Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Masterarbeit

Fenja Harbke

DSL zur automatisierten Generierung von Comics

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Fenja Harbke

DSL zur automatisierten Generierung von Comics

Masterarbeit eingereicht im Rahmen der Masterprüfung

im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Philipp Jenke
Zweitgutachter: Prof. Dr. Zhen Ru Dai

Eingereicht am: 26. Mai 2016

Fenja Harbke

Thema der Arbeit
DSL zur automatisierten Generierung von Comics

Stichworte
Comic, domänenspezifische Sprache, Bildgenerierung, nicht-fotorealistisches Rendern, Computergrafik, Cartoon-Shader, LibGDX, Meta Programming System, Java

Kurzzusammenfassung
In dieser Arbeit wird die Implementierung einer DSL beschrieben. Mit ihr werden Eigenschaften formuliert, anhand derer ein unterliegendes Java-Programm einen Comic generiert.
Dieser bisher einzigartige Ansatz nutzt 3D-Modelle als Grundlage für die darzustellenden Charaktere und ein passendes Grafikframework, um die in einem Comic-Panel dargestellte Szene zu generieren. Umgesetzt werden u.a. verschiedene Gesten, Sprechblasen und variable Panels.
Das fertige System ermöglicht die einfache und schnelle Produktion von Comics.

Fenja Harbke

Title of the paper
DSL for automatic comic-generation

Keywords
comics, domain specific languages, picture generation, non photorealistic rendering, computer graphics, cartoon shaders, libGDX, Meta Programming System, Java

Abstract
This thesis describes the implementation of an DSL to describe properties for an underlaying Java system to generate a comic.
This unique approach uses 3D-modells to display Characters and a graphical framework to build the scene in the comic panel. Shown are amongst others different gestures, speachballoons and variable panels.
The finished system allows quick and easy production of comics.

Inhaltsverzeichnis

Abbildungsverzeichnis

1. Einleitung

In der heutigen Informationsgesellschaft geht es immer mehr darum, möglichst viele Informationen in möglichst kurzer Zeit aufzunehmen. Die meisten Informationen liegen hierbei in Textform vor. Gebrauchsanweisungen, Projektdokumentationen, Nachrichten. Diese Massen an Text sind häufig eine undankbare Informationsquelle, denn kaum jemand mag sich durch zweitausend Seiten an technischer Dokumentation arbeiten, auch wenn Diagramme und Schaubilder etwas Abwechslung anbieten.
Das bisher vernachlässigte Medium Comic bietet hier die Möglichkeit, Informationen über mehrere Kanäle zu vermitteln. Stephanie Maurer [2, S. 61] fand im Rahmen einer medizinischen Doktorarbeit heraus, dass Comics als bewegte Bilder und Sprache wahrgenommen werden. Und auch Scott McCloud stellte treffend fest: „Put comics in front of most people and they'll find them harder not to read than to read!“ [3, S.85]

Comics sind ein aktuelles Thema. Sie legen allmählich ihr Image als Kinderliteratur ab und werden als erfolgreiche Blockbuster verfilmt. Schon länger existieren Comic-Strips in Zeitungen, Karikaturen zur Auflockerung von Berichten und als alternative Form der Anleitung. Bei all den Vorteilen, die Comics zu bieten haben, liegt es nahe, dieses Medium umfangreicher zu Nutzen. Comics zu produzieren, wie aktuell Textdokumente oder Webseiten. Und das möglichst ohne, dass sich ein Künstler zeitaufwändig mit der Gestaltung und Umsetzung einzelner Panels beschäftigen muss. Ein Lösungsansatz hierfür wird in der vorliegenden Arbeit umgesetzt.

1.1. Motivation

Ausgangsidee für diese Arbeit war es, langatmige, wenig gelesene Projektdokumentationen in Comic-Form zu bringen, um ihr Lesen attraktiver und angenehmer zu gestalten. Fast jeder Mensch liest wohl lieber einen bebilderten, humorvollen Comic, als unendlich erscheinende Seite mit Text und Diagrammen. Nicht nur Dokumentationen, sondern jegliche Informationen soll in einen Comic verpackt werden können, um dieses Medium zu einem anerkannten Kommunikationsmittel zu erheben. Da die Herstellung solcher Comics für gewöhnlich Zeichner und Zeitressourcen kostet, ist ein einfacher, automatisierter Herstellungsprozess gefragt. Der

Computer soll einen Comic nach den Beschreibungen des Benutzers generieren.
Weiter gesponnen ermöglicht solch ein Comic-Generator zum einen die Verwendung eines angenehm zu lesenden Vermittlungsmediums für Dokumentationen, Newsletter usw., zum anderen kann es aber auch als Hilfestellung für Künstler, die ihre Werke planen wollen, als Storyboard für Theater und Film, oder ganz allgemein als Unterhaltungswerkzeug für Geschichtenerzähler dienen.

1.2. Ziel der Arbeit

Im Rahmen der Masterarbeit soll ein System zur Comic-Generierung entstehen, das eine Arbeit mit den Grundelementen des Comics ermöglicht. In Form einer Comic-DSL beschreibt der Anwender einen Comic nach seinen Vorstellungen. Die abstrakte Repräsentation wird von einem Java-System, unter Nutzung eines Grafik-Frameworks, in einen Comic umgewandelt, der den Beschreibungen der DSL entspricht.
Die Comic-DSL soll möglichst leicht zu erlernen sein und Comic-Vokabular nutzen, um unter anderem auch Programmier-Neulinge und Künstler anzusprechen. Anhand des Quellcodes sollen die Hierarchie des späteren Comics, sowie Kernelemente möglichst einfach abzulesen sein. Der Funktionsumfang des Systems soll dem entsprechen, was innerhalb des Zeitrahmens der Masterarbeit umgesetzt werden kann. Grundfunktionalitäten sind die Repräsentation verschieden großer Panels auf einer Seite, sowie die Anzeige von Charaktere und Sprechblasen.

1.2.1. Anforderungen

Das während der Masterarbeit entstehende System soll eigenständig nutzbar sein und ohne großen Aufwand eingerichtet werden können. Dies ist vor allem für eine Zielgruppe wichtig, die wenig IT-Erfahrungen aufweist. Fertige Comics sollen publiziert werden können. Entsprechend muss ein Format angeboten werden, das plattformunabhängig und auf möglichst vielen Geräten angezeigt werden kann.

1.2.2. Abgrenzung

Bei der Implementierung des Systems wird viel Wert auf Erweiterbarkeit gelegt, vor allem da der im Rahmen der Masterarbeit geleistete Umfang wenig mehr als die geforderten Grundfunktionalitäten beinhaltet. So ist es mehr als eine Machbarkeitsanalyse zu sehen, aus welcher mit entsprechendem Mehraufwand ein ggf. marktfähiges Produkt entstehen kann. Aufgrund technischer Einschränkungen wird das System nicht für iOS entwickelt oder getestet. Ebenso

wenig kann eine Kompatibilität mit sämtlichen Betriebssystemen und Hardware gewährleistet werden.
Die mit dem System entstehenden Comics erheben keinen Anspruch auf die Gleichstellung mit einem, durch einen Comic-Künstler erschaffenes Werk. Der Künstler soll nicht vom System ersetzt, sondern unterstützt werden, indem das Medium populärer und verbreiteter wird. Lediglich die handwerkliche Messlatte wird aufgrund der technischen Präzision etwas höher gelegt.

1.3. Struktur der Arbeit

Die vorliegende Arbeit stellt verwandte Arbeiten und das Medium Comic, sowie Grundlagen der DSL vor. Es folgt die Vorstellung der technologieunabhängigen Konzeptionen, sowie des konkreten Systems. Abschließend finden Evaluation und Zusammenfassung statt.

Kapitel 2 stellt mehrere verwandte Arbeiten mit Comic-Schwerpunkt dar, deren Erkenntnisse als Inspiration oder Unterstützung für die vorliegende Arbeit dienen. In Kapitel 3 wird das Medium Comic, seine Historie, sowie seine Eignung zur Informationsvermittlung vorgestellt. Künstlerisches Vorbild sind hier die so genannten Instruktions-Comics (3.2). Weiterhin wird ein Überblick über die comic-eigenen Stilmittel (3.3) und Grundlagen (3.4) geboten. Kapitel 4 stellt die domänenspezifische Sprache an sich vor, zeigt Design-Pattern auf (4.2) und welche Anforderungen (4.3) an eine Comic-DSL gestellt werden müssen.
Die von der Technik unabhängige Konzeption wird in Kapitel 5 präsentiert. Thematisiert werden hier der DSL-Entwurf nach Design-Pattern (5.1), die Architektur des Systems (5.2) und der abwechslungsreiche Bilder ermöglichenden Determinismus (5.3). Weiterhin werden comicspezifisch die zugrunde liegende 3D-Szenerie (5.4), der für den typischen Comic-Stil nötige Cartoon-Shader (5.5), sowie für das Erscheinungsbild des Comics getroffene Entscheidungen (5.7) erläutert.

In Kapitel 6 wird die konkrete Implementierung der Comic-DSL aufgezeigt. Beginnend bei der Auswahl einer geeigneten Language Workbench (6.1) zur spezifischen Umsetzung mit dem Meta Programming System (MPS) (6.2). Der MPS-eigene Aufbau begleitet die nächsten Schritte; die Festlegung einer logischen Struktur (6.3), der Entwurf eines Editors (6.4) und die Spezifikation eines Generators (6.5). Letzterer nutzt die vom zweiten Teil des Systems bereitgestellte Schnittstelle (6.7). Abgeschlossen wird das Kapitel mit einem Aufzeigen der umgesetzten Qualitätssicherung (6.8), sowie einer Zusammenfassung (6.9) der Comic-DSL.

Kapitel 7 zeigt die Grafik-Komponente des Systems. Beginnend bei der Auswahl eines Grafik-Frameworks (7.1), bis zur konkreten Umsetzung mit LibGDX (7.2). Kernelement des Comic-Generators sind die 3D-Modelle der Charaktere, die mit einem passenden System erstellt (7.3) und gewissen Anforderungen genügen müssen (7.4). Der Comic-Generator ist wiederum in mehrere Sub-Komponenten mit klar abgegrenzter Aufgabenteilung untergliedert. Comic-Generator (7.5) dient als Schnittstelle für die DSL und als koordinierende Hauptklasse. Der SceneBuilder (7.6) erstellt die darzustellende Szene unter Zuhilfenahme der 3D-Modelle. Der PanelRenderer (7.7) fügt die Sprechblasen und den passenden Comic-Stil hinzu. Ein Export (7.8) ermöglicht die Darstellung des Comics auf verschiedenen Plattformen. Zugehörige Qualitätssicherung (7.9) und eine Zusammenfassung des Comic-Generators (7.10) schließen das Kapitel ab.

Mit einem Blick von außen evaluiert Kapitel 8 die Arbeit. Betrachtet wird der aktuelle *State of the art* (8.1). Ein Durchlauf des Comic-Generierungs-Prozesses wird mit einem zuvor erstellten Wunsch-Ergebnis in Form eines Comics unternommen (8.2). Zusammenfassend wird noch einmal der aktuelle Projektumfang betrachtet (8.3), das Ziel der Erweiterbarkeit beleuchtet (8.4) und auf während der Arbeit entstandene Schwierigkeiten (8.5) eingegangen.
Im letzten Kapitel (9) wird noch einmal die dieser Arbeit übergeordnete Vision vorgestellt, aus beliebigem Text eine Comic-Repräsentation zu generieren (9.3). Nächste Schritte zur Weiterführung der Arbeit (9.4) werden aufgezählt, ein Fazit gezogen (9.5) und ein Ausblick auf weitere Arbeiten, sowie die Zukunft der Comic-Generierung (9.6) gegeben.

2. Verwandte Arbeiten

Es gibt viele Arbeiten, die sich mit literarischen oder pädagogischen Aspekten von Comics beschäftigen. Neil Cohn [4, 5, 6] beschäftigt sich ausgiebig mit der dem Comic eigenen visuellen Sprache und einige andere ordnen Filme oder Fotos im comic-typischen Layout an. Die Generierung von Comic-Inhalten und eines Comics, ohne dass ein Anwender bei den Zwischenschritten Feder führt, ist ein nicht allzu häufig behandeltes Thema.
Im Folgenden werden einige Arbeiten vorgestellt, die vorrangig mit dem Thema Comic in Verbindung stehen. Sie behandeln Aspekte und Algorithmen, die für die vorliegende Masterarbeit übernommen wurden oder als Inspiration dienten.
Nach Vorstellung der Arbeiten wird jeweils noch einmal auf den Kernaspekt eingegangen, und inwiefern ein Mehrwert für die Masterarbeit daraus gezogen werden konnte.

2.1. Comic Chat

Der auf 2D-Grafiken basierende Comic Chat wurde 1996 von Microsoft veröffentlicht und war später auch als Microsoft Chat bekannt. Besonders ist die Darstellung des Chats nicht als Text, sondern als Comic. Bis zu seiner Abschaltung 2001 war er der offizielle Chat Client des Microsoft Netzwerkes (MSN). David Kurlander entwickelte das Projekt im Microsoft Research, zusammen mit Tim Skelly und David Salesin in C++. Als Comic-Künstler wurde Jim Woodring in das Projekt geholt. Woodring extrahierte die Regeln, die er zum Zeichnen befolgte und diese wurden im System eingesetzt [7].
Comic Chat kommuniziert über das Internet via IRC-Protokoll (Internet Relay Chat) und kann noch heute über entsprechende Server genutzt werden. Obwohl die Publikation von 1996 ist, hat sie keinesfalls an Aktualität eingebüßt. Seit ihrer Veröffentlichung wird sie von einer Mehrheit der Arbeiten, die sich mit Comics beschäftigen, zitiert.

Kernaspekt der Arbeit sind die von einem Künstler geschaffenen Grafiken, auf deren Pool der Comic-Chat beschränkt ist. Anhand von Schlüsselwörtern in den Chat-Texten werden passende Gesten und Gesichtsausdrücke ausgewählt. So winkt ein Avatar beispielsweise, wenn der Anwender „Hallo“ schreibt. Alternativ und ergänzend steht ein *Emotion-Wheel* zur Verfügung,

über welches der Anwender die darzustellende Emotion auswählen kann.
Sehr erwähnenswert sind die in der Arbeit vorgestellten Algorithmen zur Charakter- und Sprechblasen-Platzierung. Berücksichtigt werden hierbei die Anzahl der Sprecher, deren Positionierung zueinander und im Bezug auf die Sprechblasen, dass diese in der richtigen Reihenfolge angezeigt werden und sich die Dornen der Sprechblasen nicht überkreuzen. Weiterhin stehen mehrere Sprechblasen-Typen zur Verfügung. Variabilität wird mit Hilfe des Zooms erzeugt. Zur Verfügung stehen mehrere Zoom-Stufen, wobei möglichst keine zwei aufeinander folgenden Panels die gleiche Einstellung haben sollen.

Der im Comic-Chat verfolgte 2D-Ansatz wurde für die Masterarbeit verworfen. Ein Pool mit 3D-Modellen der Charaktere bietet zum einen erheblich mehr Variabilität, was Gestik und Perspektive betrifft, zum anderen muss für die Einführung eines neuen Charakters nur ein neues Modell erstellt werden. Beim Comic-Chat würde eine solche Neueinführung einen Aufwand von mehreren zu zeichnenden Gesten und Gesichtsausdrücken bedeuten.

2.2. Comic Book Markup Language

Die Arbeit über die Comic Book Markup Language (CBML)[1] von 2012 stellt ein System vor, mit welchem Comics mit Tags versehen werden, um ihre Inhalte im Rahmen eines Bibliotheks-Systems zu durchsuchen [8]. Die CBML basiert auf der *Text Encoding Initiative P5: Guidelines for Electronic Text Encoding and Interchange.*

Die CBML ist eine Sammlung von Tags (maschinenlesbarer, textueller Code), zur Codierung und Analyse von Comics, Comicbüchern und Graphic Novels, um eine Übersicht über die Inhalte der Comic-Publikationen zu geben. Somit soll nach Bibliothek-Standards in Comic-Dokumenten gesucht werden können. Besonderes Augenmerk liegt hierbei auf der Beschreibung und Analyse des Zusammenspiels von Text und Bild.
Verschiedene Textinhalte werden mit entsprechenden Tags beschrieben. So z.B. die Textbox („caption“), die Sprechblase („balloon“) mit Typ („type“) und Sprecher („who“), sowie Lautwörter („sound“). Der übergeordnete Panel-Tag führt die enthaltenen Charaktere ("characters") auf und nutzt die von Scott McCloud [9] vorgestellte Panel-Transition (z.B. „action-to-action“). Weiterhin wird eine Nummerierung der Panels durchgeführt. Hier kommt es zu Schwierigkeiten bei komplexen Panel-Layouts. Auch eine allgemeingültige Beschreibung der Bilder-Inhalte ist nicht möglich, für außergewöhnliche Inhalte kann allerdings das „note“-Tag zur Hilfe ge-

[1] Webseite des Projektes auf www.cbml.org

(a) Eigenes Beispiel-Panel

```
<cbml:panel
    n="7"
    characters="#fenja"
    ana="#moment-to-moment"
    xml:id="ma 000"
        xmlns:cbml="http://www.cbml.org/ns/1.0">
<cbml:caption>
    Gut zu wissen
</cbml:caption>
<cbml:balloon xml:id="ma 005" type="speech" who="#fenja">
    Die <emph rendition="#b">Szene</emph> ist
    das wichtigste
    im Panel
</cbml:balloon>
<sound>PLOP</sound>
</cbml:panel>
```

(b) Zugehöriger Panel-Tag der CBML

Abbildung 2.1.: Beschreibung eines Panels in CBML

nommen werden. Ein Auszug aus CBML-Code ist in Abbildung 2.1 zu sehen.
Die in der CBML umgesetzte Hierarchie der Tags eignet sich als Vorlage für die in der Masterarbeit genutzte DSL. Sowohl Vokabular, Elemente, wie auch Attribute können fast eins-zu-eins übernommen werden. Da die Arbeit von Walsh erst nach dem Entwurf der DSL gefunden wurde, fand mehr ein Abgleich von DSL und CBML statt, der eine sehr große Übereinstimmung zum Ergebnis hatte. Lediglich einige Begrifflichkeiten wurden der CBML angepasst, da zuvor kein geeigneter Fachbegriff vorlag.

2.2.1. Text Encoding Initiative

Die Text Encoding Initiative (TEI)[2] ist ein internationaler und interdisziplinärer Standard, der von Bibliotheken, Museen, Verlagen und Universitäten genutzt wird, um Literatur und linguistische Texte zu repräsentieren. Genutzt wird hierzu ein Schema, das möglichst ausdrucksstark ist und nach Möglichkeit nicht (oder nur minimal) veraltet.
Bei der TEI-Repräsentation von Comics ist die Herausforderung, dass der in Textboxen und Sprechblasen enthaltene Text soweit möglich durch Beschreibungen des Panel-Inhaltes, also Charaktere, Symbole und Handlungen, ergänzt werden muss. Die CBML stellt hier den entsprechenden Lösungsansatz vor.

[2] Webseite der TEI unter www.tei-c.org

2.3. Comic Computing

Hiroaki Tobitas Vision [10] baut darauf auf, dass eine Kommunikation durch Comics visuell und somit interessanter wird. Mit seinem System „Comic Computing“ können User mit einfachen Mitteln Comics erstellen und sie mit ihren Freunden teilen. Comic Computing nutzt Comics zur Repräsentation komplexer Informationen. Im Gegensatz zu *information visualization* (IV) zur Datenvisualisierung werden in Comics Geschichten visualisiert.
Ein großer Teilaspekt von Tobitas Arbeiten beruht darauf, dass große Elemente immer im Fokus liegen. Beim Umblättern der Seite eines Comic-Buches liegt der Fokus immer zuerst auf dem größten Panel („big frame“), während kleine Panels unterstützend wirken („small frame“). Ebenso gibt es innerhalb des Panels Objekte, die durch ihre Größe und Detailreichtum im Fokus liegen und die unterstützenden Objekte. Eine andere Arbeit von Tobita, die auch Teil von Comic Computing ist, beschäftigt sich mit dem Hervorheben bestimmter Regionen von Fotos mit Hilfe des Fischaugen-Effektes, um diesen Fokus hervorzurufen. So wird beispielsweise der ausholende Arm eines Baseball-Werfers hervorgehoben, um das Bild insgesamt interessanter zu machen [10, S. 92].

Comic Engine Ein weiteres Prinzip, das Tobita in seinem Comic Computing einsetzt, ist das im Comic Engine [11] erarbeitete „Attention cuing“. Hierbei wird der Lesefluss beeinflusst. Die Aufmerksamkeit des Lesers wandert von Elementen zum Text. Wichtig sind Winkel und Länge zwischen dem aktuellen und dem nächsten Element. Je spitzer der Winkel, desto schneller der Lesefluss.

Comic Computing™ beta Die Ende 2014 nicht mehr verfügbare Beta-Version des Comic Computing ermöglicht die einfache Erstellung eines Comics. Der User kann eigene Fotos einfügen, den Comic Style nutzen und Effekte hinzufügen. Aktuell (März 2016) ist die zugehörige Webseite nicht mehr online. Im ersten Schritt wird ein Seitenlayout ausgewählt, in dessen Panel per Drag & Drop eigene Fotos gezogen werden können. Der Benutzer fügt Sprechblasen hinzu, die er mit Text füllt und für die Sprechblase einen Stil auswählt (z.B. Denkblase). Die Bilder können zusätzlich mit comic-typischen Effekten bearbeitet werden und werden zuletzt durch einen Shader in Cartoon-Fotos deformiert [12].

Der Comic wird letztendlich im XML-Format abgespeichert. Hierarchisch sind „ story“, „page“, „frame“ und „frame elements“ geschachtelt. Zusätzlich werden Informationen wie „image“, „effect“, „scale“ und „position“ angegeben, die unter anderem für die Fischaugen-Deformation von Comic Engine genutzt werden. Ein sinngemäßer Auszug aus einem solchen XML-Code ist

```
<?xml version="1.0" encoding="UTF-8"?>
<comic>
    <title>ComicTitle</title>
    <page num="1">
        <frame x="8.5" y="1" w="11" h="14">
            <frameAnimation time="-1">normal</frameAnimation>
            <focusLine x="5" y="6">normal</focusline>
            <background>back01.jpg</background>
            <character x="2" y="10" w="4" h="6" direction="front">Chara01</character>
            <bubble type="A" x="6" y="2.5" w="10" h="4">text01</bubble>
            <bubble type="A" x="9" y="10" w="8" h="7">text02</bubble>
        </frame>
        <frame x="1" y="1" w="6.5" h="14">
            <frameAnimation time="-1">normal</frameAnimation>
            <focusLine x="3.5" y="6.5">normal</focusline>
            <background>back02.jpg</background>
            <bubble type="B" x="4.5" y="2" w="8" h="3">text03</bubble>
        </frame>
    </page>
</comic>
```

Abbildung 2.2.: Comic Computing XML

in Abbildung 2.2 zu sehen.

Die Generierung des Comics erfolgt bei Tobita auf Grundlage von bereits fertigen Bildern, den Fotos der Anwender. Ein für diese Arbeit übernehmenswerter Aspekt ist jener der hierarchischen Speicherung der Panels. Wie in Abbildung 2.2 zu sehen, werden Comic, Seiten und Panels entsprechend geschachtelt, sodass die Hierarchie am Quelltext abzulesen ist.

2.4. Cartoon-Looking Rendering of 3D-Scenes

Im Rahmen seiner Doktorarbeit entwickelte Philippe Decaudin einen Algorithmus, um Bilder mit typischem Cartoon-Look zu erstellen [1]. Aus 3D-Beschreibungen der Szene, statisch oder aber animiert, wird ein traditioneller Cartoon geschaffen. Die zum Rendern genutzten Techniken ermöglichen ein Umranden von Profil und Kanten in schwarz, eine einheitliche Kolorierung innerhalb der Outlines, sowie das Einfügen von Schatten anhand von Lichtquellen. Während sich die meisten Rendering-Verfahren damit beschäftigen Objekte so realistisch wie möglich aussehen zu lassen, nutzt das Non-Photorealistic Rendering das stilisierte Aussehen von Cartoons.

Objekte und Charaktere werden im Cartoon durch ihre Outlines, sowie die darin eingeschlossene, diskretisierten Farbplateaus definiert. Um den Cartoon-Look für Bilder zu erhalten, sind Outlines konstanter Dicke und einheitlich ausgefüllte Flächen nötig. Anstatt mit Farbe könne solche Flächen auch mit Texturen gefüllt werden. Lichtreflexe geben Aufschluss über die Be-

schaffenheit von Materialien. Beim Schatten unterscheidet man zwischen den der Lichtquelle abgewandten Flächen des Objektes („backface shadow“) und dem Schlagschatten, den das Objekt wirft ("projected shadow"). Einfache Cartoons enthalten meist überhaupt keine Schatten.

Die Arbeit von Greif und Gübürz [13] beschäftigt sich ebenfalls mit dem Non-Photorealistic Rendering, wobei ihre Ergebnisse mehr das Ziel eines skizzenhaften Aussehens verfolgen. Der Algorithmus von Decaudin [1] wurde für den eigenen Cartoon-Shader übernommen (beschrieben in Kapitel 5.5). Entsprechende Vertex- und Fragmentshader erkennen die Kanten einer zuvor angepassten Szene (Tiefen- bzw. Normalenmap) und zeichnen diese schwarz nach.

2.5. Comix I/O

Der tschechische Programmierer Antonin Hildebrand entwickelte 2013 die JavaScript Bibliothek Cmx.js[3], welche die Programmierung eines Comics im xkcd[4]-Stil ermöglicht. Die Comics von Randall Munroe zeichnen sich durch stilisierte Strichmännchen, aber intelligente Texte aus, sodass der Schwerpunkt auf dem Text liegt.
Die Idee hinter dem Projekt ist es, dass Programmierer Programmcode zur Erstellung von Comic-Strips nutzen können. Bisher ist es ein Prototyp und nicht für die Nutzung durch Nicht-Entwickler gedacht. Die Veröffentlichung des Codes auf GitHub soll das Projekt für die Öffentlichkeit zugänglich machen und so von anderen mit weiterentwickelt werden können. Allerdings hat das Projekt seitdem (2013) keine Erweiterungen erfahren [14].

[3] Comix I/O - Open Source von Antonin Hildebrand „Create your own xkcd-style comics using HTML markup“ (http://cmx.io)
[4] xkcd - ein Webcomic von Randall Munroe mit Themen aus der Informatik, Mathematik, Sprache und Romantik (xkcd.com)

3. Comic

Nach dem „golden Age“ des amerikanischen Comics in den 1940ern, erlebt das Medium Comic heute eine Renaissance. Allen voran die japanischen Manga befüllen deutsche Buchmessen und das Kino-Programm ist von Comic-Blockbustern aus dem Marvel-Universum geprägt. Doch auch neben der Pop-Kultur steigt die bildhafte Visualisierung von Sachverhalten in ihrer Beliebtheit. Erklärende Comic-Strips und Info-Grafiken sind hier nur der Anfang. Beispielsweise hat sich das hamburger Unternehmen Dialogbild[1] auf die Darstellung von komplexen und vielschichtigen Vorgängen, wie am Hamburger Flughafen, durch detailverliebte Wimmelbilder spezialisiert.

Das folgende Kapitel soll das Medium Comic definieren und vorstellen. Die zur bildhaften Erklärung genutzten „Instruktions-Comics“ sind Vorbild für die Grundidee dieser Arbeit. Weiterhin werden die comic-typischen Stilmittel vorgestellt, sowie der Grundbaukasten für Comiczeichner und das System Comic-Generator.

3.1. Definition

McCloud definiert den Comic als „zu räumlichen Sequenzen angeordnete, bildliche oder andere Zeichen, die Informationen vermitteln und/oder eine ästhetische Wirkung beim Betrachter erzeugen“ (vgl.[9]). Der Brockhaus definiert den Comic als eine „literarisch-künstlerische Erzählform, bei der die Erzählung vorwiegend über das Bild transportiert wird“ (siehe [15, S.6]). Dies soll als Definition für diese Arbeit gelten, grundsätzlich gibt es allerdings keine einheitlich akzeptierte Definition von Comics [16, S.30].
Weiterhin sei - keine allgemeingültige, aber für diese Arbeit verwendete - Abgrenzung der Comic-Begriffe eingeführt.

