Diplomarbeit

Unified Typesetting API

Entwurf einer vereinheitlichten Schriftsatz-Schnittstelle

Diplomand: Christian Ziesemer
Referent: Prof. Dr. Karim Kremer
Korreferent: Prof. Dr. Wolf-Rainer Novender

Sommersemester 2004

Fachhochschule Gießen-Friedberg
Bereich Friedberg
Fachbereich IEM
Fachrichtung Medieninformatik

„Wir haben ganze Kästen voller Buchstaben [...] Wir können jedes beliebige Wort formen.“

— Die volle Wahrheit, Terry Pratchett

Mein Dank geht an Christian Kumpe, Christian Manz, Daniel Müller-Pathle, Tobias Schmid, Andrea Schmideler und Stefan Ziesemer, die diese Arbeit Korrektur lasen. Zusätzlich danken möchte ich Christian Kumpe und meinem Bruder, die mir bei der Beschaffung von Literatur eine unschätzbare Hilfe waren.

Ein Dank geht auch an Gerd Neugebauer, der meine Arbeit um einen hochinteressanten Aspekt bereichert hat.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Vorwort

Dieses Dokument beschreibt die Unified Typesetting API (UTA) in der Version M1. UTA ist eine Schnittstelle für textverarbeitende Anwendungen. Sie assistiert diesen Anwendungen beim Schriftsatz. Der Entwicklungsschwerpunkt liegt bei Modularität und Allgemeingültigkeit. Sie soll einfache wie komplexe Implementierungen gleichermaßen ermöglichen, sowie in interaktiven und nicht interaktiven Textverarbeitungsprogrammen Anwendung finden können.

Zusätzlich zu diesem Dokument wurde im Rahmen der Diplomarbeit Software entwickelt. Sie ist im Internet unter der URL

http://inghuimische.drhuim.de/uta

zu finden. Diese Software ist die Referenzimplementierung der auf den folgenden Seiten vorgestellten Konzepte. Die Software ist ausführlich durch Javadoc-Kommentare dokumentiert, die im Vereichnis `doc/api/` zu finden sind. Weitere Informationen sind der Datei `README` zu entnehmen. Es ist zu empfehlen, mit beiden Teilen parallel zu arbeiten, da das vorliegende Dokument nicht auf alle Details *der Umsetzung* eingeht. Die Software wäre doppelt dokumentiert und zusätzlich würde der Blick auf das eigentliche Konzept versperrt.

Die verwendeten Begriffe in dieser Arbeit lehnen sich stark an Java an. Wird von einem *Interface* gesprochen, sind damit die speziellen Java-Klassen ohne Methodenkörper gemeint. Um Verwirrung zu vermeiden wurde versucht ohne den Begriff auszukommen und statt dessen *Klasse* zu verwenden. Im Gegensatz dazu bezeichnet API oder Schnittstelle die Gesamtheit aller entwickelten Klassen und ist damit die Schnittstelle nach außen, wie sie von anderen Programmen verwendet werden muß.

Das Englische ist reich an Wörtern, mit denen sich zwischen einzelnen Schriftarten (Fonts) und Schriftsystemen (Scripts) unterscheiden läßt. Im Deutschen bezeichnet Schrift in der Typografie eine Schriftart, sprachwissenschaftlich ein Schriftsystem. Um hier Mißverständnisse zu vermeiden wird in dieser Arbeit Schrift ausschließlich als Synonym für Schrift-

system verwendet. Gelegentlich wird auch auf den englischen Begriff Font für Schriftart zurückgegriffen. Ein Schriftformat schließlich bezeichnet ein Dateiformat in dem Schriftarten gespeichert werden.

1 Einleitung

Bereits in den 60er Jahren des letzten Jahrhunderts wurde erfolgreich an der digitalen Umsetzung des von Gutenberg erfundenen Schriftsatzes geforscht. Wo heute mit dem Erstellen und Veröffentlichen von Texten Selbstverständlichkeit verbunden ist, stellte sich damals die Frage nach der generellen Machbarkeit. Vor rund 25 Jahren präsentierte Donald Ervin Knuth mit TEX [2, 3] ein Programm, dessen ausgefeilte Algorithmen noch heute das Maß für qualitativen Schriftsatz sind. Sie waren das Resultat jahrelanger Forschung und sind in Mathematik gegossene Regeln zur Erzeugung ästhetischer Schriftstücke. Um so verwunderlicher erscheint es, daß dieses Wissen in der Zwischenzeit nicht in alle Bereiche der digitalen Datenverarbeitung Einzug gehalten hat. Ästhetik ist schließlich nicht an ein Medium gebunden. Nicht nur Bücher und Zeitschriften profitieren davon. Auch grafische, interaktive Anwendungen und Web-Seiten, schlicht jeder Text. Als logische Konsequenz sollte jedes von einem Computer verarbeitete Stück Text höchsten Ansprüchen genügen. Dem ist nicht so und es gibt auch Gründe dafür.

Schrift ist ein vom Menschen geschaffenes, visuelles Kommunikationsmittel. Allein diese Tatsache ist für die Auswirkung auf die Definition von qualitativ hochwertigem Schriftsatz und dem damit verbundenen Aufwand der Verarbeitung verantwortlich. Die Jahrtausende andauernde, unabhängige Entwicklung verschiedener Schriftsysteme hat zu komplexen Regelwerken geführt wie eine Sprache zu Papier gebracht wird. Der Mensch schlug dabei alle nur denkbaren Wege ein. Nicht nur Schreibrichtung und Schriftzeichen unterscheiden sich, sondern auch das zu Grunde liegende System wie die einzelnen Schriftzeichen Bedeutung erlangen. In manchen Schriften erhält erst eine Reihe von Zeichen eine Bedeutung (Wort), in anderen reicht ein Zeichen (Ideogramm) aus. Diese kulturelle Entwicklung zu mechanisieren macht den Schriftsatz aufwendig und verursacht daher Kosten. Diese Kosten spiegeln sich in Arbeitszeit und Rechenleistung wider. Mehr und mehr wird ersteres zum begrenzenden Faktor. Die Gründe sind damit ökonomische und haben guten Schriftsatz in die Nische der Printmedien abgedrängt. Und selbst dort wird, aus Kostengründen, immer öfter auf ihn verzichtet.

Es liegt in der Natur des Menschen aufwendige Arbeit zu delegieren und zu automatisieren. Wo sich Knuth noch mit einer 95%igen Automatisierung begnügt und die restlichen fünf Prozent, den Feinschliff, selbst erledigen will, ist heute 100%ige Automatisierung gefordert – wenn auch nicht immer möglich. TEX hat sich als überaus stabil und erweiterbar erwiesen und wird heute noch viel verwendet – in der schnellebigen Softwarewelt eine Seltenheit. Während TEX eine Konstante darstellt, hat sich die Umgebung in der es eingesetzt wird stark verändert. Dokumente werden nicht mehr nur für ein Medium erstellt, sie sollen wiederverwertbar und automatisch verarbeitbar sein. Das Mittel der Wahl dazu ist XML. Die Konsequenz sind Lösungen, die nur bedingt ansprechende Qualität bieten können, oder die TEX im Hintergrund einsetzen und die Eingabe zuvor in TEXs eigene Sprache übersetzen müssen.

Dabei ist TEX natürlich nicht die einzige Software die es versteht Buchstaben auf anspruchsvolle Weise zu platzieren. In den grafischen Oberflächen von Linux, MacOS und Windows arbeiten immer komplexer werdende Schnittstellen, um auch andere, nicht lateinische Sprachen darstellen zu können. Gerade die Schnittstelle unter MacOS, ATSUI[1], bietet hier auch Funktionen für anspruchsvollen Schriftsatz. Im Printsektor sind Adobe und Quark die Hauptkonkurrenten. Ihre Produkte befriedigen ebenfalls hohe Ansprüche. Zudem hat hier Adobe mit dem Portable Document Format das Austauschformat für Druckmedien schlechthin entwickelt. Auch bei den Cross-Media-Publishing Funktionen dieser Produkte tut sich etwas, wenngleich hier sicherlich noch nicht alle Möglichkeiten ausgereizt sind.

Alle diese Anwendungen haben das Problem Schriftsatz für sich gelöst. Die meisten sind proprietär oder bei bestimmten Aspekten schlecht erweiterbar. Zudem sind sie wenig modular. Einzelne Komponenten können nicht, oder nur mit großem Aufwand ausgetauscht werden. Ziel dieser Arbeit ist es, eine Schnittstelle zu formulieren, die auf eine flexible Weise Lösungen für immer wieder auftretende Probleme beim Schriftsatz anbietet. Es gilt zunächst diese Probleme zu lokalisieren und in einer Schnittstelle zu modellieren. Die Implementierung ist dabei als Referenzimplementierung in einer bestimmten Programmiersprache (Java) zu sehen, die gemachten Aussagen sind allgemeingültig und somit auf andere Programmiersprachen übertragbar. Das Ziel bei der Umsetzung der Schnittstelle in Java ist ein modulares und erweiterbares Design. Applikationen, die auf dieser vereinheitlichten Schriftsatz Schnittstelle (Unified Typesetting API,

[1] URL http://developer.apple.com/intl/atsui.html

UTA) aufbauen, sollen dies auf einer möglichst stabilen Grundlage tun. Refaktorisierung und Redesign sollen weitestgehend vermieden werden. Wichtig ist es daher, daß Implementierungen verschiedener Qualitätsstufen möglich sind. Gleichzeitig soll aber auch das Austauschen einfacher gegen komplexere Komponenten möglich sein. So läßt sich mit einer einfachen Implementierung beginnen, die nach und nach mehr typografische Funktionen unterstützt. Dies erlaubt auch das Maßschneidern eines eigenen Schriftsatzsystems.

Zur Gliederung dieser Arbeit: sie ist in sechs Hauptkapitel unterteilt. Nachfolgend eine kurze Beschreibung was in jedem dieser Kapitel besprochen wird.

Herausforderung Schrift Geht auf verschiedene Schriftsysteme ein und zeigt deren Besonderheiten, die ein Satzsystem zu berücksichtigen hat.

Technischer Hintergrund Liefert die notwendigen technischen Aspekte, die in Zusammenhang mit UTA stehen. Hilft auch dabei UTA gegenüber bestehenden Technologien abzugrenzen bzw. Parallelen zu zeigen. Desweiteren gibt das Kapitel einen Überblick über das Grobdesign von UTA.

Das Umbrechen von Absätzen Bespricht das Umbruchmodul in UTA und geht auf die Umbruchproblematik an sich ein.

