns number
    return math:MinimumOf(max, math:MaximumOf(min, n))
end

action Reset
    secondsAlive = 0
    SetEnergy(75)
    SetLove(75)
    SetStatus(STATUS_NORMAL, 0)
end
end

```

Listing C.15. PetListener.quorum

```

class PetListener
    action OnPetLoveChanged(number love)

    end

    action OnPetEnergyChanged(number energy)

    end

    action OnPetStatusChanged(integer status)

    end
end

```

Listing C.16. Symbol.quorum

```

use Libraries.Game.Graphics.Drawable
use Libraries.Interface.Events.MouseListener
use Libraries.Interface.Events.MouseEvent
use Libraries.Game.Graphics.Color

class Symbol is Drawable, MouseListener
    private Drawable cross

    private integer loveChange = 0
    private integer energyChange = 0
    private boolean enabled = true
    private Pet pet

    action Create(Pet pet, integer loveChange, integer energyChange)
        me:pet = pet
        me:loveChange = loveChange
        me:energyChange = energyChange

```

```

SetOriginCenter()
cross:Load("../assets/tamagochi/cross.png")
cross:Hide()
Add(cross)
AddMouseListener(me)

end

action ApplyToPet
pet:ChangeEnergy(energyChange)
pet:ChangeLove(loveChange)
end

action ClickedMouse(MouseEvent event)
if enabled and event:IsClicked() and event:IsLeftButtonEvent()
    ApplyToPet()
end
end

action IsEnabled returns boolean
return enabled
end

action Enable
enabled = true
cross:Hide()
end

action Disable
enabled = false
cross>Show()
end
end

```

Listing C.17. CakeSymbol.quorum

```

use Libraries.Compute.Random

class CakeSymbol is Symbol

private Random random

action ApplyToPet
parent:Symbol:ApplyToPet()

if random:RandomInteger(100) < 30
    Pet pet = parent:Symbol:pet
    pet:SetStatus(pet:STATUS_SICK, 15)
end
end
end

```

Listing C.18. Bar.quorum

```

use Libraries.Game.Graphics.Drawable
use Libraries.Game.Graphics.Color

```

```
use Libraries.Game.Graphics.Label
use Libraries.Compute.Math

class Bar is Drawable
    private Math math

    private integer width = 0
    private integer height = 0
    private number percent = 100
    private Drawable fill
    private text labelText = ""
    private Label label

action Create(
    integer width,
    integer height,
    Color backColor,
    Color fillColor,
    Color borderColor,
    text labelText)
{
    me:width = width
    me:height = height
    me:labelText = labelText
    LoadFilledRectangle(width, height, backColor)

    fill:LoadFilledRectangle(width, height, fillColor)
    Add(fill)

    label:SetText(labelText + ": 100 %")
    label:SetPosition(width / 2 - label:GetWidth() / 2, height / 2 - label:GetHeight() / 2)
    Add(label)

    Drawable border
    border:LoadRectangle(width, height, borderColor)
    Add(border)
}

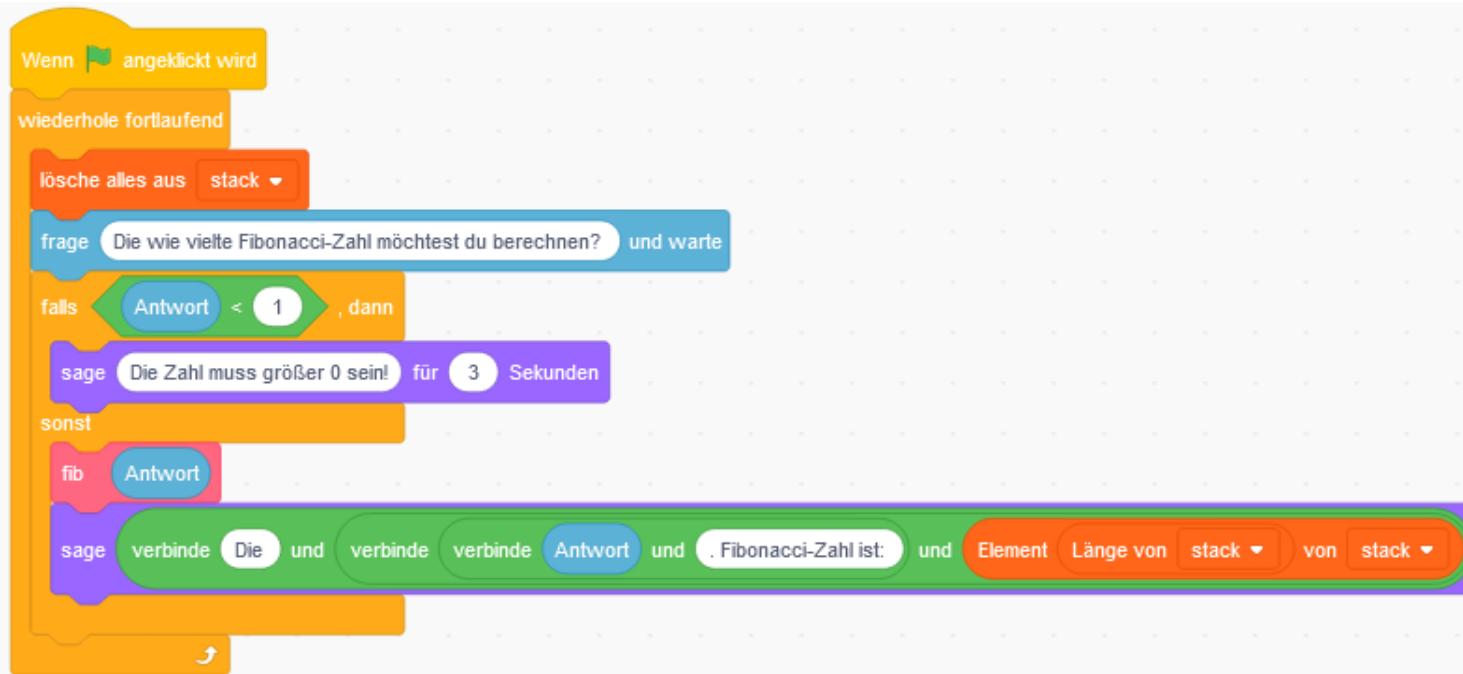
action GetPercent returns number
{
    return percent
}

action SetPercent(number percent)
{
    me:percent = percent
    number fillWidth = percent / 100 * width
    fill:SetWidth(fillWidth:GetInteger())
    label:SetText(labelText + " " + math:Round(percent):GetInteger():GetText() + "%")
}