Cartoon ist ein einzelnes Bild, meist im Comic-Stil. Zu einem Comic wird ein Cartoon erst durch die Aneinanderreihung mehrerer Bilder.

[1] Dialogbild, seit 2003, T. Becker, W. Wienecke, www.dialogbild.de

Sprechblasencomic ist die bekannteste Comic-Form, in welcher Text mit Hilfe von Sprechblasen in das Bild integriert wird. Im Rahmen dieser Arbeit werden die Begriffe Comic und Sprechblasencomic äquivalent verwendet.

Comicbuch schließt mehrere, kürzere Geschichten oder eine große Comic-Geschichte in einem (physischen) Buch zusammen.

Graphic Novel ist ein von Will Eisner (3.2.1) geprägter Begriff, um qualitativ anspruchsvollere Comics von der „komischen" Masse hervorzuheben.

Comic Strip ist ein meist einzeiliger Comic, wie er in Zeitungen abgedruckt wird.

3.1.1. Populäre Vertreter

Comics erlebten in verschiedenen Teilen der Welt unterschiedliche Ausprägungen. In Amerika erscheinen die Comics meist episodenweise in Heften. Am bekanntesten sind hier die Superhelden-Comics, unter ihnen *Batman*[2], aber auch Disney-Charaktere wie *Donald Duck*[3], die durch das „Lustige Taschenbuch" ihren Weg nach Deutschland fanden. Das Comic-Buch, welches eine komplette Geschichte beinhaltet, wurde als „Bande Designée" im frankobelgischen Raum etabliert. Hier sind *Asterix*[4] und *Tim und Struppi*[5] als einige der bekanntesten Vertreter zu nennen.
Die aus dem asiatischen, meist japanischen Raum stammenden Manga erfreuen sich aktuell wachsender Beliebtheit im europäischen Raum. Durch große Augen und einen meist niedlichen Zeichenstil werden die Protagonisten dem Leser näher gebracht, als es bei anderen Comic-Vertretern der Fall ist. Als Beispiele sind hier die Shonen-Serie (für Jungen) *Dragonball*[6] und die Shojo-Serie (für Mädchen) *Sailor Moon*[7] genannt.

3.1.2. Historie

Das erste Auftreten von Comics in der Menschheitsgeschichte ist davon abhängig, wie eng man die Definition sieht. Weit gefasst beginnt es bei Höhlenmalereien, ägyptischen Hieroglyphen und dem Teppich von Bayeux [9]. Als 1450 der Buchdruck begann, gab es bereits die heute

[2] Batman, erster Auftritt 1939 *Detective Comics*, Bob Kane, Bill Finger
[3] Donald Duck, seit den 1930er Jahren, Disney, Zeichner u.a. Don Rosa
[4] Asterix, der Gallier (fr. „Astérix"), seit 1959, René Goscinny, Albert Uderzo
[5] Tim und Struppi (fr. „Les aventures de TinTin"), 1907-1983, Hergé
[6] Dragonball (jap), 1984-1995, Akira Toriyama
[7] Sailor Moon (jap), 1992-1997, Naoko Takeuchi

comic-typische Unterteilung von Bildern in Panels durch ein einfaches Raster, allerdings noch ohne Text. 100 Jahre später enthielten Bilderzählungen bereits Überschriften in den Panels und Sprechblasen. Von dort war es nur noch ein kleiner Schritt zu den heutigen Comics [17].

Der heute als Sprechblasencomic bekannte Comic fand seine Verbreitung seit der Wende zum 20. Jahrhundert als Comic Strip in amerikanischen Wochen- und Tageszeitungen. Viele dieser Vertreter hatten Kinder und Jugendliche als Protagonisten, waren aber nicht eindeutig an diese Zielgruppe adressiert.
In den 1930ern entstand das Kiosk-Heft mit längeren Detektiv-, Sciencefiction-, Wildwest- und anderen Geschichten, was die Publikation von umfangreicheren, abgeschlossenen Comic-Geschichten erlaubte. Schnell folgten auch in mehreren Episoden erscheinende Fortsetzungsgeschichten, die vom Verlag später gerne als Album veröffentlicht wurden, wodurch in Frankreich bereits in den 1930ern das Buch als Medium aufkam. In Deutschland fand der Comic erst nach Ende des Zweiten Weltkrieges Verbreitung, als Bildgeschichten für Kinder und Jugendliche. Pädagogische Vorbehalte führten zu Kampagnen gegen die „Schundliteratur“, während die erfolgreichen Reihen den Comic zur Kinder- und Jugendliteratur abstempelten.

Ab Mitte der 1960er Jahre wurden einige der vorhandenen Comicreihen ernster und besonders die französischen Comics *Asterix* und *Lucky Luke* konnten durch grafische Qualität und doppelsinnigen Humor ein deutlich breiteres Publikum ansprechen. In den späten 1990ern kamen die japanischen Manga hinzu, mit ihrem eigenen Stil und Erzählart, ebenfalls in Serienform von schwarz-weißen Taschenbüchern.
Die zunehmende Differenzierung des Genres ermöglichte im deutschsprachigen Raum die Anerkennung eines Zweiges, der sich von Kinderliteratur differenziert (vgl. [18, e. Geschichte und Adressierung]). Gegenwärtig (2014) erreicht das Medium Comic eine neue Stufe der Akzeptanz. Es gilt nun die „Überzeugung, dass Comics als Medium des Schriftspracheerwerbs und der Wissensvermittlung besonders anschaulich und effizient sein können“(siehe [18, S. 458]).

Während in Frankreich und Belgien der Comic bereits als neunte Kunst anerkannt wird, in Japan mehr als 40% der Bevölkerung - aller Alters- und Bildungsklassen - Manga lesen und in Italien die Einschränkung auf Kinder-Comics durch düstere Graphic-Novels aufgehoben wurde, haben Deutschland und Großbritannien nach wie vor ein kindliches Bild vom Medium Comic [16, S.29f]. Ole Frahm spricht ihnen sogar eine Rolle als ewige Parodie zu und betont „Comic-Wissenschaft existiert nicht.“ [19, S. 143]. Dennoch wird langsam erkannt, dass über einen Comic Wissen vermittelt werden kann, das dem Leser auf anderem Wege so

nicht zuträglich wäre [16, S.33]. So sind Schüler inzwischen bereit, den Comic als Lernmittel anzuerkennen.

3.2. Instruktions-Comics

Der Instruktions-Comic (en. „instruction comic“) sei hier als Oberbegriff genannt für Comics deren Ziel es ist, Sachverhalte zu erklären. Geprägt wurde der Begriff von Will Eisner und seinem Beitrag zur Armee-Zeitung. Allgemein wird der Ansatz, Comics zur Informationsvermittlung zu verwenden, von mehreren Institutionen und für die Instruktion unterschiedlicher Entitäten, wie Spiele, Gebrauchsanweisungen oder für erklärende Darstellungen genutzt.

3.2.1. Will Eisner

Will Eisner sei hier zum einen als Zeichner der Instruktions-Comics der Armee vorgestellt, zum anderen als einflussreicher Comic-Künstler. Im Magazin „Army Motors“ zeichnete Eisner Comics, welche den Soldaten den sachgerechten Umgang mit militärischem Gerät näher bringen sollten. Mit Negativbeispielen und Humor erzielten Eisners Comics größeren Lernerfolg als die gewöhnlichen, textuellen Beschreibungen.
Eisner verfasste mehrere, grundlegende Werke über den Umgang und die Stilmittel des Comics [20, 21] und führte den Begriff „Graphic Novel“ ein. Mit seinen Werken wie „The Spirit“, von 1940-1952 wöchentlich als Comicbeilage publiziert, und seinen Graphic Novels, wie „Ein Vertrag mit Gott“ (1978), nicht zuletzt wegen seiner Experimentierfreude mit dem Medium, war und ist er Vorbild für viele ihm nachfolgenden Künstler.

3.2.2. Comic-Anleitungen

Abseits der zahlreichen Publikationen Will Eisners gibt es auch neuere Ansätze, Comics zur Instruktion zu nutzen, wenngleich diese in ihrem Umfeld eher experimenteller Natur sind. Das Brettspiel *Piranha Pedro*[8] wurde mit einer zusätzlichen Spielregel in Comic-Form herausgebracht. Zur Einführung in das Spiel Go wurde ein *Go-Comic*[9] entworfen, in welchem zwei Go-Steine dem Schach-König die Spielregeln erklären. Zuletzt sorgte der von Scott McCloud gezeichnete *Google Chrome Comic*[10], in welchem die Funktionsweise des Browsers erklärt wird, für Begeisterung.

[8] Piranha Pedro, 2004, Goldsieber, Jens Peter Schliemann, Grafik von Marcel-Andrè Casasola Merkle
[9] Go - Eine Einführung, 2000, Deutscher Go-Bund e.V. (www.dgob.de), Andreas Fecke
[10] The Google-Chrome Comic, 2008, www.google.com/googlebooks/chrome/, Scott McCloud

3.3. Stilmittel

Scott McCloud, Verfasser einiger populären Comics über Comics, beschäftigt sich in seinen Werken mit der Wirkung von Bildern auf den Leser: Welche Regeln existieren, um Zeit zu visualisieren, Gefühle darzustellen und den Leser möglichst tief in die Geschichte eintauchen zu lassen. Allen voran in „Making Comics“ [3] gibt er als Leitfaden einen Überblick über die Werkzeuge eines Comic-Künstlers. Viele davon können als Hilfestellung für einen Anwender dienen, der mit Hilfe der Comic-DSL seinen eigenen Comic beschreibt. Einige der Stilmittel seien an dieser Stelle vorgestellt.

Übertreibung Um die Botschaft sicher an den Leser zu vermitteln, werden Gesten, Sprachen und vor allem Emotionen in Comics übertrieben dargestellt. Gesten sind ausgeprägter, die Arme schwingen weiter aus, Schritte sind größer und ein Fausthieb wird durch Perspektive noch unterstrichen. Sprechblasen zeigen den Ton gesprochener Worte auf und Formatierung hebt einzelne Wörter hervor, damit sie nicht überlesen werden.
Vor allem in Manga werden Bewegungen und Emotionen mit einer Anpassung der Umgebung unterstützt. Für Frohsinn scheint die Sonne, bei Angst verdunkelt sich alles und bei Schrecken zucken Blitze auf. Durch Überzeichnung der Gesten, verschiedene Sprechblasen und einen passenden Hintergrund kann also leichter vermittelt werden.

Szenenaufbau Die Szene in einem Panel lässt sich in unterschiedliche Klassen einteilen. Wichtig ist hierbei der Zoomlevel, aber auch die Perspektive. Ein Zoom von der Landschaftsaufnahme über die Totale bis zur Detailaufnahme des Gesichtes gibt dem Leser entweder einen Überblick über die Szene oder lässt hautnah mit dem Protagonisten mitfiebern. In seiner Arbeit unterscheidet Neil Cohn verschiedene Panelarten [4].
Die Perspektive unterstützt, wie die dargestellte Szene vom Leser wahrgenommen wird. Eine Szene auf Augenhöhe wirkt informativ, aus der Vogelperspektive wird ein Überblick über die Situation gegeben und eine Froschperspektive kann bedrohlich wirken [20]. Nicht nur die Szene selbst also, sondern auch der Blickwinkel, der dem Leser dargeboten wird, sind für die übermittelten Informationen wichtig.

Humor Die Wortherkunft des Comics vermittelt bereits, dass es sich meist um komische Inhalte handelt. Das Verpacken von Informationen in lustige Comic-Strips ist bereits von Will Eisner erfolgreich umgesetzt worden (siehe hierzu 3.2.1). Für den Autoren eines Comics bietet es sich also an, wenn und wo möglich einen Gag einzubauen.

Attention-Cuing Tobita [11] beschäftigt sich mit dem Weg, den das Auge des Lesers über eine Comicseite nimmt. Zumeist springt die Aufmerksamkeit des Lesers hierbei von Sprechblase zu Charakter, je nach Leserichtung von rechts nach links oder umgekehrt. Die zwischen diesen Punkten liegenden Winkel beeinflussen die Geschwindigkeit, in welcher das Lesen des Comic empfunden wird (nicht unbedingt die tatsächliche Lesegeschwindigkeit).
Ein flacher Winkel sorgt für langsames Tempo. Paradebeispiel wäre hier, immer das gleiche Panel mit einem Charakter und einer Sprechblase darüber. Spitze Winkel sorgen dafür, dass das Auge hin und her springt und so die wahrgenommene Geschwindigkeit steigt.

3.3.1. Scott McCloud

An dieser Stelle wird noch einmal genauer der Comic-Künstler und -Theoretiker Scott McCloud vorgestellt. Neben seinen gewöhnlichen Publikationen wie *Zot!*[11] wurde er vor allem durch sein Werk „Understanding Comics“[9] und die darauf folgenden Werke [22, 3] bekannt.
Aus einer Familie von Wissenschaftlern stammend begann er schon früh, in aktuellen Comics nach Mustern zu suchen, selbst welche zu zeichnen und sie zu verstehen [17]. Obwohl sie ein visuelles Medium sind können Comics alle fünf Sinne ansprechen, indem sie die Realität mit Zeichnungen abstrahieren. Die außerdem enthaltenen Texte abstrahieren noch weiter [17].
Eine von McClouds wichtigsten Erkenntnissen ist, dass der Leser den Raum zwischen zwei Panels selbst ausfüllt und somit die Erzählung unterstützt. Als Beispiel dient ihm ein erstes Panel mit einem wütenden Messerträger, ein zweites Panel mit einem Schrei. Dazwischen ist im Kopf des Lesers ein Mord geschehen, der nicht explizit dargestellt wurde [9].

Mit der Arbeit am Computer beginnt McCloud sich die Zukunft des modernen Comics auszumalen [17]. Die neue Technologie kann Ton und Bewegung hinzunehmen und ist nicht mehr an den starren Rahmen einer Seite Papier gebunden [3]. Für diese Arbeit vor allem interessant ist das Aufzeigen der Vermittlungsmöglichkeiten eines Comics, so schreibt er: „... potential of comics to communicate ideas – maybe its greatest promise — is, do date, just its best-kept secret“ (siehe [3, S.84]).

3.4. Grundlagen

Einige Stilmittel des Mediums Comic, die mehr noch als Grundregeln bezeichnet werden können, lassen sich so weit abstrahieren, dass sie, wie schon beim Comic Chat [7] (Kapitel 2.1), innerhalb des Systems implementiert werden können.

[11] Zot! (1984-1991) von Scott McCloud, erschienen bei Harper Collins

3.4.1. Panel

Das Panel ist das zentrale Element des Comics. Ein Panel ist ein einzelnes Bild, das erst zusammen mit weiteren Bildern einen Comic ergibt. Enthalten sind Bild und Text in beliebigem Zusammenspiel (siehe hierzu [9, S.152 ff][4]). Die Größe (Höhe und Breite) dieses Bildes orientiert sich meist an einem Grid, dem Grundraster einer Seite. Je größer ein Panel, desto wichtiger ist sein Inhalt für das Erzählte und umgekehrt. Weiterhin entspricht die Länge oder Größe eines Panels der vergehenden Zeit, sodass ein Leser die Zeit eines großen Panels als länger empfindet als die eines kleinen [9, S. 101].
Der so genannte „4th-Wall-Break“ lässt den Inhalt des Panels über dessen Begrenzung (Rahmen) hinaus ragen. Der entstehende Effekt lässt dem Leser den entsprechenden Inhalt aus der Seite entgegen springen und zieht so besondere Aufmerksamkeit auf sich.

3.4.2. Sprechblasen

Sprechblasen sind das Mittel, um Text und Bild miteinander zu vereinen. Man spricht ihretwegen auch von „Sprechblasen-Comics“. Sie sind die Haupttechnik, um Bild und Sprache zusammen zu bringen. Im Optimalfall ergänzt der Text in den Sprechblasen das Bild und reichert es somit an [9].
Es existieren verschiedene Typen von Sprechblasen, die in Abbildung 3.1 dargestellt sind. Sie beeinflussen, wie der Text ausgesprochen wird. Normal, in Gedanken, geschrien oder geflüstert. Darüber hinaus wirkt sich auch die Schriftart auf diesen Eindruck aus [6] . Über einen so genannten Dorn (en. „tail“) wird die Sprechblase einem Sprecher zugeordnet. Der Dorn weist hierbei auf den Mund des Sprechers. Neben Text können auch Symbole in Sprechblasen enthalten sein (siehe hierzu Kapitel 3.4.5).

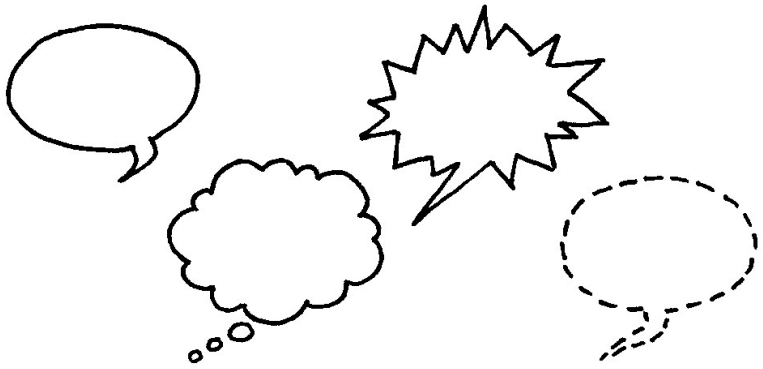


Abbildung 3.1.: Verschiedene Typen von Sprechblasen

3.4.3. Charaktere

Abseits von ihrer Entwicklung und dem für eine Erzählung relevanten Entwurf des Wesens sind Charaktere in Comics meist vereinfacht dargestellt. Dies ermöglicht eine leichtere Identifikation mit demselben, je realistischer, desto fremdartiger [3, S. 31 ff]. Charaktere haben den Vorteil, zusätzlich zum Text mit Mimik und Gestik Informationen an den Leser weitergeben zu können, ebenso wie ein menschliches Gegenüber. Sogar meist noch wirkungsvoller, da Emotionen und Gesten in Comics nach Möglichkeit überzeichnet werden.

3.4.4. Umgebung

Die Welt, in der die Charaktere leben, trägt maßgeblich zur Atmosphäre des Comics bei. Der Hintergrund (im Panel) oder Objekte liefern zusätzliche Informationen oder schaffen Stimmung. Weiterhin kann die Wahl von Lichtverhältnissen die Szene beeinflussen. Durch einen dunklen Hintergrund kann Bedrücken dargestellt werden, ein heller Hintergrund Fröhlichkeit zum Ausdruck bringen. Das Hinzufügen oder Weglassen von Details kann von Charakteren ablenken, oder stärker auf sie hinweisen. Zusätzliche Stilelemente wie Speedlines oder Blitze können Aktionen innerhalb eines Panels anzeigen und verstärken.

3.4.5. Symbole

Unabhängig von Sprache und Lautwörten in den Comics haben sich Symbole zur abstrakten Darstellung bestimmter Umstände etabliert. Einige von ihnen gelten sogar weltweit [4]. So stehen Herzen für Verliebtheit, Dollarzeichen in den Augen für Geldgier und über einen Haufen Müll müssen lediglich einige Fliegen gesetzt werden, um ihn stinken zu lassen. Solche Symbole können eingesetzt werden, wenn eine Mimik oder Geste zu schwierig zu deuten wäre, oder eine sprachliche Beschreibung zu ausschweifend würde. Sie können anstatt von Text in Sprechblasen, oder direkt neben die Gesichter der Charaktere gesetzt werden, um den Gesichtsausdruck zu unterstreichen.

4. DSL

Das folgende Kapitel definiert die domänenspezifische Sprache (DSL) und ihre Eigenschaften, die alle Ausprägungen gemein haben. Zu beachtende Design-Pattern werden vorgestellt und Anforderungen erarbeitet, die an eine Comic-DSL gestellt werden. Im Rahmen dieser Arbeit ist die DSL das ausgewählte Werkzeug, um einen Comic zu programmieren. Sie sollte in Struktur und Syntax soweit möglich an einen realen Comic angelehnt sein.

4.1. Definition

Eine DSL ist eine spezialisierte, explizit entworfene Sprache, welche in Kombination mit umwandelnden Funktionalitäten den Abstraktionsgrad einer Software erhöhen und die Softwareentwicklung erleichtern kann [23, S.1]. Dabei fokussiert sie auf einen bestimmten Problembereich, die Domäne [24, S.16]. Diese Sprachen können verschiedenen Mustern folgen, baumartig oder strukturlos sein, mit grafischer oder textueller Notation, im deklarativen oder imperativen Stil [23].
Das Fachgebiet, auf welches sich die DSL dieser Arbeit spezialisiert ist der Comic. Benutzer sollen Künstler, Programmierer, Verfasser von Dokumentationen und Comic-Fans sein können.

4.1.1. Aufbau

Jede DSL verfügt über drei Grundelemente.

- **Schema**: abstrakte Syntax, Beschreibung der Elemente einer Sprache.
- **Editor**: Quellsprache, in welcher der Benutzer Code verfasst.
- **Generator**: erstellt Code in der Zielsprache.

Der Editor definiert die Syntax der Quellsprache. Aus der Quellsprache wird, meist mit Hilfe eines Interpreters, die abstrakte Syntax extrahiert. Aus der abstrakten Syntax - nicht aus dem Quell-Code - kompiliert ein Generator den Code in die Zielsprache. Veranschaulicht ist dies in Abbildung 4.1.

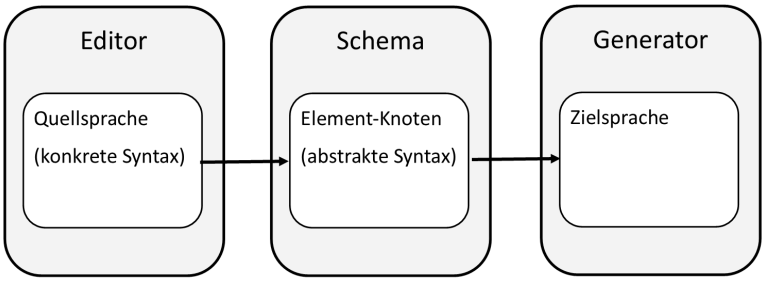


Abbildung 4.1.: Veranschaulichung der Code-Verarbeitung einer DSL

4.1.2. Intern vs. extern

Man unterscheidet zwischen internen und externen DSL. Externe DSL („free-standing“) sind unabhängig von bisher existierenden Sprachen. Es wird also eine neue (Programmier-) Sprache entworfen, mit eigener abstrakter und konkreter Syntax. Die Sprache wird nach Bedarf mit Hilfe eines Generators in die gewünschte Zielsprache umgewandelt. Interne DSL („embedded“) werden im Rahmen einer Host-Sprache implementiert. Hierbei wird die Syntax einer bereits existierenden Sprache genutzt und ggf. erweitert. Ein Anwendungsbeispiel hierfür ist *Scala*, welche in Java integriert ist [25].
Die Comic-DSL ist dieser Definition nach eine externe DSL, da eine eigene Syntax erarbeitet und die abstrakte Syntax in Java umgewandelt wird. Entsprechend wird sich der Herausforderung gestellt, eine beschreibende Sprache für Comics zu entwickeln.

4.1.3. Syntax

Bei dem Design einer neuen DSL muss sowohl die konkrete, als auch die abstrakte Syntax entworfen werden. Die konkrete Syntax beschreibt hierbei den Code, den der Benutzer schreibt. Also unter Einhaltung der Regeln für die DSL erstellte Beschreibungen. Die abstrakte Syntax beschreibt das unter der konkreten Syntax liegende logische Gebilde [26]. Im Falle der Comic-DSL ist dies die baumartige Struktur des abstrakten Comics.
Ein Generator wandelt die abstrakte Syntax in Quellcode um, für die Umwandlung von konkreter in abstrakte Syntax ist zumindest bei textuellen Language Workbenches (TLWB) ein Parser zuständig.

4.2. Design-Pattern

Für den Entwurf einer Programmiersprache gibt es eine Auflistung sinnvoller Design-Pattern von Pane und Myers. Die Richtlinien sollen hauptsächlich den Einstieg für Programmierneulinge erleichtern [27, S.5]. Da als Zielgruppe auch solche angesprochen werden sollen, die noch keine oder wenig Erfahrung mit Programmierung haben, und die Pattern auch für jede neue Sprache gelten sollten, dienen sie als Vorlage für den Entwurf der Comic-DSL.

Im Folgenden sind die Haupt-Pattern kurz vorgestellt und zusammengefasst, inwiefern eine Sprache nach ihnen entworfen werden sollte.

- **Sichtbarkeit des System-Status**: Der Benutzer sollte dem System immer ansehen, was gerade vor sich geht. Zusammenhängendes sollte nebeneinander angezeigt und nicht ausgelagert werden. Generierung und Test müssen unverzüglich Rückmeldung über Erfolg oder Misserfolg geben.
- **Übereinstimmung des Systems und der echten Welt**: Die Sprache sollte nicht so weit abstrahieren, dass der Bezug zur Realität verloren geht. Das Beschriebene sollte weitestgehend aus der Syntax erkennbar sein.
- **Kontrolle und Freiheit des Anwenders**: „undo“ und „redo“ sollten unterstützt werden. Kleine Änderungen im Programm sollten ohne viel Aufwand möglich sein.
- **Konsistenz und Standards**: Befehle und Schlüsselwörter sollten eindeutig zu passenden Ereignissen führen. Kein Parameter sollte Redundanz erzeugen oder in anderem Kontext seine Bedeutung ändern.
- **Wiedererkennung statt Erinnerung**: Benötigte Informationen sollten ohne großen Nachschlage-Aufwand zur Verfügung stehen.
- **Ästhetik und minimalistisches Design**: Dialoge sollten keine überflüssigen oder nur selten nützliche Informationen enthalten, um den Benutzer nicht zu verwirren.
- **Anwender unterstützen bei der Wiedererkennung, Diagnose und Entdeckung von Fehlern**: Fehlermeldungen sollen dem Benutzer in Klartext (nicht Code) vermitteln, warum oder wo ein Fehler aufgetreten ist, um die Ursache möglichst zielsicher beheben zu können.
- **Hilfe und Dokumentation**: Auch wenn ein System eigentlich ohne zusätzliche Dokumentation auskommen sollte, ist es ratsam dem Benutzer eine solche zur Verfügung zu

stellen. Informationen sollten leicht gesucht werden können und Anleitungen übersichtlich gestaltet sein.

In welchem Umfang und in welcher Form diese Design-Pattern im Rahmen der Comic-DSL umgesetzt wurde, ist unter Abschnitt 5.1 (DSL Entwurf) einzusehen.

4.3. Anforderungen

Jede DSL sollte den Jargon der Domäne, für welche sie verfasst ist, nutzen. Dies umfasst typische Vokabeln, aber auch bereits vorhandene Strukturen, die soweit möglich in die DSL übertragen werden sollten. Ziel ist eine möglichst schnelle Eingewöhnung für einen Nutzer, der mit der Domäne vertraut ist. Weiterhin sollte eine neu verfasste DSL die vorgestellten Design-Pattern einhalten und in diesem Falle im Rahmen dieser Arbeit zumindest prototypisch umsetzbar sein.

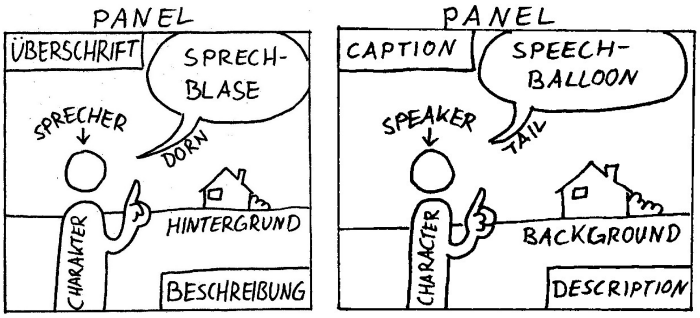
4.3.1. Vokabular

Für die Umsetzung einer Comic-DSL sollte möglichst der Comic-Jargon verwendet werden. Entsprechendes Vokabular wird schon in der CBML (siehe Kapitel 2.2) erschlossen [8] .
Ein Comic-Buch ist in Seiten („pages“) unterteilt. Eine Seite hält in einem Gitter („grid“) mehrere Panels. Diese sind Kernelement des Comics, sie enthalten Bilder und Text. Der Text kann in Form von Sprechblasen („(speech) balloons“) oder Lautwörtern („sound“), sowie über vom Bild abgegrenzte Kästen, die Überschrift („caption", oben links) oder Beschreibung („description“, unten rechts) eingebunden werden. Die Benennung der Elemente eines Panels werden in Abbildung 4.2 noch einmal übersichtlich dargestellt.