Der Satzvorgang Behandelt den Satzvorgang wie er in UTA abläuft und die damit verbundenen Klassen.

Qualitätsmanagement Dokumentiert die Qualitätsmanagement-Komponente in UTA. Sie dient dazu Aussagen über Funktionsumfang und Qualität eines Satzsystems machen zu können.

Ausblick Behandelt Ergänzungsmöglichkeiten zum aktuellen Stand und Schildert derzeit in UTA vorhandene Einschränkungen.

2 Herausforderung Schrift

Dieses Kapitel widmet sich Eigenschaften einzelner Schriften und damit verbundenen Auswirkungen auf ihren Satz. Aus der stark europäischen und amerikanisch geprägten Entwicklungsgeschichte des Computers ergibt sich noch immer eine gewisse Problematik im Umgang mit anderen Sprachen und Schriften, die nicht auf dem lateinischen Alphabet beruhen.

Bevor auf einige Sprachen näher eingegangen wird, zunächst ein paar Worte wie Schriften kategorisiert werden. Zu unterscheiden sind drei Arten (vgl. [4]):

1. Alphabetschriften, bei denen ein Zeichen einem Laut entspricht (wie Deutsch oder Arabisch)

2. Silbenschriften, bei denen ein Zeichen einer Silbe zugeordnet wird. Darunter fallen zwei japanische Schriften, wie wir gleich sehen werden.

3. Ideographische Schriften. Dort wird jedem Zeichen ein Begriff (Wort) zugewiesen, ein Beispiel ist Chinesisch.

Praktisch kommen meist Mischformen oder Abwandlungen dieser drei Kategorien vor.

Exemplarisch einige Schriften, die in der mulitlingualen Textverarbeitung oft für Beispiele herangezogen werden, da sie außergewöhnliche Merkmale aufweisen.

2.1 Thai

Einige nah- und fernöstliche Schriften werden aufgrund ihrer Schreibregeln als komplex bezeichnet. Darunter fallen Arabisch und Thai. Da deren Satz besondere Anforderungen an ein Satzsystem stellt, ist das nicht von der Hand zu weisen, andererseits erscheinen einem Thailänder Trennregeln, wie sie gerade die alte deutsche Rechtschreibung vor 1996 vorschrieb, wahrscheinlich nicht weniger komplex.

ไทย

Abbildung 2.1: Thai

Die thailändische Schrift kann als eine der am schwersten zu erlernenden bezeichnet werden und stellt besondere Anforderungen an ein Satzsystem [5]. So werden im thailändischen fünf verschiedene Betonungen einer einzelnen Silbe unterschieden, die in der Schrift entsprechend kenntlich gemacht werden müssen[1]. Dies erledigen diakritische Betonungszeichen (Häkchen, Striche, usw. die an oder um ein Schriftzeichen platziert werden) über den einzelnen Schriftzeichen. Auch Vokale werden auf diese Weise rund um einen Konsonanten positioniert (links, rechts, darüber und darunter). Desweiteren wird in Thai meist kein Worttrennzeichen verwendet. Leerzeichen finden sich, wenn überhaupt, nur zwischen vom Schreiber definierten Sinnabschnitten. Das hat Auswirkungen auf den Zeilenumbruch, der nur wörterbuchbasiert implementiert werden kann.

2.2 Arabisch

Arabisch ist eine Sprache die bidirektional ist. Text wird von rechts nach links, Zahlen hingegen von links nach rechts geschrieben. Ein Satzsystem muß solche Richtungsänderungen erkennen und verarbeiten. Die gleichen Anforderungen ergeben sich auch wenn deutscher und arabischer Text gemischt wird. Zudem sind beliebige Verschachtelungen von Schreibrichtungen denkbar links nach rechts (LNR) Text, der innerhalb eines rechts nach links (RNL) Textes zitiert wird, der wiederum selbst von einem LNR-Text umgeben wird [6][7].

اللغة العربية

Abbildung 2.2: Arabisch

Arabische Buchstaben können je nach Kontext unterschiedliche optische Repräsentanten haben, je nach dem, ob sie allein, am Anfang, am Ende

[1] Beispiele zu den hier erwähnten Schriftsystemen bietet die Webseite Omniglot im Internet unter http://www.omniglot.com

oder in der Mitte eines Wortes stehen. Zur Unterscheidung von arabisch-indischen Zahlen wird bei manchen Zeichen eine weitere Zeichenvariante benötigt. Zudem ist arabisch eine Schreibschrift, einzelne Schriftzeichen müssen ineinander übergehen. Der Blocksatz wird im Arabischen erzielt, indem die Darstellung der einzelnen Buchstaben gedehnt wird, um auf gleichlange Zeilen zu kommen. Trennung von Wörtern ist im Arabischen nicht möglich. Ligaturen sind zur korrekten Darstellung zwingend notwendig.

2.3 Nastaliq und Devanagari

Aus dem Arabischen abgeleitet ist Nastaliq, eine Schrift, in der mehrere indische Sprachen, u. a. Urdu, geschrieben werden. Wie Arabisch ist Nastaliq ebenfalls eine Schreibschrift. Als Besonderheit werden Schriftzeichen nicht zwangsläufig auf einer horizontalen Grundlinie platziert. Werden Schriftzeichen kombiniert, kann auch ein vertikaler Versatz auftreten. Handelt es sich in europäischen Schriften bei der Grundlinie tatsächlich um eine horizontale Linie, könnte man die Nastaliq-Grundlinie grob mit einer Sägezahnkurve vergleichen[2].

Abbildung 2.3: Devanagari

Devanagari ist ein weiteres, in Indien gebräuchliches Schriftsystem mit einer außergewöhnlichen Grundlinie. Sie wird als hängend bezeichnet, die einzelnen Schriftzeichen werden wie an einer Wäscheleine aufgehängt, wie Abbildung 2.3 zeigt.

2.4 Chinesisch

1955 reformierte China sein Schriftsystem [8]. Aus dem bis dahin gebräuchlichen traditionellen Chinesisch mit seinen Langzeichen ging das vereinfachte Chinesisch hervor, im EDV-Bereich oft englisch als *Simplified Chinese* anzutreffen. Diese Reform ging nicht nur mit einer Vereinfachung der Ideogramme einher, sondern auch mit einer Änderung der Schreibrichtung. Während traditionell in Spalten von oben nach unten geschrieben

[2]Unter http://www.omniglot.com/writing/urdu.htm findet sich ein Beispiel

wird, die von rechts nach links angeordnet werden, wird im vereinfachten Chinesisch, wie im Deutschen, in Zeilen von links oben nach rechts unten geschrieben. Da inzwischen wieder vermehrt auf die traditionelle Schreibweise zurückgegriffen wird, finden sich in Zeitungen beide Methoden.

2.5 Japanisch

Einen Schritt komplexer als Chinesisch ist die japanische Schreibkultur. Sie ist gekennzeichnet durch Aufgeschlossenheit anderen Schriftkulturen gegenüber und einer gewissen Freude am Experiment mit der eigenen. Dies hat dazu geführt, daß insgesamt vier Schriftsysteme Anwendung finden, schließt man das lateinische Alphabet mit ein [9].

日本語

Abbildung 2.4: Japanisch

Die Japaner besitzen sowohl eine vom Chinesischen beeinflußte Ideogrammschrift namens Kanji, als auch eine an Sanskrit orientierte Silbenschrift (Kana, die sich aus Kanji entwickelte). Kana läßt sich weiter in Hiragana und Katakana unterteilen. Abbildung 2.4 zeigt japanisch in Kanji.

Ein interessanter Aspekt ist, daß Katakana in Texten zur Hervorhebung verwendet wird, ähnlich wie kursiv gestellter Text im Deutschen. Im Japanischen wird also nicht die Zeichenform variiert sondern es werden gleich andere Schriftzeichen benutzt. Technisch ergeben sich aus der Tatsache, daß in japanischen Texten vier verschiedene Zeichensysteme parallel vorkommen können keine weiteren Anforderungen, auch kursive Schriftzeichen müssen entweder durch eine andere Schriftart oder einen anderen Zeichencode verfügbar gemacht werden.

2.6 Lateinische/Europäische Schriften

Lateinische Schriften haben ebenfalls ihre Eigenheiten. Diakritische Zeichen sind in den deutschen Umlauten zu finden, ebenso wie die Trennung von Wörtern, oder kontextabhängige Formen von Schriftzeichen. Im Griechischen hat das Sigma am Wortende eine andere Form als in der Mitte: „ς“ vs. „σ“. Im Altdeutschen wird in Abhängigkeit von Lautverbindungen

bzw. der Position im Wort das lange S „ſ“ oder das kurze S „ſ“ verwendet (vgl. [10], [11, S. 75-76]). Zudem lernt jedes Kind auch bei uns zwei Schriftsysteme, die Blockschrift und die Schreibschrift.

2.7 Zusammenfassung

Der XSL-Standard [12] beschreibt im Anhang A *Additional ‚writing-mode‘ values* weitere Schreibrichtungen. Darunter auch Richtungen, in denen von unten nach oben geschrieben wird. Wie groß die praktische Relevanz ist läßt sich nicht sagen, da keine Sprachen angegeben sind, die diese Schreibrichtungen verwenden. Zusammengefasst können jedoch gewisse Anforderungen definiert werden:

- Verschiedene Sprachen kombinieren zwei oder mehr Zeichen zu einem. Eine solche Kombination mag optional sein (Ligaturen im Deutschen) oder zwingend (wie im Arabischen).
- Diakritische Zeichen müssen in beliebiger Anzahl beliebig platzierbar sein.
- Ein Zeichen kann je nach Kontext eine andere Darstellung verlangen.
- Nicht jede Sprache benutzt Worttrennzeichen, Umbrechen von Text wird dadurch komplizierter.
- Einige Sprachen erlauben die Trennung von Wörtern, wobei ein Trennsymbol eingesetzt werden kann.
- Blocksatz wird unterschiedlich realisiert, mit Wortzwischenräumen, Zeichenzwischenräumen oder durch Dehnung/Änderung von Schriftzeichen.
- Eine Schriftart reicht evtl. nicht aus um alle benötigten Schriftzeichen bereit zu stellen, da eine Schriftart nicht alle Zeichen der Welt vereinen kann.
- Je nach Sprache unterscheiden sich Interpunktions- und Umbruchregeln.
- Schriftzeichen werden nicht ausschließlich entlang waagrechter oder senkrechter Linien platziert. Es gilt unterschiedliche Grundlinien und Schreibrichtungen zu beachten. Unter Umständen kommen in einem Dokument mehrere Schreibrichtungen gemischt vor.