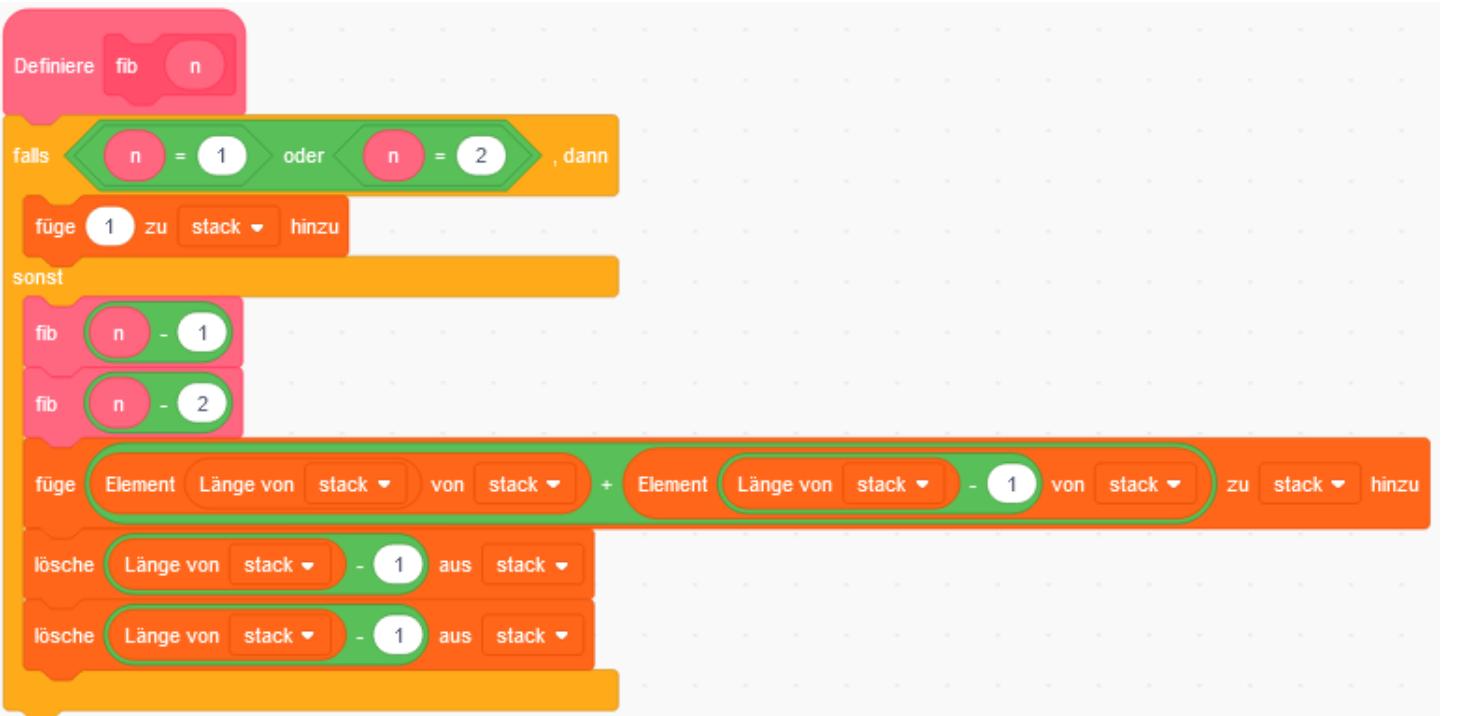
end
end
```

C.3. Implementierungen in Scratch

C.3.1. Fibonacci



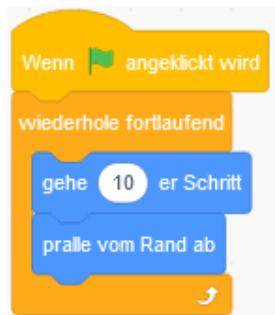
(a)



(b)

C.3.2. Bouncing Ball

Abb. C.2. Ball



C.3.3. Tic-Tac-Toe

Abb. C.3. Bühne

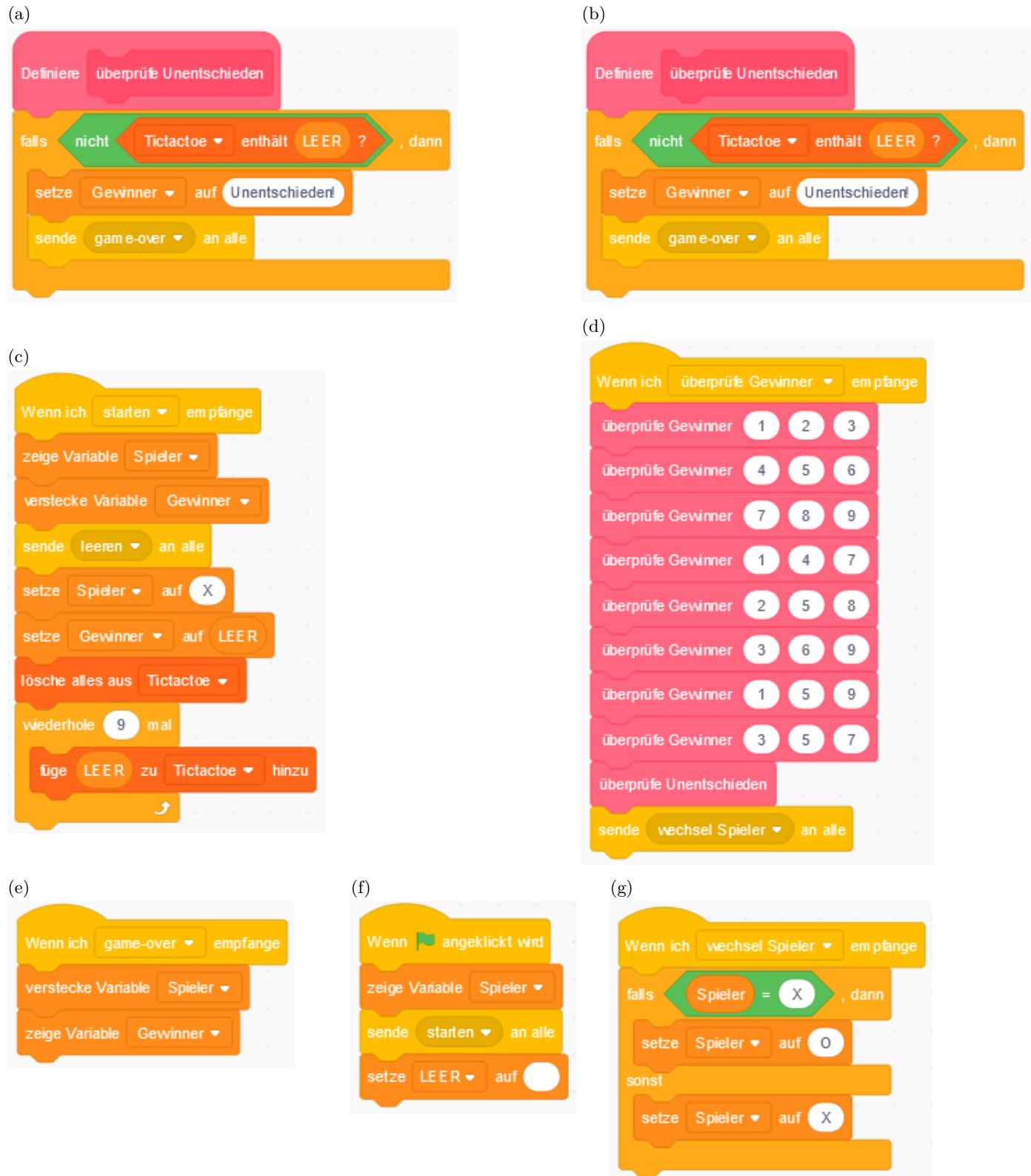


Abb. C.5. Feld

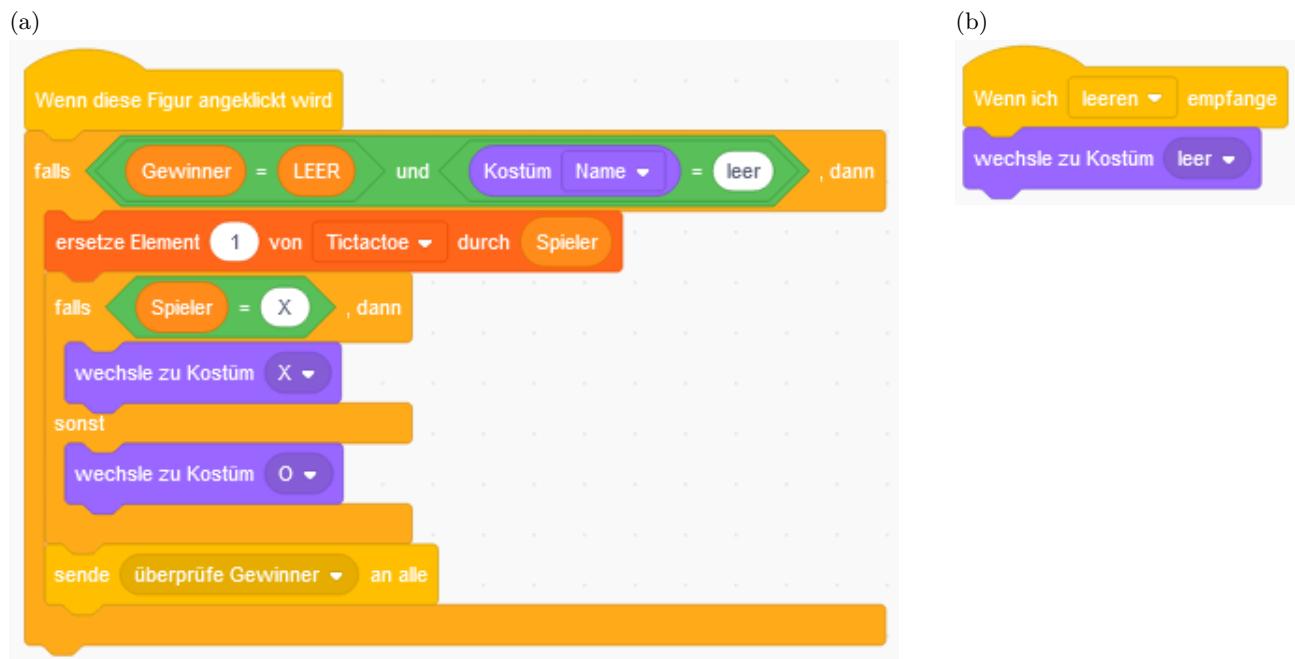


Abb. C.7. Button



C.3.4. Flappy Bird

Abb. C.9. Bühne

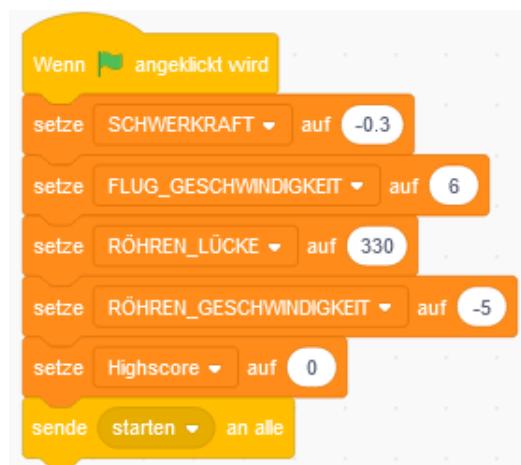


Abb. C.10. Papagei

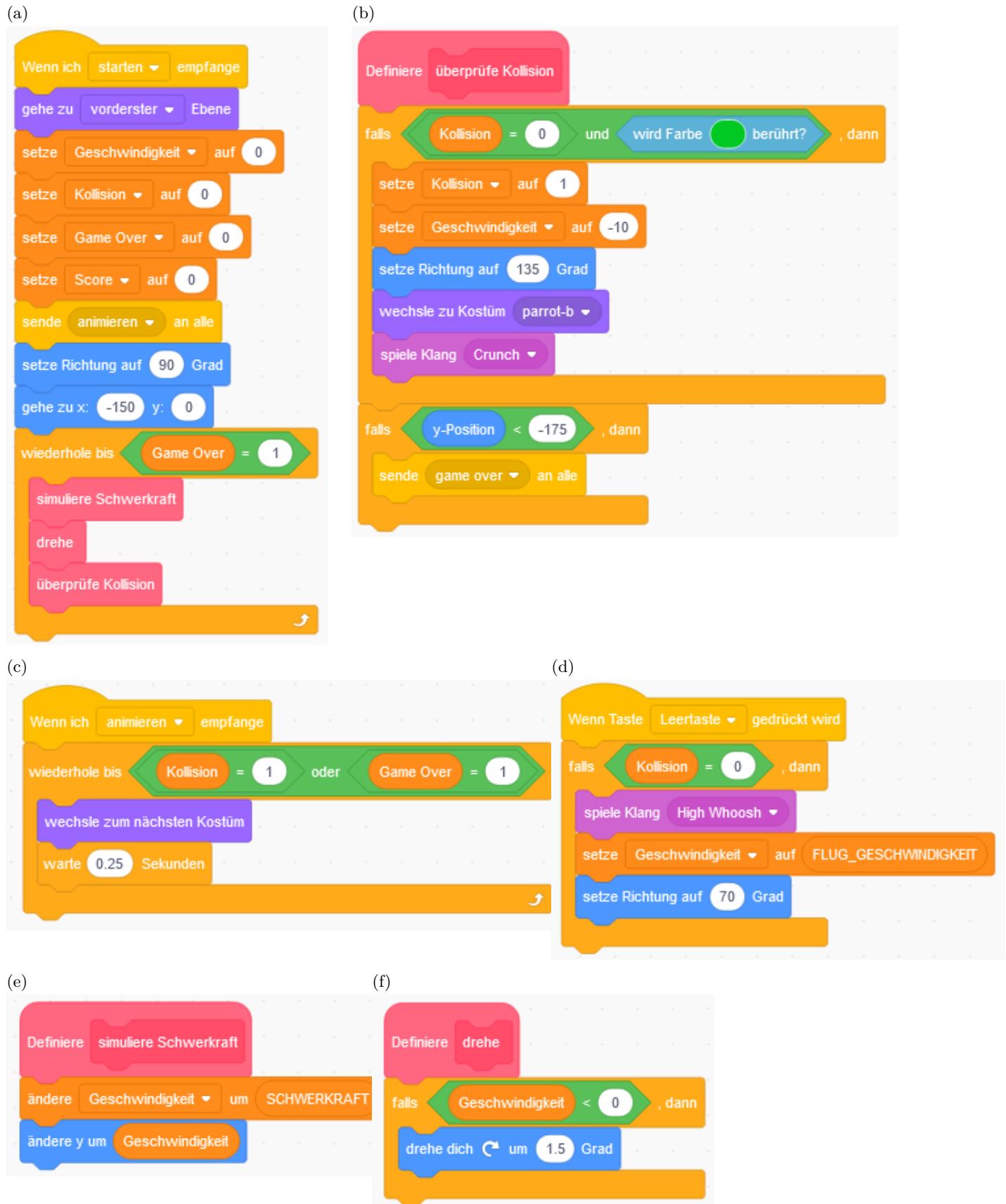


Abb. C.12. obere Röhre

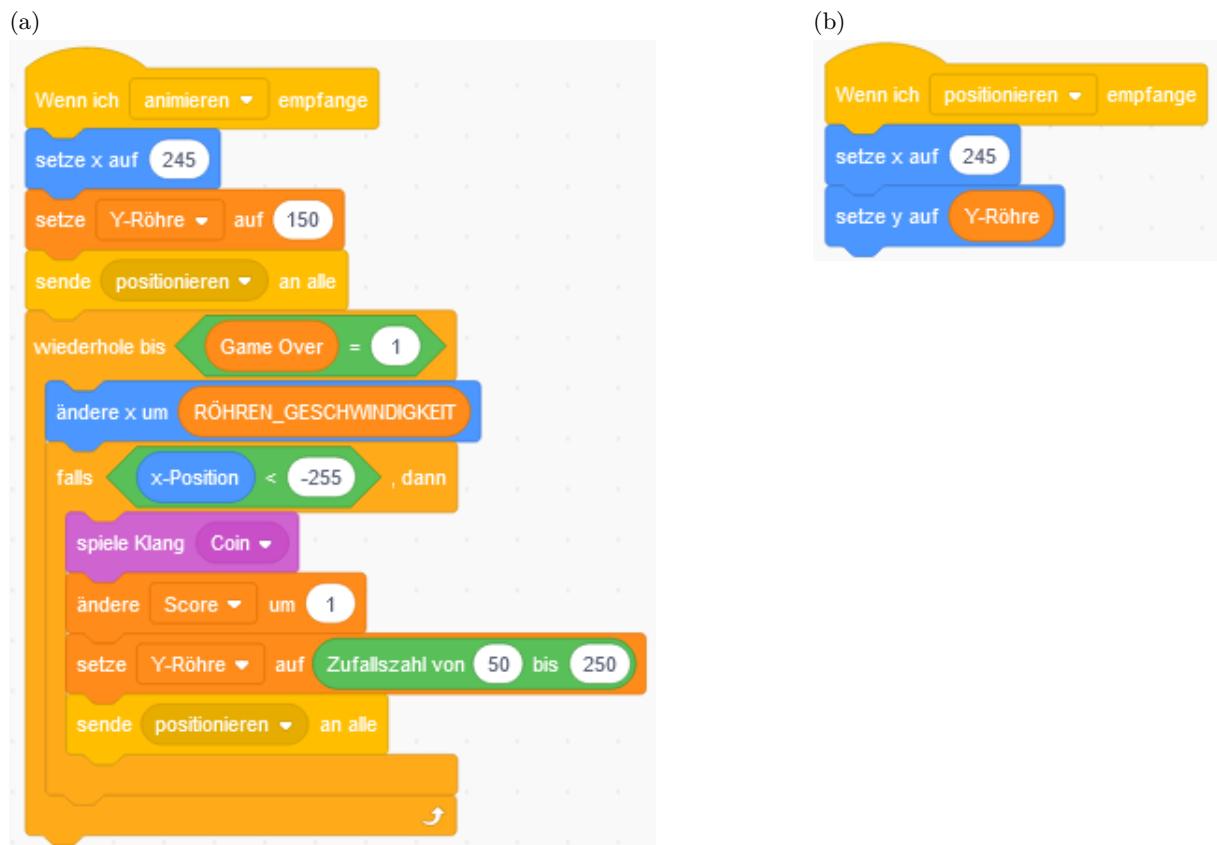


Abb. C.14. untere Röhre

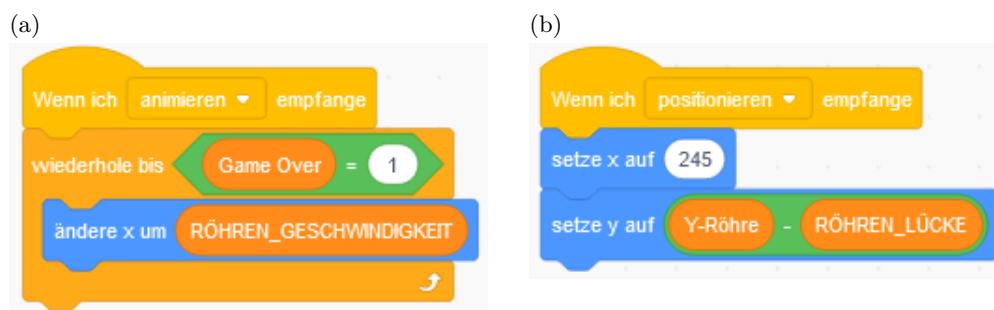


Abb. C.16. Button

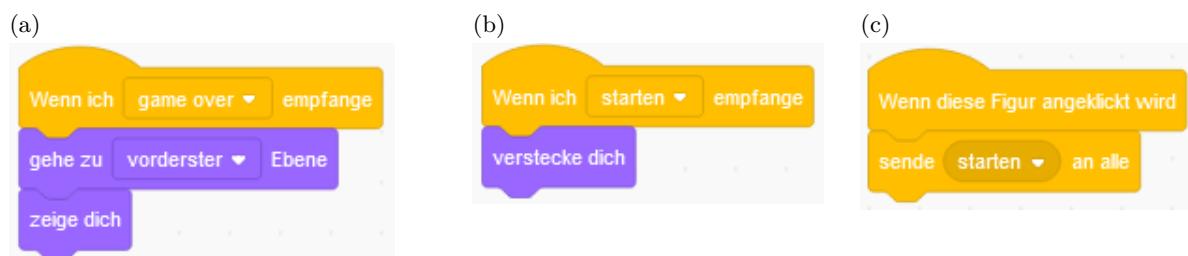


Abb. C.18. Game Over

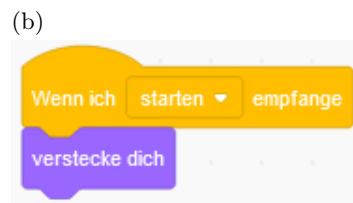
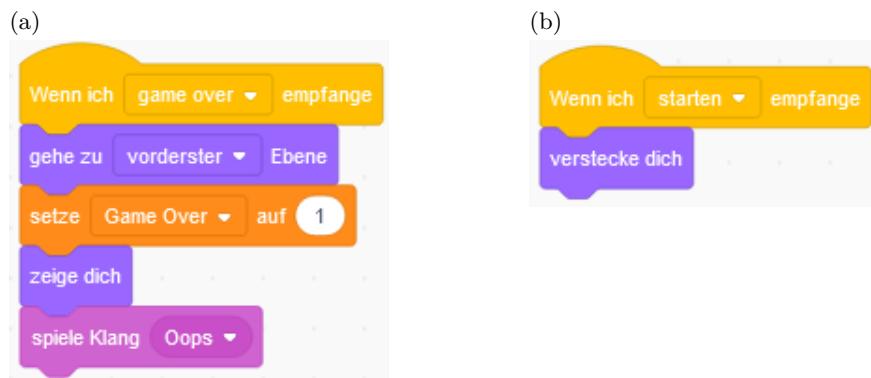
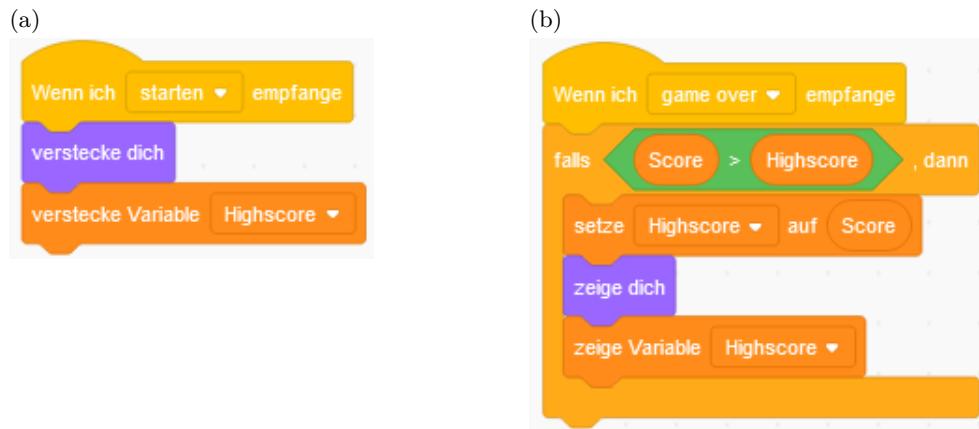


Abb. C.20. Highscore



C.3.5. Tamagochi

Abb. C.22. Bühne



Abb. C.23. Haustier

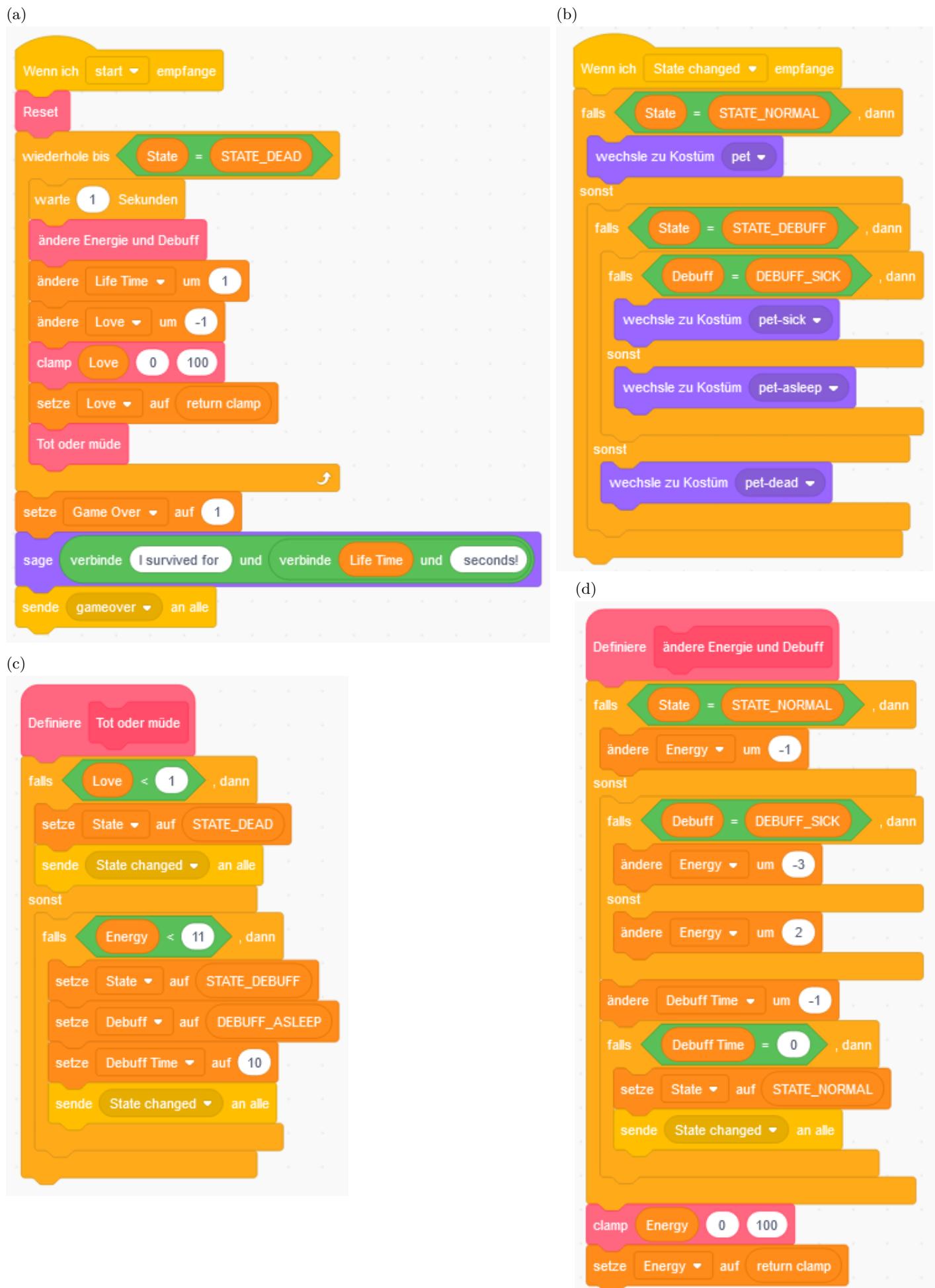


Abb. C.25. Haustier

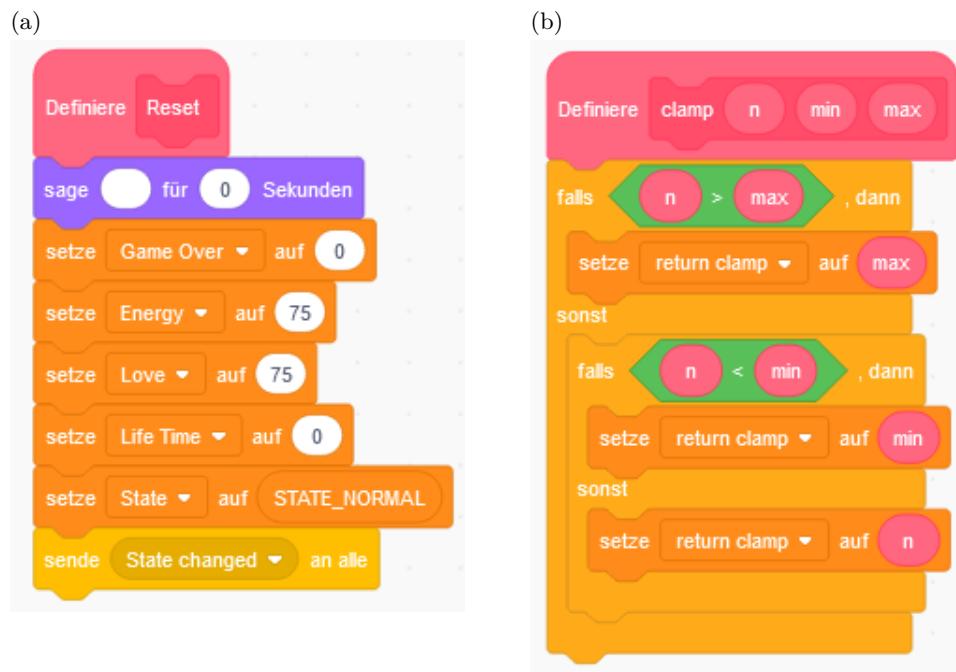


Abb. C.27. Herz

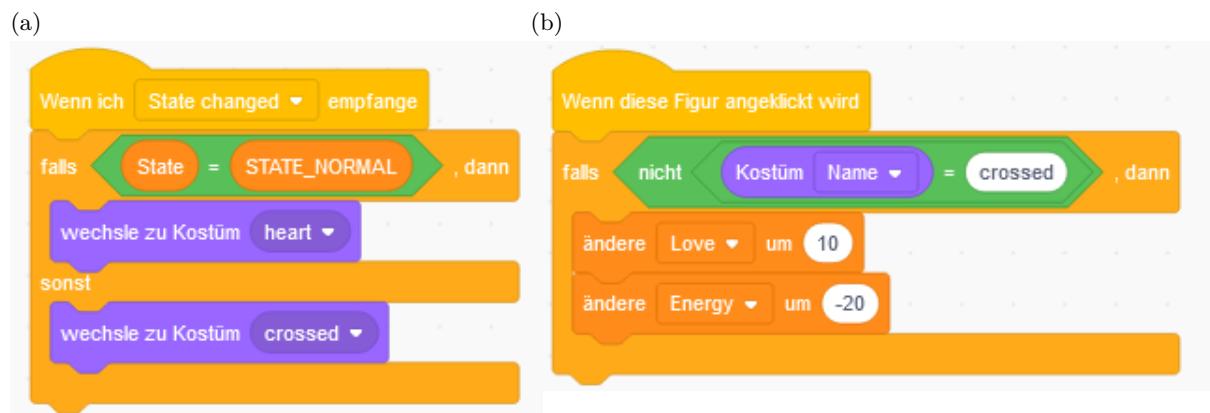


Abb. C.29. Brokkoli

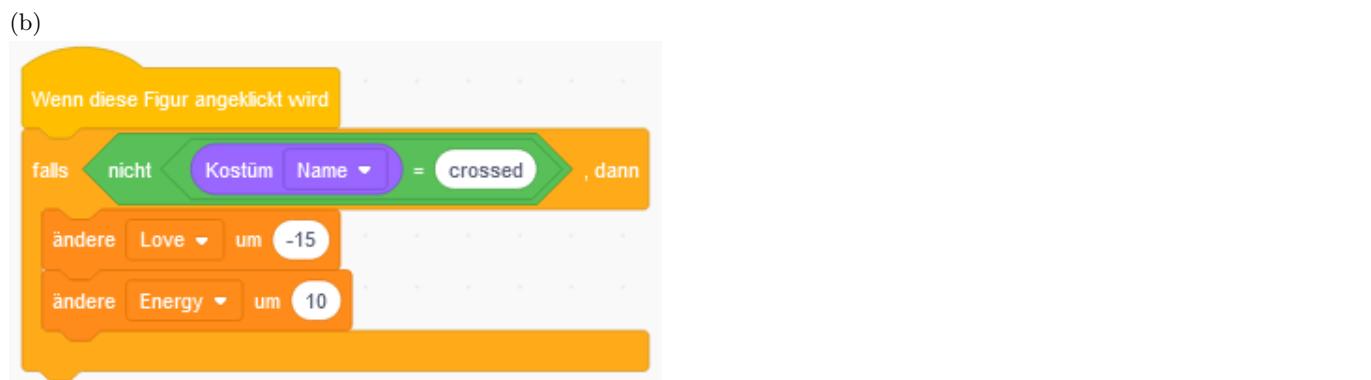
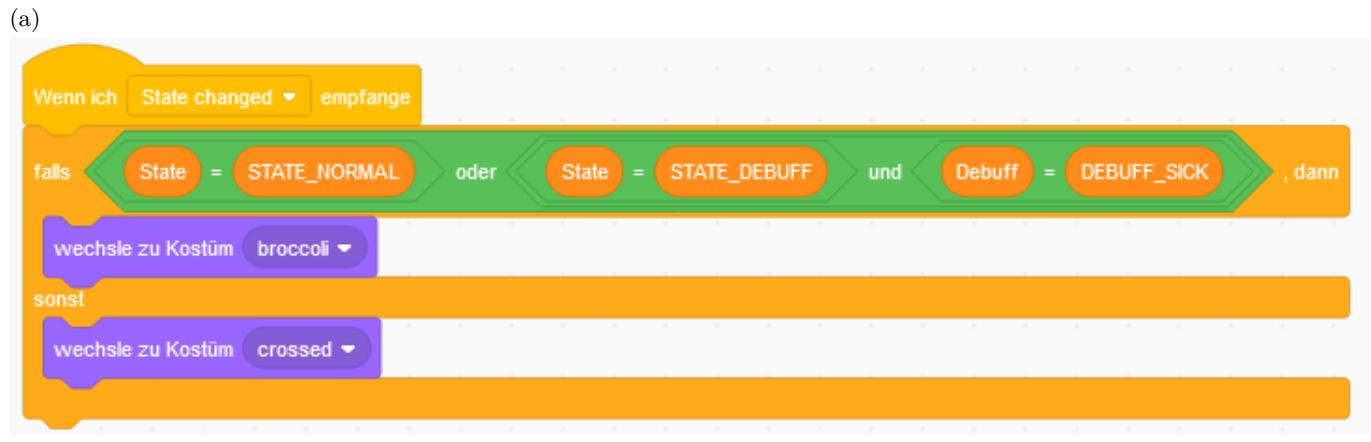


Abb. C.31. Kuchen

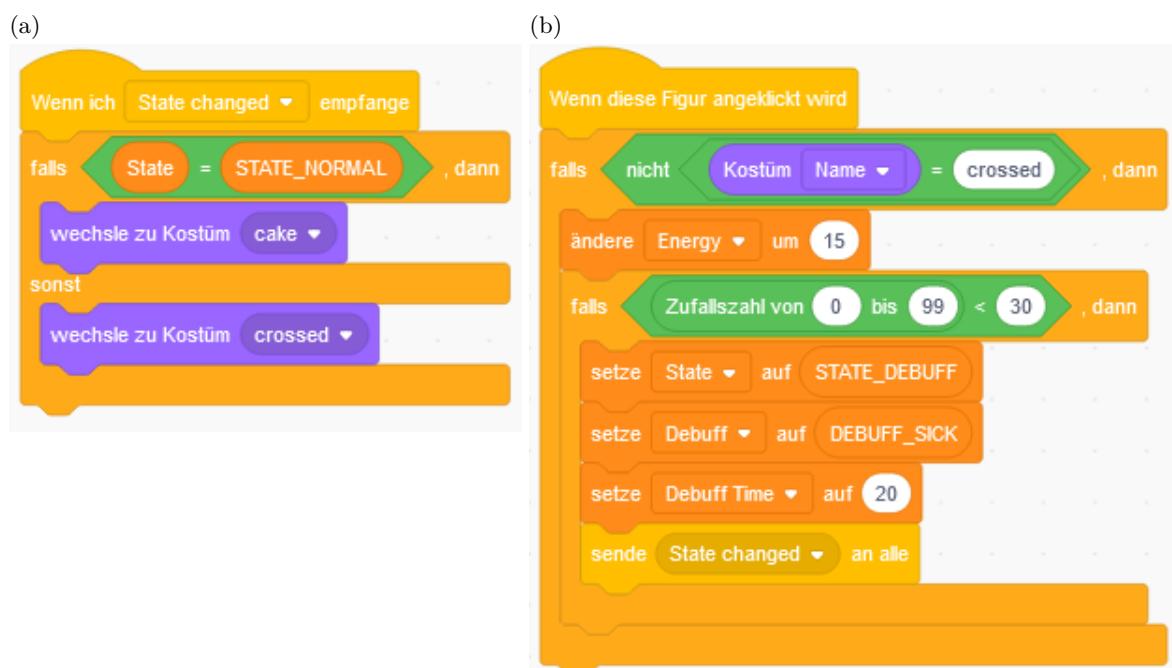


Abb. C.33. Stift

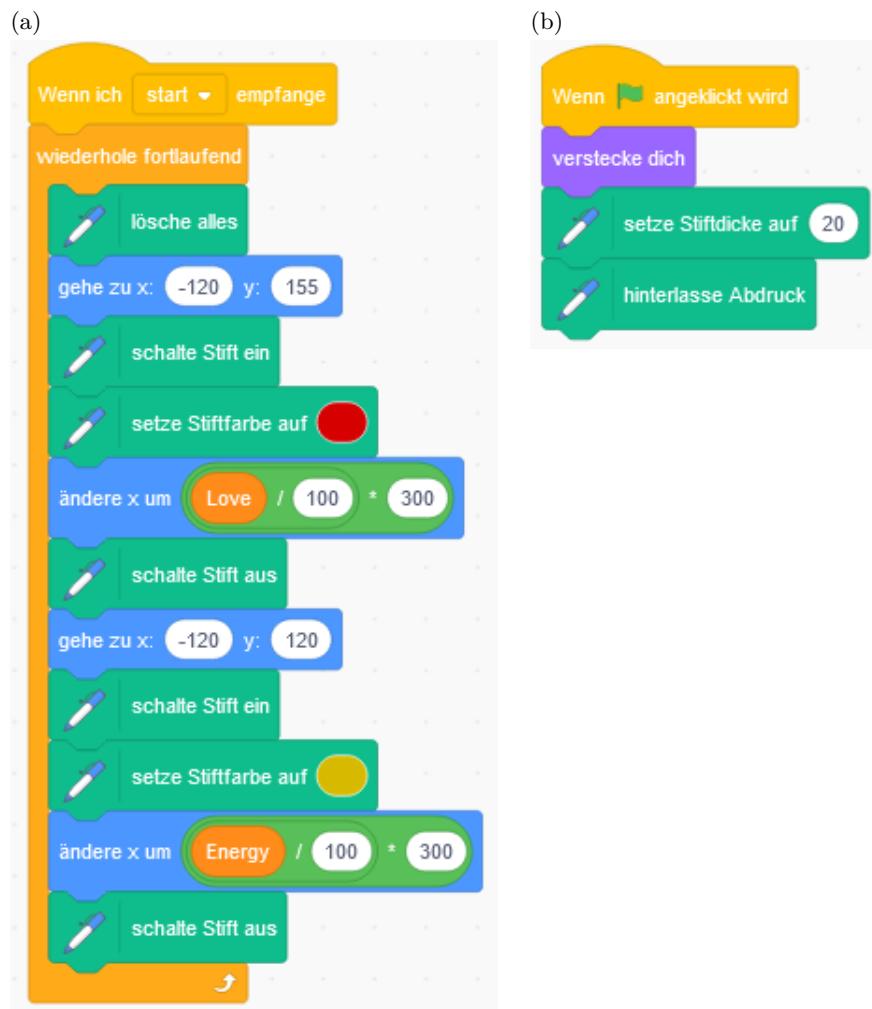
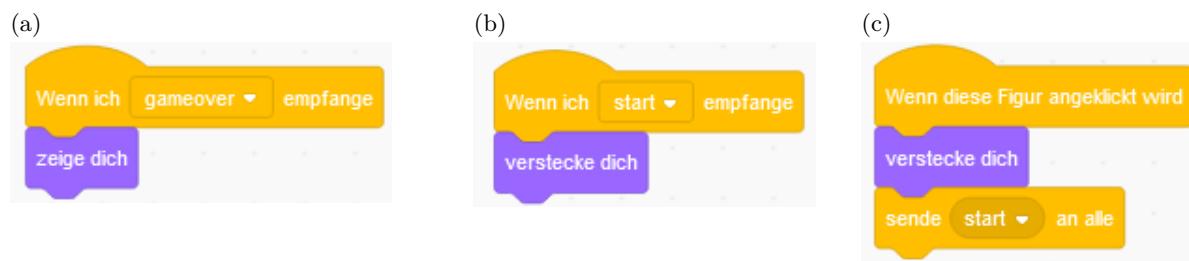


Abb. C.35. Button



C.4. Implementierungen in ToonTalk

C.4.1. Fibonacci

Listing C.19. fibonacci.xml