Um dem Comic weitere Meta-Informationen mitzugeben, wurde sich an Parametern von möglichen Export-Formaten orientiert (besonders am ePub-Format, siehe hierzu Kapitel 5.7.5). So besitzt ein Comic auch einen Autor („creator“), eine Sprache („language“) und weitere Informationen, die in der Comic-DSL aufgegriffen werden.

4.3.2. IDE

Um in der DSL-Syntax programmieren zu können ist eine passende IDE hilfreich. Sie soll den Benutzer bei der Eingabe, Syntaxprüfung und Fehlerbehebung unterstützen und ist deshalb einem gewöhnlichen Texteditor vorzuziehen. Es gibt die Möglichkeit einer eigenständigen IDE,



(a) Deutsche Vokabeln (b) Englische Vokabeln

Abbildung 4.2.: Vokabel-Beschriftung eines Panels

welche ausschließlich der Code-Erstellung der DSL dient, oder Plugins für größere, verbreitetere IDEs (wie IntelliJ oder Eclipse), um diese auch als Editor für die DSL nutzen zu können.

5. Konzeption

Das folgende Kapitel führt die technologieunabhängigen Ideen dieser Arbeit auf. Dies sind abstrakte oder rein logische Überlegungen und Konzepte, deren konkrete Umsetzung im Rahmen der Masterarbeit stattgefunden hat.
Ein Entwurf der Komponenten, der Umgang mit dem 3D-Raum sowie unabhängige Konstrukte, die als Grundlagen dienen, werden hier aufgezeigt. Ihre Umsetzung kann mit verschiedenen, geeigneten Werkzeugen geschehen. Die Implementierung wird, soweit noch nicht von diesem Kapitel abgedeckt, in den späteren Kapiteln zur Comic-DSL (6) und dem Comic-Generator (7) thematisiert.

5.1. DSL Entwurf

Der Entwurf einer geeigneten Comic-DSL beinhaltet den Entwurf sowohl der abstrakten, als auch der konkreten Syntax. Es gilt die logische Struktur eines Comics zu beschreiben, wie auch das Aussehen des Quellcodes, in welchem der Anwender den Comic beschreibt.

Für die abstrakte Syntax der DSL wurde eine Baumstruktur gewählt, welche die Hierarchien innerhalb eines Comics geeignet darstellt. Wurzelknoten ist der Comic, dieser hält als Kindknoten beliebig viele Seiten, die wiederum Panels enthalten. Jedes Panel kann entsprechende Panel-Elemente beinhalten, so wie Charaktere, Sprechblasen, Hintergründe und weitere Elemente. Erweitert werden die Knoten des Baumes durch passende Attribute, welche z.B. die Breite eines Panels oder die Geste eines Charakters beschreiben. Der konkrete, abstrakte Syntax Baum (AST) wird noch einmal in Kapitel 6.3.1 vorgestellt.

Die konkrete Syntax der Comic-DSL soll zum einen die unterliegende Hierarchie des AST widerspiegeln und zum anderen so weit möglich die unter 4.2 vorgestellten Design-Pattern umsetzen, um vor allem Nicht-Programmierern einen einfachen Einstieg zu bieten. Bei dem Grundentwurf wurde sich an der Auszeichnungssprache HTML orientiert. Dies hat den Vorteil, dass der Künstler als potentieller Anwender häufig schon Berührung mit HTML hatte und umgekehrt nach dem Erlernen der Comic-DSL einen einfacheren Einstieg in HTML hat.

Ähnlichkeit besteht ebenso zur CBML (2.2), beide Referenzsprachen zeigen die Hierarchie des Comics optimal auf.

5.1.1. Design-Pattern

Die in 4.2 vorgestellten Design-Pattern wurden beim Entwurf der Comic-DSL berücksichtigt. Die Umsetzung konkreter Pattern fand wie im Folgenden beschrieben statt:

- **Sichtbarkeit des System-Status**
  Der wichtigste Zustand der DSL bezieht sich darauf, ob der eingegebene Code valide ist und einen Comic generieren kann. Dank des projektionalen Editors wird korrekter Code daran erkannt, dass er keine Fehler wirft. Bei Ausführung des Codes wird höchstens noch auf invalide Pfade zu Comic-Speicherort oder Modellen hingewiesen. Valider Code führt immer zur vollständigen Generierung eines Comics.

- **Übereinstimmung des Systems und der echten Welt**
  Damit der DSL-Code den beschriebenen Comic anschaulich repräsentiert, wird zum einen Comic-Jargon verwendet und zum anderen die Comic-Hierarchie berücksichtigt. Wie bei bereits bestehenden Arbeiten [10, 8], sind einzelne Elemente logisch geschachtelt, sodass ihre Struktur am Code zu erkennen ist. So enthält eine Seite mehrere Panels und diese wiederum weitere Elemente wie Sprechblasen und Charaktere.

- **Kontrolle und Freiheit des Users**
  Der projektionale Editor von MPS sorgt dafür, dass Eigenschaften der Knoten, wie z.B. gesprochener Text oder die Schriftgröße, einfach geändert werden können. Die JetBrains-Umgebung sorgt außerdem für einfaches Editieren, auch mit „undo“ und „redo“ Funktionalitäten, sowie das Duplizieren von DSL-Code.

- **Konsistenz und Standards**
  Die Autovervollständigung in MPS, insbesondere das Einfügen neuer Knoten in den AST, ist an Schlüsselwörter gebunden. Diese sind einmalig und können nur unterhalb bestimmter Eltern-Knoten auftreten. So können Sprechblasen nur innerhalb eines Panels eingefügt werden. Eigenschaften wie die Variabilität können an mehreren Stellen auftreten, haben dann aber immer die gleiche Bedeutung.

- **Wiedererkennung statt Erinnerung**
  Die Autovervollständigung ermöglicht dem Benutzer per Tastenkürzel alle Möglichkeiten angezeigt zu bekommen. So kann er für die Panel-Breite zwischen den verschiedenen, möglichen Werten auswählen oder die Inhalte eines Panels einsehen.

- **Ästhetik und minimalistisches Design**
  Der projektionale Editor verhindert Dialoge, die sonst von einem Parser ausgegeben werden müssten. Fehler im Code werden in JetBrains-Manier angezeigt. Weitere auftretende Fehler, wie ungültige Referenz-Pfade, werden dem Benutzer in einem entsprechenden Dialog angezeigt.

- **Anwender unterstützen bei der Wiedererkennung, Diagnose und Entdeckung von Fehlern**
  Wiederum beschränken sich mögliche Fehler durch MPS auf ein Minimum. Meldungen weisen auf ungültige oder nicht ausgefüllte Eigenschaften hin, oder aber auf ungültige Pfade.

- **Hilfe und Dokumentation**
  Als Hilfe zum anfänglichen Einstieg wurde eine Anleitung erstellt (mit Hilfe der Comic-DSL). Diese ist in Anhang C zu finden. Weiterführende und detailliertere Dokumentation soll folgen. Bei einer Veröffentlichung als Open Source Projekt wird die Doku im zugehörigen Wiki platziert.

Bei der Erweiterung der DSL gilt es, diese Pattern weiterhin zu berücksichtigen.

5.1.2. HTML als Vorlage

Beim Entwurf der Comic-DSL wurde sich an der Auszeichnungssprache HTML orientiert.
Die XML-typischen Tags und Einrückungen sorgen für strukturierten Code und lassen schnell die Hierarchie erkennen. Selbstschließende Tags sind für Elemente ohne weiter unterteilten Inhalt vorgesehen (z.B. Charaktere in Panels). In den öffnenden Tags aufgeführte Parameter ermöglichen genaueres Modifizieren, an einem für alle Elemente gleichermaßen gültigen Platz.

5.2. Architektur

Das System Comic-Generator ist zweigeteilt. Zum einen existiert die in MPS umgesetzte DSL mit Struktur, Editor und Generator, zum Anderen ein Java-System, welches unter Nutzung des LibGDX-Frameworks Panels aus dem Input der DSL generiert und selbst in Komponenten zum Szenenaufbau (SceneBilder) und Rendern im Comic-Stil (PanelRenderer), sowie vielen zusätzlichen Komponenten aufgeteilt ist. Die Aufteilung der Komponenten ist dargestellt in Abbildung 5.1.
Während bei der DSL die Architektur durch MPS vorgegeben wird, ist diese beim Java-System lediglich an die LibGDX-Technologie angelehnt und um eine Schnittstelle zur DSL erweitert.

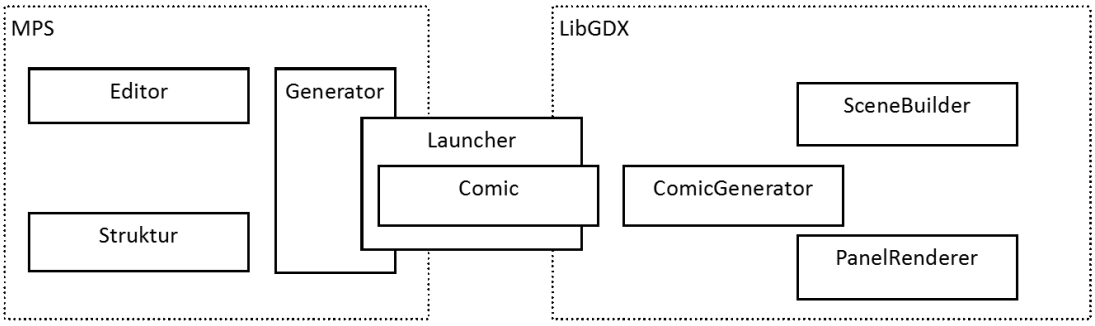


Abbildung 5.1.: Sinngemäße Architektur des Systems

5.2.1. Game-Loop

Das Framework LibGDX (siehe Kapitel 7.1.3 für eine detailliertere Vorstellung) wird häufig zur Spielprogrammierung genutzt und funktioniert entsprechend mit einer Game-Loop. Nach Ablauf einer konfigurierbaren Zeitspanne werden die Update- und die Render-Methode des Systems aufgerufen. In einem Spiel wird mit dem Update auf Eingaben des Spielers oder Events in der Umgebung reagiert und der Zustand entsprechend angepasst. Diese Änderung wird im Render dem Spieler mit einem neuen Bild angezeigt. Paradebeispiel ist ein Charakter, der sich durch seine Umwelt bewegt. Für jede Fortbewegung wird er an einer anderen Stelle dargestellt und ggf. die Kamera angepasst.

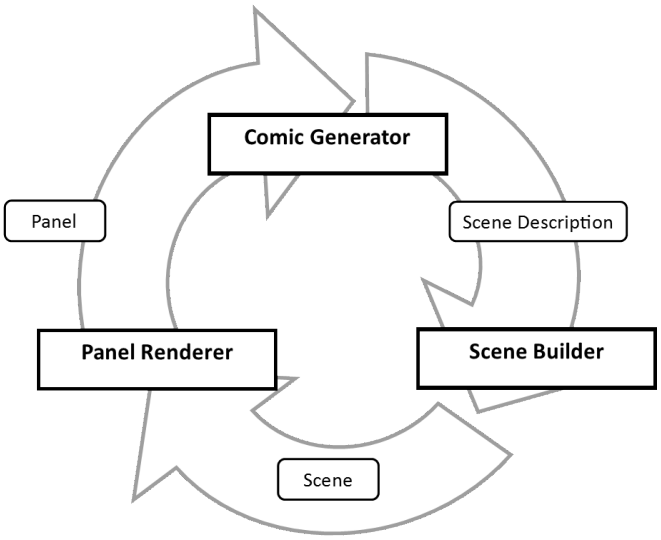


Abbildung 5.2.: „Game-Loop“ des Java ComicGenerators

Für das vorliegende System ist eine Game-Loop mit 60 fps nicht nötig bzw. nicht erwünscht. Für jedes Panel soll genau ein Bild generiert werden. Die Spiele-Game-Loop wird hierfür zur Hilfe genommen. Pro Panel wird einmal ein Update und ein Render durchgeführt. Veranschaulicht ist dies in Abbildung 5.2.

5.3. Determinismus

Eine wichtige Entscheidung fiel über den Determinismus des Comic-Generators. Zum einen ist eine Variabilität der Panels erwünscht, um den Leser nicht zu langweilen. So soll der gleiche Charakter mit einer Sprechblase nicht jedes Mal das gleiche Panel ergeben. Zum anderen ist ein gewisser Determinismus wünschenswert, um nicht bei jedem Durchlauf einen komplett anderen Comic zu erhalten. Beispielsweise soll das Ändern eines Textes in einer Sprechblase nicht zur Umordnung des Panelinhaltes führen.
Weiterhin soll aus einem Comic-AST, der die abstrakte Repräsentation des Comics darstellt, immer der gleiche Comic entstehen. Dies ermöglicht die Weitergabe des AST, aus welchem mit Hilfe des Comic-Generators der Comic generiert werden kann, ohne dass der komplette Comic verschickt werden muss.

Um dennoch eine gewisse Abwechslung zu erhalten, wurden zufällige Variabilitätswerte in den AST eingebaut. Beim Anlegen eines neuen Knotens (z.B. ein neues Panel) wird dieser Wert generiert und als Attribut des Knotens gespeichert. Wird nun aus dem AST der Comic generiert, so werden hierbei diese Werte mit heran gezogen.
Variabilitätswerte werden bisher für die Gesten der Charaktere, die Auswahl des Hintergrundausschnittes und die Positionierung der Kamera verwendet. Der übergebene Wert sorgt hier für leichte Variation in der Ausführung, wie es in Abbildung 5.3 veranschaulicht ist.

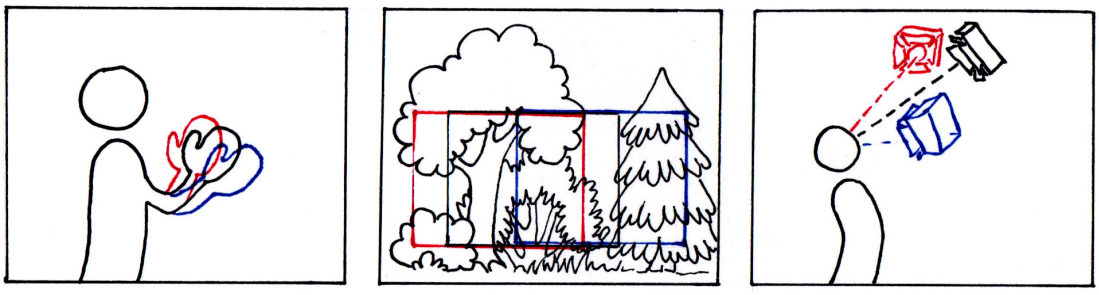


Abbildung 5.3.: Variabilität der Szene: Geste, Hintergrund und Kameraposition

5.4. 3D-Szene

Die Arbeit im 3D-Raum ist der Kern-Aspekt, der diese Arbeit von den meisten anderen Comic-Ansätzen, wie z.B. dem Comic Chat 2.1 unterscheidet. Der Umgang ist komplexer, als das Austauschen von 2D-Grafiken, bietet dafür das Mehr an Variabilität und Spannung für den Leser. Im Rahmen dieser Arbeit wird nicht weiter auf Grundlagen der Computergrafik bezüglich dreidimensionaler Repräsentationen eingegangen. Hierfür sei auf „Real-Time Rendering“ von Akenine-Möller [28] verwiesen. Innerhalb des 3D-Raumes müssen Charakter-Modelle platziert, posiert und von der Kamera aufgenommen werden. Die verschiedenen, zugehörigen Aspekte werden im Folgenden vorgestellt.

5.4.1. Charakter-Gestik

Innerhalb der SceneBuilder-Komponente gibt es eine ModelPoser-Komponente. Diese nimmt Beschreibungen der Szene und speziell der Charaktere entgegen und setzt die gewünschten Gesten um. Implementiert wurde eine Sammlung von Gesten, die in der Beschreibung verwendet werden können. Eine Erweiterung ist durch die Implementierung der Geste im ModelPoser und das Hinzufügen eines weiteren Schlüsselwortes möglich.

Comic-Gesten sollten grundsätzlich übertriebener dargestellt werden, um die Geste zu ver-

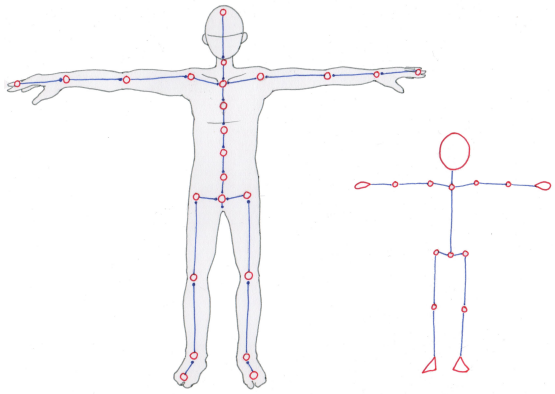


Abbildung 5.4.: Abstraktion des Charakters zu einem Drahtmännchen

deutlichen. Für den Entwurf der Gesten wird der abstrakte Charakter zu einem so genannten Drahtmännchen abstrahiert, wie in Abbildung 5.4 gezeigt. Mit Hilfe dieses Drahtmännchens werden die Gesten entworfen, wobei deutlich die zu verändernden Knochen sichtbar sind. Eine Auswahl der mit Drahtmännchen modellierten Gesten wird in Abbildung 5.5 dargestellt.

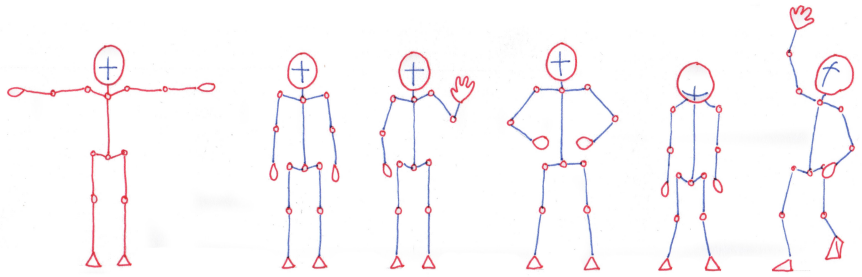


Abbildung 5.5.: Gestensammlung, Darstellung durch Drahtmännchen: ohne Geste (*none*), *neutral*, *wink*, *confident*, *shy*, *high wink*

Um die Charaktermodelle in die durch die Geste definierte Form zu bringen, muss ausgehend von einem unveränderten Skelett (in Abbildung 5.5 ganz links, *none*) jeder abweichende Knochen angepasst werden. Hierfür werden Quaternionen genutzt, welche die Drehung im Raum beschreiben. Vier Werte geben die Drehungen um die drei Achsen und um den Bone selbst an. So werden die Gesten Bone für Bone beschrieben und auf das Charakter-Modell angewendet. Für eine Einleitung in die Quaternionen sei an dieser Stelle auf „Freie Rotation im Raum: Quaternionen und Matrizen“ [29] als grundlegende Literatur verwiesen. Um für Variabilität zu sorgen - wie unter 5.3 beschrieben - werden für manche Bestandteile der Gesten die zuvor berechneten Zufallswerte mit einbezogen. So wird beispielsweise der Winkel des Unterarms bei einer winkenden Geste entsprechend dieses Wertes variiert.

5.4.2. Charakter-Mimik

Die Mimik eines Charakters ist mindestens so wichtig wie seine Gestik. Vor allem, wenn bei einer Nahaufnahme lediglich das Gesicht zu sehen ist. Als Basis-Mimik können die von McCloud vorgestellten Emotionen Zorn, Abscheu, Angst, Freude, Verzweiflung und Überraschung verwendet werden [3, S. 83]. Eine sinnvolle Erweiterung wäre die Kombination dieser Gesichtsausdrücke mit verschiedener Intensität, wie es ebenfalls von McCloud gezeigt wird [3, S. 84f].
Aus technischen Gründen wird die Charakter-Mimik im Rahmen dieser Arbeit nicht umgesetzt. Das verwendete Grafik-Framework unterstützt die entsprechenden Funktionalitäten noch nicht. Die im Modellierungswerkzeug für Mimik zuständigen Shape-Keys werden weiterhin nicht in das durch LibGDX lesbare Format exportiert. Hierzu besteht bereits ein Ticket[1], nach dessen Lösung eine Implementierung von Mimik möglich wäre.

[1] LibGDX-Issue 2633: Support for morph targets/blend shapes in 3D models (open 9.12.2014, closed due to no further activity 2.2.2015)

5.4.3. Kamera-Positionierung

Für die Ausrichtung der Kamera auf die fertige Szene ist in der konkrete Implementierung eine eigene Komponente (ViewFinder) zuständig. Die Kamera wird auf ein Modell ausgerichtet und kann hierbei unterschiedlich große Ausschnitte oder Perspektiven nutzen. Unabhängig davon soll immer das Gesicht eines Charakters zu sehen und Platz für Sprechblasen vorhanden sein. Für die gewünschte Position der Kamera sind je nach gewolltem Endergebnis andere Werte von Nöten. Soll der abzubildende Charakter zentriert im Panel erscheinen, so sind immer die Höhe (y-Position) der Kamera sowie deren Entfernung zum Charakter (z-Distanz) wichtig. Einige Positionen der Kamera seien im Folgenden exemplarisch vorgestellt. Passend dazu zeigt Abbildung 5.6 die zugehörigen, mathematischen Skizzen.

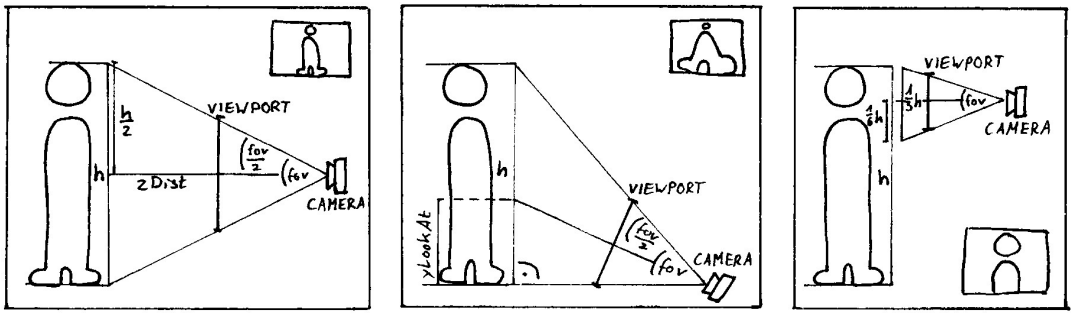
Die Totale stellt den Charakter in seiner Gänze dar. Er ist von Kopf bis Fuß zu sehen. Die Kamera wird auf halber Höhe positioniert, wobei die Höhe des Charakters bei Bedarf um Platz für Sprechblasen erweitert wird. Gegeben ist weiterhin das *Field of View* (fov). Berechnet wird die z-Distanz, um abhängig von Höhe und fov den Charakter vollständig im Panel anzuzeigen. Der fehlende Wert kann mit Hilfe von Trigonometrie ermittelt werden (siehe hierzu auch Abbildung 5.6(a)).

Die Froschperspektive wird erreicht, indem die Kamera auf Bodenhöhe platziert wird und zum Charakter hinauf sieht. Um den Charakter (und Platz für Sprechblasen) zur Gänze aufzunehmen, muss die y-Position ermittelt werden, auf welche die Kamera schaut. Dieser Wert ist abhängig vom fov und der z-Distanz zum Charakter. Abbildung 5.6(b) zeigt den Aufbau.

Das Portrait wird erhalten, indem nur ein Teilausschnitt des Charakters gezeigt wird. Für den Comic-Generator wurde sich hier für das oberste Drittel (zuzüglich Platz für Sprechblasen) entschieden. Die Berechnung der z-Distanz entspricht der, wie bei der Totalen (mit anderem Höhen-Wert) und die y-Position entspricht der Mitte des darzustellenden Abschnitts (hier $\frac{5}{6}$ von der Gesamthöhe des Charakters). Dargestellt ist dies in Abbildung 5.6(c).

Determinismus der Kamera

Um auch bei gleicher Perspektive Variabilität zu ermöglichen wird - wie in 5.3 bereits erwähnt - der entsprechende Wert genutzt, um die Kamera um das Modell herum zu variieren. So können die beiden Porträt-Ansichten, sowie Zwischenschritte ermöglicht werden. Die Verschiebung anhand des Variabilitätswertes wird in Abbildung 5.7 veranschaulicht. Die zugehörige Neuberechnung der Kameraposition ist konzeptionell in Abbildung 5.8 dargestellt. Hierbei wird die



(a) Totale (b) Froschperspektive (c) Portrait

Abbildung 5.6.: Positionierung der Kamera (Abstand und Höhe), zur Erreichung unterschiedlicher Zoom-Level und Perspektiven

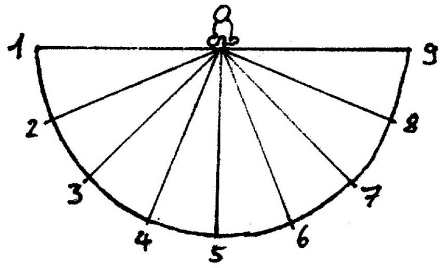


Abbildung 5.7.: Variabilität der Kamera

Kamera nur auf der x-z-Ebene verschoben, und zwar wie folgt:
$\alpha = -90 + (v - 1) \cdot 22.5$
$b = \cos(\alpha) \cdot a$
$\Delta z = a - b$
$\Delta x = \sin(\alpha) \cdot signum(\alpha)$
Der Wert v der Variabilität aus dem Intervall $[1 - 9]$. Bei $v = 10$ findet keine Verschiebung der Kamera statt.

Drehung bei mehreren Charakteren

Bei der Anzeige mehrerer Charaktere in einem Panel wird die Bounding-Box, an welcher sich die Kamera zur Positionierung orientiert, um die jeweiligen Charaktere ergänzt, sodass gewährleistet ist, dass alle Charaktere im Bild sind. Weiterhin können die einzelnen Charaktere über ihre eigenen Variabilitätswerte gedreht werden, um sie so zueinander oder voneinander weg zu drehen. Die DSL sorgt automatisiert dafür, dass zwei Charaktere in einem Panel (leicht

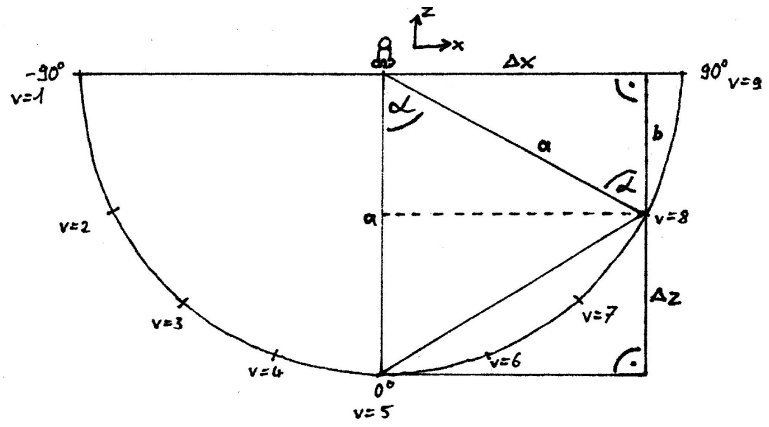


Abbildung 5.8.: Rechenskizze zur Verschiebung der Kamera

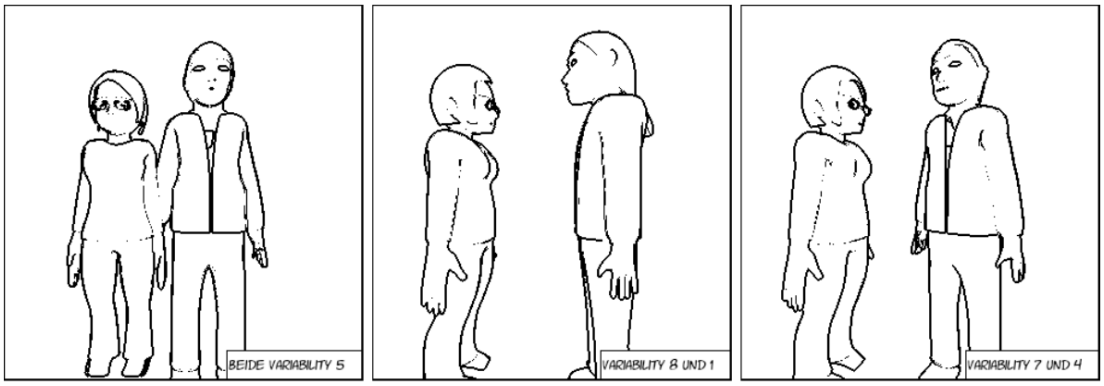


Abbildung 5.9.: Zwei Charaktere in einem Panel

variabel) zueinander gedreht werden. Diese Drehung erfolgt anhand der gleichen Logik wie in Abbildung 5.7 abgebildet. Beispielhafte Anwendung für verschiedene Kombinationen von Variabilitätswerten sind in Abbildung 5.9 aufgezeigt. Die generelle Berechnung der Höhe bleibt auch bei mehreren Charakteren gleich.