Es läßt sich erahnen, daß ein multilinguales Satzsystem bei der Platzierung einzelner Schriftzeichen sehr flexibel sein muß und eine alleinige Konzentration auf eine Schreibrichtung Einschränkungen mit sich bringt, die nachträglich schwer zu eliminieren sind.

3 Technischer Hintergrund

Dieses Kapitel beschäftigt sich mit Technologien, die in Zusammenhang mit UTA stehen. Sei es direkt, weil sie benutzt werden, oder indirekt, weil sie ähnliche Probleme behandeln. Zudem hilft dieses Kapitel UTA in das Verhältnis zu anderen Technologien zu setzen. Das klärt die Frage welche Funktionen UTA zur Verfügung stellt und welche Funktionen bereits anderweitig vorhanden sind und deshalb nicht Teil von UTA sind.

Zunächst zu drei grundlegenden Themen. Zu den Formaten wie Schriftzeichen und deren grafische Repräsentation für die Speicherung auf Datenträgern kodiert werden. Danach zu Technologien, an denen sich UTA orientiert bzw. die UTA einsetzen können. Schließlich wird auf das Grobdesign von UTA eingegangen.

3.1 Character-Glyph-Modell

Zunächst zu der wichtigen Unterscheidung zwischen Schriftzeichen und Glyphen, auch als Character-Glyph-Modell bezeichnet. Die Kernforderung des Modells ist die Trennung von Zeichen und diese Zeichen repräsentierenden Glyphen. Während ein Schriftzeichen der logische Repräsentant eines Schriftelements ist, ist ein Glyph der optische Repräsentant.

Am ersichtlichsten wird der Unterschied zwischen Zeichen und Glyph, wenn man das gleiche Zeichen in verschiedenen Schriftarten betrachtet (Abbildung 3.1). Alle stellen das gleiche Zeichen, „M“ dar, allerdings mit unterschiedlichen Formen. Ursprünglich gab es diese Unterscheidung nicht. Lange Zeit war der Zeichencode identisch mit dem Index des Glyphen in

Abbildung 3.1: Verschiedene Glyphen, gleiches Zeichen. Was Schriftarten unterscheidet sind ihre unterschiedlichen Repräsentationen ein und desselben Schriftzeichens.

der Schriftart. In modernen Schriftformaten ist dies nicht mehr unbedingt der Fall.

3.2 Unicode

Unicode ist der Versuch, alle auf der Welt verwendeten Schriftzeichen in einer großen Zeichenreferenz zu vereinigen. Damit wird der kontextabhängigen Interpretation von ASCII codierten Texten ein Ende gesetzt, bei der ein ASCII-Code je nach Sprache ein anderes Schriftzeichen repräsentiert. Unicode bietet in Version 4.0 [14] Platz für 1.114.112 Zeichen (0-10FFFF in hexadezimaler Schreibweise).

3.2.1 Code Points

Die Repräsentation eines Zeichens übernimmt ein *Code Point*. Code Points werden als U+⟨Ziffer⟩ notiert. ⟨Ziffer⟩ ist dabei eine sechsstellige Hexadezimalzahl. Um Unicode speichern zu können wird zwischen unterschiedlichen Kodierungen unterschieden. Diese sind UTF-8, UTF-16 und UTF-32. Bei UTF-8 wird minimal ein Byte zur Kodierung benötigt, maximal sind es vier. UTF-16 benötigt minimal zwei, maximal vier Bytes, UTF-32 schließlich verwendet stets vier Bytes. Die Kodierung entspricht *nicht* einem einfachen Aufteilen der sechsstelligen Hexzahl (Code Point) in einzelne Bytes.

Um in Java einen nicht über die Tastatur erreichbaren Code Point anzusprechen, wird in einem String `\u`⟨`ziffer`⟩ notiert. Java verwendet intern UTF-16. Die für Zeichen vorgesehene Datenstruktur `char` repräsentiert nur 16 Bit und kann somit lediglich $2^{16} = 65536$ verschiedene Werte annehmen. Daher wird in Java keine sechsstellige, sondern lediglich eine vierstellige ⟨Ziffer⟩ interpretiert. Um Code Points über `\uFFFF` hinaus verwenden zu können, müssen zwei Ersatzzeichen miteinander kombiniert werden, sog. *surrogate characters*.

Neben Schrift- und Steuerzeichen existieren weiter Code Points, die bei der Interpretation von Texten hilfreich sind. Ein Beispiel dafür ist der Object Replacement Character (U+FFFC), der einen Platzhalter für ein einzubindendes Objekt darstellt. Oftmals werden Grafiken oder Formeln in den Text eingebettet. Diese komplexen Strukturen lassen sich nur so sinnvoll in den Textfluß integrieren.

Die Unicode-Zeichentabelle beinhaltet nicht nur Code Points für ganze Zeichen, sondern auch für Teile von Zeichen. Gerade in mathemati-

schen Formeln ist es notwendig bestimmte Glyphen zusammenbauen zu können. Hier reicht ein einfaches Glyph, das gedehnt wird, nicht aus, da es zu sehr unansehnlichen Ergebnissen kommt, wie Abbildung 3.2 zeigt. Links die geschweifte Klammer in Schriftgröße 12, wie sie auf der Tastatur direkt abgerufen werden kann. Rechts daneben die gedehnte Variante. Die Einzelteile der zusammengesetzten Klammer sind ebenfalls jeweils in Schriftgröße 12.

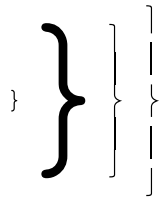


Abbildung 3.2: Gedehnte Klammer und aus mehreren Einzelteilen zusammengesetzte Klammer. Die gedehnte Klammer ist viel zu kräftig. Rechts die Einzelteile, aus denen die mittlere Klammer zusammengesetzt ist.

In erster Linie sind die Glyphen die zusammengebaut werden Klammern, Integral-, Summen- oder Wurzelzeichen, die mitunter sehr große Ausdrücke umspannen müssen. Da mathematische Formeln nur auf diese Weise setzbar sind und die Einzelteile nur innerhalb einer Schriftart zusammenpassen, muß Unicode derartige Code Points bereitstellen, auch wenn das Ergebnis eine geschweifte Klammer ist, für die es einen eigenen Code Point gibt.

3.2.2 Anhänge und Metadaten

Unicode stellt nicht nur für jedes Schriftzeichen einen Code zur Verfügung, sondern auch Zusatzinformationen zu jedem dieser Zeichen und empfiehlt Methoden, wie diese Zeichen verarbeitet werden sollten. Die Zusatzinformationen dienen dabei hauptsächlich der Kategorisierung. Wichtige Kategorien sind Buchstabe, Ziffer, Trenner und Interpunktion. Andere Zusatzinformationen sind für Zeilenumbruchalgorithmen von Bedeutung.

Die Methoden wie Teilprobleme des Schriftsatzes angegangen werden sollen, werden in Anhängen beschrieben, den sog. Unicode Standard Annexes, kurz UAX. Tabelle 3.1 führt die für diese Arbeit relevanten mit zugehöriger Nummer auf.

Es ist nicht das Ziel all diese Anhänge, jeder für sich sehr komplex und umfangreich, ausführlich zu diskutieren. Vielmehr geht es darum die mit

UAX	9	The Bidirectional Algorithm
UAX	14	Line Breaking Properties
UAX	15	Unicode Normalization Forms
UAX	29	Text Boundaries

Tabelle 3.1: Relevante Unicode Standard Annexes (UAX)

den Anhängen verbundenen Probleme und die Lösung kurz zu erläutern um an geeigneter Stelle auf sie verweisen zu können.

3.2.3 Bidirektionaler Textfluß

Das bidirektionaler (bidi) Text besondere Ansprüche an ein Satzsystem stellt wurde schon im Kapitel *Herausforderung Schrift* klar. Deshalb bietet Unicode in Anhang 9 [7] einen Algorithmus als Hilfestellung. Um den Algorithmus verstehen zu können, zunächst eine wichtige Unterscheidung in der Reihenfolge, wie Zeichen aneinandergereiht werden.

Logische und optische Reihenfolge

Im Schriftsatz ist eine Unterscheidung zwischen zwei Reihenfolgen der einzelnen Schriftzeichen notwendig. Hier und in der restlichen Arbeit kennzeichnet ɹɔɐidɐɹA arabischen Text. Die „Sprache“ wird durch gespiegelte Schriftzeichen dargestellt und von rechts nach links gelesen.

Die logische Reihenfolge beschreibt die Reihenfolge der Zeichen im Speicher. Sie sind so abgelegt, wie sie eingegeben werden. Das im Textfluß folgende Zeichen steht hinter seinem Vorgänger. Das „A“ wurde nach dem „h“ und dem Leerzeichen eingetippt und steht daher auch im Speicher hinter diesen Zeichen:

Deutsch AɹɐdiɐɔɹI

Zur besseren Unterscheidung wird Text in logischer Reihenfolge in einer `diktengleichen` Schriftart gesetzt.

Bei der optischen Anordnung folgen die Zeichen so aufeinander, wie sie auf einem Bildschirm dargestellt werden. Hier folgt auf „h“ und Leerzeichen von Deutsch, das „ɹ“ von ɹɔɐidɐɹA.

Deutsch ɹɔɐidɐɹA

Die Umstellung von der logischen in die optische Reihenfolge wird als optische Umordnung bezeichnet. Dieser Umordnung kommt ein erheblicher Teil des Bidi-Algorithmus zu.

Der Bidi-Algorithmus

Der Algorithmus arbeitet dreistufig. Zunächst teilt er den Text in Absätze auf. Auf jeden dieser Absätze wird dann der eigentliche Kernalgorithmus angewandt. Dieser besteht aus Schritt zwei, der Berechnung der verschiedenen Embedding-Level, und aus Schritt drei, der optischen Umordnung. Ein Embedding-Level gibt an, wie tief ein Zeichen in umgebenden Text eingebettet ist. Im Folgenden wird die Zahl, die diese Tiefe angibt, als Ordnung bezeichnet. Die Ordnung ist umso höher, je größer die Zahl ist. Gleichzeitig gilt: Ist das Embedding-Level gerade handelt es sich um Text mit der Leserichtung links nach rechts, ist es ungerade, ist die Leserichtung rechts nach links. Zahlen erhalten stets ein Level, das gerade und höher ist als das des umgebenden Textes. Der Grund dafür ist, daß auch arabische und hebräische Zahlen von links nach rechts geschrieben werden. Obiges Beispiel hat folgende Embedding-Level (Ordnung 0, bzw. 1):

```
Deutsch ɐɹɐqıɔǝɥ
0000000011111111
```

Bei einer Zahl würde es wie folgt aussehen:

```
Deutsch 123
00000000222
```

Das Finden der Embedding-Level funktioniert in den meisten Fällen ohne weiteres Zutun des Benutzers, es ist keine weitere Auszeichnung des Textes als Hilfestellung notwendig. Um das zu erreichen, teilt der Algorithmus Zeichen in verschiedene Kategorien ein, die angeben, wie stark ein Zeichen einer Leserichtung zugeordnet wird. Ein starkes Zeichen prägt bspw. neutralen Zeichen das Embedding-Level auf. In obigem Beispiel ist das Leerzeichen ein solches neutrales Zeichen. Für schwierige Situationen berücksichtigt der Algorithmus spezielle Unicode Code Points, die die vom Algorithmus festgestellte Richtung überschreiben können. Diese liegen zwischen U+202A Left-To-Right Embedding bis U+202E Right-To-Left Override einschließlich.