```
<?xml version="1.0"?>
<!-- This is the result of saving a ToonTalk object. You can include
this in files, email, or whatever. To decode it just use any program
that can copy this to the Windows clipboard. For example, you can use
Windows Notepad or WordPad program. Select all of the text
and then use the Copy command in the Edit menu. Then it will appear
in the ToonTalk remote control for the clipboard. You can find the
clipboard remote control on page 30 of the Sensor notebook. Just use
Dusty to vacuum it off the clipboard. Control-v is a shortcut for pasting.
What follows is in a special code and was produced by
ToonTalk 3 (version 3.191). To learn more visit www.toontalk.com.
-->
<ToonTalkObject Language="1" Version="4" CreatedUsing="ToonTalk 3 (version 3.191)" xmlns="http://www.
toontalk.com">
<Robot Version="3">
<Label><! [CDATA[fib1]]></Label>
<ThoughtBubble>
<Box>
<NumberOfHoles>2</NumberOfHoles>
<Hole>1
<Integer>1
<Active/>
</Integer>
</Hole>
<Hole>2
<Bird Parameter="8">
<Active/>
<SeeSome/>
<Erased/>
</Bird>
</Hole>
<GeometricRelationToContainer RecordedLeaderWidthSquared="14638276" RecordedLeaderHeightSquared="
16834609" WidthTimesLeaderWidth="9802212" HeightTimesLeaderHeight="6749435"
XOffsetTimesLeaderWidth="2544290" YOffsetTimesLeaderHeight="8091116" Z="-2"/>
<Active/>
</Box>
<GeometricRelationToContainer RecordedLeaderWidthSquared="19882681" RecordedLeaderHeightSquared="
21344400" WidthTimesLeaderWidth="18473637" HeightTimesLeaderHeight="20780760"
XOffsetTimesLeaderWidth="13840736" YOffsetTimesLeaderHeight="23478840" Z="0"/>
<Active/>
<SeeAll/>
</ThoughtBubble>
<Actions ID="160" Version="2" MadeOrMovedCount="2">
<MoveTo>NumberStack</MoveTo>
<MadeOrMoved TypeCode="0">1</MadeOrMoved>
<Grasp>1</Grasp>
<GiveBird>MyBox.2</GiveBird>
<MoveTo>BombStack</MoveTo>
```