5.5. Cartoon-Shader

Die erstellte 3D-Szene wird mit einem Cartoon-Shader gerendert, um den typischen Comic-Stil zu erhalten. Dieser Shader entspricht dem aus der Arbeit von Decaudin [1]. Die Kanten des Objektes werden erkannt und schwarz dargestellt. Ein ähnliches Ergebnis entsteht bei Greif und Gürbüz [13], hier haben die entstehenden Bilder einen skizzenhaften Charakter, indem verschieden dunkle Grautöne genutzt werden.

Ausgangspunkt des Shadings ist eine fertige 3D-Szene. Die Modelle sind angepasst und in einer Umwelt positioniert und die Kamera hat eine Position erhalten, in welcher sie die Szene einfängt. Eine Übersicht über den im Folgenden vorgestellten Algorithmus wird in Abbildung 5.10 gegeben.

5.5.1. Tiefen-Map (äußere Kanten)

Um die Outlines (äußeren Kanten) nachzuziehen, wird die mit einem Kantendetektor ermittelte Silhouette benötigt. Im ersten Schritt wird hierfür eine Tiefen-Map erstellt. Auf dieser enthält jeder Vertex seinen Abstand zur Kamera (Vertices des Modells mit geringerem Abstand, Vertices des Hintergrundes mit maximalem oder sehr großem Abstand). Im zweiten Schritt werden nebeneinander liegende Fragmente verglichen. Überschreitet die Differenz einen konfigurierbaren Schwellenwerte (bspw. Hintergrund-Fragment und Charakter-Fragment), so wird das entsprechende Pixel schwarz dargestellt.
Auf diese Weise wird das Modell mit einer schwarzen Linie umrandet sowie Linien an überlappenden Teilen des Modells gezogen (z.B. ein vor dem Torso angewinkelter Arm).

5.5.2. Normalen-Map (innere Kanten)

Nicht durch die äußeren Kanten abgedeckt sind die Knickkanten (inneren Kanten). Der Abstand der entsprechenden Vertices zur Kamera ist sehr ähnlich, lediglich die Ausrichtung der zugehörigen Flächen macht das Zeichnen einer Linie nötig, um das Modell ausreichend zu beschreiben. Hierfür wird eine Normalen-Map erstellt, die durch das Setzen von gerichteten Lichtquellen an den Achsen des karthesischen Koordinatensystems erhalten wird. Jede Fläche ist daraufhin entsprechend ihrer Ausrichtung in x-, y- oder z-Richtung eingefärbt. Äquivalent zur Kantenerkennung aus der Tiefen-Map werden nun die Farbwerte der Fragmente verglichen und bei entsprechender Differenz das Pixel schwarz eingefärbt.

5.6. Sprechblasen-Platzierung

Die Möglichkeiten der Sprechblasenplatzierung werden bisher noch nicht ausgeschöpft. Jede Sprechblase erhält einen Anteil der Panel-Breite zugeteilt, entsprechend der Anzahl enthaltener Sprechblasen. Um in diesen Bereich zu passen werden bei Bedarf Zeilenumbrüche in den Text eingefügt. Die Reihenfolge entspricht der, in welcher Eintragungen in die DSL gemacht wurden. Weiterführende Ansätze bieten [7, 30, 6].

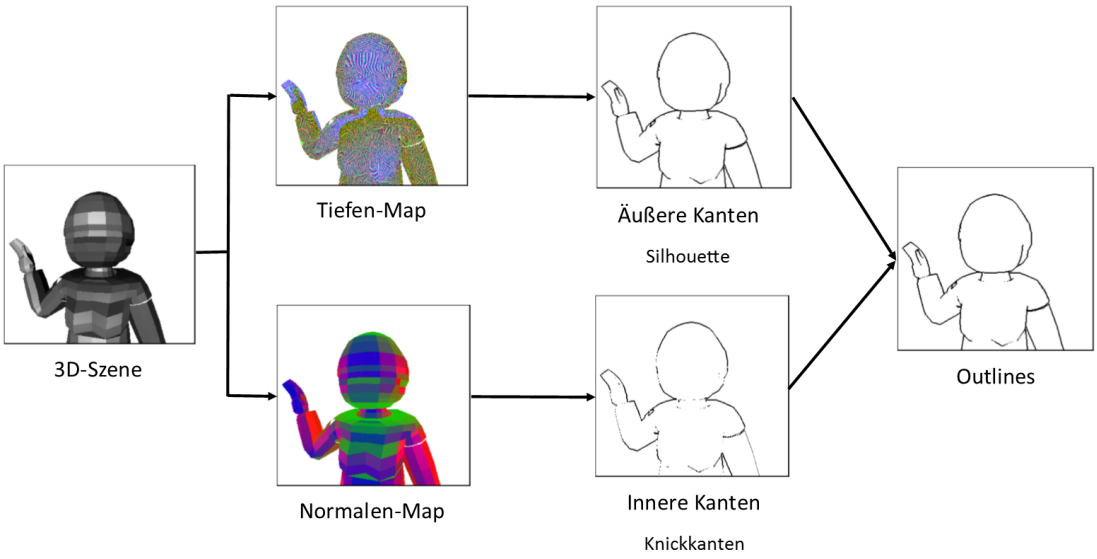


Abbildung 5.10.: Teilschritte des Cartoon-Shading-Algorithmus nach Decaudin [1]

5.7. Comic

Neben den in Kapitel 3.4 aufgeführten Grundbausteinen eines Comics gibt es einige Designentscheidungen, die Auswirkung auf die technische Umsetzung haben. Zum Teil geht es hierbei um die Umsetzung eines typischen, wiedererkennbaren Comic-Stils in computergenerierten Grafiken.

5.7.1. Farbgebung

Farbtechnisch existieren mehrere Ausprägungen von Comics. Bunte Comics, für die neben dem Zeichner auch ein Kolorist nötig ist, schwarz-weiße Comics, vor allem aus dem *Crime Noir* Genre und Zeichnungen, die mit Schattierungen aufgewertet werden. Hier sind vor allem die japanischen Manga zu nennen, die so genannte Rasterfolien verwenden.

Im Rahmen dieser Arbeit wurde sich aus mehreren Gründen für eine Umsetzung ohne Farbe entschieden. Zuerst ist die Colorierung ein weiterer Arbeitsschritt, bei der Textur der 3D-Modelle, aber auch im Bezug auf farbliche Abstimmung von Charakteren, Objekten und Hintergrund ist eine Umsetzung mit Farben deutlich aufwändiger.
Eine schwarz-weiße Darstellung hat, neben der einfacheren Umsetzung, einige Vorteile, die vor allem das Endmedium betreffen. Ein Druck in schwarz-weiß ist günstiger als ein Farbdruck. In Japan werden Comichefte auf unterschiedlich farbigem Papier gedruckt, um einzelne

Geschichten voneinander zu unterscheiden und Abwechslung zu schaffen. Auch das Ausspielen auf Geräten ohne Farbdarstellung ist unproblematisch, hier sei vorrangig der eReader (ePub-Format) genannt.

5.7.2. Stilmittel

Die Kunstform Comic lebt vor allem von den verwendeten Zeichenstilen. Diese sind vom ausführenden Künstler abhängig und können stark variieren. Beim Comic-Generator wird der Stil bestimmt durch die Strichstärke, Hintergrundbilder und besonders die den Charakteren zu Grunde liegenden 3D-Modelle. Die 3D-Modellierung bietet dem Künstler hier die entsprechenden Freiheiten. So kann durch ein abgewandeltes Modell der gewünschte Stil dargestellt werden.

5.7.3. Leserichtung

Die Leserichtung für westliche Comics ist von links nach rechts und von oben nach unten. Dies gilt sowohl für die Seiten, als auch für jedes einzelne Panel. Diese Leserichtung wird dadurch befolgt, dass Sprechblasen von links nach rechts (in Sprechreihenfolge) in das Panel gesetzt, und die Panel auf der Seite mit einem *float: left* (per CSS) ausgerichtet werden.

5.7.4. Charaktere

Die (Haupt-) Charaktere eines Comics sind das Herzstück. Ob der Leser Gefallen an ihnen findet (im ästhetischen Sinne) und sich mit ihnen identifizieren kann, wirkt sich auf den Willen und das Durchhaltevermögen beim Lesen aus. Um die Darstellung der Charaktere möglichst abwechslungsreich und somit für den Leser interessant zu gestalten, wird ein 3D-Ansatz gewählt.
Die im Comic verwendeten Charaktere werden mit einem entsprechenden Werkzeug modelliert und in der DSL referenziert. Ihre Erstellung bleibt der umfangreichste Schritt in der Comic Generierung mit der Comic-DSL. Ist allerdings ein Pool von Charakteren vorhanden, so kann der Benutzer auf diesen zurückgreifen und ist nur durch die vorhandene Auswahl eingeschränkt, sofern er nicht selbst neue Charaktere modelliert und einpflegt.

5.7.5. Export

Endprodukt der Comic-Generierung ist eine Sammlung von Panels in Bildform. Die Aufteilung auf die Comic-Seiten ist durch die Ordnerstruktur gegeben. Um die seitenweise Ansicht des generierten Comics zu ermöglichen, ist eine Einbindung der Panel-Bilder in eine übergeordnete

Struktur nötig.
Später sollen mehrere Export-Formate zur Verfügung gestellt werden, um eine Ansicht auf möglichst vielen Geräten zu ermöglichen. Der wichtigste Export ist der nach HTML, da die Ansicht im Browser für die meisten Geräte möglich ist und das Format als Ausgangspunkt für weitere Exportformate dient. Die Hierarchie des Comics ist, wie schon bei Tobita [10], fester Bestandteil des Dokuments. Die Seiten sind jeweils in einzelnen Containern enthalten ($< div\ class = "page" >$), in welchen die Bilder eingefügt werden.
Sind alle Panels einer Seite generiert, so wird ein einzelnes Page-Bild erzeugt. Dieses vereinfacht den weiteren Export, da es alle Panel zusammen fasst.

HTML Für den Export wird nach Abschluss der Generierung über die Ordnerstruktur iteriert und so ein HTML-Dokument zusammen gebaut. Für Seiten-Ordner wird ein *div*-Tag angelegt, in welches per *img*-Tag die Page-Bilder eingepflegt werden. Weiterhin wird ein im Projekt hinterlegtes CSS-Dokument in den Zielordner kopiert, um die Bilder auf einer Webseite korrekt anzuordnen.

PDF Der Export in eine PDF-Datei findet mit *iText*, Version 2.1.7[2] statt. Die Page-Bilder werden hier jeweils auf einer eigenen Seite angezeigt.

ePub Der ePub-Export setzt eine xhtml-Datei voraus, die dem exportierten HTML-Dokument entpricht. In einer weiteren Datei werden Informationen zum Comic, wie der Autor, Verlag, etc. vermerkt. Eine vorgefertigte Ordnerstruktur wird programmatisch gepackt und die Endung in „.ePub“ umbenannt. Der Comic kann damit auf eReadern und entsprechenden ePub-fähigen Geräten gelesen werden. Damit die Page-Bilder den Bildschirm des eReaders optimal ausnutzen, werden die Page-Divs und die enthaltenen Bilder über die volle Breite ausgespielt.

CBML Der Export in eine XML-Datei nach dem CBML-Format, welches in Kapitel 2.2 vorgestellt wurde, kann nicht wie die anderen Exports nach der Generierung des Comics erfolgen, da die im Panel enthaltenen Informationen wichtig sind. Bei jeder Panel-Generierung muss hier ein Eintrag in die zugehörige Datei erfolgen. Ein $< cbml : panel >$ wird mit der Nummerierung und den Charakteren als Attributen und ggf. Tags für Sprechblasen und Überschriften angelegt, bevor der Panel-Tag wieder geschlossen wird. Eine beispielhafte CBML-Datei ist in Anhang A.4 zu sehen, diese ist zum Use-Case-Comic zugehörig.

[2] itextpdf.com Bis Version 2.1.7 unter der LGPL, danach unter AGPL. Bei kommerzieller Nutzung ist hier der Kauf einer iText-Lizenz angebracht.

6. Comic-DSL

Teil der Masterarbeit ist die Implementierung einer Comic-DSL, einer Sprache, in welcher der Nutzer seinen eigenen Comic beschreiben kann. Diese soll möglichst leicht zu erlernen sein und in ihrer Struktur die gegebene Hierarchie des Comics aufgreifen. Der Input der DSL muss jene Informationen enthalten, die an den Generator (im Kapitel 7) weiter gereicht und zu einem Comic umgeformt werden. Es muss also eine ausreichende Beschreibung der darzustellenden Szenen und Panelinhalte stattfinden.
Hauptanforderung ist, dass valider Code immer zu einem Comic führt. Außerdem soll die gleiche, abstrakte Syntax auch den gleichen Comic hervor bringen (vgl. Kapitel 5.3). Um eine DSL, wie in Kapitel 4 zu beschreiben, ist eine so genannte Language Workbench nötig. Mit ihr werden Struktur, Editor und Generator definiert und die Anforderungen an Aussehen und Funktionalität der DSL umgesetzt.

Dieses Kapitel definiert die Language Workbench, trifft eine Auswahl aus verschiedenen Vertretern, um die Comic-DSL umzusetzen und beschreibt die Komponenten der DSL, sowie deren Schnittstelle zum Comic-Generator.

6.1. Language Workbench

Eine DSL wird mit Hilfe einer Language Workbench (LWB) beschrieben. Es gibt verschiedene Ansätze für die Umsetzung.

6.1.1. Definition

Eine Language Workbench ist ein System zur Erstellung einer DSL. Sie ermöglicht die Spezifizierung der Grundelemente einer DSL, also den logischen Aufbau des Schemas, die Syntax des Editors und nach welchen Regeln der Generator aus den Eingaben Quellcode generieren soll [26].

6.1.2. Alternativen

Es gibt zwei Hauptansätze für Language Workbenches: textuelle und projektionale Language Workbenches. Hauptvertreter für *Textual Language Workbenches* (TLWB) ist Xtext, weitere das Textual Editing Framework (TEF), Textual Conrete Syntax (TCS) und EMFText. Die *Projectional Language Workbenches* (PLWB) haben bisher nur MPS als nennenswerten Vertreter, die Entstehung weiterer PLWBs wird allerdings prognostiziert [31, 32]. Vorrangig muss also eine Entscheidung für TLWB oder PLWB getroffen werden.

6.1.3. Anforderungen

Gegeben durch die Problemstellung einer Comic-DSL und den Rahmen dieser Arbeit gelten die folgenden Anforderungen für die Auswahl einer geeigneten LWB:

- **Zielsprache Java**, da der unterliegende Comic Generator durch die Verwendung von LibGDX in Java verfasst ist.
- **Erweiterbarkeit** wird angestrebt, da im Rahmen der Masterarbeit nicht alle Möglichkeiten ausgeschöpft werden können und viele Ideen für Erweiterungen vorliegen.
- Eine **Einarbeitung** in die LWB soll im Rahmen eines Semesters (Projekt 2, siehe Kapitel 9.2) möglich sein, sodass innerhalb des Semesters eine einsetzbare DSL erhalten wird.

6.1.4. Auswahl

Verglichen werden die beiden Hauptvertreter Xtext und MPS und ihre Tauglichkeit anhand der Anforderungen geprüft.

Xtext ist ein 2006 entstandenes Open-Source-Framework, das seit 2008 im Rahmen des Eclipse Modeling Project weiterentwickelt wird. Mit Hilfe von Xtext[1] können in einer Eclipse-basierten Umgebung DSLs und GPLs erstellt werden. Hierbei sorgt Xtext nicht nur, wie andere Parsergeneratoren, für einen Parser, sondern auch für die folgenden Bestandteile einer Sprache [32, S.23f]:

- **Parser** auf Basis einer Beschreibung der Sprache
- Modellierung des **semantischen Modells**
- Bau des **Abstrakten Syntax Baumes**

[1] Xtext (eclipse) www.eclipse.org/Xtext - Language Engineering For Everyone!

- **Eclipse-Plugin** zur Nutzung einer Eclipse-Entwicklungsumgebung

Vorteile von Xtext sind die einfache Beschreibung der Grammatik, das Eclipse-Plugin, mit dessen Hilfe sich die Sprache leicht nutzen lässt, sowie der Umstand, dass sich die Funktionalitäten von Xtext leicht erweitern lassen. Nachteile sind die umständliche Erweiterung einer bereits erstellten Sprache, die damit verknüpften Abhängigkeiten, sowie nicht zuletzt die Festlegung auf Eclipse [32, S.50f].

MPS Das *Meta Programming System* (MPS) ist der bisher einzige Vertreter von PLWBs. Vorteile von MPS sind die modulare Spracherweiterung und Java als Grundlage für dieselben, sowie der Umstand, dass keine Grammatik für die Spracherweiterung entwickelt werden muss, da der abstrakte Syntaxbaum (AST) direkt bei der Eingabe erzeugt wird. Nachteile sind, dass man durch fehlende textuelle Repräsentationen an MPS als Entwicklungsumgebung gebunden ist, die Versionierung erfolgt in XML-Form der Modelle und als Zielsprache steht bisher nur Java zur Verfügung [32, S.51].

Für eine Abwägung werden die zuvor aufgestellten Anforderungen herangezogen. Java als Zielsprache wird von beiden Frameworks angeboten. Die Erweiterbarkeit der DSL ist bei Xtext in der Form wie bei MPS nicht möglich und gestaltet sich umständlich. Über den Umfang und die Einarbeitungszeit kann vor Beginn der Arbeit keine Aussage getätigt werden.
Eine Entscheidung für Xtext könnte bei einer vorliegenden Eclipse-Präferenz gewählt werden. Dies entfällt, da bevorzugt IntelliJ genutzt wird. Beide Frameworks fordern eine Festlegung bezüglich der Entwicklungsumgebung, sodass dieser Aspekt keinen Unterschied macht. Die Problematik von MPS bei der Versionskontrolle beschränkt sich auf ein für Menschen unleserliches Format, das die konkrete Implementierung nicht aus den versionierten Dateien ersichtlich werden lässt.

6.2. Umsetzung mit MPS

JetBrains *Meta Programming System* (MPS) ist eine Umgebung zur Sprachen-Definition, eine Language Workbench und eine IDE für DSLs. Alle MPS Versionen erlauben das Beschreiben der Sprache, allerdings hatten einige Versionen bekannte Bugs bei der Generierung von build-Solutions (benötigt für die Standalone-IDE), sodass vorrangig mit MPS 3.0 gearbeitet wurde. 2016 wurde das Projekt auf die neueste Version, MPS 3.3.2 migriert.

6.2.1. Aufbau

Die Implementierung der DSL erfolgt in einem so genannten Language-Projekt (im Vergleich dazu das Solutions-Projekt, in Kapitel 6.2.2). Neu angelegt enthält es Order für die verschiedenen Knoten, wichtig sind vor allem jene für Struktur, Editor und Generator. Diese werden in den folgenden Kapiteln vorgestellt.

6.2.2. Solution

Die Solutions in MPS werden auf verschiedene Arten genutzt. Für einen Benutzer am wichtigesten ist die Funktion, Code in der per Language definierten DSL zu verfassen. Hierzu wird die Comic-Language geerbt und ein neuer Knoten angelegt. In diesem kann - nach der im Editor der Language beschriebenen konkreten Syntax - ein neuer Knoten beschrieben werden, im speziellen Fall ein neuer Comic.
Um die Klassen des Comic Generators bekannt zu machen und die Interfaces zu nutzen, wird eine entsprechende Solution benötigt. Diese fügt die jar-Datei des Comic Generators hinzu (siehe auch Kapitel 6.7.2). Die Solution wird nun von der Language genutzt, um auf die Klassen zugreifen zu können. Zuletzt kann eine build-Solution erstellt werden, um eine Standalone IDE zu generieren. Dies ist näher im Kapitel 6.2.4 beschrieben.

6.2.3. Erweiterbarkeit

Die Erweiterbarkeit von MPS zeichnet sich dadurch aus, dass meist nur ein neuer Knoten angelegt und eingebunden werden muss. Sollen beispielsweise Lautwörter eingefügt werden, so muss auf Seiten von MPS nur ein neuer Knoten hierfür angelegt, eingebunden und mit Editor und Generator-Code versehen werden. Detailliertere theoretische Erweiterungsmöglichkeiten werden in Kapitel 8.4 vorgestellt.

6.2.4. Standalone IDE

Mit Hilfe einer build-Solution kann aus MPS eine Standalone-IDE für die entwickelte DSL generiert werden. Es handelt sich hierbei um eine Instanz von MPS, in welche die benötigten Solutions und Languages bereits eingebunden sind. Hier kann durch Anlegen einer neuen Solution mit der Comic-DSL programmiert werden.
Obwohl MPS die build-Solution automatisch generiert, muss bei der Nutzung eines Artefakts dieses von Hand unter Plugins eingetragen werden. Das Ausführen generiert Distributionen für verschiedene Betriebssysteme unter */build/artifacts*. Ein Screenshot der Windows-Anwendung ist in Abbildung 6.1 zu sehen.

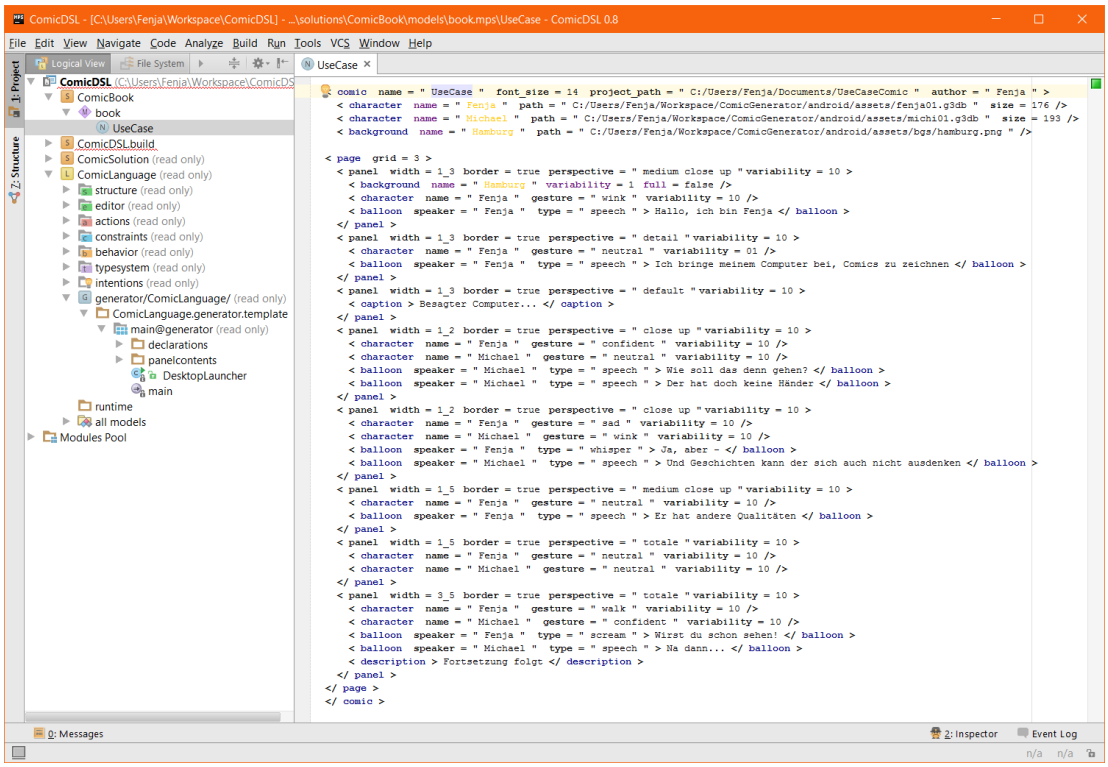


Abbildung 6.1.: ComicDSL 0.8 als Standalone-IDE von MPS

6.3. Struktur

Die Struktur beschreibt die abstrakte Syntax der Sprache. Nicht zu verwechseln mit der konkreten Syntax; diese ist Gegenstand der Beschreibung des Editors. Bei MPS ist die Struktur in Knoten organisiert. Diese besitzen Eltern- und Kindknoten, sowie beschreibende Attribute. Durch die Knoten wird eine baumartige Struktur beschrieben, ein so genannter AST, der das Skelett der Sprache darstellt.

6.3.1. AST

Der AST (*abstrakter Syntaxbaum*, en. „abstrakt syntax tree“) ist die abstrakte Syntax der Sprache. Hier ist beschrieben, welche Knoten wie miteinander verknüpft sind und welche Attribute sie halten. Ein AST beschreibt einen konkreten Comic, der per DSL beschrieben wurde. So sind die Kindknoten eines Comic-Knotens mehrere Page-Knoten, die wiederum auf Panel-Knoten verweisen. Der Comic-Knoten hat Eigenschaften, wie den Namen des Comics, den Autoren und den Dateipfad zum Speicherort. Beispielhaft ist ein solcher AST in Abbildung 6.2 dargestellt.

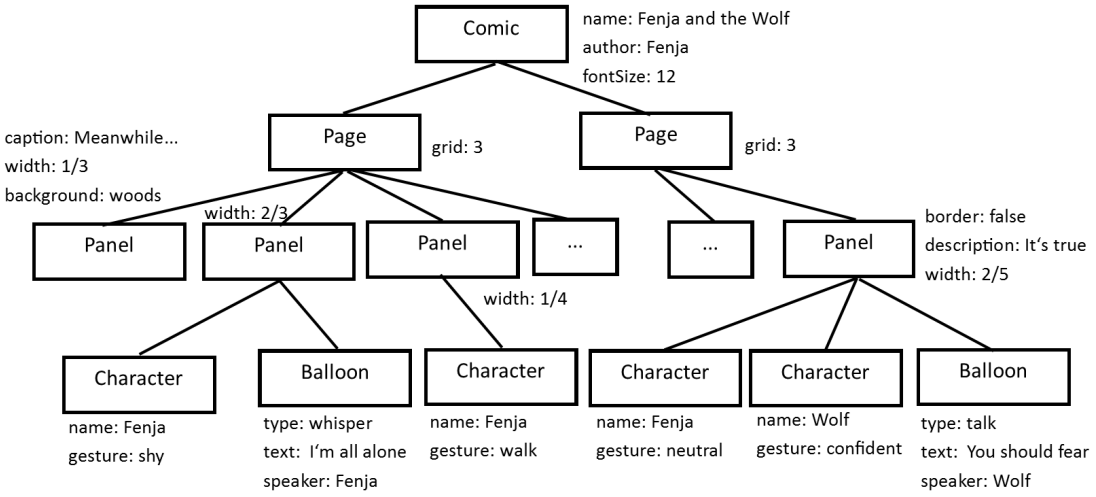


Abbildung 6.2.: Relevanter Ausschnitt aus einem Comic-Baum

MPS definiert die Regeln, nach denen ein AST aufgebaut wird. So ist beschrieben, ob und welche Kindknoten erlaubt sind und welche (Pflicht-) Attribute enthalten sind. Der User beschreibt den AST in konkreter Syntax mit Hilfe eines projektionalen Editors. Dieser erlaubt nur die Eingabe validen Quellcodes, also valide Knoten und Attribute, sodass der unterliegende Baum immer valide ist. Weiterhin wird über den Editor direkt der unterliegende AST verändert, sodass kein Parser nötig ist.

6.3.2. Begriff

Ein Begriff (en. „concept“) ist ein MPS-Konstrukt zur abstrakten Beschreibung eines Knotens. Jeder Knoten erhält einen Namen (z.B. „Page“) und kann bei Bedarf von anderen (abstrakten) Knoten erben. Als Eigenschaften können primitive Datentypen gesetzt werden (z.B. „int: fontSize“), oder auch Kindknoten (z.B. $panels : Panel[0..1]$), wobei angegeben wird, wie vielen Kindknoten erlaubt sind (z.B. $[1]$, $[0..n]$). Die Implementierung eines Knotens füllt die (Pflicht-) Attribute mit konkreten Werten aus.