Im dritten Schritt werden die einzelnen Zeichen von der logischen in die optische Reihenfolge gebracht. Dieser Schritt wird laut Unicode nach dem Zeilenumbruch auf die einzelnen Zeilen angewandt. Die komplette Umordnung erfolgt, in dem vom höchsten gefundenen Level ausgehend, jedes Level mit untergeordneten Leveln umgeordnet wird. Ist das niedrigste Level 0 erreicht, ist keine weitere Umordnung notwendig.

```
Deutsch ARABISCHER 123 TEXT
00000000111111111122211111
Deutsch ARABISCHER 321 TEXT
```

Zuerst wird Level 2 umgeordnet, dann Level 1 mit dem untergeordneten Level 2. Der Text befindet sich in der richtigen Reihenfolge:

Deutsch TEXT 123 ARABISCHER

Soviel zunächst zum Unicode Bidi-Algorithmus, im Kapitel *Satzvorgang* werden wir sehen, welche dieser Schritte für UTA in welcher Form relevant sind und wann sie durchgeführt werden.

3.2.4 Mögliche Umbrüche, Wort- und Satzgrenzen

Der Unicode Anhang 14 beschreibt Eigenschaften von Zeichen, die wichtig sind um gültige Umbruchstellen zu finden. Desweiteren beschreibt er einen Algorithmus, der diese gültigen Umbruchstellen findet. Was dieser Algorithmus nicht macht, dafür aber eine Kernfunktion von UTA ist, sagt dieser Satz:

> „The definition of optimal line break is outside the scope of this document. Different formatting algorithms may use different methods of determining an optimal break.“ [15, Kap. 3]

Die gültigen Umbruchstellen, die dieser Algorithmus findet sind eine nahezu vollständige Untermenge aller möglichen Umbruchstellen. UTA wiederum kümmert sich nicht darum mögliche Umbruchstellen zu finden, sondern nur die optimalen. UTA stellt eine Schnittstelle bereit um qualitativ unterschiedliche Umbruchalgorithmen zu ermöglichen. Deren Qualität entscheidet darüber, wie optimal die gewählten Umbrüche unter dem Gesichtspunkt der Ästhetik sind.

Sehr eng mit UAX 14 verwandt ist der Anhang 29 [16]. Er beschreibt wie in einem Unicode-Zeichenstrom zusammengehörende Zeichen gefunden werden können, die

- ein zusammengesetztes Glyph,
- ein Wort oder
- einen Satz bilden.

In besonderem Maße sind diese Positionen im Strom für Editoren wichtig. Häufig wird in solchen Anwendungen mit einem Doppelklick ein Wort, mit einem Dreifachklick der ganze Satz markiert. Derartige Funktionen sind nicht Bestandteil von UTA, da sie in einem nicht interaktiven Textprozessor nicht benötigt werden. Allerdings ist es typografisch vertretbar hinter einem Punkt, der einen Satz beendet, etwas größeren Leerraum zu lassen, als bspw. hinter einem Komma oder einer Abkürzung. TEX geht z. B. so vor. Fehlerfreiheit kann die Erkennung von Satzgrenzen dabei ohne semantische Analyse nicht garantieren.

Wichtig sind diese Anhänge, weil sie nicht nur Umbrüche für lateinische Schriften finden, sondern auch für ostasiatische.

3.2.5 Normalisierung

Unicode Anhang 15 [17] beschreibt Methoden, wie Unicode Text normalisiert werden kann. Die Normalisierung ist deshalb wichtig, weil ein bestimmtes Glyph auf mehrere Weisen in Unicode codiert werden kann. Wie erwähnt bietet Unicode für jedes Schriftzeichen einen Code an. Also sowohl für diakritische Zeichen als auch für die Kombinationen eines solchen Zeichens mit einem normalen Buchstaben. Der Umlaut „ö“ hat ebenso einen eigenen Code Point wie die beiden Zeichen „o“ und die Diärese „¨“ (Combining Diaeresis, U+0308) und kann folglich als „ö“ bzw. „o + ¨“ dargestellt werden.

Die Normalisierung unterscheidet insgesamt vier Vorgehensweisen, die sog. *Normalization Forms*: NFD, NFC, NFKD, NFKC. In allen Fällen findet zunächst eine Aufteilung von kombinierten Zeichen in die Einzelbestandteile statt *(Decomposition)*. Es ist nach diesem Schritt garantiert, daß keine kombinierten Codes mehr existieren. Bei NFC und NFKC werden danach wieder alle Zeichen kombiniert *(Composition)*, bei NFD und NFKD nicht. Die NFK Varianten wenden dabei die sog. *Compatibility Decomposition* an, welche sich auf Zeichen bezieht, die aus dem Grund in Unicode aufgenommen wurden, weil einige Schriftarten seit jeher einen eigenen Code dafür verwenden. So bspw. für Ligaturen wie fi oder ffi. Das Wissen um diese Normalisierungsformen spielt daher auch ein Rolle für ein Satzsystem bzw. ein vorbereitendes System, wie einen Editor, der den Text zum Setzen übergibt. Zum einen muß sichergestellt sein, daß ein kombiniertes Glyph auch als solches verarbeitet wird und nicht zwei dafür dargestellt werden (im obigen Beispiel also „ö“ und nicht „o + ¨“). Zum anderen darf auch nicht soweit gegangen werden, daß der Text durch Normalisierung ungewollt verändert wird. Das ist bei Ligaturen wichtig

– je nach Wort sind sie gewollt oder nicht. In „auffinden“ ist eine fi-Ligatur im Wort „finden“ angebracht, allerdings wegen der Betonung keine ffi-Ligatur. Welche Normalisierung gewählt wird, entscheidet damit auch darüber wieviel Logik ein Satzalgorithmus mitbringen muß.

3.3 Schriftformate

Der wichtigste Aspekt, wenn es um Schriftsatz geht, ist natürlich die grafische Repräsentation der einzelnen Zeichen, die es hintereinander zu platzieren gilt. Glyphen werden in Schriftarten zusammengefasst. Schriftarten wiederum werden in unterschiedlichen Schriftformaten kodiert und gespeichert. Dieser Abschnitt beschäftigt sich mit solchen unterschiedlichen Schriftformaten.

3.3.1 Formate

Es gibt aktuell zwei weit verbreitete Schriftformate. Dies sind TrueType [18] Schriftarten sowie Type1 bzw. Type2 Schriftarten. Type1/2 Schriftarten stammen ursprünglich von Adobe und sind letztendlich nichts anderes als PostScript-Programme. Type1 war eine Zeit lang proprietär was Apple veranlasste ein eigenes Format, TrueType, zu entwickeln. Wie Type1 verbirgt sich auch hinter TrueType eine regelrechte Programmiersprache, die der Komplexität von Sprachen Rechnung trägt. Inzwischen favorisieren Apple, Adobe und Microsoft das gemeinsam definierte OpenType-Format [19]. Letztendlich handelt es sich dabei um ein Containerformat, das entweder Type2 (eine leichte Modifizierung von Type1, weshalb beide Begriffe im Rest des Kapitels synonym verwendet werden) oder TrueType Daten enthalten kann. Die wichtigste Neuerung ist sicherlich die Unterstützung von Unicode.

Da die Entwicklung des Computers hauptsächlich im englisch/lateinisch geprägten Kulturkreis stattfand, waren in den ursprünglichen Schriftarten nur 256 unterschiedliche Zeichen vorgesehen, was selbst für Englisch eine Minimallösung darstellt, bezieht man mathematische Symbole und dergleichen mit ein. Deshalb bieten TrueType und Type2 die Möglichkeit von Unterschriften. Schriftformate sind in Tabellen organisiert, die einem Code ein Glyph zuweisen. Um mehr als 256 Glyphen bereitstellen zu können, kann anstatt auf ein Glyph auch auf eine Untertabelle verwiesen werden. Diese Schachtelung impliziert eine zusätzliche Logik innerhalb der Schriftart bei der Suche nach dem passenden Glyph.

TrueType wie Type1/2 sind sog. Outline-Schriftarten, die die Hülle eines Glyphen beschreiben. Die beiden anderen Möglichkeiten sind Bitmap-Schriftarten, bei denen jedes Glyph ein eigenes Bild, vergleichbar einem Icon, ist, und Stroke-Schriftarten. Bei letzteren wird, ähnlich wie bei Handschrift, ein Pinsel über eine Zeichenfläche geführt. Verschiedene Pinselformen und -größen ergeben unterschiedliche, winkelabhängige Strichstärken. Stroke-Schriftart wäre damit im Sinne von Linien- oder Pinselschriftart zu übersetzen. METAFONT [13] beschreibt Glyphen auf diese Weise. Will man eine eigene Schriftart nach dieser Methode erstellen, verlangt es ein gewisses Gespür dafür, welche Spur ein Pinsel hinterlässt und wie dessen Form zu variieren ist, um zum gewünschten Ergebnis zu gelangen.

Für Outline- wie Stroke-Schriftarten gilt, daß Glyphen als eine Reihe von Punkten gespeichert werden. Der komplette Pfad ergibt sich durch eine mathematische Interpolationsfunktion, TrueType benutzt hier ein Polynom zweiten Grades, Type1/2 und METAFONT Polynome dritten Grades, unter dem Namen Bézierkurven bekannt. Dies stellt auch den Hauptunterschied zwischen TrueType und Type1 dar. TrueType läßt sich daher schneller verarbeiten, da die Mathematik einfacher ist als bei Polynomen dritten Grades. Polynome dritten Grades lassen hingegen eine feinere Kontrolle über die Pfade zu, es können im Optimalfall Punkte eingespart werden. Die beiden Formate können ohne größere Verluste ineinander konvertiert werden. Besonders leicht natürlich von TrueType nach Type2; jedes Polynom zweiten Grades ist ein Polynom dritten Grades bei dem das höchstgradige Glied Null ist.