```
<MadeOrMoved TypeCode="13">2</MadeOrMoved>
<Grasp>2</Grasp>
<Use></Use>
</Actions>
<Next>
<Robot Version="3">
<Label><! [CDATA[fib2]]></Label>
<ThoughtBubble>
<Box>
<NumberOfHoles>2</NumberOfHoles>
<Hole>1
<Integer>2
<Active/>
</Integer>
</Hole>
<Hole>2
<Bird Parameter="8">
<Active/>
<SeeSome/>
<Erased/>
</Bird>
</Hole>
<GeometricRelationToContainer RecordedLeaderWidthSquared="14638276" RecordedLeaderHeightSquared=
    "16834609" WidthTimesLeaderWidth="9802212" HeightTimesLeaderHeight="6749435"
    XOffsetTimesLeaderWidth="2544290" YOffsetTimesLeaderHeight="8091116" Z="-2"/>
<Active/>
</Box>
<GeometricRelationToContainer RecordedLeaderWidthSquared="19882681" RecordedLeaderHeightSquared="
    21344400" WidthTimesLeaderWidth="18473637" HeightTimesLeaderHeight="20780760"
    XOffsetTimesLeaderWidth="13840736" YOffsetTimesLeaderHeight="23478840" Z="2"/>
<Active/>
<SeeAll/>
</ThoughtBubble>
<Actions ID="161" Version="2" MadeOrMovedCount="2">
<MoveTo>NumberStack</MoveTo>
<MadeOrMoved TypeCode="0">1</MadeOrMoved>
<Grasp>1</Grasp>
<GiveBird>MyBox.2</GiveBird>
<MoveTo>BombStack</MoveTo>
<MadeOrMoved TypeCode="13">2</MadeOrMoved>
<Grasp>2</Grasp>
<Use></Use>
</Actions>
<Next>
<Robot Version="3">
<Label><! [CDATA[add]]></Label>
<ThoughtBubble>
<Box>
<NumberOfHoles>3</NumberOfHoles>
<Hole>1
<Integer>1
<Active/>
<Erased/>
</Integer>
```