6.3.3. Implementierung

Die Implementierung eines konkreten Begriffs führt den Namen des Knotens, seine Eigenschaften und Kindknoten auf. Die Implementierung des Knotens „Panel“ ist in Abbildung 6.3 zu sehen. Abbildung 6.4 zeigt die abstrakte Beschreibung eines AST. Ein konkreter AST hält ent-

```
concept Panel extends     BaseConcept
                implements <none>

  instance can be root: false
  alias: panel
  short description: comic canvas

  properties:
  border      : boolean
  width       : PanelWidth
  perspective : Perspective
  variability : integer

  children:
  panelContents : PanelContent[0..n]
```

Abbildung 6.3.: Struktur-Knoten „Panel“ in MPS 3.0

sprechende Knoten und ausgefüllte Attribute, wie in Abbildung 6.2. Die Knoten „Declaration“ und „PanelContent“ sind hierbei abstrakt.

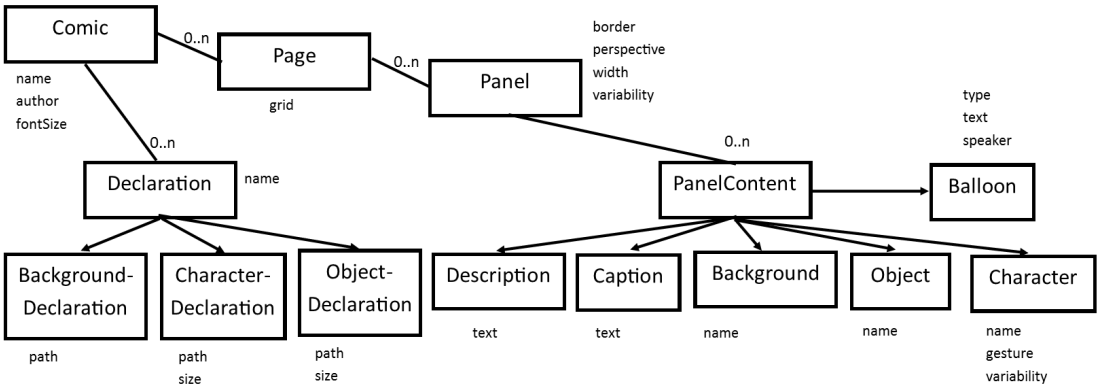


Abbildung 6.4.: AST - Knoten mit Kindknoten und Eigenschaften

6.4. Editor

Der Editor beschreibt die konkrete Syntax, in welcher ein Benutzer die Comic-DSL verfasst. Hier werden Schlüsselwörter, Sonderzeichen, Reihenfolgen und die Einbindung von Attributen und Kindknoten beschrieben. Grundsätzlich ist es möglich, mehrere Editoren zu verfassen, die auf dem gleichen AST arbeiten, sodass eine alternative Comic-DSL (konkrete Syntax) ohne Neuimplementation verfasst werden könnte.

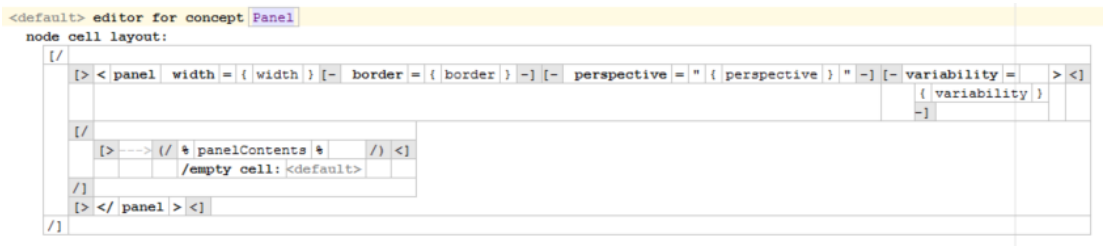


Abbildung 6.5.: Editor-Knoten „Panel“ in MPS 3.0

6.4.1. Editor-Knoten

Zu jedem Begriff-Knoten wird ein Editor-Knoten verfasst. Innerhalb dessen kann auf die Attribute und Kindknoten des Begriffknotens zugegriffen werden. So wird beschrieben, in welcher textuellen Notation Attribute mit Werten belegt werden können. Im Editor können String-Konstanten, die vom Benutzer eingegeben, aber nicht verändert werden können (meist Schlüsselwörter), Attribute (der Benutzer gibt dort den gewünschten Wert ein) und Kindknoten, welche das Einbinden des zugehörigen Editor-Knotens bewirken, beschrieben werden.

6.4.2. Implementierung

Wie in Kapitel 5.1.2 bereits beschrieben, wird sich bei der konkreten Syntax an der von HTML orientiert. Der Name des Knotens wird hier zur Tag-Bezeichnung, die Attribute wie bei HTML üblich in den öffnenden Tag mit ihrer Bezeichnung eingebunden (z.B. $name = "MyComic"$). Kindknoten werden mit ihren eigenen Tags zwischen das öffnende und schließende Tag des Knotens eingefügt. Zur besseren Übersichtlichkeit werden so eingeschlossene Knoten-Tags eingerückt. Die konkrete Implementierung des Editor-Knoten „Panel“ ist in Abbildung 6.5 zu sehen.

6.4.3. Syntax Highlighting

Einzelne String-Konstanten, Schlüsselwörter oder Attribut-Werte können mit den Editor-Regeln eingefärbt werden. In der Beschreibung des jeweiligen Elements kann ein Style gesetzt werden. MPS bietet eine Sammlung eigener Styles an (z.B. KeyWord), die mit einem eigenen, zusätzlichen Stylesheet erweitert wurden.

6.5. Generator

Der Generator ist dafür verantwortlich, den AST in Zielsprache umzuwandeln. Aus der abstrakten Syntax wird hierbei Quellcode in der Zielsprache (im Falle von MPS also Java). Aus einer Hauptklasse heraus kann auf die Knoten zugegriffen werden, um Code zu generieren. Hierbei wird um MPS-Funktionen erweiterte Java-Syntax verwendet. Ebenso wie beim Editor ist es auch hier möglich, mehrere Generatoren für den gleichen AST zu nutzen, um alternativen Code zu generieren.

6.5.1. Generator-Klasse

Der Generator-Knoten ist eine Java-Klasse, die bei Bedarf mit weiteren Klassen kombiniert werden kann, so für Entscheidungsmechanismen (welcher Knoten von mehreren möglichen wird umgesetzt) oder zur Erhöhung der Übersichtlichkeit. Eine „mapping configuration“ listet auf, für welchen Knoten im AST welcher Code eingefügt wird. Am wichtigsten ist hierbei die „root mapping rule“ welche besagt, dass aus dem Comic-Knoten die Hauptklasse generiert wird. Im vorliegenden Falle ist der AST-Wurzelknoten „Comic“ Ausgangspunkt und mit einer Java-Klasse umgesetzt.

Die einfache Implementierung nutzt eine main-Methode, die beim Ausführen des fertig generierten Codes ausgeführt werden kann. Bei variablen Reihenfolgen oder Kindknoten werden „template declarations“ genutzt, um nach Bedarf passende „template fragments“, also Code-Schnipsel einzufügen. Mit mehreren Funktionalitäten können die Knoten zur Code-Generierung genutzt werden.
Ein *Property-Makro* ersetzt einen Platzhalter durch den spezifischen Wert des aktuell betrachteten Knotens. So wird aus „name“ der für den Comic vergebene Name (zu sehen in Abbildung 6.7 in der Zeile $generator.setComicName(\text{"}\$[Comic]\text{"})$.
Mit einem *LOOP-Makro* kann beispielsweise über alle Page-Knoten eines Comic-Knotens iteriert und im vorliegenden Beispiel jeweils eine Methode „createPage“ aufgerufen werden.
Gerade bei der Verwendung von LOOP-Makros ist eine automatische Benennung der generierten Methoden nötig, um die Eindeutigkeit der Namen zu wahren. Hierfür können *Value- und Referenz-Makros* genutzt werden, welche die Methoden und ihre Aufrufe mit entsprechenden ID-Anhängen versehen.
Das *CopySrcNode-Makro* fügt Code-Schnipsel ein, entsprechend dem eingebundenen Knoten. So wird im $createPanel$ über alle Knoten vom Type „PanelContent“ iteriert (mit dem LOOP-Makro), wohinter sich z.B. Charaktere, Balloons, oder Textfelder verbergen können. Je nach

Typ des jeweiligen Knotens wird das zugehörige „template fragment“ eingefügt. Die Zuordnung findet wiederum in der „mapping configuration“ unter „reduction rules“ statt.

Auf weitere Makros zur Code-Generierung wird an dieser Stelle nicht eingegangen, hierfür sei auf die MPS-Dokumentation verwiesen [33].

6.5.2. Zielsprache Java

Input für den Generator ist eine Implementierung des Wurzelknotens innerhalb eines Solution-Projekts. Ist der eingegebene Code valide, so kann per Rechtsklick der generierte Java-Code angezeigt werden („Preview generated Text“). Hieran wird die Arbeitsweise des Generators deutlich. Properties werden ersetzt, aus LOOP-Makros gehen Code-Wiederholungen hervor, die Dank der Value-Makros eindeutig benannt sind. Zu sehen ist das Ergebnis in Abbildung 6.6, zum Vergleich die Generator-Klasse in Abbildung 6.7.

6.5.3. Implementierung

Wie bereits erwähnt wird in der Hauptklasse des Generators für jeden Page-Knoten ein $createPage$ mit eindeutiger ID erzeugt. Übergeben werden beschreibende Parameter, der wichtigste zur Angabe des Grids, worüber die Höhe der einzelnen Panel bestimmt wird. Innerhalb der createPage-Methode wird für jeden Panel-Knoten (der unter dem jeweiligen Page-Knoten hängt) eine createPanel-Methode erstellt. Beide Methoden nutzen die als Stub eingehängte Schnittstelle (siehe hierzu Kapitel 6.7), um Aufrufe an den Comic-Generator zu stellen. Der Generator-Code ist in Abbildung 6.7 zu sehen.

6.6. Weitere MPS-Begriffe

Ergänzend zu Editor, Struktur und Generator gibt es noch weitere MPS-Konstrukte, welche bei der Umsetzung der DSL unterstützend wirken.

6.6.1. Actions

In Actions ist definiert, wie mit im Editor neu erzeugten Knoten umgegangen werden soll. So werden mit einer „Node Factory“ Knoten beim Anlegen mit default-Parametern befüllt. Beispielsweise erhalten alle Panels standardmäßig einen Rahmen (*border*) und alle betroffenen Knoten einen zufällig generierten Variabilitätswert.

```java
package book;

/*Generated by MPS */

import de.haw.ma.ComicGeneratorImpl;
import de.haw.ma.interfaces.ComicGenerator;
import de.haw.ma.descriptions.PanelDescription;
import de.haw.ma.descriptions.CharacterDescription;
import de.haw.ma.descriptions.Gesture;
import de.haw.ma.descriptions.BalloonDescription;
import de.haw.ma.descriptions.BalloonType;

public class map_Comic {


  public static void main(String[] args) {
    ComicGeneratorImpl generator = new ComicGeneratorImpl();
    generator.setComicName("MyComic");
    generator.setPageParametersDin(500);

    boolean validPath = generator.setProjectPath("path/to/comic");
    if (!(validPath)) {
      System.out.println("invalid project path");
      generator.exit();
    }

    generator.loadCharacter("Fenja", "path/to/model/Fenja.g3db", 1.80);

    createPage_a(generator);

    generator.exportToHTML();
  }



  private static void createPage_a(ComicGenerator generator) {
    generator.generatePage(3);
    createPanel_a0a(generator);
  }



  private static void createPanel_a0a(ComicGenerator generator) {
    PanelDescription panel = new PanelDescription();
    panel.setVariability(0);

    addCharacterToPanel_a0a0(panel);

    addBalloonToPanel_a0a0(panel);


    generator.generatePanel(true, 1, panel);
  }
```

Abbildung 6.6.: Vorschau des aus der DSL generierten Java-Codes

6.6.2. Constraints

Constraints werden als zusätzliche Regeln angelegt und schränken die Eigenschaften eines einzelnen Knoten ein. So wird beispielsweise gewährleistet, dass ein Knoten mit Hintergrund auch einen Rahmen (*border*) besitzt. Weiterhin können Typ und Anzahl der Kindknoten, sowie weitere Attribute validiert werden.

6.7. Schnittstelle

Die eigentliche Generierung des Comics findet erst durch den Comic-Generator (Kapitel 7) statt, anhand der Beschreibungen, die durch die Comic-DSL geliefert werden. Diese Zusammenarbeit wird umgesetzt, indem die Comic-DSL die Schnittstelle des Comic-Generators nutzt und gezielt mit Parametern anspricht. Damit die generierenden Methoden auf diesem Wege aufgerufen werden können, muss das Java-Projekt dem MPS-Projekt bekannt gemacht werden. Dies geschieht durch das Einbinden eines Artefaktes und das Setzen von Abhängigkeiten.

6.7.1. Abhängigkeiten

Jeder Knoten in MPS kann Abhängigkeiten haben und Sprachen erben. Abweichend von einem gewöhnlichen MPS-Projekt ist hierbei das Einbinden des Comic-Generators als Artefakt nötig. Im folgenden Kapitel wird die Stubs-Solution beschrieben. Darüber hinaus muss der Generator-Knoten diese als Abhängigkeit exportieren, sowie der Knoten „main@generator“ die für den Generator-Code benötigten Sub-Packages des Artefakts einfügen. Eine Übersicht über die relevanten Abhängigkeiten wird in Anhang B gegeben.

6.7.2. Stubs

Die Stubs-Solution *ComicSolution* wird von der Comic-Language geerbt und enthält mit dem Comic-Generator-Artefakt alle benötigten Modelle, um die Schnittstelle des Comic-Generators ansprechen zu können. In den Abhängigkeiten der ComicSolution ist das Artefakt als Quelle für Java-Klassen eingetragen, wodurch die enthaltenen Klassen dem Pool an Modellen im Knoten hinzugefügt werden.

6.7.3. DesktopLauncher

Um LibGDX zur Comic-Generierung nutzen zu können, muss ein Launcher mit entsprechender Konfiguration eine weitere Klasse aufrufen, in welcher erst ein „create“ und dann in konfigurierbaren Intervallen ein „render“ aufgerufen wird. Es bestand die Schwierigkeit, sowohl

den Launcher, als auch die aufzurufende Hauptklasse in MPS zu integrieren. Erstere wird zur Ausführung benötigt und die Hauptklasse, um die Methoden des Comic-Generators aufzurufen. Gelöst wurde das Ganze mit *Java Inner Classes*. So wird nur eine Klasse generiert und der Launcher kann seine innere Klasse mit der benötigten Konfiguration aufrufen.
Der zugehörige Code ist in Abbildung 6.7 aufgezeigt. Das hiermit gelöste Problem ist bisher einzigartig, da MPS noch keine allzu weite Verbreitung findet, schon gar nicht in Kombination mit LibGDX.

6.8. Qualitätssicherung

Die Qualitätssicherung der Comic-DSL kann auf zwei Ebenen stattfinden. Programmatisch wird getestet, ob der AST richtig aufgebaut und Regeln eingehalten werden. Weiterhin kann die Usability im Umgang mit der DSL und MPS überprüft werden.

6.8.1. MPS-Test

Im Rahmen einer Solution, die im vorliegenden Fall TestSolution genannt wurde, können Test-Module angelegt werden. Weiterhin kann unterhalb der Language ein Test-Aspekt angelegt werden. Die Test-Module sind hierbei die flexiblere Lösung für umfangreiche Testcodes. In beiden Bereichen können wie gewohnt Test-Klassen, aber auch gezielt Tests für Knoten und Editor verfasst werden.
In diesen Tests kann die Einhaltung implementierter Beschränkungen oder die automatische Generierung überprüft werden.

6.8.2. Usability Lab

Test zur Benutzbarkeit der Comic-DSL konnten aus zeitlichen Gründen nicht im Rahmen dieser Arbeit durchgeführt werden. Es ist jedoch empfehlenswert, solche zur Verbesserung und Weiterentwicklung der DSL durchzuführen. Testpersonen mit und ohne Erfahrungen in den Bereichen Comic und HTML sollen einen Comic erstellen können. Aspekte hierbei sind die Aussagekraft des DSL-Codes, sowie der Umgang mit dem projektionalen Editor. Weiterhin können solche Tests dazu genutzt werden um herauszufinden, welche Erweiterungen am ehesten von Anwendern gewünscht werden.

```
root template
input Comic

public class DesktopLauncher {
  public static void main(string[] args) {
    LwjglApplicationConfiguration config = new LwjglApplicationConfiguration();
    new LwjglApplication(new Comic(), config);
  }

  public static class Comic extends ApplicationAdapter {
    public Comic() {
      <no statements>
    }
    /*package*/ ComicGenerator generator;

    public void create() {
      generator = new ComicGeneratorImpl();
      generator.setComicName("$[Comic]");
      generator.setFontSize($[12]);
      generator.setPageParametersDin(500);

      boolean validPath = generator.setProjectPath("$[path/to/dir]");
      if (!validPath) {
        System.out.println("invalid project path");
        generator.exit();
      }
      $LOOP$[$COPY_SRC$[System.out.println("declarations"); ]]
    }

    private void characterLoad(String name, String path, int size) {
      if (!generator.loadCharacter(name, path, size)) {
        System.out.println("invalid path " + path + " for character " + name);
        generator.exit();
      }
    }
```

Abbildung 6.7.: Allgemeine Beschreibung der Klasse DesktopLauncher in MPS. Gezeigt ist der Ausschnitt zum Laden von Charakteren.

6.9. Zusammenfassung

Mit der PLWB MPS von Jetbrains wurde eine Comic-DSL erstellt. Diese ermöglicht mit HTML-ähnlicher, konkreter Syntax die Beschreibung eines Comics. Hierzu wird der AST in Java-Code umgewandelt und ruft den im folgenden Kapitel vorgestellten Comic-Generator auf. MPS benötigte einige Einarbeitungszeit und besonders die Schnittstelle zum Java-Projekt war eine, noch nicht zuvor behandelte, Herausforderung.

6.9.1. Erfahrungen mit MPS

Die größte Besonderheit von MPS ist der projektionale Editor. Dieser sorgt für eine längere Eingewöhnungszeit, was in der späteren Nutzung für eine Einstiegshürde sorgen könnte. Allerdings überwiegen die Vorteile mit einer höheren Entwicklungsgeschwindigkeit und einer kaum benötigten Fehlerbehandlung.
Wie zuvor bei der Abwägung für eine LWB festgestellt ist die Versionskontrolle unübersichtlich. Ein Sichten der Dateien auf GitHub ist so gut wie unmöglich; das Projekt kann erst nach dem Klonen und Aufrufen in MPS in Augenschein genommen werden.
Gerade im Bezug auf Abhängigkeiten ist MPS verwirrend und es ist empfehlenswert zu notieren, welcher Knoten was benutzen, implementieren oder exportieren muss, um ein lauffähiges Projekt zu erhalten. Auf Nachfragen in Online-Foren gab es allerdings immer schnelle Antworten von Seiten der MPS-Entwickler, sodass die Erfahrungen mit MPS unterm Strich positiv sind.

6.9.2. Schwierigkeiten

Schwierigkeiten im Umgang mit MPS als LWB bestanden vorrangig mit den projektinternen Abhängigkeiten. Jeder Knoten kann Sprachen und Module einbinden, erben oder exportieren; und um korrekte Funktionalität zu gewährleisten, müssen diese Abhängigkeiten korrekt aufgelöst werden. In der MPS-Dokumentation [33] existiert hierfür ein eigenes Kapitel, trotzdem wurde hierfür bei den Entwicklern nachgefragt und besagtes Kapitel dahingehend sogar ergänzt. Entsprechend sind die vorliegenden Abhängigkeiten im Anhang B aufgeführt.

Im Bezug auf die Funktionalität, eine Standalone-IDE zu exportieren, gab es Probleme mit einigen der MPS-Versionen, sodass hauptsächlich mit *MPS 3.0* gearbeitet wurde. Die Versionen 3.1 und 3.2 waren jeweils mit Bugs belastet, welche den build-Prozess verhinderten. Version 3.3 wurde erst zum Ende der Arbeit hin veröffentlicht.

Zuletzt gab es noch einige Schwierigkeiten bei der korrekten Einbindung des Java-Comic-Generators in MPS, um die Generierung mit den erwünschten Parametern anzustoßen. Hierfür wurde ein neuer Lösungsansatz entwickelt, der in Kapitel 6.7.3 näher beschrieben wird.

6.9.3. Ausblick

Bei der Implementierung neuer Funktionalitäten für den Comic-Generator muss die Comic-DSL entsprechend nachziehen. Automatisierte Parameter sollten vorborgen werden und erst auf Wunsch des Benutzers hin angezeigt werden, wie etwa die Variabilität oder Perspektive im Panel. MPS bietet für die DSL noch einiges mehr an Möglichkeiten, die ausgeschöpft werden können. So wären besonders Hinweise für den Benutzer wünschenswert, die mit Zeichentipps gleichzusetzen sind. Dies ist denkbar im Bezug auf Abwechslung in der Darstellung, Textmenge oder weitere Stilmittel.
Ebenso schön wäre ein grafischer Editor, in dem ein Comic nicht programmiert, sondern zusammen geklickt werden kann. Dies ist ebenfalls mit MPS umsetzbar, da lediglich der Editor betroffen ist, AST und Generator jedoch gleich bleiben.

7. Comic-Generator

Kernelement dieser Arbeit ist ein Java-System, das für die Generierung der Comic-Panel zuständig ist. Jeder Methodenaufruf an die Schnittstelle generiert ein Panel, entsprechend der übergebenen Parameter.
Die im Panel dargestellte Szene basiert auf 3D-Modellen, anders als in anderen Ansätzen, wo mit einer Sammlung an 2D-Grafiken gearbeitet wird. Der dreidimensionale Ansatz wurde gewählt, um mehr Möglichkeiten für Perspektive und Gesten zu erhalten. Charaktere werden in die Szene gesetzt, posiert, ausgerichtet und die Kamera entsprechen platziert. Der oder die Charaktere sollen im View-Port sichtbar sein, nach Möglichkeit auch der Mund. Ebenfalls ist Platz für die Platzierung der Sprechblasen einzuplanen, von denen aus auf den jeweiligen Sprecher gewiesen wird. Sprechblasen und weitere Text-Kästen sorgen für die Anzeige von Text im Comic und werden ebenfalls per Parameter übergeben.
In diesem Kapitel werden die genutzten Tools vorgestellt, sowie komponentenweise der Java-Comic-Generator.

7.1. Grafik-Framework

Durch die Szenen-Modellierung im 3D-Raum musste ein entsprechendes Werkzeug gewählt werden, mit dem 3D-Modelle platziert und eine Kamera positioniert werden kann. Möglichst viel sollte hierbei programmierbar sein, von der Kameraposition bis zu den Gelenken im Skelett der Charaktere. Weiterhin besteht der Bedarf, mit Hilfe eines Shaders die Szene in Comic-Stil umzuwandeln.

7.1.1. Alternativen

Ein Java-Framework, das für Spiele und andere Anwendungen genutzt wird, ist LibGDX[1]. three.js[2] nutzt ebenso wie LibGDX OpenGL und basiert auf JavaScript.

[1] libGDX libgdx.badlogicgames.com - Desktop/Android/BlackBerry/iOS/HTML5 Java game development framework
[2] three.js threejs.org - Javascript 3D library

7.1.2. Anforderungen

Den meisten Frameworks gemein ist, dass sie OpenGL[3] oder eine Abwandlung davon nutzen. Es ist eine Spezifikation für eine plattform- und sprachenunabhängige Programmierschnittstelle zur Entwicklung von 2D- und 3D-Computergrafikanwendungen. Nötig sind die Erstellung und Modifikation einer dreidimensionalen Szene, der Zugriff auf die Knochen des Modells, um Gesten darzustellen, sowie die Anwendbarkeit eines Comic-Shaders.

7.1.3. Auswahl

LibGDX wurde auf Empfehlung von Kommilitonen ausgewählt. Es nutzt Java, was für eine geringere Einstiegshürde sorgt. Weiterhin existieren umfangreiche Anleitungen für die Arbeit mit Shadern und das Modifizieren von Knochen in 3D-Modellen, womit die gestellten Anforderungen an den Funktionsumfang bereits abgedeckt sind.

7.1.4. LibGDX

LibGDX ist ein Java-Framework für plattformunabhängige Spielentwicklung. Es wurde von Mario Zechner geschaffen und Open Source auf GitHub zur Verfügung gestellt. Die Struktur trennt das Kernprojekt in den Ordner „core“ und erstellt für jede Plattform ein Unterprojekt, wie z.B. „desktop“, „android“ oder „html“, in der Starter-Klassen eine neue Instanz des Kernprojektes erstellen und ausführen. Somit wird auch plattformspezifischer Code ermöglicht. Eine für diese Arbeit relevante Bibliothek in LibGDX ist OpenGL.

7.2. Umsetzung mit LibGDX

Zur Umsetzung des Comic-Generators mit LibGDX gilt es, einige Besonderheiten und Standards zu beachten. Das verwendete Koordinatensystem weicht von dem, im 3D-Modellierungstool genutzten ab und die Modelle müssen im g3db-Format geladen werden. Der benötigte Shader kann als Shaderprogramm eingebunden werden.

7.2.1. Game-Loop

Die bereits in Kapitel 5.2.1 vorgestellte Game-Loop entspricht der Standard-Architektur von LibGDX. Normalerweise ruft ein gestarteter Launcher abwechselnd „update“ und „render“ einer Methode auf und aktualisiert somit das angezeigte Bild. Im Rahmen des Comic-Generators ist

[3] OpenGL (Khronos Group) Open Graphics Library www.opengl.org

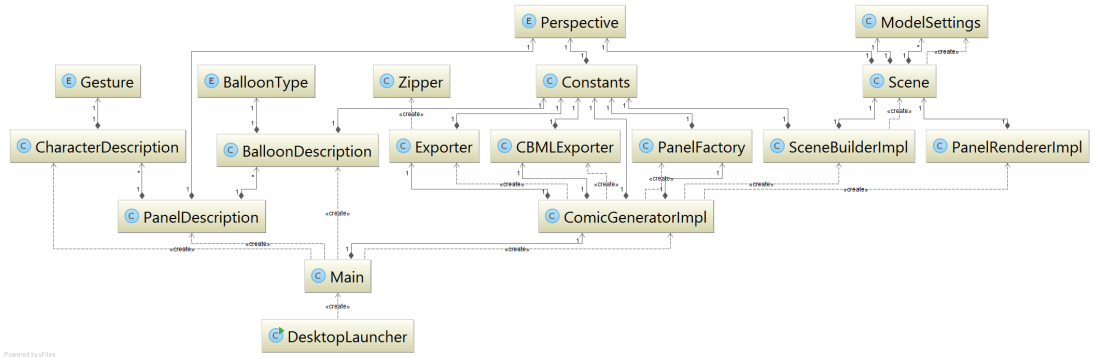


Abbildung 7.1.: Klassendiagramm des Systems „Comic Generator“, erstellt mit IntelliJ IDEA 15.0.4. Darstellung ohne Interfaces und statische Klassen.

lediglich ein solcher Durchlauf pro Panel erwünscht, sonst bleibt die Struktur dieser Schleife gleich.

7.2.2. Architektur

Die Architektur des Comic-Generators ist an die anderer LibGDX-Projekte angelehnt. Meistens findet man hier Spiele, die als Beispielimplementation gezeigt werden. Unterteilt wird grob in Hauptkomponenten (ComicGenerator, SceneBuilder, PanelRenderer), *Assets*, zum Laden von Modellen, Bildern und Schriften, *Launcher* für verschiedene Plattformen (Desktop, Android, ...), welche die Plattformunabhängigkeit ermöglichen. Hinzu kommen weitere Hilfskomponenten, wie z.B. ViewFinder und ModelPoser, Modelle zur Beschreibung von Szenen, Panels und Charakteren. Einen Überblick über die Architektur bietet Abbildung 7.1.
Was normale Spiele-Projekte nicht benötigen, in diesem Projekt zur Erhöhung der Erweiterbarkeit aber bedacht ist, sind Interfaces, um die Hauptkomponenten einfacher austauschen zu können. Nicht zuletzt muss eine Schnittstelle zur Comic-DSL geschaffen werden. Diese ist in gewöhnlichen LibGDX-Projekten nicht vorgesehen und bedarf einigen zusätzlichen Überlegungen.