Eine Schriftart ist damit im Grunde nichts anderes als eine Ansammlung von Vektorgrafiken, angereichert mit, zugegebenermaßen sehr viel, Metainformation. Die wichtigste dieser Zusatzinformation ist welches Schriftzeichen welchem Glyph zugeordnet wird. Um genauer zu sein: Welche Folge von Schriftzeichen welcher Folge von Glyphen zugeordnet wird. Diese Zuordnung ist nicht immer 1 : 1 und es wird dann von Glyphsubstitution gesprochen.

3.3.2 Glyphsubstitution

Ersetzungsregeln sind durch Glyphindizes definiert. Das heißt, um kontextabhängige Operationen durchführen zu können, müssen die kodierten Zeichen zunächst in Glyphindizes konvertiert werden. Substitutionen sind in Schriftarten daher unabhängig von der ursprünglichen Kodierung der Zeichen. So wie es das Character-Glyph-Modell verlangt.

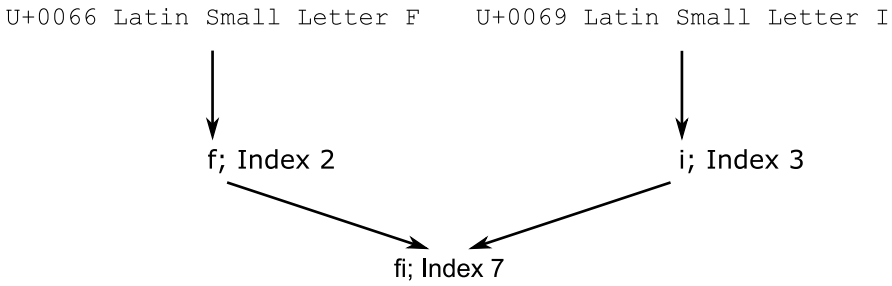


Abbildung 3.3: Zeichencodes und Glyphindizes

Eine eins zu eins Umsetzung, in der Unicode Code Points identisch den internen Indizes sind, erscheint sinnvoll, ist aber nicht Voraussetzung, wie Abbildung 3.3 zeigt. Die Unicode-Zeichen U+0066 „f“ und U+0069 „i“ werden in unserer virtuellen Schriftart auf die Glyphen mit den Indizes 2 und 3 abgebildet. Gleichzeitig zeigt die Grafik eine Ersetzung. Treten Glyph 2 und 3 direkt hintereinander auf, sollen sie durch Glyph 7 (fi-Ligatur) ersetzt werden. Eine formale Regel könnte bspw. so aussehen:

$$2, 3 \rightarrow 7$$

Dies ist ein Beispiel für eine äußerst simple Ersetzung. In Wahrheit sind meist weitaus komplexere Regeln erforderlich. So ist nicht in jedem Fall eine Ligatur angebracht. Im zusammengesetzten Wort „auffinden“ ist aufgrund der Betonung keine ffi-Ligatur zu verwenden – wir kennen den Fall bereits. Solche Anforderungen haben zu regelrechten Zustandsautomaten geführt, die in Schriftarten eingebettet sind. Auch die Programmiersprachen, die nötig sind um solche Ersetzungen auszudrücken, sind wesentlich komplexer als in obigem Beispiel.

Graphite[1], ein Projekt von SIL International[2], hat eine solche Programmiersprache als Bestandteil. Graphite erstellt zusätzliche Tabellen für TrueType-Fonts, die es unter Windows zur Textverarbeitung verwenden kann.

3.3.3 Hinting

Die Schriftgröße kann in zweierlei Hinsicht Probleme bereiten. Zum einen scheinen die Proportionen nicht mehr zu stimmen, wenn ein Glyph sehr

[1]URL http://sourceforge.net/projects/silgraphite
[2]URL http://www.sil.org

groß dargestellt wird, obwohl mathematisch korrekt skaliert wurde. Dies ist ein wahrnehmungstechnischer Aspekt. Zum anderen muss ein Glyph bei der Ausgabe meist gerastert werden (eigentlich immer, außer man besitzt ein nicht punktbasiertes Ausgabemedium, welches in der Lage ist, Kurven direkt darzustellen, wie bspw. ein Oszilloskop). In eine Schriftart eingebettete Informationen, die bei der Rasterung und Skalierung behilflich sind, nennt man Hints. Um eine bessere Darstellungsqualität zu erreichen werden z. B. die Kontrollpunkte am Raster ausgerichtet, was die Glyphform minimal verändert. Hinting spielt bei der endgültigen Ausgabe eine Rolle, im Rahmen von UTA hingegen keine.

3.3.4 Handhabung von Metainformationen

Prinzipiell gilt: Umso besser die Schriftart, umso besser sind die Ergebnisse bei verschiedenen Schriftgrößen, umso mehr Glyphen sind verfügbar, umso geringer ist die Wahrscheinlichkeit, daß auf eine andere Schriftart ausgewichen oder ein Approximationsverfahren eingesetzt werden muss. Eine gute Schriftart zeichnet sich damit nicht nur durch wohlgeformte Glyphen aus, sondern auch durch das Drumherum und trägt maßgeblich zu der Gesamtqualität des Schriftsatzes bei.

Das Problem ist, daß meist mehr Potential in einer Schriftart steckt, aber es nicht immer voll ausgeschöpft wird, weil die Schriftart selbst nicht alle notwendigen Informationen bietet. Entwirft ein Schriftschneider in einer Schriftart ein A mit Ring „Å“ (U+00C5), aber kein Ångström (U+212B), so hätte theoretisch die Schriftart das Potential auch ein Ångström darzustellen indem einfach das A mit Ring genutzt wird. Wird dies aber nicht in der Schriftart erwähnt, kann eine externe Anwendung nicht sicher sein, daß sich die Glyphen nicht doch unterscheiden, eine eigenmächtige Ersetzung ist immer mit einem gewissen Risiko verbunden. Allerdings ist es sicherlich besser auf ein anderes Glyph der gleichen Schriftart auszuweichen, als ein entsprechendes aus einer anderen Schriftart verwenden zu müssen. Ein Ångström läßt sich auch durch die Kombination des großen „A“ mit einem Ring „ ˚ “ erreichen. Allein für dieses Beispiel gibt es damit schon drei verschiedenen Möglichkeiten das gleiche Glyph zu erzeugen.

Ein ähnliches Beispiel sind die Zeichen „¡“ und „¿“, die im Spanischen zur Einleitung eines Ausrufs bzw. einer Frage genutzt werden. Eine Rotation des Ausrufezeichens „!“ respektive Fragezeichens „?“ um 180° bringen das gewünschte Ergebnis. Es fällt schwer zu entscheiden, wo derartiges Wissen besser aufgehoben ist. In der Schriftart an sich, oder im Textpro-

zessor. Der große Nachteil wenn derartige Informationen in der Schriftart platziert werden ist, daß sie in jeder Schriftart vorhanden sein müssen, was den Grad der Redundanz erhöht.

Zu letztem Beispiel sollte es sehr selten Ausnahmen geben. Gemeint ist damit, daß sich die Glyphen selten von ihren rotierten Gegenstücken unterscheiden. Das Wissen in den Textprozessor zu integrieren, wäre in diesem Fall die bessere Alternative. Anders sieht es aus, wenn es um zusammengesetzte Glyphen geht, wie bei der entsprechenden Lösung im ersten Beispiel. Während die eine Schriftart hier einen Satz vorgefertigter Glyphen mitbringt, kann eine andere nur Einzelstücke mitbringen, aber dafür einen Regelsatz, wie diese zu kombinieren sind. Hier gibt es also keine andere Möglichkeit, als die Information in der Schriftart zu belassen.

Entsprechend liegen je nach Technologie die Prioritäten anders. Der Ansatz in Java, die Positionierung der einzelnen Glyphen vollständig der Schriftart zu überlassen, entspricht auch dem Vorgehen in ATSUI. Wenn man so will ist eine solche Schriftart ein Fat-Client mit aller notwendigen Verarbeitungslogik. OpenType sieht die Zuständigkeit einer Schriftart hingegen eher im Glyphlieferanten, also im Thin-Client. Von der Theorie her erschwert dieser Unterschied in der „Philosophie“ die Implementierung von Anwendungen, die beide Schriftformate unterstützen. Eine OpenType-basierte Anwendung bringt schon viel Logik mit, die in einer ATSUI Anwendung nicht benötigt wird. In der Realität werden diese Unterschiede geringer ausfallen, es zeigt aber, welcher Aufwand im schlimmsten Fall betrieben werden muß, um das darzustellende Glyph zu finden. Zusammenfassend wird dieser Aufwand als *Font-Management* bezeichnet, was die notwendigen Funktionen einschließt bei fehlender Schriftart eine möglichst gleichwertige Alternative zu bieten. Desweiteren gehört dazu auch die Fähigkeit verschiedene Schriftformate einlesen zu können.

3.4 Verwandte Technologien

Während die bisher erwähnten Technologien direkt oder indirekt von UTA benutzt werden, nun zu solchen, die UTA als Teilkomponente benutzen könnten oder ähnliche typografische Probleme zu lösen haben.

3.4.1 TEX und Derivate

Bereits in der Einleitung war mehrfach von TEX die Rede. Der Name, der in einem Atemzug mit TEX genannt wird ist Donald Ervin Knuth. Er

entwickelte TEX aus der Unzufriedenheit über die in den 70er Jahren vorherrschende Qualität des Schriftsatzes heraus. Ausschlaggebend war der Probedruck seines Werkes *The Art of Computer Programming Volume 2*. Aus der Entscheidung das Problem selbst zu lösen folgten mehrere Jahre intensiver Forschung auf dem Bereich der digitalen Typografie. Daraus hervor ging nicht nur das komplette Satzsystem TEX, sondern auch das bereits erwähnte Schriftformat mit dem Namen METAFONT.

Die noch heute einflußreichsten Teilkomponenten dieser Arbeit sind der mathematische Satz, der Umbruch- sowie der Trennalgorithmus. Knuth hat dabei nicht sämtliche Arbeit alleine erledigt, der Umbruchalgorithmus ist auch mit dem Namen Michael F. Plass in Verbindung zu bringen, der Trennalgorithmus mit Frank M. Liang.