```

</Hole>
<Hole>2
<Integer>1
<Active/>
<Erased/>
</Integer>
</Hole>
<Hole>3
<Bird Parameter="8">
<Active/>
<SeeSome/>
<Erased/>
</Bird>
</Hole>
<GeometricRelationToContainer RecordedLeaderWidthSquared="14638276" RecordedLeaderHeightSquared=
  ="16834609" WidthTimesLeaderWidth="9802212" HeightTimesLeaderHeight="6749435"
  XOffsetTimesLeaderWidth="2544290" YOffsetTimesLeaderHeight="8091116" Z="-2"/>
<Active/>
</Box>
<GeometricRelationToContainer RecordedLeaderWidthSquared="19882681" RecordedLeaderHeightSquared=
  ="21344400" WidthTimesLeaderWidth="18473637" HeightTimesLeaderHeight="20780760"
  XOffsetTimesLeaderWidth="13840736" YOffsetTimesLeaderHeight="23478840" Z="4"/>
<Active/>
<SeeAll/>
</ThoughtBubble>
<Actions ID="162" Version="2" MadeOrMovedCount="2">
<Grasp>MyBox.1</Grasp>
<Drop>1</Drop>
<Grasp>MyBox.2</Grasp>
<DropOn>1</DropOn>
<Grasp>1</Grasp>
<GiveBird>MyBox.3</GiveBird>
<MoveTo>BombStack</MoveTo>
<MadeOrMoved TypeCode="13">2</MadeOrMoved>
<Grasp>2</Grasp>
<Use></Use>
</Actions>
<Next>
<Robot Version="3">
<Label><! [CDATA [fib(n)] ]></Label>
<ThoughtBubble>
<Box>
<NumberOfHoles>2</NumberOfHoles>
<Hole>1
<Integer>3
<Active/>
<Erased/>
</Integer>
</Hole>
<Hole>2
<Bird Parameter="8">
<Active/>
<SeeSome/>
<Erased/>

```