7.2.3. Artefakt

Um den Comic-Generator in MPS und somit für die DSL nutzbar zu machen, wird ein Artefakt erzeugt. Ein jar-Verzeichnis wird mit dem gesamten Projekt (inklusive Desktop-Ordner) und zugehörigen Bibliotheken erstellt. Wichtig ist außerdem das Hinzufügen des Android-Assets-Ordners, damit dessen Inhalt korrekt angesprochen wird. Eine Main-Klasse zur Ausführung

besteht in diesem Falle nicht, da der Aufruf durch eine weitere, in MPS generierte Klasse stattfindet.

7.3. 3D-Modellierung

Basis für die Comiczeichnungen dieser Arbeit sind 3D-Charaktere. Alternative Ansätze nutzen einen 2D-Ansatz. Dieser ist jedoch deutlich aufwändiger, da sämtliche genutzte Charakter-Grafiken vorgezeichnet werden müssen und die Darstellungsvariationen auf die Kombinationsmöglichkeiten beschränkt sind. Die Einführung eines neuen Charakters würde somit die Erstellung einer großen Anzahl an neuen Grafiken erfordern.
Der in dieser Arbeit verfolgte Ansatz basiert auf 3D-Modellen und deren Manipulation, um Gesten und Mimik darzustellen. Die Anpassung des Charakters geschieht mit Hilfe des Grafik-Frameworks in Java. Ein neuer Charakter kann hier einfach durch das Hinzufügen eines neuen 3D-Modells erstellt werden.

7.3.1. Definition

Während sich so genannte 3D-Modellierungswerkzeuge hauptsächlich um die dreidimensionale, virtuelle Generierung von Geometrie kümmert, ermöglicht es 3D-Grafik-Software zusätzlich mit Hilfe von Computergrafik dreidimensionale Szenen zu virtualisieren und/oder zu rendern. Für die vorliegende Arbeit genügt grundsätzlich ein Modellierungswerkzeug, da die Virtualisierung und das Rendern der Szene vom Grafikframework übernommen werden. Allerdings muss auch ein Skelett vorhanden sein, welches das modellierte Mesh beeinflusst, was wiederum für ein 3D-Grafikprogramm spricht.

7.3.2. Alternativen

Die verbreitetsten 3D-Grafikprogramme sind Blender[4], Maya[5], Cinema 4D[6], oder auch LightWave 3D[7]. Jedes dieser Programme benötigt aufgrund der Komplexität, die 3D-Modellierung mit sich bring, eine längere Einarbeitungszeit, sodass lediglich eines ausgewählt wird.

[4] Blender (Blender Foundation) www.blender.org
[5] Maya (Autodesk) www.autodesk.de/products/maya
[6] Cinema 4D (Maxon) www.maxon.net/de/products/cinema-4d-studio
[7] LightWave (NewTek) www.lightave3d.com

7.3.3. Anforderungen

Die Bedienung des Modellierungstools sollte ihm Rahmen der Masterarbeit zu erlernen sein und die möglichst einfache Erstellung von 3D-Charakteren ermöglichen. Einem 3D-Modellierungswerkzeug entsprechend muss ein Mesh modelliert und modifiziert werden können; ein Glätten des Meshes ist wünschenswert, um nach Anwendung des Comic-Shaders ein glattes Ergebnis zu erhalten. Dem 3D-Mesh muss ein Skelett hinzugefügt werden können, dessen einzelne Bones das Mesh beeinflussen.
Weiterhin muss ein Export in ein von LibGDX lesbares Format möglich sein. Aus dem Framework heraus muss auf das Modell und besonders die Bones zugegriffen werden können, um die Gestik zu ermöglichen. Dies geht im Optimalfall mit dem entsprechenden Dateiformat einher.

7.3.4. Auswahl

Die Auswahl des 3D-Grafikprogramms fiel auf Blender. Vorrangig auf Empfehlung eines Kommilitonen und weil die Übertragung der Modelle über den Export nach *fbx* gut dokumentiert ist und somit ein schnelles Vorankommen ermöglicht wird [34]. Die Kompatibilität mit alternativer Software soll aber in Zukunft noch ermöglicht werden.

7.3.5. Blender

Blender ist ein Open Source 3D-Grafikprogramm, das vielfach Verwendung findet. Es ist für iOS, Windows, Linux und weitere Betriebssysteme verfügbar und enthält eine umfangreiche Werkzeugsammlung für Modellierung, Animation, Malen, Texturieren, Rendern und Videoeditierung. Weiterhin ist eine Python-Engine enthalten [34, Blender For Programmers - Introduction],[35, S. x ff].

7.4. 3D-Modelle

Die dreidimensional modellierten Charaktere sind Kernaspekt des generierten Comics. Sie sollen über Bildsprache Informationen vermitteln, die über den in Sprechblasen und Boxen enthaltenen Text hinausgehen, bzw. diese ergänzen.

7.4.1. Modellierung

Erster Schritt bei der Erzeugung der benötigten Comic-Charaktere ist die Modellierung eines 3D-Meshes. Als Hilfestellung wurden hierfür erst Skizzen der Charaktere angefertigt und als Hintergrundbild in die Blender-Oberfläche eingebunden, um diese als Vorlage nutzen zu

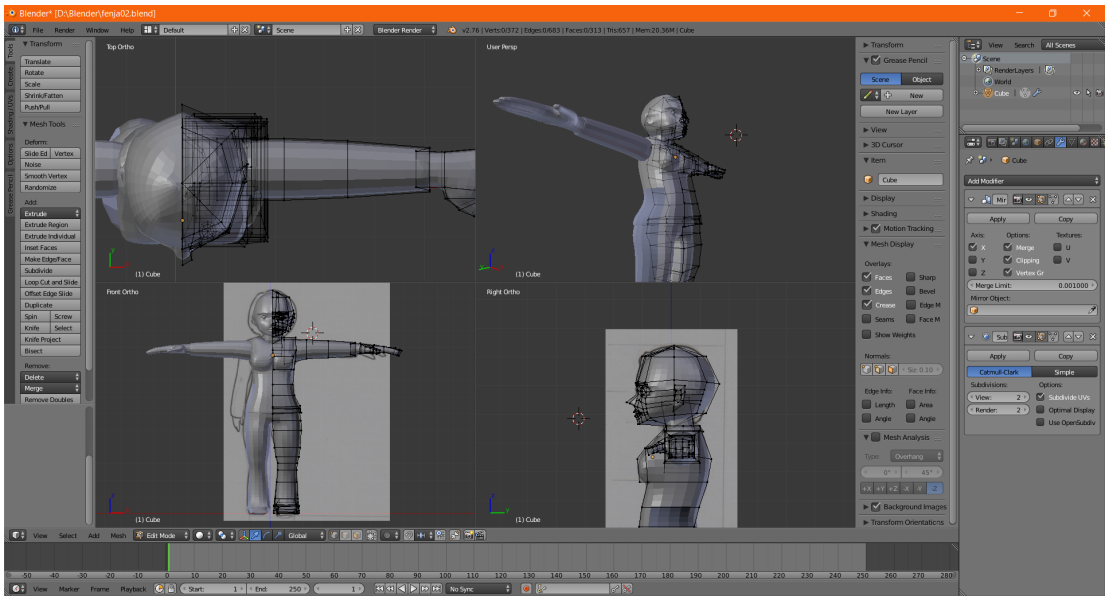


Abbildung 7.2.: Modellierung eines 3D-Charakters in Blender 2.76

können. Dieses Vorgehen ist in Abbildung 7.4.1 und auch im Blender-Buch [36] beschrieben und hat zum Ziel, dass die später gerenderten Modelle den Comic-Entwürfen so ähnlich wie möglich sind.

7.4.2. Skelett

Dem fertig modellierten Mesh wird ein Skelett (im Blender-Vokabular „armature“) eingesetzt. Ebenso wie das Mesh werden Knochen („Bones“) hinzugefügt und erweitert. Das Skelett für sich gesehen entspricht einer Drahtpuppe, wie sie in Kapitel 5.4.1 vorgestellt wurden. Bei der Erstellung einzelner Knochen ist eine korrekte Benamung derselben nötig, um sie später gezielt ansprechen zu können.
An der Anatomie orientiert gibt es die Bones „spine1“ bis „spine5“, „neck“, „head“, sowie für die Gliedmaßen mit für rechts und links entsprechenden Namen, z.b. „upper_leg.l“ für den linken Oberschenkel-Knochen. Das Mesh mit Skelett und benannten Knochen ist in Abbildung 7.3 dargestellt.
Das fertige Skelett muss mit dem Mesh verbunden werden, damit eine Manipulation der einzelnen Knochen auch Auswirkungen auf das Mesh hat. Die entsprechende Funktionalität bietet Blender. In einem entsprechenden Modus kann die Verformung des Meshes durch das Skelett überprüft werden.

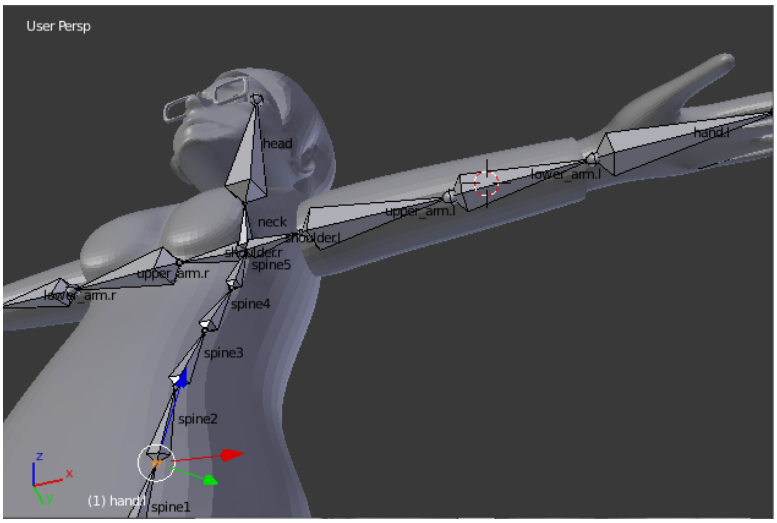


Abbildung 7.3.: Modell in Blender mit angezeigten Bones-Benamungen

7.4.3. Verformung

Bei dem Verbinden von Skelett und Mesh werden automatische Gewichtungen der Knochen gesetzt, um den Einfluss des Meshes durch das Skelett zu definieren. Durch fehlerhafte Modellierung des Meshes oder besonders im Bereich der Gelenke, kann die durch das Skelett hervorgerufene Verformung fehlerhaft sein. Hier kann in Blender nachgearbeitet werden, indem in einem eigenen Modus die Beeinflussung einzelner Knochen auf das Mesh angepasst wird.

7.4.4. Export

Zur Nutzung des erstellten 3D-Modells im Comic-Generator muss es in einem geeigneten Dateiformat vorliegen. In diesem Falle muss es von LibGDX verwertet werden können. LibGDX nutzt obj- und g3db-Dateien.
Ein direkter Export der Blender-Datei in eines dieser Formate ist mit der aktuellen Version (2.7) nicht möglich. Stattdessen findet ein Export in das Zwischenformat fbx statt. Dies kann im Export-Menü von Blender ausgewählt werden. Bei diesem Export ist weiterhin darauf zu achten, dass alle später benötigten Informationen mit exportiert werden. Im vorliegenden Fall werden das Mesh, die Textur (es wird ein weißes png verwendet, ohne Textur schlägt die Anzeige fehl) und das Skelett für den Export ausgewählt.

7.4.5. g3db

Um das Modell von fbx nach g3db zu konvertieren, ist ein eigener Konverter nötig[8], der per Konsole ausgeführt werden kann. Dieser ermöglicht das Generieren von g3db- oder g3dj-Dateien. Die g3dj-Datei enthält die Informationen der g3db-Datei in einem lesbaren Format. Hier werden die einzelnen Knochen und Teil-Meshes mit ihrem Namen aufgeführt.

7.5. ComicGenerator

Die Klasse ComicGenerator ist die Schnittstelle zur Ansprache durch die DSL. Hier werden globale Parameter - wie die Seiten-Abmaße oder Schriftgrößen - gesetzt und die Anfragen zur Panel-Generierung an den SceneBuilder weiter geleitet. Verschiedene *Descriptions* enthalten eine genauere Beschreibung der zu generierenden Szene (welche Charaktere, Hintergrund, Text, usw.). Weiterhin kümmert sich der ComicGenerator um die Game-Loop, also das abwechselnde Aufrufen von SceneBuilder und PanelRenderer sowie die Verwaltung der Comic-Seiten.

7.5.1. PanelDescription

Beim Aufruf zur Panel-Generierung wird eine PanelDescription mitgeliefert, welche die Inhalte des Panels enthält. Textboxen, Hintergrund, Variabilitätswerte (siehe hierzu Kapitel 5.3) und Charaktere und Sprechblasen in Form entsprechender Beschreibungen (Character- bzw. BalloonDescription) werden angezeigt.

7.5.2. CharacterDescription

Die CharakterDescription enthält den Namen des darzustellenden Charakters, anhand dessen er aus der Liste von geladenen Modellen herausgesucht werden kann. Die Geste bestimmt die Verformung des Modells.

7.5.3. BalloonDescription

Die BalloonDescription hält den textuellen Inhalt einer Sprechblase, ihren Typ (Sprech-, Schrei-, Denkblase o.ä.), sowie den zugehörigen Sprecher, um den Dorn zu zeichnen.

[8] fbx-conv Kommandozeilen-Funktion auf Basis der FBX SDK zur Konvertierung von .obj-Dateien in ein Laufzeit-freundliches Format

7.6. SceneBuilder

Die SceneBuilder-Komponente ist zuständig für den Aufbau der darzustellenden 3D-Szene. Nach den per Descriptions erhaltenen Vorgaben werden Charaktere eingefügt. Diese werden per ModelPoser in die gewünschte Gestik gebracht. Mehrere Charaktere werden nebeneinander gestellt und anhand von Variabilitätswerten zueinander gedreht. Weiterhin wird im SceneBuilder die Kamera platziert, um die Szene wie gewünscht aufzunehmen und zuletzt die enthaltenen Sprechblasen gesetzt. Die fertige *Scene* wird durch die Komponente PanelRenderer zum Comic-Panel.

7.6.1. ModelPoser

Der ModelPoser bringt den ihm übergebenen Charakter in Position, setzt also die Gestik um. Hierzu wird auf die Bones zugegriffen, die im 3D-Modell enthalten sind und das Mesh beeinflussen. Mit Hilfe von Quaternionen werden die Gelenke verbogen. Dies ist nur dann einheitlich für alle 3D-Modelle möglich, wenn in Blender die *Armature* vor dem Export normiert wird. Weiterhin wird hier der Variabilitätswert genutzt, um die in den Gesten verwendeten Winkel leicht zu variieren (siehe hierzu Kapitel 5.3).

7.6.2. Gesten

Eine vorgefertigte Sammlung an Gesten kann im Comic-Generator und somit der Comic-DSL verwendet werden. Für jede Geste ist im ModelPoser ein Codeabschnitt für eine passende Modifikation des Meshes zuständig. Mit der Auswahl und Implementierung der Gesten beschäftigt sich Kapitel 5.4.1.

7.6.3. ViewFinder

Nachdem die 3D-Szene mit den Charakteren fertig erstellt ist, sorgt der ViewFinder für eine Platzierung und Ausrichtung der Kamera, mit der alle wichtigen Inhalte eingefangen werden. Hierfür müssen die richtige Entfernung zu den Charakteren, die genaue Position im Raum und die passende Höhe der Kamera gefunden werden. Die Konzeption hierzu, sowie unterschiedliche Perspektiven und Zoomlevel, beschreibt Kapitel 5.4.3.

7.6.4. BalloonPlacer

Bereits der ViewFinder beachtet, ob in einem Panel Sprechblasen enthalten sind. Entsprechend wird der Bildausschnitt so gewählt, dass oberhalb der Charaktere Platz bleibt. Der Platzierungs-

mechanismus für die Sprechblasen ist aktuell so einfach wie möglich gehalten und ordnet die Sprechblasen von links nach rechts, oberhalb der Köpfe an, um die Sprechreihenfolge einzuhalten.
Eine abwechslungsreichere und ansprechendere Anordnung kann in Zukunft unter Zuhilfenahme des Algorithmus von Kurlander [7] geschehen, oder sogar unter Nutzung des passiven Raumes innerhalb eines Panels [6].

7.6.5. Scene

Die Scene wird vom SceneBuilder an den PanelRenderer weiter gereicht. Sie enthält die 3D-Szene, die Abmessungen des Panels, sowie Textboxen und Sprechblasen.

7.7. PanelRenderer

Die Komponente PanelRenderer stellt die 3D-Szene im Comic-Stil dar. Dies wird durch Anwendung eines Comic-Shaders erreicht. Weiterhin werden von der Szene unabhängige Elemente wie Sprechblasen, Textboxen und der Hintergrund in Position gebracht. Ist das Panel fertig gerendert, so wird es mit dem typischen, schwarzen Rahmen umzogen und abgespeichert.

7.7.1. Comic-Shader

Der eingesetzte Comic-Shader arbeitet nach dem Vorbild von Decaudin [1]. Seine Funktionsweise ist in Kapitel 5.5 beschrieben. LibGDX ermöglicht hier die einfache Einbindung von Vertex- und Fragment-Shadern in der OpenGL Shading Language (GLSL). Für die Normalenmap werden gerichtete, gefärbte Lichtquellen gesetzt und die Ergebnisse der einzelnen Shading-Schritte mit Hilfe eines Framebuffers (FBO - Frame Buffer Object) kombiniert.

7.7.2. Texte

Sprechblasen, Textboxen und Symbole werden mit Hilfe einer Stage über die Comic-Szene gelegt. Anhand der in der *Scene* übergebenen Werte werden sie positioniert und angezeigt. Die LibGDX-Stage ermöglicht hierbei die einfache Nutzung einer 2D-Fläche über einer 3D-Umgebung.

7.7.3. Screenshot

Das fertige Panel wird nun abgespeichert, indem ein Screenshot erstellt und im gewünschten Dateiformat und Ordnerstruktur abgelegt wird. Im vorliegenden Fall wird mit png gearbeitet,

das Bild anhand der Panelnummer auf der aktuellen Seite benannt und in einem Ordner für die Seite abgespeichert. Dazu mehr in Kapitel 7.8.1.

7.8. Export

Eine zusätzliche Komponente sorgt für den Export in verschiedene Formate, um die generierten Bilder im Kontext einer Comicseite anzeigen zu können. Hierfür sind eine sinnvolle Ablage der Bilder und ein Gerüst, um diese passend einbinden zu können nötig.

7.8.1. Ordnerstruktur

Innerhalb des zu Beginn mit einem Pfad angegebenen Ordners wird ein neuer Ordner mit dem Namen des Comics angelegt. Innerhalb dessen gibt es für jede Seite einen eigenen Ordner, der entsprechend benannt ist (z.B. „page_1“). Diese Ordner halten die auf der Seite enthaltenen Panel mit äquivalenter Benamung (z.B. „panel_4.png“). Beispielhaft ist dies in Abbildung 7.4 dargestellt. Diese Anordnung ermöglicht strukturierte Referenzen für die Export-Dateien. Es kann einfach über alle Seiten bzw. Panel iteriert werden.

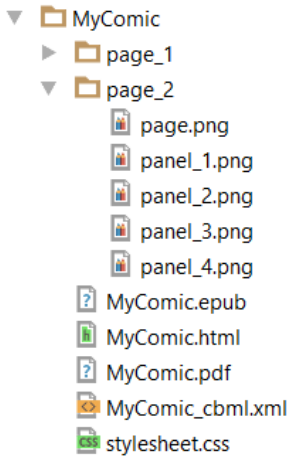


Abbildung 7.4.: Beispielhafte Ordnerstruktur

7.8.2. HTML

Beispielhaft sei hier der Export nach HTML vorgestellt. Das entstehende HTML dient als Grundlage für weitere Export-Formate, auf die in Kapitel 5.7.5 näher eingegangen wurde. Die Methode $exportToHTML()$ schreibt HTML-Syntax in ein neues Dokument. Innerhalb von div-Tags für jede Seite werden img-Tags für die Seiten eingefügt. Zusätzlich wird ein vorgefertigtes css in den Comic-Ordner kopiert.

7.9. Qualitätssicherung

Für die Qualitätssicherung eines Comic-Generators liegt es nahe zu überprüfen, ob die generierten Bilder ästhetisch ansprechend und sinnvoll zusammengesetzt sind. Dies würde im vorliegenden Falle allerdings in einer detaillierten Bilderkennung resultieren. Stattdessen werden wie gewohnt sämtliche gesetzten und berechneten Parameter überprüft. Tests können

überprüfen, ob alle Kombinationen (z.B. bzgl. Charakter und Gesten) möglich sind und ob beim Export alle benötigten Dateien generiert werden.
Allerdings ist auch ein Bildabgleich umsetzbar, um zu gewährleisten, dass der gleiche Methodenaufruf zur Panelgenerierung (bei gleichen Variabilitätswerten) in dem selben Bild resultiert (siehe hierzu Kapitel 5.3).

7.9.1. JUnit

Innerhalb des core-Moduls sind JUnit-Tests angelegt. Diese teilen sich auf in allgemeine Tests und solche, die aufgrund der Schnittstelle zur DSL durchgeführt werden müssen.

Dateipfade Bevor die Generierung eines Comics gestartet wird, überprüft das System, ob die angegebenen Pfade korrekt sind. Also ob die angegebenen Dateien existieren, das richtige Format haben und im Falle der Charaktere, ob ein korrekt benanntes Skelett enthalten ist. Schlagen diese Überprüfungen fehl, so wird dem Benutzer eine entsprechende Meldung angezeigt mit der Bitte, den Pfad zu korrigieren.

Schnittstellen-Tests Damit die Schnittstelle zum Comic-Generator von dem Generator der DSL aus genutzt werden kann, musste diese um einige Methoden erweitert werden. Das Problem ist hierbei, dass MPS lediglich die Datentypen Integer, String und Boolean unterstützt, sodass Repräsentationen der eigentlich benötigten Werte korrekt umgewandelt werden müssen. So wird aus einem Wert für die Panel-Breite (z.B. 12 bzw. “1_2“) ein passender Float (hier 0.5 bzw. 1/2).

7.10. Zusammenfassung

Der Comic-Generator ist technisch gesehen ein Zusammenspiel von Blender-Modellen und einem Szenenaufbau per LibGDX. Beides funktioniert fast reibungslos miteinander, kann aber in Zukunft noch besser werden. Besonders zu nennen sind hier der Umgang mit den Koordinatensystemen und den Informationen, die zwischen Blender-Modellen und LibGDX ausgetauscht werden können, mit dem aktuellen Entwicklungsstand des Frameworks.

7.10.1. Erfahrungen mit LibGDX

LibGDX hat sich als eine gute Wahl für das Grafik-Framework erwiesen und ist trotz eines Spiele-Schwerpunktes gut für die Aufgabe der Bildgenerierung geeignet. Aktuell wird es aktiv

weiterentwickelt und eine große Community hilft bei Fragen und Schwierigkeiten. Für den Umgang mit 3D-Modellen bietet LibGDX Funktionen zur Manipulation der Modelle, die Nutzung von Shadern und das Prinzip einer Stage, die normalerweise für Hintergrund-unabhängige Steuerelemente genutzt wird und hier als Plattform für Sprechblasen dient. Weiterhin ist ein Zuschneiden der Bildauswahl zu einem Panel mit Hilfe eines Screenshots problemlos möglich.

Der Umgang mit Shape-Keys aus Blender ist aktuell noch nicht implementiert, würde allerdings für eine Umsetzung der Mimik benötigt. Für das Problem besteht bereits ein Ticket und bei Bedarf kann ein Pull-Request mit der zusätzlichen Funktionalität zu einer Erweiterung von LibGDX führen. Dank der Nutzung von Gradle ist ein LibGDX-Projekt sehr einfach zu aktualisieren. So stellten Versionswechsel im Laufe des Projekts kaum ein Problem dar.

7.10.2. Erfahrungen mit Blender

Eine unerwartete Problematik bestand mit den Bounding-Boxen um Blender-Modelle herum, die sich am Skelett, statt am Mesh orientierten. Da diese für die Platzierung des Charakters im Panel benötigt wurden, mussten hier eigene Berechnungen angestellt werden.
Der Umgang mit Blender ist mit Kenntnis der grundlegenden Tastenkürzel zum Mesh-Aufbau möglich. Die Qualität der modellierten Meshes ist hierbei natürlich nach oben offen, wobei immer bedacht werden sollte, dass ein detaillierteres Mesh auch mehr Arbeitsspeicher benötigt.

7.10.3. Schwierigkeiten

Die größte Schwierigkeit bei der Arbeit mit Blender und LibGDX lag dort, wo beide Systeme die Informationen unterschiedlich verarbeiten. So unterscheiden sich die karthesischen Koordinatensysteme und die Bounding-Boxen aus Blender erfüllten nicht die im LibGDX-Code erforderlichen Bedingungen. Diese Probleme sind allerdings bekannt und entsprechend gut dokumentiert.

7.10.4. Ausblick

Zukünftig soll der Comic-Generator noch komplexere Szenen handhaben können. So zum Beispiel viele Charaktere, eine dreidimensionale Umwelt oder Interaktion von Charakteren und Objekten. Auf der zweidimensionalen Ebene können noch mehr Effekte, wie Speedlines oder Texturen ergänzt werden. Auf jeden Fall soll aber eine Ergänzung um die Mimik der Charaktere erfolgen, auch wenn dies eine Erweiterung des LibGDX-Frameworks voraussetzt.

8. Evaluation

Das folgende Kapitel soll rückblickend die erarbeiteten Ergebnisse dieser Arbeit bewerten. Welche neuen Funktionalitäten wurden geschaffen? Was funktioniert nicht und welche Erweiterungsmöglichkeiten bestehen für die Zukunft?

8.1. State of the art

Bei der Suche nach „Comic Generator“ im Internet trifft man auf eine Vielzahl von Ergebnissen. Die meisten sind hierbei Anwendungen, mit denen der Benutzer sich per Drag & Drop einen Comic zusammen klicken kann. Je nach Lösung ist die Auswahl an Hintergründen und posierten Charakteren kleiner oder größer. Immer bleibt das Erschließen der Szene und die Positionierung der Charaktere jedoch Aufgabe des Benutzers.
Die unter Kapitel 2 vorgestellten Arbeiten 2.1 Comic Chat und 2.5 Comix.js sind die einzigen, welche dem Benutzer das Erstellen des Bildes abnehmen. Comix.js benötigt hierbei allerdings weiterhin umfangreiche Eingaben durch einen Programmierer.
Comic Chat von 1996 hat als Input die Chat-Nachrichten eines IRC-Chats in einen Comic umgewandelt. Gesten und Mimik der Charaktere können vom Benutzer ausgewählt werden oder werden anhand von Schlüsselwörtern in den Chat-Nachrichten festgelegt. Platzierung von Charakteren und Sprechblasen findet anhand von Algorithmen statt, die bei Bedarf ein neues Panel beginnen lassen. Perspektive wird nicht genutzt, der Zoom variiert jedoch, um Abwechslung zu schaffen, oder alle Charaktere im Panel anzeigen zu können.

8.2. Use-Case

Um die Funktionsweise der Comic-DSL zu gewährleisten wurde von Hand ein Comic gezeichnet, der den Use-Case darstellt (siehe Anhang A.1). Dieser wurde mit passendem DSL-Code beschrieben, der in Anhang A.2 aufgeführt wird. Je näher der daraus resultierende Comic dem im Voraus erstellten Werk entspricht, desto besser funktioniert der Generator. Das aktuelle Ergebnis ist in Anhang A.3 zu sehen.

Bei der Umsetzung von erweiterten Funktionalitäten wurde sich am Use-Case-Comic orientiert, um die enthaltenen grafischen Elemente als Minimum umzusetzen. Die Mimik konnte aus bereits aufgeführten Gründen nicht umgesetzt werden, ebenso ist die aktuelle Sprechblasenplatzierung deutlich statischer. Um die genauen Perspektiven zu erzwingen, können die Variabilitätswerte als optionale Parameter von Hand in den DSL-Code eingetragen werden. Dies soll bei einer gewöhnlichen Umsetzung natürlich automatisiert geschehen, ermöglicht dem Benutzer aber größere Kontrolle, sofern er sie wünscht.