Obwohl viele Programme einzelne Teile der damals entwickelten Techniken einsetzen, sind solche, die alle Komponenten in einem Satzsystem vereinen, selten. Am nächsten kommt dem InDesign von Adobe, welches einen an TEX angelehnten Umbruchalgorithmus verwendet. Liangs Trennmuster werden hingegen nahezu in allen (vornehmlich freien) Programmen verwendet, die Silbentrennung beherrschen.

TEX leistet noch immer, nach mehr als 20 Jahren, gute Dienste und gilt als fehlerfrei und äußerst stabil. Allerdings sind die Anforderungen mit der Zeit gestiegen. Knuth hat sein Programm allein für seine persönlichen Zwecke entwickelt. In erster Linie heißt das für die englische Sprache. Weitsichtig hat er zwar zahlreiche Erweiterungsmöglichkeiten eingebaut, mit der sich gravierend veränderten Umgebung, in der TEX heute eingesetzt wird, erscheinen diese Erweiterungen aber als umständlich. Knuth sagt dazu:

> „[. . .] I was expecting that the really special applications would be done by changing things in the compiled code. But people didn't do that; they wanted to put low-level things in at a higher level.“ [6, S. 648]

So entstand eine kaum mehr überschaubare Menge an Makropaketen und Hilfsprogrammen, von denen viele effizienter im eigentlichen Programm erledigt werden könnten. Hier seien exemplarisch die Handhabung von Farben und das Erstellen eines Index genannt.

Die veränderte Umgebung verwendet auch andere Ein- und Ausgabeformate. Nicht METAFONT ist das Schriftformat der Wahl, sondern TrueType und Type1 mit dem gemeinsamen Nachfolger OpenType. Neue Anforderungen wie internationaler Schriftsatz und Cross-Media-Publishing waren

ursprünglich nicht angedacht. Daher kommen existierende Erweiterungen, die in der Lage sind aus einem XML-Dokument ein PDF-Dokument zu erzeugen, nicht ohne mehrfache Konvertierungen aus. Zwangsläufig bleibt dabei die Feinkontrolle über die Ausgabe durch den Anwender auf der Strecke. Diese Unzulänglichkeiten zusammen mit dem Mangel an Alternativen sind der Hauptgrund für die Existenz von UTA.

UTAs Ziel ist es daher in erster Linie eine flexible Plattform für die bisher am wenigstens berücksichtigten Teilkomponenten bereitzustellen. Dies ist der Umbruch von Absätzen und die Unterstützung mikrotypografischer Feinheiten wie Unterschneidungen und Ligaturen. Letzteres Ziel muß zunächst hinter der Notwendigkeit für internationalen Schriftsatz zurückstehen. Um, wie in der Einleitung angesprochen, umfangreiche Umbauarbeiten zu vermeiden, muß bereits zu Beginn auf die unterschiedlichen Forderungen Rücksicht genommen werden. Das hat vorangehendes Kapitel gezeigt.

Die TEXnologie auf moderne Füße zu stellen ist seit nunmehr über einer Dekade das Anliegen der TEX-Gemeinde. Zu diesem Zweck wurde Anfang der 90er das $\mathcal{NTS}$ Projekt ins Leben gerufen. $\mathcal{NTS}$ steht für *New Typesetting System* und soll die Beschränkungen von TEX beseitigen. Als Meilensteine zum endgültigen, neuen System können Anwendungen wie ε-TEX und pdfTEX gesehen werden. ε-TEX erweitert TEX um grundlegende Funktionen zum Satz von bidirektionalem Text, pdfTEX ist in der Lage PDF-Dokumente zu generieren. 2001 endete mit der Veröffentlichung einer 1.0 Beta Version ein weiteres Subprojekt, das ebenfalls den Namen $\mathcal{NTS}$ trug, und zum Ziel hatte eine Reimplementierung von TEX in Java zu erstellen. ε_{χ}TEX[3], das als $\mathcal{NTS}$ Nachfolgeprojekt bezeichnet werden kann, steht noch am Anfang. Es ist zu erwarten, daß ε_{χ}TEX ein modernes Font-Management bieten wird mit Unterstützung für TrueType, OpenType und METAFONT. Mit einem modularen Design sollen die Einzelkomponenten auch mit anderen Eingabesprachen als TEX verwendbar sein. Ω schließlich ist ein weiteres TEX-Derivat, spezialisiert auf multilingualen Satz. Ω lockert weitere Einschränkungen, vornehmlich solche, die den Speicherverbrauch betreffen.

3.4.2 XSL

XSL steht für *Extensible Stylesheet Language* [12] und besteht aus einer Reihe von Unterstandards. Diese sind XSLT *(XSL Transformations)* [20],

[3] URL http://www.extex.org

XPath [21] sowie XSL-FO *(XSL-Formatting Objects)*. XSLT dient der Transformation von XML Dokumenten allgemein. Gemeint ist dabei in erster Linie die Veränderung der Struktur des Eingangsdokument in die des Zieldokuments. Diese Veränderung wird durch ein XSL-Stylesheet beschrieben. Ein Beispiel ist die Umwandlung eines beliebigen XML Dokuments in HTML zur Darstellung in einem Web-Browser. Die Dateien *eingabe.xml* und *stylesheet.xsl* miteinander verknüpft ergeben *ausgabe.html*.

```
<?xml version="1.0" encoding="UTF-8"?>
<beispiel>Text der HTML-Seite.</beispiel>
```

Listing 3.1: eingabe.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
     xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
 <xsl:template match="/">
  <html>
   <head>
    <title>Eine einfache Transformation von XML
           nach HTML</title>
   </head>
   <body>
     <xsl:value-of select="." />
   </body>
  </html>
 </xsl:template>
</xsl:stylesheet>
```

Listing 3.2: stylesheet.xsl

```
<html>
 <head>
  <title>Eine einfache Transformation von XML
         nach HTML</title>
 </head>
 <body>Text der HTML-Seite.</body>
</html>
```

Listing 3.3: ausgabe.html

XSLT bietet die Möglichkeit Ergebnisdokumente zusammenzubauen, ohne den Unterstandard XPath wäre dies nicht möglich. Bereits das kleine Stylesheet oben enthält zwei XPath Ausdrücke. Dies ist zum einen der Slash bei `match="/"` im `xsl:template` Element, zum anderen der Punkt im `xsl:value-of` Element (`select="."`). Wie der Name schon nahe legt, geht es bei XPath um Pfadangaben um XML-Elemente und -Attribute

adressieren zu können. Ein XML-Dokument läßt sich als Baumstruktur darstellen, ganz ähnlich einem Dateisystem. Daher liegt ein Vergleich nahe. Der Slash bezeichnet in XML das Wurzelelement, wie in unixartigen Betriebssystemen das Wurzelverzeichnis. Der Punkt bezeichnet das aktuell gewählte Element. In einem Dateisystem ist es das aktuelle Verzeichnis. Obigem XSL Ausdruck entspräche von der Navigation in etwa einem