```
</Bird>
</Hole>
<GeometricRelationToContainer RecordedLeaderWidthSquared="14638276"
    RecordedLeaderHeightSquared="16834609" WidthTimesLeaderWidth="9802212"
    HeightTimesLeaderHeight="6749435" XOffsetTimesLeaderWidth="2544290"
    YOffsetTimesLeaderHeight="8091116" Z="-2"/>
<Active/>
</Box>
<GeometricRelationToContainer RecordedLeaderWidthSquared="19882681"
    RecordedLeaderHeightSquared="21344400" WidthTimesLeaderWidth="18473637"
    HeightTimesLeaderHeight="20780760" XOffsetTimesLeaderWidth="13840736"
    YOffsetTimesLeaderHeight="23478840" Z="6"/>
<Active/>
<SeeAll/>
</ThoughtBubble>
<Actions ID="164" Version="2" MadeOrMovedCount="18">
    <Grasp>MyBox.2</Grasp>
    <Drop>1</Drop>
    <MoveTo>BoxStack</MoveTo>
    <MadeOrMoved TypeCode="10">2</MadeOrMoved>
    <Grasp>2</Grasp>
    <Drop>2</Drop>
    <MoveTo>BoxStack</MoveTo>
    <MadeOrMoved TypeCode="10">3</MadeOrMoved>
    <Grasp>3</Grasp>
    <DropOnRightSide>2</DropOnRightSide>
    <MoveTo>BoxStack</MoveTo>
    <MadeOrMoved TypeCode="10">4</MadeOrMoved>
    <Grasp>4</Grasp>
    <DropOnRightSide>2</DropOnRightSide>
    <Grasp>1</Grasp>
    <DropOn>2.3</DropOn>
    <Grasp>MyBox</Grasp>
    <Drop>MyBox</Drop>
    <GraspMagicWand>MagicWand</GraspMagicWand>
    <UseMagicWand>MyBox</UseMagicWand>
    <Drop>5</Drop>
    <DropMagicWand>MagicWand</DropMagicWand>
    <MoveTo>NumberStack</MoveTo>
    <MadeOrMoved TypeCode="0">6</MadeOrMoved>
    <Grasp>6</Grasp>
    <TypeTo KeyDescription="Backspace key" KeyCode="8">InHand</TypeTo>
    <TypeTo KeyDescription="2" KeyCode="50">InHand</TypeTo>
    <TypeTo KeyDescription="-" KeyCode="45">InHand</TypeTo>
    <TypeTo KeyDescription="2" KeyCode="50">InHand</TypeTo>
    <TypeTo KeyDescription="Backspace key" KeyCode="8">InHand</TypeTo>
    <TypeTo KeyDescription="Backspace key" KeyCode="8">InHand</TypeTo>
    <TypeTo KeyDescription="Backspace key" KeyCode="8">InHand</TypeTo>
    <TypeTo KeyDescription="-" KeyCode="45">InHand</TypeTo>
    <TypeTo KeyDescription="2" KeyCode="50">InHand</TypeTo>
    <DropOn>MyBox.1</DropOn>
    <MoveTo>NumberStack</MoveTo>
    <MadeOrMoved TypeCode="0">7</MadeOrMoved>
    <Grasp>7</Grasp>
```

```
<TypeTo KeyDescription="-" KeyCode="45">InHand</TypeTo>
<DropOn>5.1</DropOn>
<MoveTo>NestStack</MoveTo>
<MadeOrMoved TypeCode="6">8</MadeOrMoved>
<GraspNest>8</GraspNest>
<MadeOrMoved TypeCode="7">9</MadeOrMoved>
<Drop>8</Drop>
<Grasp>9</Grasp>
<DropOn>MyBox.2</DropOn>
<GraspNest>8</GraspNest>
<DropOn>2.1</DropOn>
<MoveTo>NestStack</MoveTo>
<MadeOrMoved TypeCode="6">10</MadeOrMoved>
<GraspNest>10</GraspNest>
<MadeOrMoved TypeCode="7">11</MadeOrMoved>
<Drop>10</Drop>
<Grasp>11</Grasp>
<DropOn>5.2</DropOn>
<GraspNest>10</GraspNest>
<DropOn>2.2</DropOn>
<Grasp>MyBox</Grasp>
<Drop>MyBox</Drop>
<Grasp>5</Grasp>
<Drop>5</Drop>
<MoveTo>TruckStack</MoveTo>
<MadeOrMoved TypeCode="8">12</MadeOrMoved>
<Grasp>12</Grasp>
<Drop>12</Drop>
<TypeTo KeyDescription "+" KeyCode="43">MagicWand</TypeTo>
<GraspMagicWand>MagicWand</GraspMagicWand>
<TypeTo KeyDescription "+" KeyCode="43">InHand</TypeTo>
<UseMagicWand></UseMagicWand>
<Drop>13</Drop>
<DropMagicWand>MagicWand</DropMagicWand>
<Grasp>13</Grasp>
<DropInTruck>12</DropInTruck>
<GraspMagicWand>MagicWand</GraspMagicWand>
<TypeTo KeyDescription="-" KeyCode="45">InHand</TypeTo>
<TypeTo KeyDescription="-" KeyCode="45">InHand</TypeTo>
<DropMagicWand>MagicWand</DropMagicWand>
<GraspMagicWand>MagicWand</GraspMagicWand>
<DropMagicWand>MagicWand</DropMagicWand>
<MoveTo>TruckStack</MoveTo>
<MadeOrMoved TypeCode="8">14</MadeOrMoved>
<Grasp>14</Grasp>
<Drop>14</Drop>
<MoveTo>TruckStack</MoveTo>
<MadeOrMoved TypeCode="8">15</MadeOrMoved>
<Grasp>15</Grasp>
<Drop>15</Drop>
<GraspMagicWand>MagicWand</GraspMagicWand>
<TypeTo KeyDescription "+" KeyCode="43">InHand</TypeTo>
<TypeTo KeyDescription "+" KeyCode="43">InHand</TypeTo>
<UseMagicWand></UseMagicWand>
```

```

<DropInTruck>14</DropInTruck>
<UseMagicWand></UseMagicWand>
<DropInTruck>15</DropInTruck>
<DropMagicWand>MagicWand</DropMagicWand>
<Grasp>5</Grasp>
<DropInTruck>12</DropInTruck>
<Grasp>MyBox</Grasp>
<DropInTruck>14</DropInTruck>
<Grasp>2</Grasp>
<DropInTruck>15</DropInTruck>
<MoveTo>BombStack</MoveTo>
<MadeOrMoved TypeCode="13">18</MadeOrMoved>
<Grasp>18</Grasp>
<Use></Use>
</Actions>
<GeometricRelationToContainer RecordedLeaderWidthSquared="16957924" RecordedLeaderHeightSquared=
    ="17757796" WidthTimesLeaderWidth="16957924" HeightTimesLeaderHeight="17757796"
    XOffsetTimesLeaderWidth="10764452" YOffsetTimesLeaderHeight="14146398" Z="5" />
<Active/>
</Robot>
</Next>
<GeometricRelationToContainer RecordedLeaderWidthSquared="16957924" RecordedLeaderHeightSquared=
    ="17757796" WidthTimesLeaderWidth="16957924" HeightTimesLeaderHeight="17757796"
    XOffsetTimesLeaderWidth="10764452" YOffsetTimesLeaderHeight="14146398" Z="3" />
<Active/>
</Robot>
</Next>
<GeometricRelationToContainer RecordedLeaderWidthSquared="16957924" RecordedLeaderHeightSquared="17757796" WidthTimesLeaderWidth="16957924" HeightTimesLeaderHeight="17757796"
    XOffsetTimesLeaderWidth="10764452" YOffsetTimesLeaderHeight="14146398" Z="1" />
<Active/>
</Robot>
</Next>
<Geometry Z="-1">
    <Width>4118</Width>
    <Height>4214</Height>
</Geometry>
<Active/>
</Robot>
</ToonTalkObject>

```

C.4.2. Bouncing-Ball

Listing C.20. bouncing-ball.xml

```

<?xml version="1.0"?>
<!-- This is the result of saving a ToonTalk object. You can include
this in files, email, or whatever. To decode it just use any program
that can copy this to the Windows clipboard. For example, you can use
Windows Notepad or WordPad program. Select all of the text
and then use the Copy command in the Edit menu. Then it will appear
in the ToonTalk remote control for the clipboard. You can find the
clipboard remote control on page 30 of the Sensor notebook. Just use
Dusty to vacuum it off the clipboard. Control-v is a shortcut for pasting.
What follows is in a special code and was produced by

```

```
ToonTalk 3 (version 3.191). To learn more visit www.toontalk.com.  
-->  
<ToonTalkObject Language="1" Version="4" CreatedUsing="ToonTalk 3 (version 3.191)" xmlns="http://www.  
toontalk.com">  
<Picture Parameter="2">  
<ColorIndex>2</ColorIndex>  
<Shape Code="0" Description="a rectangle"/>  
<Geometry Z="-1">  
<Width>15997</Width>  
<Height>12000</Height>  
</Geometry>  
<Active/>  
<SeeSome/>  
<OnFront>  
<Picture Parameter="1">  
<ColorIndex>1</ColorIndex>  
<Shape Code="0" Description="a rectangle"/>  
<GeometricRelationToContainer RecordedLeaderWidthSquared="256000000" RecordedLeaderHeightSquared="  
144000000" WidthTimesLeaderWidth="322160000" HeightTimesLeaderHeight="180144000"  
XOffsetTimesLeaderWidth="-36048000" YOffsetTimesLeaderHeight="138036000" Z="3220"/>  
<Active/>  
<SeeSome/>  
<GeometryWhenFreeAndUnflipped>  
<Width>20135</Width>  
</GeometryWhenFreeAndUnflipped>  
</Picture>  
</OnFront>  
<OnFront>  
<Picture Parameter="1">  
<ColorIndex>1</ColorIndex>  
<Shape Code="0" Description="a rectangle"/>  
<GeometricRelationToContainer RecordedLeaderWidthSquared="256000000" RecordedLeaderHeightSquared="  
144000000" WidthTimesLeaderWidth="322160000" HeightTimesLeaderHeight="180144000"  
XOffsetTimesLeaderWidth="249472000" YOffsetTimesLeaderHeight="-21912000" Z="1954"/>  
<Active/>  
<SeeSome/>  
<GeometryWhenFreeAndUnflipped>  
<Width>20135</Width>  
</GeometryWhenFreeAndUnflipped>  
</Picture>  
</OnFront>  
<OnFront>  
<Picture Parameter="1">  
<ColorIndex>1</ColorIndex>  
<Shape Code="0" Description="a rectangle"/>  
<GeometricRelationToContainer RecordedLeaderWidthSquared="256000000" RecordedLeaderHeightSquared="  
144000000" WidthTimesLeaderWidth="322160000" HeightTimesLeaderHeight="180144000"  
XOffsetTimesLeaderWidth="-39024000" YOffsetTimesLeaderHeight="-175224000" Z="740"/>  
<Active/>  
<SeeSome/>  
<GeometryWhenFreeAndUnflipped>  
<Width>20135</Width>  
</GeometryWhenFreeAndUnflipped>  
</Picture>
```

```
</OnFront>
<OnFront>
<Picture Parameter="1">
<ColorIndex>1</ColorIndex>
<Shape Code="0" Description="a rectangle"/>
<GeometricRelationToContainer RecordedLeaderWidthSquared="256000000" RecordedLeaderHeightSquared="144000000" WidthTimesLeaderWidth="322160000" HeightTimesLeaderHeight="180144000" XOffsetTimesLeaderWidth="-314176000" YOffsetTimesLeaderHeight="-34104000" Z="1857"/>
<Active/>
<SeeSome/>
<GeometryWhenFreeAndUnflipped>
<Width>20135</Width>
</GeometryWhenFreeAndUnflipped>
</Picture>
</OnFront>
<OnFront>
<Picture GUID="kpndefipgjdelhepj1jbjplnjcmplji">
<BuiltIn Code="54" PictureNumber="0" Description="a ball"/>
<GeometricRelationToContainer RecordedLeaderWidthSquared="255904009" RecordedLeaderHeightSquared="144000000" WidthTimesLeaderWidth="19068424" HeightTimesLeaderHeight="15408000" XOffsetTimesLeaderWidth="177358739" YOffsetTimesLeaderHeight="41700000" Z="2547"/>
<Active/>
<SeeSome/>
<SpeedToRight>-200</SpeedToRight>
<SpeedToTop>200</SpeedToTop>
<OnBack>
<Robot Version="3">
<Label><! [CDATA[set speed]]></Label>
<ThoughtBubble>
<Box Parameter="4">
<NumberOfHoles>4</NumberOfHoles>
<Hole>1
<Integer>20
<Active/>
<Erased/>
</Integer>
</Hole>
<Hole>2
<RemoteControl Code="3" Description="SpeedtoTop" Version="2">
<Value>
<RemoteControl>0</RemoteControl>
</Value>
<Active/>
<Erased/>
</RemoteControl>
<Label><! [CDATA[Up Speed]]></Label>
</Hole>
<Hole>3
<RemoteControl Code="2" Description="SpeedtoRight" Version="2">
<Value>
<RemoteControl>0</RemoteControl>
</Value>
<Active/>
```