8.3. Projektumfang

Im Folgenden wird stichwortartig aufgeführt, welche Funktionalitäten im aktuellen Stand der Comic-DSL enthalten sind. Als Kern-Funktionalitäten ist eine DSL mit HTML-ähnlicher Syntax umgesetzt, die ein eingebundenes Java-Programm ausführt. Dieses erzeugt Comic-Panel, entsprechend der übergebenen Parameter. Konkrete Funktionen sind die folgenden:

- Anzeige von Sprechblasen verschiedener Typen
- Anzeige von Textboxen *caption* und *description*
- Umsetzung von Gesten aus einer vorgegebenen Sammlung
- Szene mit einem oder mehreren posierten Charakteren
- Platzieren der Kamera in Vogel- oder Froschperspektive
- Kamera-Zoom von *Totale* bis *Portrait*
- Export des fertigen Comics in verschiedene Formate

8.3.1. Panels

Als Hauptelement eines Comics treten Panels in verschiedenen Maßen auf. Gegeben sind die Abmessungen vorrangig durch die Größe der Seite, sowie deren Grid (3x3 oder 4x4 Panels). Während das Grid die Höhe der Panels festlegt kann die Breite per Parameter bestimmt werden. $< panel\ width = $"1_3" $>$ legt ein Panel mit einem Drittel der Gesamtbreite der Seite an. Eine Auswahl verschiedener Panelgrößen ist in Abbildung 8.1 zu sehen.
Nicht möglich sind bisher ungewöhnliche Panelformen, wie runde, rautenförmige oder grundsätzlich ungleichmäßige Panels. Ebenso sind keine Panels angedacht, die über die Zeilenabmessungen hinaus ragen. Entsprechend ist auch der in Comics verbreitete 4th-wall-break, also das Hinausragen des Panelinhaltes über den Rahmen, vorerst nicht machbar.

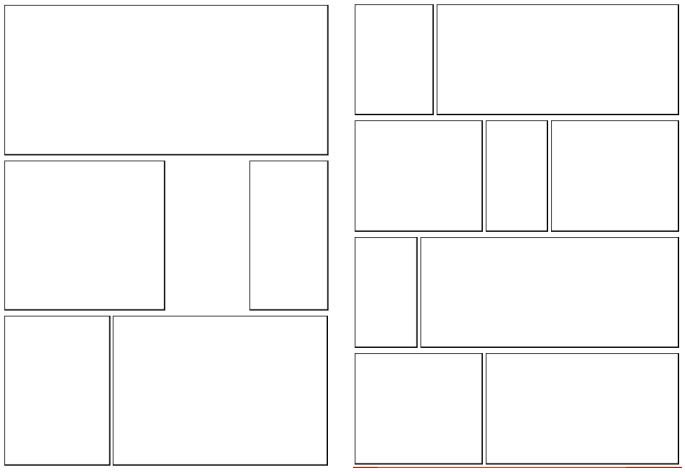


Abbildung 8.1.: Umsetzung verschiedener Panel-Breiten und Page-Grids

8.3.2. Textboxen

Textboxen als Überschrift und zur Beschreibung sind umgesetzt und werden nach dem Comic-Shading über die Szene gelegt. Abbildung 8.2 zeigt den Einsatz der Textboxen. Eingefügt werden sie im Panel-Tag: $< caption > Meahwhile... < /caption >$.

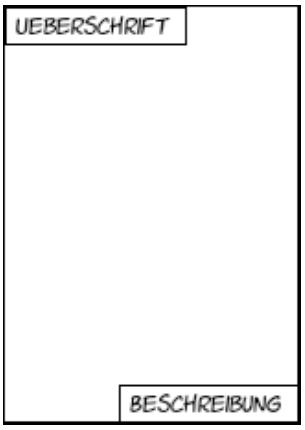


Abbildung 8.2.: Einsatz von Textboxen

8.3.3. Sprechblasen

Die Sprechblasen sind eigene 3D-Meshes, der Code stammt von Raimund Wege. Es existieren die unter 3.4.2 vorgestellten Grundtypen, die sich programmatisch nach belieben anpassen lassen. Ebenso zeigt der Dorn jeder Sprechblase auf den jeweiligen, im Panel befindlichen Sprecher. Beispielhaft abgebildet ist dies in Abbildung 8.3.

Nicht umgesetzt ist eine komplexe Sprechblasenplatzierung, wie etwa bei Kurlanders Comic Chat [7]. Schön wäre außerdem ein Ausnutzen des passiven Raumes im Panel, sowie eine Verschmelzung zusammengehöriger Sprechblasen.

Abbildung 8.3.: Verschiedene Typen von Sprechblasen und Sprechern

8.3.4. Charaktere und Objekte

Charaktere und Objekte werden zu Beginn des DSL-Codes deklariert, indem Pfade zu den jeweiligen Modellen angegeben und ein Name zur Referenzierung vergeben wird. Zum Beispiel: $< character\ name = "Fenja"\ path = "path/to/fenja.g3db"\ size = 176/ >$
Sofern die Charaktere in einem lesbaren Format vorliegen und ein korrekt benanntes, normiertes Skelett besitzen, können vorimplementierte Gesten für jedes Modell umgesetzt werden. Abbildung 8.4 zeigt eine Auswahl von angewendeten Gesten. Die Einbindung erfolgt innerhalb eines Panel-Tags, z.B. wie folgt: $< character\ name = "Fenja"\ gesture = "wink"/ >$.

Mimik konnte wie bereits in Kapitel 5.4.2 dargelegt, noch nicht umgesetzt werden. Ebenso kann noch keine Interaktion zwischen Charakteren oder mit Objekten stattfinden. Äquivalent zu

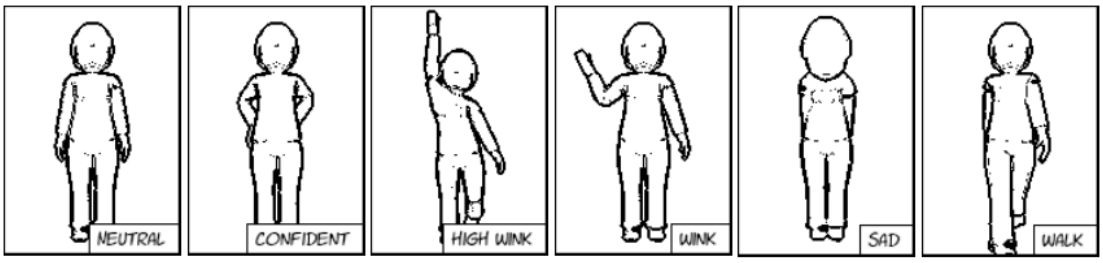


Abbildung 8.4.: Implementierte Gesten

Charakteren können auch allgemein Objekte dargestellt werden. Diese enthalten kein Skelett, sind also sehr statisch und variieren ausschließlich durch die Kameraposition.

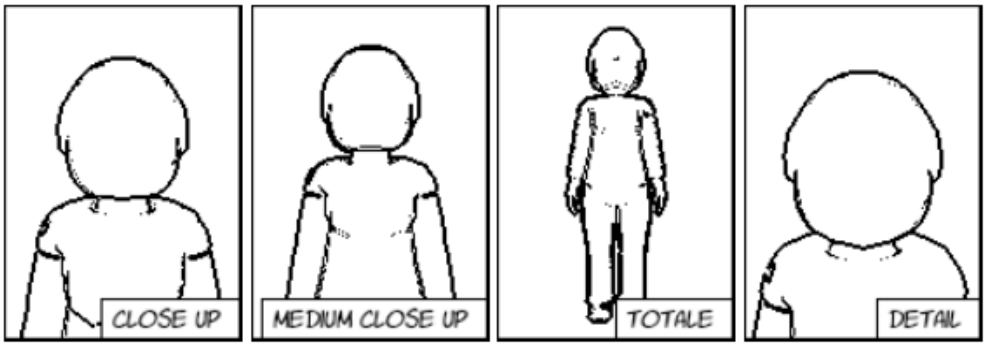


Abbildung 8.5.: Implementierte Kamera-Zoomeinstellungen

8.3.5. Kamera

Für die Kamera sind verschiedene Zoomstufen und Perspektiven einsetzbar (genauer in Kapitel 5.4.3), die über den Parameter $perspective$ im Panel-Tag modifizierbar ist. Der Default sorgt dafür, dass die im Panel genutzte Anzeige eine andere ist, als im vorherigen Panel, um für den Leser Abwechslung zu schaffen. Eine Vorschau der möglichen Perspektiven zeigen die Panel in Abbildung 8.5.
Für die Zukunft wünschenswert wäre ein intelligenter Default, der die Perspektive anhand von Inhalt oder Stimmung des Panels auswählt.

8.3.6. Hintergrund

Als Hintergrund können bisher Bilder in der Manier, wie auch Charaktere und Objekte geladen werden. Werden sie im Panel genutzt, so wird anhand des Variabilitätswertes ein Ausschnitt gewählt. Beim Rendern der Szene wird der Hintergrundausschnitt in der Größe des Panels hinter den Modellen angezeigt. Abbildung 8.6 zeigt das Ausgangsbild und Abbildung 8.7 Ausschnitte, abhängig von der Variabilität. Der Hintergrund-Tag wird innerhalb des Panel-Tags eingefügt und sieht wie folgt aus:
$< background\ name = "hamburg"\ variability = 6/ >$

Die aktuelle Umsetzung ist unabhängig von der Kamera, was bei extremen Perspektiven falsch aussieht. In Zukunft ist das Texturieren einer Skybox im Hintergrund wünschenswert, um sich auch an die Perspektive und den Zoom anzupassen. Noch einen Schritt weiter wäre eine dreidimensionale Umgebung denkbar, die durch die Positionierung der Kamera optimal die Szene beschreibt.

Abbildung 8.6.: Hintergrundbild „Hamburg“

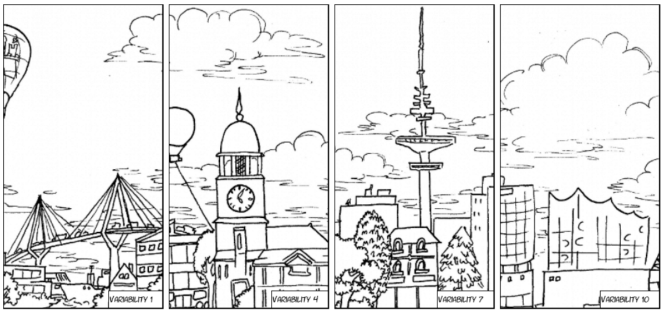


Abbildung 8.7.: Hintergrundausschnitte anhand der Variabilitäts-Werte

8.3.7. Export

Umgesetzt wurde der Export in verschiedene Formate. HTML zur Darstellung im Browser, das verbreitete Dateiformat PDF und ePub, um die generierten Comics auf eReadern lesen zu können. Ein Test für die Kompatibilität mit verschiedenen Devices und Browsern hat hierbei nicht stattgefunden. Weiterhin wird ein XML-Dokument erstellt, dass nach Vorlage der CBML (siehe hierzu Kapitel 2.2) den Inhalt des Comics beschreibt, um später Suchen zu unterstützen.

8.4. Erweiterbarkeit

Ein wichtiges Designziel ist die Erweiterbarkeit, vor allem im Hinblick darauf, dass im Rahmen der Masterarbeit aus Zeitgründen nicht alle Möglichkeiten ausgeschöpft werden können. Die Ansatzpunkte zur Erweiterung von Inhalten und Funktionalitäten müssen dennoch bereits gegeben sein. Deshalb im Folgenden eine Übersicht, wie einige der Elemente erweitert werden können.

Charaktere Am einfachsten lässt sich der Comic-Generator um neue Charaktere erweitern. Es müssen neue Modelle modelliert und dann einfach im DSL-Code referenziert werden. Bei der Erstellung neuer Modelle ist lediglich darauf zu achten, dass ein korrekt benanntes Skelett enthalten ist, damit die Gesten umgesetzt werden können. Außerdem muss das Modell im g3db-Format vorhanden sein, was meistens durch einen Export aus dem Modellierungs-Werkzeug oder mit einem Konverter bewerkstelligt wird.

Gesten Um den Pool an Gesten zu erweitern muss ein neuer Eintrag in die Gesten-Enums erfolgen. Im ModelPoser muss entsprechend die Modifikation des Skeletts für die jeweilige Geste implementiert werden. Eine Einbeziehung der variablen Werte sollte nach Möglichkeit berücksichtigt werden, um die Darstellung abwechslungsreich zu halten.

Mimik Sobald die Funktionalität der Mimik umgesetzt ist, kann auch hierfür - äquivalent zu den Gesten - eine Sammlung angelegt werden und ein FacePoser durch die Manipulation der Modell-Shape-Keys verschiedene Gesichtsausdrücke formen.

Kamera Weitere Perspektiven und Winkel können in der ViewFinder-Klasse hinzugefügt werden. Um diese Möglichkeiten explizit zu nutzen werden entsprechende Parameter in der DSL benötigt.

Sprechblasen und Textboxen Andere Designs für Sprechblasen und Textboxen müssen zusätzlich implementiert werden und könnten dann über ein Flag für den gesamten Comic, oder aber einzelne Panel ausgewählt werden.

Panel Andere Panel-Abmaße, z.B. über mehrere Zeilen, sind aktuell eher schwierig umzusetzen, da die Panels einer Seite in Zeilen angeordnet sind. Hier müsste also eine andere Strukturierung umgesetzt und komplexere Constraints für das Befüllen einer Seite mit Panels aufgestellt werden. Problemlos ist allerdings das Hinzufügen alternativer Panelbreiten, wie $\frac{1}{10}$ oder $\frac{3}{7}$. Diese müssen lediglich in der DSL erweitert werden, da der Generator nur mit den Werten rechnet.

Werkzeug zur 3D-Modellierung Um neben Blender auch andere 3D-Modellierungs- Werkzeuge zur Erstellung von Charakteren oder Objekten nutzen zu können, müssen diese zum einen die Skelettstruktur mit benannten Knochen unterstützen und einen Export in ein für LibGDX lesbares Format ermöglichen.

8.5. Schwierigkeiten

Insgesamt gab es die meisten Schwierigkeiten durch die Verwendung bisher unbekannter Werkzeuge. Häufig mussten sich zugehörige Grundlagen angelesen werden, bevor mit der Implementierung begonnen werden konnte. Einzeln zu nennen sind mathematische Fragestellungen im dreidimensionalen Raum, das Zusammenwirken von Blender und LibGDX, sowie MPS und LibGDX und nicht zuletzt die Abhängigkeiten innerhalb von MPS.

9. Zusammenfassung

Die Problemstellung Comic Generator wurde im Rahmen des Masterstudiums erfolgreich bearbeitet. Eine DSL und ein Java-System ermöglichen die automatisierte Generierung eines Comics aus DSL-Code. Der aktuell vorliegende Prototyp bietet zahlreiche Erweiterungsmöglichkeiten in alle erdenklichen Richtungen.

9.1. Comic DSL

Die in dieser Arbeit implementierte DSL zur automatisierten Generierung von Comics besteht aus zwei Hauptkomponenten: Einer DSL und einem Java-System, das für die eigentliche Generierung zuständig ist. Beide Komponenten sind über eine Schnittstelle verknüpft (siehe hierzu Kapitel 6.7). Auf Seiten des Java-Systems ist dies ein Interface, dessen Methoden die Generierung einzelner Panel anstoßen. Die DSL generiert in ihrer Generator-Komponente einen Launcher, der für LibGDX notwendig ist, sowie eine Hauptklasse, welche eine Instanz der Schnittstellenklasse erstellt und die entsprechenden Methoden aufruft.

9.1.1. DSL

Die Comic-DSL wurde mit Hilfe der PLWB MPS erstellt. Dieses bietet vielfältige Möglichkeiten für Einschränkungen, Automatisierungen und Funktionen, die nach der bisher erfolgten Einarbeitung noch nicht komplett ausgeschöpft werden konnten (siehe hierzu auch Kapitel 6.2). Eine mit einer Build-Solution erstellte Standalone-IDE (beschrieben in Kapitel 6.2.4) ermöglicht es, dass Benutzer der Comic-DSL sich ein zugehöriges Programm installieren und direkt damit arbeiten können.

DSL-Code wird in einem Editor (6.4) verfasst. Durch MPS ist dieser projektional, was bedeutet, dass viel mit Autovervollständigung gearbeitet wird. Dies bedarf einer längeren Eingewöhnungsphase, erhöht langfristig aber die Programmiergeschwindigkeit. Pattern für den Entwurf einer neuen Programmiersprache (4.2) werden zum Großteil durch MPS umgesetzt, vor allem durch den projektionalen Editor werden Fehler vermieden.

Aus dem Editor-Code wird ein AST (6.3.1) generiert, welcher in seiner Hierarchie gut zur abstrakten Darstellung eines Comics passt. Ein Generator überführt diesen AST wiederum in ausführbaren Java-Code, welcher die Generierung der Comic-Panel anstößt (beschrieben in Kapitel 6.5).

9.1.2. Java-Generator

Eine Umsetzung des Panel generierenden Java-Systems fand mit dem Grafik-Framework LibGDX statt (siehe hierzu 7.1). Die Schnittstelle des ComicGenerators bietet Methoden, die nach Vorgabe durch die Parameter, ein Comic-Panel generieren. Im fertigen System stammen diese Parameter aus dem DSL-Code.

3D-Modelle

Das besondere am vorliegenden Comic-Generator ist die Arbeit mit 3D-Modellen. Andere Ansätze arbeiten zumeist mit einem Pool an 2D-Grafiken, die weniger flexibel und aufwändiger in der Erweiterung sind. Modelliert werden Objekte und Charaktere mit dem 3D-Grafikprogramm Blender (7.3.5). Wichtig ist hierbei, dass ein Skelett mit benannten Knochen erstellt und an das Mesh geknüpft wird. Dies ermöglicht die spätere programmatische Ansprache einzelner Knochen, um konkrete Gesten umzusetzen.
Ein fertiges Modell wird über das Format fbx in das für LibGDX lesbare Format g3db (7.4.5) überführt und kann über ein referenzieren des Pfades vom System geladen und eingebunden werden.

SceneBuilder

Die Komponente SceneBuilder (7.6) wird vom ComicGenerator aufgerufen, um die mit Parametern beschriebene Szene zu bauen. Hier werden die im Panel auftretenden Charakter-Modelle geladen und durch einen ModelPoser (7.6.1) mit entsprechenden Gesten versehen. Eine Bounding-Box, die den einen oder mehrere Charaktere in der Szene umschließt, wird an die Kamera weiter gereicht. Die Kamera richtet sich so aus, dass sie die Box um die Charaktere abbildet. Je nach Parameter für Zoom oder Perspektive variiert ihre Position und Richtung (Komponente ViewFinder 7.6.3). Die fertige Szene mit Modellen, Kamera und Sprechblasen wird an die nächste Komponente weiter gereicht.

PanelRenderer

Die übergebene Szene wird vom PanelRenderer (7.7) mit Hilfe eines Comic-Shaders (7.7.1) bearbeitet, sodass das comic-typische Aussehen erhalten wird. Ein eventueller Hintergrund wird hierbei mit einbezogen. 2D-Elemente wie Textboxen und die Texte innerhalb der Sprechblasen werden nachträglich mit der LibGDX-Stage hinzugefügt. Die fertige Comic-Szene wird durch einen Screenshot als Bild in einer Ordnerstruktur abgespeichert. Ausgehend von den in Ordnern abgespeicherten Bildern findet ein Export in verschiedene Formate statt. So kann beispielsweise für ein passendes HTML über alle Ordner und Bilddateien iteriert und entsprechende Tags in eine Datei geschrieben werden.

9.2. Fortschritt im Masterstudium

Die vorliegende Arbeit entstand parallel zum Masterstudium Informatik. Entsprechende Vorlesungen und Projekte forderten die stückweise Erarbeitung des Themas. Dargestellt ist die Bearbeitungskette der Komponenten in Abbildung 9.1.
In Anwendungen 1 (AW1) wurden die Grundlagen des Themas erforscht. Hier wurde sich über die Grundlagen der Comic-Kunst, Verbreitung und Historie, sowie über erste Vergleichsprojekte informiert. Diese verwandten Arbeiten (Kapitel 2) wurden im Rahmen von Anwendungen 2 (AW2) näher betrachtet und der eigenen Arbeit gegenüber gestellt. Hier wurden vor allem der Cartoon-Shader [1] und die im Comic Computing [10] umgesetzte Hierarchie für die eigene Arbeit übernommen. Das zeitgleich mit AW2 verlaufende Projekt 1 (PJ1) hatte die Auseinan-

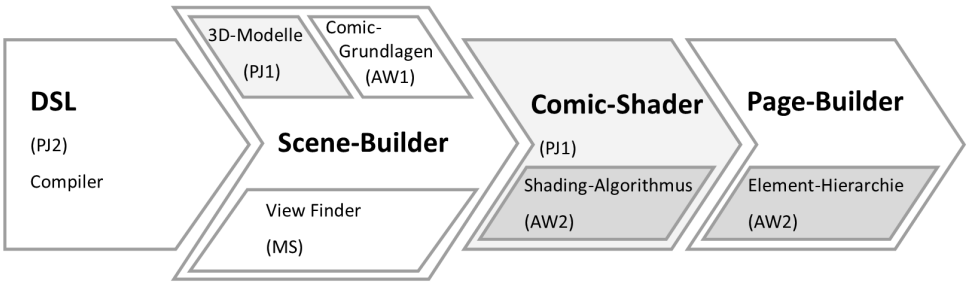


Abbildung 9.1.: Komponenten der Masterarbeit und Bearbeitungsrahmen

dersetzung mit der 3D-Modellierung, sowie die Implementierung des Cartoon-Shaders zum Thema. Im Semester darauf folgte Projekt 2 (PJ2), während dessen die Comic-DSL entworfen und soweit bereits möglich implementiert wurde. Thema des Master-Seminars (MS) war vorran-

gig ein Überblick über den Fortschritt. Die finale Umsetzung, vor allem der Kern-Komponente SceneBuilder, erfolgte im letzten Semester des Masters.

9.3. Vision

Die ursprüngliche Idee des Comic-Generators lag darin, beliebigen Text in Comicform zu bringen. Vorrangig, um Comics zur Dokumentation von beispielsweise Softwareprojekten nutzen zu können, anstatt eines viele Seiten langen Schriftstücks mit Formeln und Diagrammen. Für diese Vision ist mit der vorliegenden Arbeit der erste Schritt getan. Zur Umsetzung muss nun der Text in die Comic-DSL überführt werden. Dies ist zwar ein großer Schritt, er ließe sich jedoch mit einer Einschränkung auf ein bestimmtes Themengebiet vereinfachen. So könnten zum Beispiel Regeln für Brettspiele oder Kochrezepte automatisiert in Comics umgewandelt werden.
Auf einer höheren Ebene soll diese Arbeit die Nutzung des Comics fördern. Durch ein geeignetes Tool können Comics leichter hergestellt und somit mehr verwendet werden, z.B. für schönere und gern gelesene Softwaredokumentationen.

9.4. Nächste Schritte

Das Projekt *Comic Generierung* wird voraussichtlich über die Masterarbeit hinaus fortgeführt. Ziel ist hierbei ein fertiges Produkt zur Comic Generierung. Nächste Schritte sind hierfür die Erweiterung des Funktionsumfanges und umfangreiche Tests durch Test-Benutzer. Es gilt die verwendeten Technologien aktuell zu halten (Versionen von Blender, LibGDX und MPS) und teilweise bei der Entwicklung neuer Funktionalitäten zu unterstützen (z.B. um die Mimik in LibGDX umsetzen zu können).

9.5. Fazit

Die Bearbeitung dieser Arbeit begann mit minimalen Grundkenntnissen über 3D-Grafiken, DSLs und Comics. In jedes Teilgebiet erfolgte eine Einarbeitung und eine Auseinandersetzung mit neuen Themen, Funktionalitäten und Schwierigkeiten. Nach Überwindung der ersten Hürden wurden jedoch schnell Ergebnisse sichtbar. Die zuvor geplanten Funktionalitäten ließen sich fast ausnahmslos wie gewünscht umsetzen. Gemessen an der Einfachheit, mit der die letzten Erweiterungen im Rahmen dieser Arbeit implementiert werden konnten, zeigt sich das Potential, den Funktionsumfang auszuweiten.

Aus technischer Sicht ist die Umsetzung einer Comic-DSL und das automatisierte Generieren von Comics gelungen. Es existiert ein lauffähiger Prototyp mit bereits jetzt umfangreichen Funktionalitäten.

9.6. Ausblick

In Zukunft soll die Comic-DSL zu einem Produkt werden, das die Verbreitung des Comics unterstützt und hilft, aus ihm ein ernst zu nehmendes Medium zur Informationsvermittlung zu machen [22]. Als Werkzeug ist der Comic-Generator vielfältig einsetzbar: zur Dokumentation, für Konzeptzeichnungen oder zur Unterhaltung.
Es sind ähnliche Arbeiten denkbar, wie eine Cartoon-DSL für Filme, ein stilistischer Schwerpunkt auf japanischen Manga oder multifunktionale Panel mit Hyperlinks oder zusätzlichen Funktionen.