```
cd /
ls .
```

unter einem unixartigen Betriebssystem. Diese XPath Angaben können noch weitaus komplexer werden, um das Prinzip zu verstehen reicht dieses Beispiel aber völlig aus.

Anstatt XML in HTML umzuwandeln ist es genauso möglich XML in eine anderes XML Dokument zu transformieren. XSL-FO gibt genau so eine Struktur vor. Der Standard beschreibt Regeln wie ein XSL-FO Dokument auszusehen hat und wie diese Dokumente von einem verarbeitenden Programm zu interpretieren sind. Ein solches verarbeitendes Programm wandelt XSL-FO Dokumente in ein anderes Ausgabeformat um. Das wichtigste dürfte dabei PDF sein. XSL-FO ist ein allgemeines Daten- und Regelmodell für textverarbeitende Systeme, ausgelegt auf die Erzeugung von umfangreichen Dokumenten. Aus diesem Grund spielt Schriftsatz im Umwandlungsprozeß von XML in das Ausgabeformat eine wichtige Rolle. Im XSL Standard sind allerdings Details dazu explizit von der Spezifikation ausgenommen. UTA nimmt sich dem an, was XSL in diesem Absatz ausschließt:

> „The formatting object definitions, property descriptions, and area model are not algorithms. Thus, the formatting object semantics do not specify how the line-breaking algorithm must work in collecting characters into words, positioning words within lines, shifting lines within a container, etc. Rather this specification assumes that the formatter has done these things and describes the constraints which the result is supposed to satisfy.“ [12, Kap. 3]

3.4.3 FOP

FOP[4], der *Formatting Objects Processor*, ist eine nicht interaktive Anwendung, die XSL-FO Dokumente in eine Reihe von Ausgabeformaten um-

[4]URL http://xml.apache.org/fop/

wandeln kann. Darunter fallen PDF, RTF und Text. FOP benötigt damit nicht nur eine XSL Implementierung, sondern ist auch ein Formatierer, der die in obigem Absatz ausgeschlossenen Algorithmen implementieren muß.

Es gibt starke Ähnlichkeiten zwischen FOP und ε_{χ}TEX, was Funktionsumfang und auch Designvorstellungen angeht. Der Grund für zwei getrennte Projekte dürfte in den primären Zielen liegen. So streben beide letztendlich zwar ein nahezu identisches Ergebnis an, zunächst liegt bei FOP der Fokus aber auf einer vollständigen Implementierung von XSL. ε_{χ}TEX hingegen zielt darauf ab, auch weiterhin TEX-Dokumente interpretieren zu können und die von TEX gewohnte Qualität zu erreichen.

Die Erzeugung von Dokumenten, die FOP implementiert, läßt sich in drei Phasen gliedern. In der ersten Phase wird die Eingabe in eine FO-Baumstruktur gebracht. Dazu wird meist ein XML-Dokument durch ein XSL-Stylesheet verarbeitet. Der erzeugte FO-Baum wird daraufhin in einen sog. Area-Baum überführt, was der eigentlichen Formatierung entspricht. Eine Area ist dabei letztendlich nichts anderes als eine rechteckige Fläche. In der finalen Phase steht die Ausgabe in einem bestimmten Format an. Diese Ausgabe entspricht einer Konvertierung des Area-Baums in das gewünschte Format.

Die im Area-Baum abgelegten Informationen können nicht unbedingt für alle möglichen Ausgabeformate benutzt werden. Man denke hier nur an die eingeschränkten Möglichkeiten, die einfacher Text bietet. Unter Umständen muß an dieser Stelle eine erneute Formatierung stattfinden.

3.4.4 SVG

SVG oder *Scalable Vector Graphics* ist der Versuch ein XML-basiertes Vektorgrafikformat zu definieren. Im Unterschied zu XSL-FO zielt SVG weniger auf große Dokumente mit viel Text ab, sondern auf die freie Gestaltung. Text spielt aber auch dort eine wichtige Rolle. Hierzu die zentrale Aussage zum Schriftsatz, in der Fließtext ausgeschlossen wird:

> „SVG does not provide for automatic line breaks or word wrapping, which makes internationalized text layout for SVG relatively simpler than it is for languages which support formatting of multi-line text blocks.“ [22, Kap. 10]

SVG macht konkrete Aussagen zu der Ausrichtung von Text an einem Pfad. UTA bietet derzeit keine Unterstützung dafür, Überlegungen in diese

Richtung lehnen sich aber stark an SVG an, wie wir später im Ausblick auf zukünftige UTA Versionen sehen werden.

Wie FOP eine Implementierung von XSL ist, ist Batik[5] eine Implementierung von SVG. Batik ist, wie FOP, ein Projekt der Apache Software Foundation. Da beide Projekte XML in eine grafische Repräsentation umwandeln sind auch viele Teilprobleme ähnlich strukturiert. Diese Parallelen wurden erkannt und daher werden derzeit (Juli 2004) FOP und Batik unter einem Dach vereint, dem sog. Apache XML Graphics Project Management Committee.

3.4.5 Java Text API & ICU

Java, als von Sun entwickelte Programmiersprache, ist plattformunabhängig. Um die Plattformunabhängigkeit gewährleisten zu können bringt die Sprache eine vollständige 2D API mit, auf der eine eigene Bibliothek zur Erstellung grafischer Benutzeroberflächen aufbaut. Sie trägt den Namen Swing und muß zwangsläufig zur Darstellung von internationalem Text fähig sein, um Java-Anwendungen auch im arabischen und asiatischen Raum anbieten zu können. Entsprechend existieren auch Pakete zur Handhabung von Schriftarten und zum Layouten von Text. Ein eng damit verbundenes Projekt mit dem Namen *International Components for Unicode* (ICU) wird von IBM geleitet. Teile dieses Projekts, ICU4J[6], gehen in die Java Foundation Classes, der Sammlung standardmäßig mit Java ausgelieferten Klassen, ein. Zentrales Paket für Textlayout ist `java.awt.font`. Die darin enthaltenen Klassen verfügen über die Fähigkeit, Absätze in Zeilen umzubrechen und handhaben bidirektionalen Text. Hauptkriterium bei der Entwicklung dieser Klassen ist die Unterstützung von Sprachen auf einer typografischen Ebene, die zur Darstellung der jeweiligen Schrift ausreicht. Die Qualität der Darstellung spielt eine untergeordnete Rolle. Persönlicher Hauptkritikpunkt an der API ist die schlechte Erweiterbarkeit. Die monolitische Architektur macht es sehr schwer Teilkomponenten gegen solche auszutauschen, die bessere Ergebnisse liefern. In Java Version 1.4 wird das TrueType-Format unterstützt, andere Formate sind per Interface vorgesehen, aber nicht implementiert.

Die erwähnten Technologien beschäftigen sich, formal ausgedrückt, mit der regelbasierten Platzierung von Rechtecken. Parallelen sind deshalb immer wieder zu finden, stark unterschiedlich sind jedoch die Termini der

[5]URL http://xml.apache.org/batik/
[6]http://oss.software.ibm.com/icu4j/

einzelnen Technologien. Das ist verständlich, hat doch jeder Ansatz einen anderen Hintergrund und gerade TEX, XSL und SVG sind so umfangreich, daß es nicht in kurzer Zeit gelingt, sie sich vollständig zu erarbeiten. In der ganzen Arbeit werden von Zeit zu Zeit immer wieder Parallelen zu den erwähnten Technologien gezogen. Zum einen, um vorbelasteten Lesern zu helfen ihr bereits vorhandenes Wissen direkt übertragen zu können, zum anderen, um Lesern, die Interesse an einer der erwähnten Technologie finden, den Einstieg in diese zu erleichtern.

3.5 UTAs grundlegende Architektur

Bisher wurde UTA in Relation zu anderen Technologien gesetzt, nun zur grundlegenden Architektur. UTA ist eine Bibliothek, die textverarbeitenden Systemen typografische Dienste bereitstellt und sich auf die Bedürfnisse dieser Systeme anpassen läßt. Benötigt eine Anwendung lediglich einfachen Zeilenumbruch und keine anderen mikrotypografischen Feinheiten, wie Ligaturen und Unterschneidung, ermöglicht das Design von UTA dies ebenso, wie die Realisierung von hochqualitativem Schriftsatz. Daher wurde beim Design auf Modularität und Erweiterbarkeit geachtet, damit Teilkomponenten leicht ausgetauscht werden können und die Unterstützung für neue Schriftsysteme sukzessiv ausbaubar ist. Abbildung 3.4 zeigt, wie UTA in eine existierende Anwendung eingebunden wird.

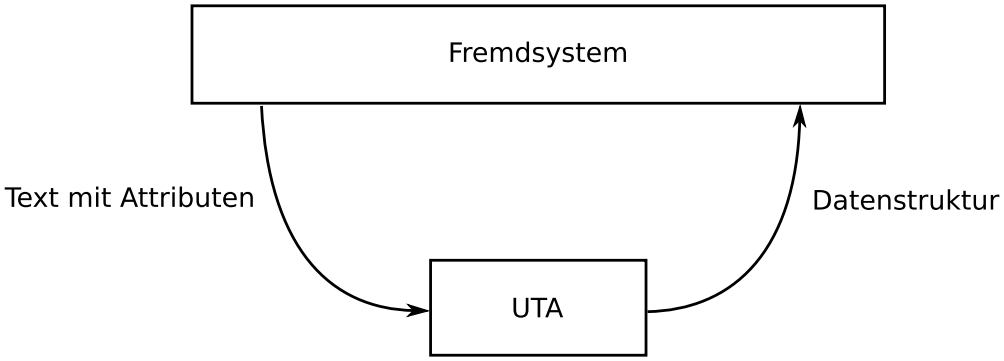


Abbildung 3.4: Einbindung von UTA in ein Fremdsystem

UTA akzeptiert als Eingabeformat Text, der mit Attributen ausgestattet ist. Die Rückgabe der Ergebnisse erfolgt über eine spezielle Datenstruktur, die wir in Kapitel 5 kennenlernen werden. Die Datenstruktur soll es ermöglichen, sowohl grafische, als auch nicht-grafische Systeme zu

bedienen. Grafische Systeme müssen den Umriss eines Glyphs pixelgenau darstellen. In nicht-grafischen Systemen kommt eine Seitenbeschreibungssprache wie PDF zum Einsatz, die exakte Form eines Glyphs wird bei der Erstellung des Dokuments nicht benötigt. Mit Adaptern läßt sich die Eingabe in ein UTA konformes Format und die Ausgabe in das Format des Fremdsystems umwandeln. Auf diese Weise lassen sich bspw. die typografischen Funktionen der Java 2D API erweitern (vgl. Abbildung 3.5).

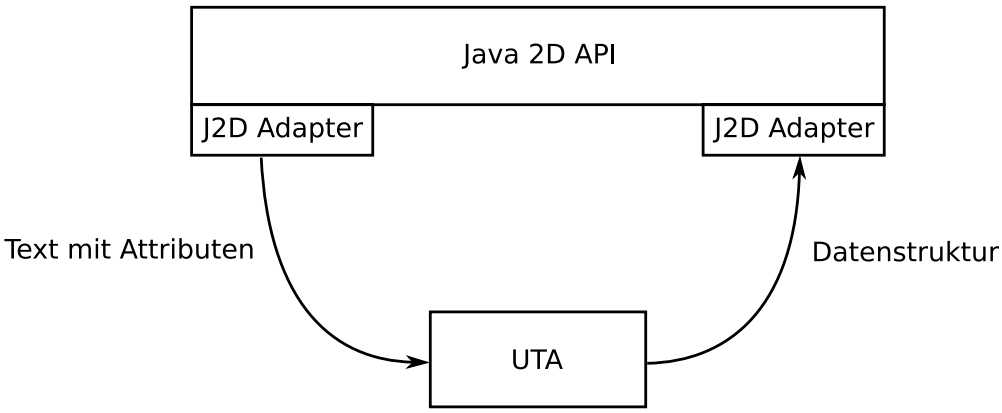


Abbildung 3.5: Einbinden von UTA in die Java 2D API

3.5.1 Schichtenmodell

UTAs grundlegende Architektur basiert auf drei Schichten. Die unterste Schicht beinhaltet Werkzeuge und wird daher im Folgenden Werkzeugschicht genannt. Als Werkzeug werden Trennalgorithmen, Font-Management und dergleichen verstanden. Die mittlere Schicht ist die schriftsystemspezifische Schicht. Sie benutzt die Werkzeuge aus der untersten Schicht. Beispielsweise können unterschiedliche Trennmuster (sprachspezifisch) für den gleichen Trennalgorithmus verwendet werden. Weitere Aufgaben sind das Finden von möglichen Umbruchstellen und Wortzwischenräumen, sowie die ggf. notwendige kontextbezogene Verarbeitung der Eingabe und zu guter Letzt die Platzierung der Glyphen. Die mittlere Schicht kommuniziert mit der obersten, der abstrakten Schicht insofern, daß gefundene Umbruchstellen und Wortzwischenräume gemeldet werden. Diese oberste Schicht abstrahiert Probleme, wie bspw. das Umbrechen von einem Absatz in einzelne Zeilen und löst sie unabhängig von dem verwendeten Schriftsystem. Diese Schicht bildet gleichzeitig die Schnittstelle für