```
<Erased/>
</RemoteControl>
<Label><! [CDATA [Right Speed]]></Label>
</Hole>
<Hole>4
<TextPad>
<TextValue><! [CDATA [yes]]></TextValue>
<Active/>
</TextPad>
<Label><! [CDATA [Dropped?]]></Label>
</Hole>
<GeometricRelationToContainer RecordedLeaderWidthSquared="14638276" RecordedLeaderHeightSquared=
    ="17181025" WidthTimesLeaderWidth="9802212" HeightTimesLeaderHeight="6818525"
    XOffsetTimesLeaderWidth="2544290" YOffsetTimesLeaderHeight="8252695" Z="-2147483648"/>
<Active/>
</Box>
<GeometricRelationToContainer RecordedLeaderWidthSquared="19882681" RecordedLeaderHeightSquared=
    ="21344400" WidthTimesLeaderWidth="18473637" HeightTimesLeaderHeight="20780760"
    XOffsetTimesLeaderWidth="11588941" YOffsetTimesLeaderHeight="22485540" Z="-2147483648"/>
<Active/>
<SeeAll/>
</ThoughtBubble>
<Actions ID="166" Version="2" MadeOrMovedCount="2">
<GraspMagicWand>MagicWand</GraspMagicWand>
<UseMagicWand>MyBox.1</UseMagicWand>
<DropOn>MyBox.2</DropOn>
<UseMagicWand>MyBox.1</UseMagicWand>
<DropOn>MyBox.3</DropOn>
<DropMagicWand>MagicWand</DropMagicWand>
</Actions>
<WorkingInfo SetDownCounter="0" RecentPageNumber="0">
<Box Parameter="4" GUID="1dgmlookeocbhenkpdgmefpnidfijkb">
<NumberOfHoles>4</NumberOfHoles>
<Hole>1
<Integer>100
<Active/>
</Integer>
</Hole>
<Hole>2
<RemoteControl Code="3" Description="SpeedtoTop" Version="2">
<For>
<Picture Link="kpndefipgjdelhepj1jibjplnjcm1pji"/>
</For>
<Value>
<RemoteControl>200</RemoteControl>
</Value>
<Active/>
</RemoteControl>
<Label><! [CDATA [Up Speed]]></Label>
</Hole>
<Hole>3
<RemoteControl Code="2" Description="SpeedtoRight" Version="2">
<For>
<Picture Link="kpndefipgjdelhepj1jibjplnjcm1pji"/>
```

```
</For>
<Value>
<RemoteControl>-200</RemoteControl>
</Value>
<Active/>
</RemoteControl>
<Label><! [CDATA [Right Speed]]></Label>
</Hole>
<Hole>4
<RemoteControl Code="12" Description="JustDropped?" Version="2">
<Appearance>
<TextPad>
<TextValue><! [CDATA [no]]></TextValue>
<Active/>
</TextPad>
</Appearance>
<For>
<Picture Link="kpndefipgjdelhepj1jibjplnjcmlpji"/>
</For>
<Active/>
</RemoteControl>
<Label><! [CDATA [Dropped?]]></Label>
</Hole>
<Active/>
</Box>
</WorkingInfo>
<Active/>
</Robot>
</OnBack>
<OnBack>
<Robot Version="3">
<Label><! [CDATA [bounce ]]></Label>
<ThoughtBubble>
<Box Parameter="4">
<NumberOfHoles>2</NumberOfHoles>
<Hole>1
<RemoteControl Code="2" Description="SpeedtoRight" Version="2">
<Value>
<RemoteControl>0</RemoteControl>
</Value>
<Active/>
<Erased/>
</RemoteControl>
<Label><! [CDATA [Right Speed]]></Label>
</Hole>
<Hole>2
<Picture>
<BuiltIn Code="53" PictureNumber="0" Description="the hit or miss cartoon"/>
<Active/>
<GeometryWhenFreeAndUnflipped>
<Width>1920</Width>
<Height>960</Height>
</GeometryWhenFreeAndUnflipped>
</Picture>
```

```

<Label><![CDATA[Right Collide?]]></Label>
</Hole>
<GeometricRelationToContainer RecordedLeaderWidthSquared="14638276" RecordedLeaderHeightSquared=
    ="17181025" WidthTimesLeaderWidth="9802212" HeightTimesLeaderHeight="6818525"
    XOffsetTimesLeaderWidth="2544290" YOffsetTimesLeaderHeight="8252695" Z="-2147483648"/>
<Active/>
</Box>
<GeometricRelationToContainer RecordedLeaderWidthSquared="19882681" RecordedLeaderHeightSquared=
    ="21344400" WidthTimesLeaderWidth="18473637" HeightTimesLeaderHeight="20780760"
    XOffsetTimesLeaderWidth="11637990" YOffsetTimesLeaderHeight="22346940" Z="-2147483648"/>
<Active/>
<SeeAll/>
</ThoughtBubble>
<Actions ID="169" Version="2" MadeOrMovedCount="1">
    <MoveTo>NumberStack</MoveTo>
    <MadeOrMoved TypeCode="0">1</MadeOrMoved>
    <Grasp>1</Grasp>
    <TypeTo KeyDescription="Backspace key" KeyCode="8">InHand</TypeTo>
    <TypeTo KeyDescription="*" KeyCode="42">InHand</TypeTo>
    <TypeTo KeyDescription="-" KeyCode="45">InHand</TypeTo>
    <TypeTo KeyDescription="1" KeyCode="49">InHand</TypeTo>
    <DropOn>MyBox.1</DropOn>
    <TypeTo KeyDescription="+" KeyCode="43">MyBox.2</TypeTo>
</Actions>
<WorkingInfo SetDownCounter="0" RecentPageNumber="0">
    <Box Parameter="4" GUID="iicjpidpeiflglgelkaijflhmoklgfopi">
        <NumberOfHoles>2</NumberOfHoles>
        <Hole>1
            <RemoteControl Code="2" Description="SpeedtoRight" Version="2">
                <For>
                    <Picture Link="kpndefipgjdelhepjljibjplnjcmpljpi"/>
                </For>
                <Value>
                    <RemoteControl>-200</RemoteControl>
                </Value>
                <Active/>
            </RemoteControl>
            <Label><![CDATA[Right Speed]]></Label>
        </Hole>
        <Hole>2
            <RemoteControl Code="13" Description="LeftRighthit?" Version="2">
                <Appearance>
                    <Picture Parameter="1">
                        <BuiltIn Code="53" PictureNumber="1" Description="the hit or miss cartoon"/>
                    <Geometry Z="-2147483648">
                        <Left>16306</Left>
                        <Bottom>13891</Bottom>
                        <Width>41</Width>
                        <Height>52</Height>
                    </Geometry>
                    <Active/>
                    <GeometryWhenFreeAndUnflipped>
                        <Width>2400</Width>
                        <Height>7400</Height>
                    </GeometryWhenFreeAndUnflipped>
                </Appearance>
            </RemoteControl>
        </Hole>
    </Box>
</WorkingInfo>

```

```

    </GeometryWhenFreeAndUnflipped>
  </Picture>
</Appearance>
<For>
  <Picture Link="kpndefipgjdelhepj1jibjplnjcmlpji"/>
</For>
<Active/>
</RemoteControl>
<Label><! [CDATA[Right Collide?]]></Label>
</Hole>
<Active/>
</Box>
</WorkingInfo>
<Active/>
</Robot>
</OnBack>
<OnBack>
<Robot Version="3">
  <Label><! [CDATA[bounce ]]></Label>
  <ThoughtBubble>
    <Box Parameter="4">
      <NumberOfHoles>2</NumberOfHoles>
      <Hole>1
        <RemoteControl Code="2" Description="SpeedtoRight" Version="2">
          <Value>
            <RemoteControl>0</RemoteControl>
          </Value>
        <Active/>
        <Erased/>
      </RemoteControl>
      <Label><! [CDATA[Right Speed]]></Label>
      </Hole>
      <Hole>2
        <Picture>
          <BuiltIn Code="53" PictureNumber="0" Description="the hit or miss cartoon"/>
        <Active/>
        <GeometryWhenFreeAndUnflipped>
          <Width>1920</Width>
          <Height>960</Height>
        </GeometryWhenFreeAndUnflipped>
      </Picture>
      <Label><! [CDATA[Right Collide?]]></Label>
    </Hole>
    <GeometricRelationToContainer RecordedLeaderWidthSquared="14638276" RecordedLeaderHeightSquared=
      ="17181025" WidthTimesLeaderWidth="9802212" HeightTimesLeaderHeight="6818525"
      XOffsetTimesLeaderWidth="2544290" YOffsetTimesLeaderHeight="8252695" Z="-2147483648"/>
    <Active/>
  </Box>
  <GeometricRelationToContainer RecordedLeaderWidthSquared="19882681" RecordedLeaderHeightSquared=
    ="21344400" WidthTimesLeaderWidth="18473637" HeightTimesLeaderHeight="20780760"
    XOffsetTimesLeaderWidth="11637990" YOffsetTimesLeaderHeight="22346940" Z="-2147483648"/>
  <Active/>
  <SeeAll/>
</ThoughtBubble>

```

```
<Actions ID="169" Version="2" MadeOrMovedCount="1">
<MoveTo>NumberStack</MoveTo>
<MadeOrMoved TypeCode="0">1</MadeOrMoved>
<Grasp>1</Grasp>
<TypeTo KeyDescription="Backspace key" KeyCode="8">InHand</TypeTo>
<TypeTo KeyDescription="*" KeyCode="42">InHand</TypeTo>
<TypeTo KeyDescription="-" KeyCode="45">InHand</TypeTo>
<TypeTo KeyDescription="1" KeyCode="49">InHand</TypeTo>
<DropOn>MyBox.1</DropOn>
<TypeTo KeyDescription="+" KeyCode="43">MyBox.2</TypeTo>
</Actions>
<WorkingInfo SetDownCounter="0" RecentPageNumber="0">
<Box Parameter="4" GUID="kljeifikhhjbpbaojaihmobfclmjognk">
<NumberOfHoles>2</NumberOfHoles>
<Hole>1
<RemoteControl Code="3" Description="SpeedtoTop" Version="2">
<For>
<Picture Link="kpndefipgjdelhepj1jibjplnjcmplji"/>
</For>
<Value>
<RemoteControl>200</RemoteControl>
</Value>
<Active/>
</RemoteControl>
<Label><! [CDATA[Up Speed]]></Label>
</Hole>
<Hole>2
<RemoteControl Code="14" Description="UpDownhit?" Version="2">
<Appearance>
<Picture Parameter="1">
<BuiltIn Code="53" PictureNumber="1" Description="the hit or miss cartoon"/>
<Geometry Z="-2147483648">
<Left>15710</Left>
<Bottom>13341</Bottom>
<Width>41</Width>
<Height>52</Height>
</Geometry>
<Active/>
<GeometryWhenFreeAndUnflipped>
<Width>2400</Width>
<Height>7400</Height>
</GeometryWhenFreeAndUnflipped>
</Picture>
</Appearance>
<For>
<Picture Link="kpndefipgjdelhepj1jibjplnjcmplji"/>
</For>
<Active/>
</RemoteControl>
<Label><! [CDATA[Up Collide?]]></Label>
</Hole>
<Active/>
</Box>
</WorkingInfo>
```