A. Use-Case

A.1. Comic

Abbildung A.1.: Use-Case-Comic

A.2. DSL-Code

```
< comic  name = " UseCase "  font_size = 14  project_path = " C:/Users/Fenja/Documents/UseCaseComic "  author = " Fenja " >
  < character  name = " Fenja "  path = " C:/Users/Fenja/Workspace/ComicGenerator/android/assets/models/fenja02.g3db "  size = 176 />
  < character  name = " Michael "  path = " C:/Users/Fenja/Workspace/ComicGenerator/android/assets/models/michi02.g3db "  size = 193 />
  < background  name = " Hamburg "  path = " C:/Users/Fenja/Workspace/ComicGenerator/android/assets/bgs/hamburg.png " />

< page  grid = 3 >
  < panel  width = 1_3  border = true  perspective = " medium close up " variability = 6 >
    < background  name = " Hamburg "  variability = 7  full = false />
    < character  name = " Fenja "  gesture = " wink "  variability = 4 />
    < balloon  speaker = " Fenja "  type = " speech " > Hallo, ich bin Fenja </ balloon >
  </ panel >
  < panel  width = 1_3  border = true  perspective = " detail " variability = 10 >
    < character  name = " Fenja "  gesture = " neutral "  variability = 1 />
    < balloon  speaker = " Fenja "  type = " speech " > Ich bringe meinem Computer bei, Comics zu zeichnen </ balloon >
  </ panel >
  < panel  width = 1_3  border = true  perspective = " default " variability = 10 >
    < caption > Besagter Computer... </ caption >
  </ panel >
  < panel  width = 1_2  border = true  perspective = " close up " variability = 10 >
    < character  name = " Fenja "  gesture = " confident "  variability = 6 />
    < character  name = " Michael "  gesture = " neutral "  variability = 3 />
    < balloon  speaker = " Michael "  type = " speech " > Wie soll das denn gehen? </ balloon >
    < balloon  speaker = " Michael "  type = " speech " > Der hat doch keine Haende </ balloon >
  </ panel >
  < panel  width = 1_2  border = true  perspective = " close up " variability = 10 >
    < character  name = " Fenja "  gesture = " sad "  variability = 7 />
    < character  name = " Michael "  gesture = " wink "  variability = 3 />
    < balloon  speaker = " Fenja "  type = " speech " > Ja, aber - </ balloon >
    < balloon  speaker = " Michael "  type = " speech " > Und Geschichten kann der sich auch nicht ausdenken </ balloon >
  </ panel >
  < panel  width = 1_5  border = true  perspective = " medium close up " variability = 10 >
    < character  name = " Fenja "  gesture = " neutral "  variability = 10 />
    < balloon  speaker = " Fenja "  type = " speech " > Er hat andere Qualitaeten </ balloon >
  </ panel >
  < panel  width = 1_5  border = true  perspective = " totale " variability = 10 >
    < character  name = " Fenja "  gesture = " neutral "  variability = 9 />
    < character  name = " Michael "  gesture = " neutral "  variability = 1 />
  </ panel >
  < panel  width = 3_5  border = true  perspective = " totale " variability = 10 >
    < character  name = " Fenja "  gesture = " walk "  variability = 1 />
    < character  name = " Michael "  gesture = " confident "  variability = 3 />
    < balloon  speaker = " Fenja "  type = " scream " > Wirst du schon sehen! </ balloon >
    < balloon  speaker = " Michael "  type = " speech " > Na dann... </ balloon >
    < description > Fortsetzung folgt </ description >
  </ panel >
</ page >
</ comic >
```

Abbildung A.2.: DSL-Code nach Vorlage des Use-Case-Comics

A.3. Generierter Comic

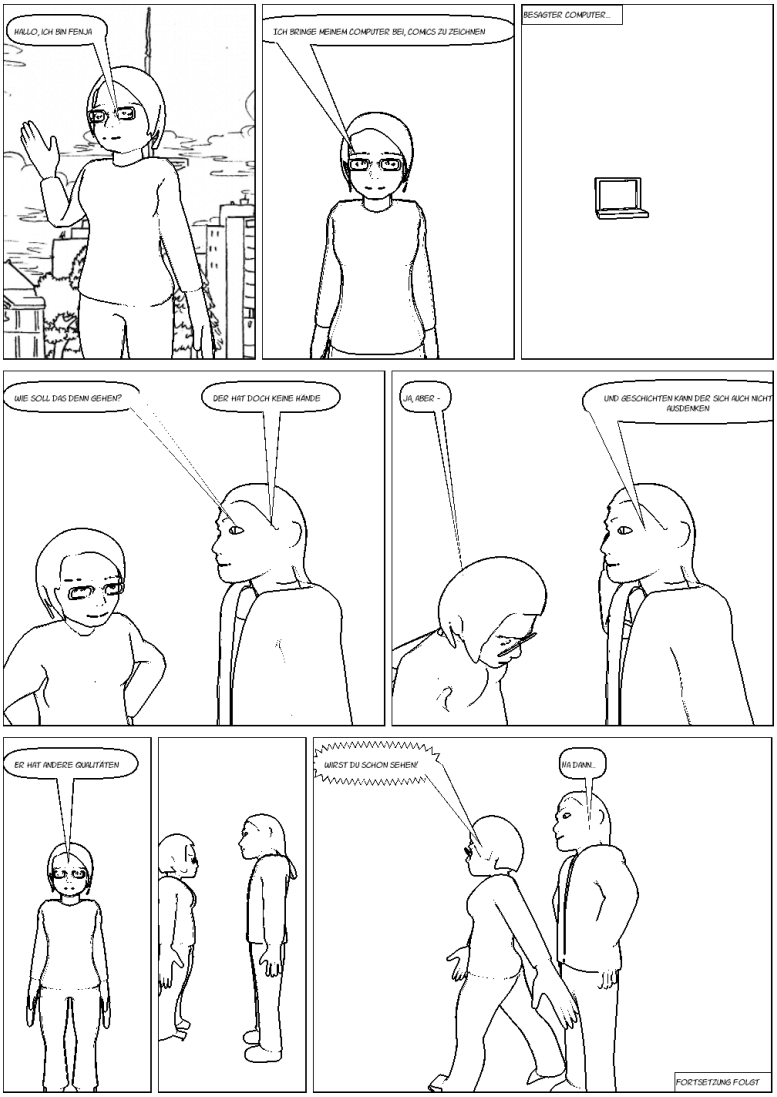


Abbildung A.3.: Generierter Comic aus dem DSL-Code

A.4. Generierte CBML

```
<div type="panelGrp">
    <cbml:panel n="1" xml:id="Us 0">
        <cbml:balloon xml:id="Us 0"  type="speech"  who="#Fenja">
            Hallo, ich bin Fenja
        </cbml:balloon>
    </cbml:panel>
    <cbml:panel n="2" xml:id="Us 1">
        <cbml:balloon xml:id="Us 1"  type="speech"  who="#Fenja">
            Ich bringe meinem Computer bei, Comics zu zeichnen
        </cbml:balloon>
    </cbml:panel>
    <cbml:panel n="3" xml:id="Us 2">
        <cbml:caption>
            Besagter Computer...
        </cbml:caption>
    </cbml:panel>
    <cbml:panel n="4" xml:id="Us 3">
        <cbml:balloon xml:id="Us 2"  type="speech"  who="#Michael">
            Wie soll das denn gehen?
        </cbml:balloon>
        <cbml:balloon xml:id="Us 3"  type="speech"  who="#Michael">
            Der hat doch keine Händer
        </cbml:balloon>
    </cbml:panel>
    <cbml:panel n="5" xml:id="Us 4">
        <cbml:balloon xml:id="Us 4"  type="whisper"  who="#Fenja">
            Ja, aber -
        </cbml:balloon>
        <cbml:balloon xml:id="Us 5"  type="speech"  who="#Michael">
            Und Geschichten kann der sich auch nicht ausdenken
        </cbml:balloon>
    </cbml:panel>
    <cbml:panel n="6" xml:id="Us 5">
        <cbml:balloon xml:id="Us 6"  type="speech"  who="#Fenja">
            Er hat andere Qualitäten
        </cbml:balloon>
    </cbml:panel>
    <cbml:panel n="7" xml:id="Us 6">
    </cbml:panel>
    <cbml:panel n="8" xml:id="Us 7">
        <cbml:balloon xml:id="Us 7"  type="scream"  who="#Fenja">
            Wirst du schon sehen!
        </cbml:balloon>
        <cbml:balloon xml:id="Us 8"  type="speech"  who="#Michael">
            Na dann...
        </cbml:balloon>
    </cbml:panel>
</div>
```

Abbildung A.4.: Zum generierten Comic zugehöriger CBML-Code

B. Abhängigkeiten der MPS-Comic-DSL

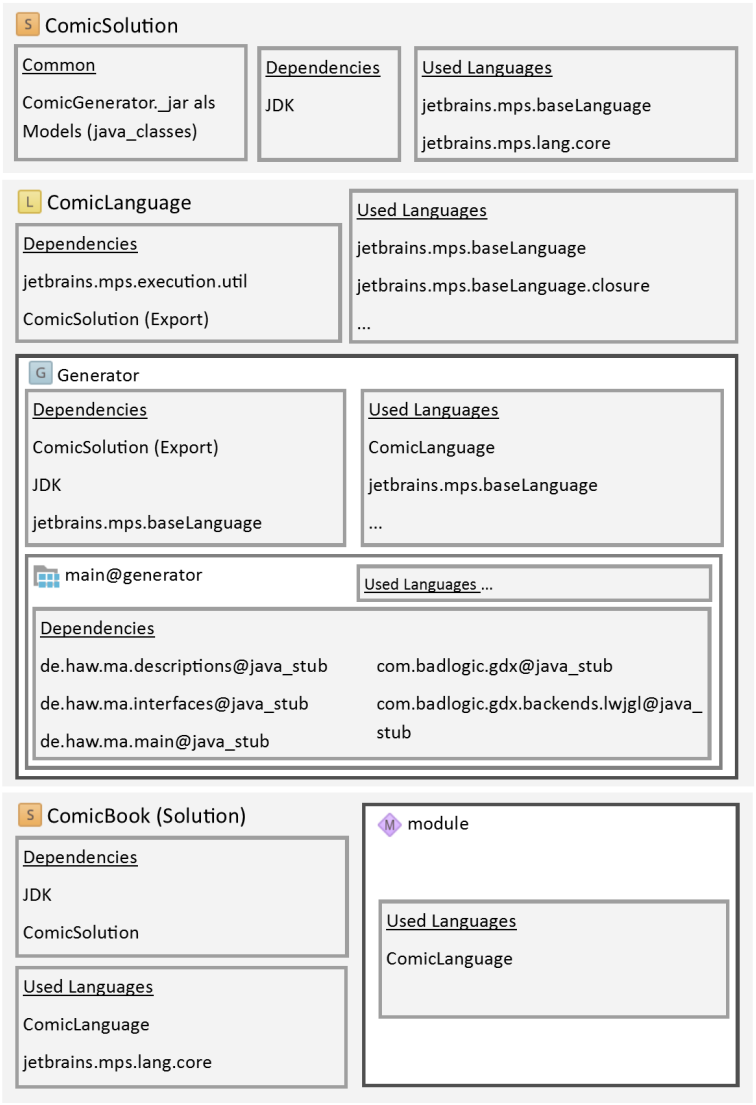


Abbildung B.1.: Abhängigkeiten innerhalb des MPS-Projektes Comic-DSL

C. Anleitung zum Comic Generator

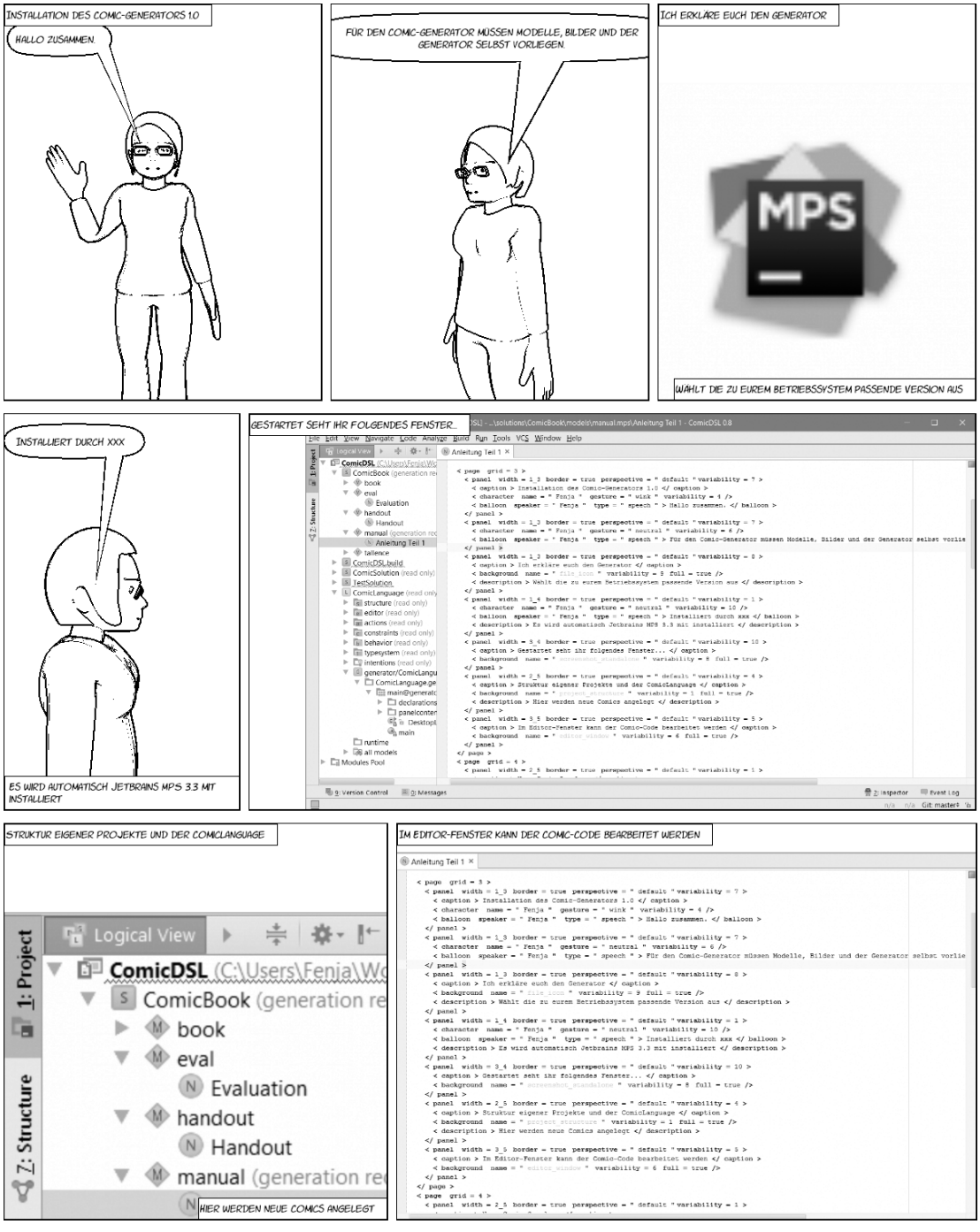


Abbildung C.1.: Anleitung zum Comic-Generator Seite 1

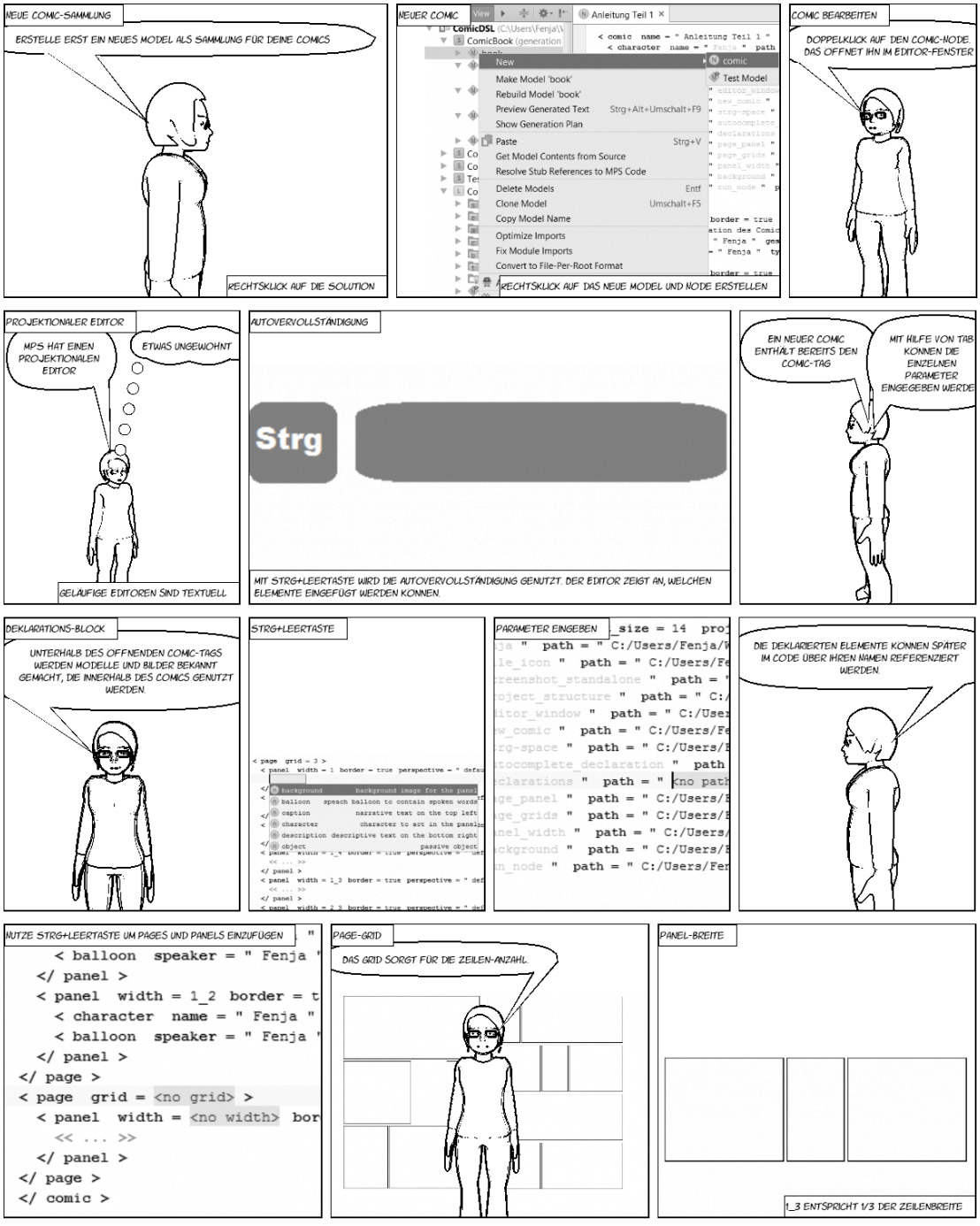


Abbildung C.2.: Anleitung zum Comic-Generator Seite 2

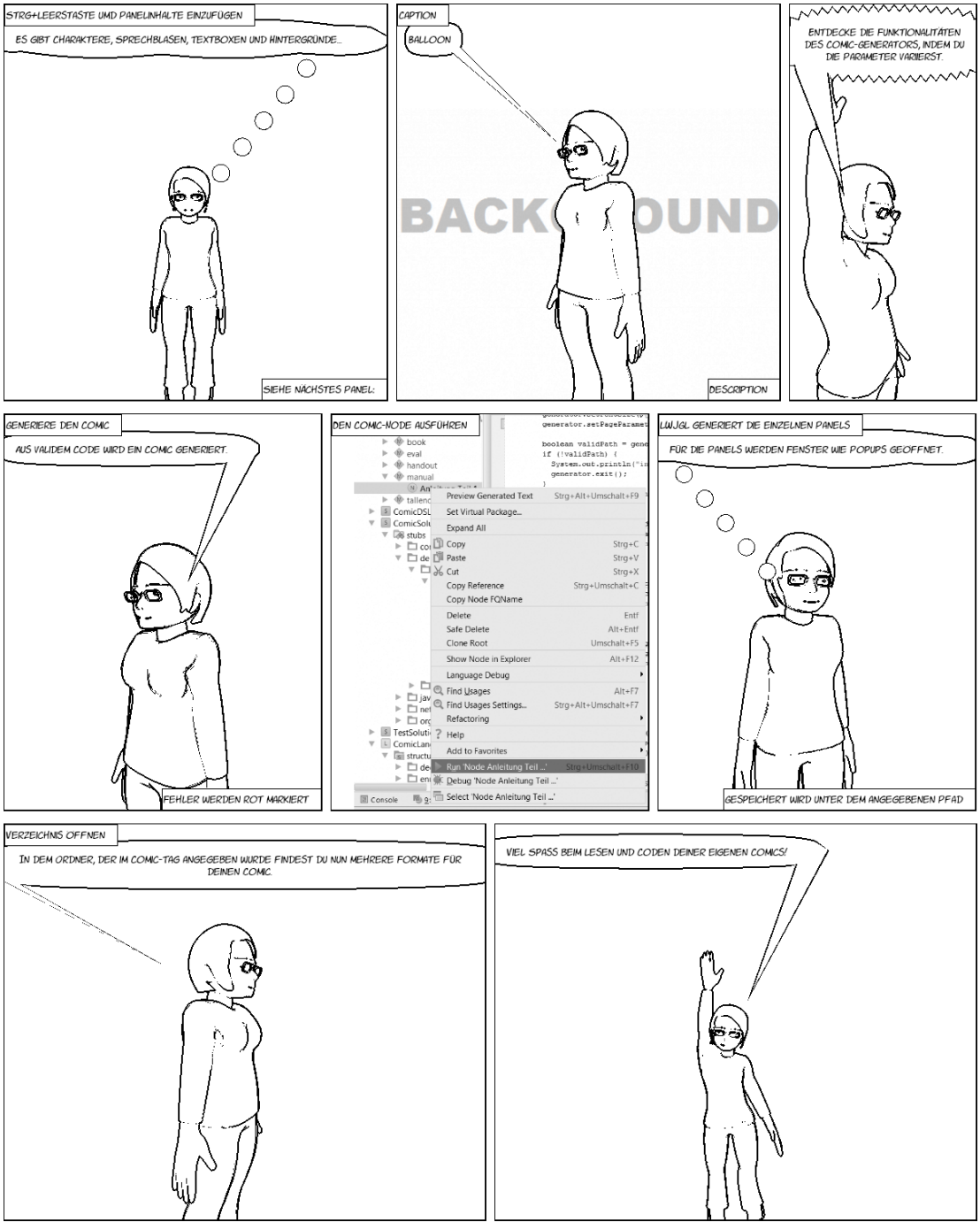


Abbildung C.3.: Anleitung zum Comic-Generator Seite 3

Literaturverzeichnis

[1] P. Decaudin, “Cartoon-looking rendering of 3D-scenes,” Syntim Project Inria, no. June, 1996, pp. 1–11.

[2] S. K. Maurer, “Kognitive Prozesse beim semantischen Verarbeiten von Text-und Bildsequenzen (Comic),” Dissertation zur Erlangung des Grades eines Doktors der Medizin, Universität des Saarlandes, 2010.

[3] S. McCloud, Making Comics: Storytelling Secrets of Comics, Manga and Graphic Novels. HarperCollins, 2011.

[4] N. Cohn, “A visual lexicon,” The Public Journal of Semiotics, vol. I, no. 1, 2007, pp. 35–56.

[5] ——, “Japanese Visual Language: The structure of manga,” Manga: An anthology of global and cultural perspectives, 2010, pp. 187–203.

[6] ——, “Beyond speech balloons and thought bubbles: The integration of text and image,” Semiotica, 2013.

[7] D. Kurlander, T. Skelly, and D. Salesin, “Comic Chat,” in Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, ser. SIGGRAPH ’96, vol. 96. New York and USA: ACM, 1996, pp. 225–236.

[8] J. A. Walsh, “Comic Book Markup Language : An Introduction and Rationale,” Digital Humanities Quarterly (DHQ), vol. 6, no. 1, 2012, pp. 1–50.

[9] S. McCloud, Understanding Comics: The Invisible Art, ser. A Kitchen Sink book. HarperCollins, 1994.

[10] H. Tobita, “Comic Computing: Creation and Communication with Comic,” in Proceedings of the 29th ACM International Conference on Design of Communication, ser. SIGDOC ’11. New York and USA: ACM, 2011, pp. 91–98.

[11] ——, "Comic engine: Interactive system for creating and browsing comic books with attention cuing," in Proceedings of the International Conference on Advanced Visual Interfaces, ser. AVI '10. New York and NY and USA: ACM, 2010, pp. 281–288.

[12] ——, "Comic Computing(TM) beta," 2014. [Online]. Available: http://www.sony.jp/hitokoto/weblabo/comic/details.html

[13] S. Greif and E. Gürbüz, "Sketch-Shading von 3D-Objekten," Paper zur Veranstaltung Vertiefung aktueller Themen in der Computer Graphik, Hochschule Darmstadt, 2007.

[14] K. Finley, "Man Builds Tool for Hacking Comic Strips," 2013. [Online]. Available: http://www.wired.com/2013/02/xkcd-style-comic-generator/

[15] E. Sackmann, "Comic. Kommentierte Definition," pp. 6–9, 2007.

[16] M. Upson and C. M. Hall, "Comic book guy in the classroom: The educational power and potential of graphic storytelling in library instruction," College and University Libraries Section Proceedings, vol. 3, no. 1, 2013, pp. 0–11.

[17] S. McCloud, "The visual magic of comics," 2005. [Online]. Available: http://www.ted.com/talks/scott_mccloud_on_comics

[18] B. Dolle-Weinkauff, "Comic, Manga, Graphic Novel," in Handbuch Kinder und Medien, ser. Digitale Kultur und Kommunikation, A. Tillmann, S. Fleischer, and K.-U. Hugger, Eds. Springer Fachmedien Wiesbaden, 2014, vol. 1, pp. 457–468.

[19] O. Frahm, "Weird Signs," in Theorien des Comics, R. Reichert, Ed. Bielefeld: transcript Verlag, 2011.

[20] W. Eisner, Theory of Comics & Sequential Art. U.S.A.: Poorhouse Press, 1985.

[21] ——, Graphic Storytelling and Visual Narrative (Will Eisner Instructional Books). WW Norton & Company, 2008.

[22] S. McCloud, Reinventing Comics: How Imagination and Technology Are Revolutionizing an Art Form. HarperCollins, 2000.

[23] B. Langlois, C.-E. Jitia, and E. Jouenne, "DSL classification," OOPSLA 7th Workshop on Domain specific modeling, 2007.

[24] S. Efftinge, "Domänenspezifische Sprachen-Sprechen Sie DSL?" JAVA Magazin, no. 12, 2009, p. 14.

[25] V. Subramaniam, "Creating DSLs in Java," 2008. [Online]. Available: http://www.javaworld.com/article/2077865/core-java/creating-dsls-in-java--part-1--what-is-a-domain-specific-language-.html

[26] M. Fowler, "Language Workbenches: The Killer-App for Domain Specific Languages?" 2005. [Online]. Available: http://martinfowler.com/articles/languageWorkbench.html

[27] J. F. Pane and B. A. Myers, "Usability Issues in the Design of Novice Programming Systems," Education, no. August, 1996, p. 78.

[28] T. Akenine-Möller, T. Möller, and E. Haines, Real-Time Rendering, 2nd ed. Natick, MA, USA: A. K. Peters, Ltd., 2002.

[29] P. Hofmann, "Freie Rotation im Raum: Quaternionen und Matrizen," 2009.

[30] W.-t. Chu and C.-h. Yu, "Optimized Speech Balloon Placement for Automatic Comics Generation," in Proceedings of the 3rd ACM International Workshop on Interactive Multimedia on Mobile & Portable Devices, ser. IMMPD '13. New York and NY and USA: ACM, 2013, pp. 1–6.

[31] B. Merkle, "Textual Modeling Tools: Overview and Comparison of Language Workbenches," Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, 2010, pp. 139–148.

[32] S.-A. Pahl, "Entwicklung einer domänenspezifischen Sprache zur Modellierung und Validierung eines Architekturentwurfes nach den Regeln der Quasar-Standardarchitektur," Bachelor Thesis, Hamburg University of Applied Science, 2011.

[33] V. Pech and M. Muhin, "MPS User's Guide - MPS 3.3 Documentation," 2015. [Online]. Available: https://confluence.jetbrains.com/display/MPSD33/MPS+User{%}27s+Guide

[34] J. van Ham, "game from scratch," 2015, web-Blog. [Online]. Available: http://www.gamefromscratch.com/

[35] T. Mullen, Introducing Character Animation with Blender. Wiley Publishing, 2007.

[36] C. Wartmann, Das Blender-Buch. dpunkt, 2011.

Glossar

Abstrakter Syntaxbaum (AST) (en. „abstract syntax tree“) abstrakte Syntax einer Sprache mit baumartiger Hierarchie.

Actions MPS-Modell. Definiert Aktionen, die beim Einfügen neuer Knoten ausgeführt werden.

Caption „Überschrift“ in einem Panel. Meist verwendet zur Beschreibung der Szene / Situation.

Comic Book Markup Language (CBML) Auszeichnungssprache zur Beschreibung der Inhalte von Comics (2.2).

Constraint MPS-Modell. Definiert Regelungen, die innerhalb eines Knoten eingehalten werden müssen.

Dorn (en. „tail“) weist von einer Sprechblase auf den Mund des Sprechers.

Editor MPS-Modell. Definiert die Quellsprache, in welcher der User den Code verfasst.

Generator MPS-Modell. Erstellt Code in der Zielsprache (z.B. Java).

Grid „Gitter“ zur Aufteilung einer Comic-Seite. Geläufig sind 3 mal 3 oder 4 mal 4 Panel.

Hypertext Markup Language (HTML) Auszeichnungssprache, die von Browsern interpretiert werden kann.

Information Visualization (IV) Die Informationsvisualisierung beschäftigt sich mit der computer-gestützten, grafischen Repräsentation von Daten.

Language Workbench (LWB) Software zur Erstellung von DSLs.

Meta Programming System (MPS) PLWB von JetBrains.

OpenGL Shading Language (GLSL) Programmiersprache, um mittels OpenGL auf dem Grafikprozessor eigene Shader-Programme auszuführen.

Panel Logische Einheit eines Comics. Einzelnes, meist umrahmtes Bild, das Objekte und Text darstellen kann.

Projectional Editor Mögliche Knoten sind vorgegeben, die Eingabe modifiziert ohne Parser den Syntaxbaum. Gegenstück ist ein Texteditor.

Projectional Language Workbench (PLWB) LWB mit projektionaler Anpassung der abstrakten Syntax. Nutzt einen projectional Editor.

Speedlines Einer Bewegung folgende Linien, welche diese anzeigen und verstärken sollen.

Sprechblase (en. „balloon“) meist rundes Behältnis für gesprochene Wörter innerhalb eines Comic-Panels.

Struktur MPS-Modell. Abstrakte Syntax, Beschreibung der Elemente einer Sprache.

Text Encoding Intiative (TEI) Standard zur Dokumentenrepräsentation in XML-Dokumenten (2.2.1).

Textual Language Workbench (TLWB) LWB mit textueller Beschreibung der abstrakten Syntax. Umformung unter Einsatz eines Parsers.

Xtext TLWB von Eclipse.

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 26. Mai 2016 Fenja Harbke