```
<Active/>
</Robot>
</OnBack>
<OnBack>
<Robot Version="3">
<Label><! [CDATA[sound]]></Label>
<ThoughtBubble>
<Box Parameter="4">
<NumberOfHoles>2</NumberOfHoles>
<Hole>1
<Picture>
<BuiltIn Code="53" PictureNumber="0" Description="the hit or miss cartoon"/>
<Active/>
<GeometryWhenFreeAndUnflipped>
<Width>1920</Width>
<Height>960</Height>
</GeometryWhenFreeAndUnflipped>
</Picture>
<Label><! [CDATA[Collide?]]></Label>
</Hole>
<Hole>2
<BuiltInSound Index="48">
<TextValue><! [CDATA[Twing]]></TextValue>
<Active/>
</BuiltInSound>
</Hole>
<GeometricRelationToContainer RecordedLeaderWidthSquared="14638276" RecordedLeaderHeightSquared=
  ="17181025" WidthTimesLeaderWidth="9802212" HeightTimesLeaderHeight="6818525"
  XOffsetTimesLeaderWidth="2544290" YOffsetTimesLeaderHeight="8252695" Z="-2147483648"/>
<Active/>
</Box>
<GeometricRelationToContainer RecordedLeaderWidthSquared="19882681" RecordedLeaderHeightSquared=
  ="21344400" WidthTimesLeaderWidth="18473637" HeightTimesLeaderHeight="20780760"
  XOffsetTimesLeaderWidth="11972415" YOffsetTimesLeaderHeight="22485540" Z="-2147483648"/>
<Active/>
<SeeAll/>
</ThoughtBubble>
<Actions ID="170" Version="2" MadeOrMovedCount="0">
<TypeTo KeyDescription=" " KeyCode="32">MyBox.2</TypeTo>
</Actions>
<WorkingInfo SetDownCounter="0" RecentPageNumber="0">
<Box Parameter="4" GUID="acgpgmgoiloeibabajgegnjldlimbp">
<NumberOfHoles>2</NumberOfHoles>
<Hole>1
<RemoteControl Code="7" Description="Colliding?" Version="2">
<Appearance>
<Picture Parameter="1">
<BuiltIn Code="53" PictureNumber="1" Description="the hit or miss cartoon"/>
<Geometry Z="-2147483648">
<Left>16255</Left>
<Bottom>13341</Bottom>
<Width>41</Width>
<Height>52</Height>
</Geometry>
```

```

<Active/>
<GeometryWhenFreeAndUnflipped>
<Width>2400</Width>
<Height>7400</Height>
</GeometryWhenFreeAndUnflipped>
</Picture>
</Appearance>
<For>
<Picture Link="#kpndefipgjdelhepj1jibjplnjcmplpji"/>
</For>
<Active/>
</RemoteControl>
<Label><! [CDATA[Collide?]]></Label>
</Hole>
<Hole>2
<BuiltInSound Index="#48">
<TextValue><! [CDATA[Twing]]></TextValue>
<Active/>
</BuiltInSound>
</Hole>
<Active/>
</Box>
</WorkingInfo>
<Active/>
</Robot>
</OnBack>
<OnBack>
<Box Link="#ldgmlookleocbhenkpdgmefpnidfijkb"/>
</OnBack>
<OnBack>
<Box Link="#iicjpipdeiflglgelkaijflhmoklgfopi"/>
</OnBack>
<OnBack>
<Box Link="#kljeifikhhjbpbaojaihmobfcilmjognk"/>
</OnBack>
<OnBack>
<Box Link="#acgpgmgoilioeiebibajgegnjldlimbp"/>
</OnBack>
<GeometryWhenFreeAndUnflipped>
<Width>1260</Width>
<Height>1260</Height>
</GeometryWhenFreeAndUnflipped>
</Picture>
</OnFront>
<GeometryWhenFreeAndUnflipped>
<Width>16000</Width>
</GeometryWhenFreeAndUnflipped>
</Picture>
</ToonTalkObject>

```

C.4.3. Tic-Tac-Toe

siehe auf *GitHub* [Gob21b]

C.5. Implementierungen in Game-Changineer

C.5.1. Fibonacci

Listing C.21. Code

```
There is a dino.
The initial f1 of the dino is 1.
The initial f1 of the dino is displayed.
The initial f2 of the dino is 1.
The initial temp of the dino is 1.
The initial num of the dino is 1.

When the dino is not dead, the temp of the dino is the f2 of the dino.
The f2 of the dino increases by the f1 of the dino.
When the dino is not dead, the f1 of the dino is the temp of the dino.
The num of the dino increases by 1.

When the num of the dino is 10, game over.
```

C.5.2. Bouncing Ball

Listing C.22. Code

```
There is a ball.
The angle of the ball is 45 degrees.
The speed of the ball is 10 pixels per frame.
The ball starts by moving forward.
When the ball touches a border, it reverses direction.
```

C.5.3. Tic-Tac-Toe

Listing C.23. Code

```
// Setze die Farbe des Canvas.
The canvas is teal.

// Plaziere die Tictactoe-Felder.
There are 9 topazes aligned 3 by 3 in the center.
The size of the topazes is 60.
When all the topazes are gone, game over.

// Der Ball kontrolliert das Spiel.
// Wenn auf ihn geklickt wird, dann ist Game-Over um das Spiel neuzustarten.
The size of the ball is 50.
When the ball is clicked, it turns yellow.
When the ball is yellow, game over.

When the original topaz is clicked, the topaz becomes notified for 0.1 second.
When the topaz is notified, the ball becomes notified for 0.1 second.
When the original ball becomes notified,
the ball becomes green and the topazes become kittyfied for 0.1 second.
When the green ball becomes notified,
the ball becomes original and the topazes become puppyfied 0.1 second.
```

When the notified topaz becomes kittyfied, the topaz inserts a kitten.
 When the notified topaz becomes puppyfied, the topaz inserts a puppy.

The size of the puppies is 30.
 The size of the kittens is 30.

// Die Felder verschwinden, sodass sie nicht nochmal angeklickt werden koennen.
 When a puppy touches a topaz, the topaz disappears.
 When a kitten touches a topaz, the topaz disappears.

Listing C.24. Map

 --1-----

C.5.4. Flappy-Bird

Listing C.25. Code

// Setting up the turtle timer.
 The turtle moves right.
 The speed of the turtle is 4 pixels per frame.
 When the turtle reaches the right border, it wraps around and it becomes yellow for 0.1 second.
 When the turtle is yellow, the turtle inserts[4, 600, 600, 0, 130] a brick and the bricks are notified
 for 0.1 second.
 When the turtle touches the spinstar it becomes excited for 0.1 second.
 When the turtle is excited, the score increases by 1.

The spinstar is invisible.

// Spawning the lower brick
 When the brick is notified, it inserts[2, 400] a brick.

// Defining the bricks.
 There are 0 bricks.
 The size of the bricks is 300.
 The angle of the bricks is 90.
 The bricks are moving left at 200 pixels per frame.
 When a brick is not dead, it becomes purple. // setting initial color
 When the position_x of the brick is less than 0, it disappears.

// Setting up the bird.
 The player controls the bird.
 The bird falls.
 The size of the bird is 40.
 The bird is fit.
 When space is pressed, the bird becomes energized for 0.1 second and the angle of the bird equals 0.
 When the fit bird is energized, it jumps by 5 pixels per frame.
 When the direction_y of the bird is less than 0, the angle of the bird decreases by 3.
 Otherwise, the angle of the bird increases by 3.
 When the angle of the bird is greater than 80, the angle of the bird equals 80.
 When the bird collides with a brick, it is not fit.
 When the bird is not fit, it moves down at 600 pixels per frame.
 When the bird is not fit, the angle of the bird increases by 1.

When the bird touches the bottom border, game over.

When the bird lands on a brick, game over.

Listing C.26. Map

```
-----s-T-
-----
-----
-----
-----
-----
-----
-----
-----
-----
-----b-----
```

C.5.5. Tamagochi

Listing C.27. Code

The canvas is pink.

There is a unicorn in the center.

The size of the unicorn is 200.

The energy of the unicorn is 75.

The love of the unicorn is 75.

The energy of the unicorn is displayed.

The love of the unicorn is displayed.

The size of the broccoli is 70.

The size of the cake is 70.

The size of the apricot is 70.

The love of the unicorn decreases by 0.0333.

When the love of the unicorn is less than 0, the unicorn explodes.

When the unicorn is dead, game over.

When the unicorn is blue, the energy of the unicorn increases by 0.0666.

When the unicorn is green, the energy of the unicorn decreases by 0.0999.

Otherwise the the energy of the unicorn decreases by 0.0333.

The score is initially 0.

When the love of the unicorn is greater than 0, the score increases by 0.0333.

When the love of the unicorn is greater than 100, the love of the unicorn equals 100.

When the energy of the unicorn is greater than 100, the energy of the unicorn equals 100.

When the energy of the unicorn is less than 0, the energy equals 0.

When the energy of the unicorn is less than 11, the unicorn turns blue for 10 seconds.

When the original broccoli is clicked, the broccoli becomes active for 0.1 second.

When the original cake is clicked, the cake becomes active for 0.1 second.

When the original apricot is clicked, the apricot becomes active for 0.1 second.

When the broccoli is active, the unicorn becomes healthy for 0.1 second.

When the unicorn is healthy, the love of the unicorn decreases by 1.
When the unicorn is healthy, the energy of the unicorn increases by 1.5.

When the apricot is active, the unicorn becomes loved for 0.1 second.
When the unicorn is loved, the love of the unicorn increases by 1.
When the unicorn is loved, the energy of the unicorn decreases by 1.5.

When the cake is active, the unicorn become sweet for 0.1 second.
When the unicorn is sweet, the energy of the unicorn increase by 1.5.
When the unicorn is sweet, the unicorn turns green for 15 seconds with a 30 percent probability.

When the unicorn turns blue, the broccoli turns red for 0.1 second.
When the unicorn turns blue, the apricot turns red for 0.1 second.
When the unicorn turns blue, the cake turns red for 0.1 second.

When the unicorn turns green, the cake turns red for 0.1 second.
When the unicorn turns green, the apricot turns red for 0.1 second.

Listing C.28. Map

```
-----
-----
-----
----@----L----#----
-----
-----
-----
-----
-----
-----
-----
-----
-----
-----
-----u-----
```