Fremdsysteme. Abbildung 3.6 fasst die Aufgaben der einzelnen Schichten zusammen.

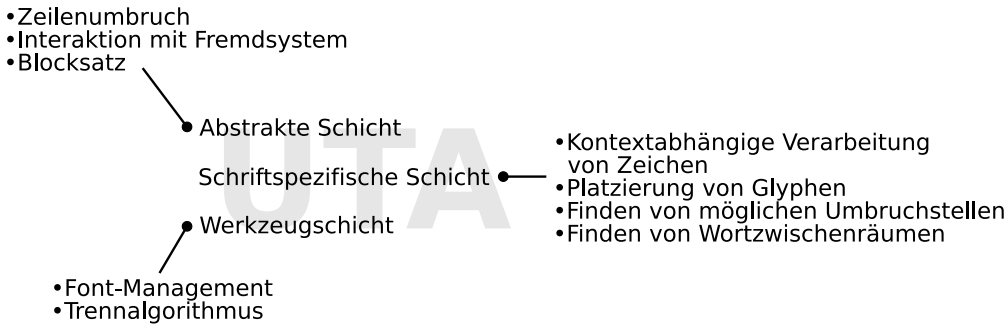


Abbildung 3.6: Das Schichtenmodell von UTA

Die Spezifikation der einzelnen Schichten erfolgt von oben nach unten, beginnend mit der abstrakten Schicht. Diese Arbeit beschreibt ausschließlich die abstrakte Schicht. Gerade auf der Werkzeugebene werden Symbioseeffekte mit anderen Projekten angestrebt. So hat FOP bereits eine Implementierung eines Trennalgorithmus. Bei FOray[7], einem Ableger von FOP, genießt die Entwicklung einer Font-Managementkomponente höchste Priorität.

3.5.2 Einheiten

UTA benutzt intern Scaled Points als grundlegendes Maß. Dies ist identisch zu TEX [2, Kap. 10], mit dem Unterschied, daß in UTA nicht ganzzahlig, sondern mit Gleitkommazahlen gerechnet wird. Dies ist der Datentyp double in Java. Scaled Points beziehen sich auf den amerikanischen Punkt. UTA stellt eine Klasse UnitConverter bereit, die bei diesen Umrechnungen behilflich ist. Dabei kommt Tabelle 3.2 zum Einsatz.

Der Big Point ist identisch zur Definition des PostScript Point, der ebenfalls 1/72 eines Inch mißt. Die Einheiten Point und Pica sind in den USA und Großbritannien gebräuchlich, der Didot Punkt sowie Cicero in Kontinentaleuropa. Die Tabelle ist so zu interpretieren, daß die Zeile die Einheit und den Faktor beinhaltet um zu der Einheit in der Spalte zu kommen. $2^{16} = 65536$ Scaled Points ergeben einen Punkt, $72,27$ Punkte ergeben ein Inch. Der von TEX definierte Punkt [2, S. 58] unterscheidet sich vernachlässigbar von der offiziellen Definition, weshalb hier nicht weiter darauf Rücksicht genommen wird.

[7] URL http://foray.sourceforge.net/

Einheit	sp	pt	pc	bp	in	cm	dd	cc
Scaled Point sp	1	2^{16}						
Point pt	$\frac{1}{2^{16}}$	1	12		$72,27$		$\frac{1238}{1157}$	
Pica pc		$\frac{1}{12}$	1					
Big Point bp				1	72			
Inch in		$\frac{1}{72,27}$		$\frac{1}{72}$	1	$\frac{1}{2,54}$		
Zentimeter cm					$2,54$	1		
Didot Punkt dd		$\frac{1157}{1238}$					1	12
Cicero cc							$\frac{1}{12}$	1

Tabelle 3.2: Die Tabelle enthält die Umrechnungsfaktoren, die auch im TEXbook angegeben sind.

4 Das Umbrechen von Absätzen

Ein elementarer Bestandteil eines Satzsystems ist die Fähigkeit einen Absatz in Zeilen gegebener Länge umzubrechen. Für UTA ist dabei ein Aspekt von besonderer Bedeutung: das Ziel ist es, einfache und komplexe Implementierungen mit der gleichen Schnittstelle zu erlauben. Daher stellt sich die Frage, ob sich der Vorgang des Zeilenumbruchs aus dem Gesamtsatzvorgang in einen separaten Algorithmus extrahieren und ob sich ein unabhängiges Interface definieren läßt, das einen Umbruchalgorithmus beschreibt. Unabhängig heißt dabei, daß nur wenig, im Idealfall gar kein Wissen über den Rest des Systems vorhanden sein muß. Dadurch lassen sich Algorithmen leicht austauschen und wiederverwenden. Dieses Kapitel versucht solch eine Schnittstelle zu finden. Dazu ist ein allgemeines Verständnis des Problems notwendig und ein Blick auf mögliche Lösungen. Damit ausgestattet lassen sich Aussagen über ein tragfähiges Umbruchmodell machen. Das Modell kann dann um weitere, speziellere Anforderungen erweitert werden.

4.1 Allgemeines Vorgehen und Anforderungen

Ein Umbruchalgorithmus hat die Aufgabe optimale Umbruchstellen innerhalb eines Absatzes für eine gegebene Zeilenlänge zu finden. Die Unterschiede zwischen Umbruchalgorithmen verschiedener Qualität liegen darin, welche der möglichen Umbruchstellen als optimal erachtet werden. Allen gemeinsam ist, die Breite aufeinanderfolgender Objekte zu messen und mit der gewünschten Zeilenbreite zu vergleichen. Die aufeinanderfolgenden Objekte machen dabei den Absatz aus. Dies können, müssen aber nicht, Glyphen sein. Ein Gegenbeispiel wäre eine eingebettete Grafik. Es läßt sich sogar noch weiter abstrahieren, denn der Inhalt eines solchen Objektes interessiert einen Umbruchalgorithmus nicht. Es ist keine Unterscheidung zwischen Glyphen, Grafiken und dergleichen notwendig. Gravierend zur Qualität tragen Objekte bei, deren Breite nicht fest vorgegeben ist, sondern die bei Bedarf auch gestaucht oder gedehnt werden können. Dies ist die zentrale Voraussetzung um überhaupt Blocksatz realisieren zu können.

Umbruchalgorithmen sind entweder lokal oder global optimierend. Grob lassen sich lokal optimierende Algorithmen als zeilen-, global optimierende als absatzbasiert bezeichnen. Ein global optimierender Algorithmus tendiert immer dazu, gleichlange Zeilen, de facto also Blocksatz (vgl. Abbildung 4.1 weiter unten), zu erzielen. Damit fallen auch bei Flattersatz (linksbündiger Satz) die Zeilenlängen gleichmäßiger aus als bei einem lokal optimierenden Algorithmus. Absatzbasierte Algorithmen sind generell zu bevorzugen. Guter Umbruch ist nur mit ihnen zu erreichen.

4.2 Verschiedene Umbruchalgorithmen

Donald Knuth und Michael Plass haben in ihrem Artikel *Breaking Paragraphs Into Lines* (zu finden in [6, Kap. 3]) unterschiedliche Umbruchalgorithmen untersucht und mit dem in TEX implementierten verglichen. Neben den drei unten vorgestellten Algorithmen, wurde auch ein Blick auf historische Ansätze geworfen. In den meisten Fällen steht aber deren Implementierungsaufwand in keinem Verhältnis zu der erzielten Qualität. Als Beispiel sei hier ein Brute-Force Algorithmus genannt, der sämtliche Umbruchmöglichkeiten durchprobiert. Aus diesem Grund werden hier nur die drei relevantesten Algorithmen besprochen, in Anlehnung an Knuth und Plass im Folgenden First-Fit, Best-Fit bzw. Total-Fit Algorithmus genannt.

4.2.1 First-Fit

Die einfachste Methode, Absätze umzubrechen, ist dabei auch die am häufigsten anzutreffende. Im Folgenden einfacher Zeilenalgorithmus oder First-Fit Algorithmus genannt. Er ahmt den Vorgang eines menschlichen Setzers nach. Er packt so viele Objekte in eine Zeile wie Platz haben, ohne die vorgegebene Zeilenlänge zu überschreiten. Wird die Zeilenlänge überschritten, kommt das Objekt in eine neue Zeile. Damit ist für diesen Algorithmus die optimale Umbruchstelle gefunden und er kann sich der nächsten Zeile widmen. Offensichtlich ist dieser Algorithmus leicht zu implementieren. Die Objektlängen werden so lange aufaddiert, solange die Summe kleiner als die vorgegebene Zeilenlänge ist. Dieser Algorithmus ist in nahezu allen Anwendungen zu finden, bspw. in mehrzeiligen Textfeldern, Browsern, Editoren usw. Für lange Zeilen mag er akzeptable Resultate erzielen, wobei es auch hier im Flattersatz zu ungewollten Formsatzeffekten aufgrund stark unterschiedlicher Zeilenlänge kommen kann.

Allgemein erzeugt er einen unruhigen Rand. Im Blocksatz, bei schmalen Spalten, kommt es zu sehr großen Lücken zwischen den einzelnen Wörtern. Zudem kann es zu inkompatiblen Zeilen kommen (zwei Zeilen stark unterschiedlicher Länge folgen aufeinander). Erweiterungen zu diesem Algorithmus, z. B. eine Berücksichtigung von Trennungsregeln, verbessern das Ergebnis nur marginal.

Bei diesem Algorithmus drängt sich ein kleiner Gedankengang auf. Nicht nur Text muß in grafischen Applikationen angeordnet werden, auch Grafiken und sog. Container, die wiederum andere Elemente enthalten können, wie bspw. Knöpfe oder Textfelder. Java geht dieses Problem mit seinen Layout-Managern an. Einer davon, FlowLayout, versucht so viele Elemente wie möglich in eine Zeile zu packen, bevor er eine neue beginnt. Auch die Terminologie in der Java-Klasse Box lehnt sich an TEX an, dort gibt es eine Methode `createGlue`. Nicht zuletzt benutzt auch FOP ein eigenes LayoutManager-Framework um das eigentliche Positionieren der Elemente zu delegieren. Die Parallelen in den Anforderungen und den Lösungsansätzen sind offensichtlich. Es wäre eine interessante Übung, einen auf FlowLayout basierenden Umbruchalgorithmus zu implementieren. Auch der umgekehrte Weg ist natürlich denkbar, Umbruch- & Satzalgorithmen zum Layouten von grafischen Oberflächen zu benutzen. Schaltflächen wären dann die zu platzierenden Objekte.

4.2.2 Best-Fit

Der Best-Fit Algorithmus arbeitet bereits mit alternativen Zeilen, wägt also zwischen möglichen verschiedenen Umbruchstellen ab. Wie erwähnt, ist es für hochwertigen Umbruch unerläßlich, Komponenten mit variabler Breite zu unterstützen. Diese sind auch der Ursprung für alternative Zeilen. Die Anzahl der in einer Zeile platzierten Objekte variiert mit deren Breite unter unterschiedlichen Vorbedingungen. Kalkuliert man die minimale Breite aller Objekte wird die Zeile sehr gedrängt, die Chancen steigen, daß ein weiteres Objekt Platz findet. Die Zeile wird maximal gequetscht. Im umgekehrten Fall wird die Zeile maximal gedehnt. Die maximale Breite aller Objekte wird einkalkuliert, es finden weniger Objekte in einer Zeile Platz. Für die alternativen Zeilen wird ein Bewertungsschema notwendig. Deshalb berechnet der Algorithmus im Gegensatz zum einfachen Zeilenalgorithmus einen Strafwert, der eine Abweichung von der ästhetischsten Zeile repräsentiert. Dieser auch im Total-Fit verwendete *Badness-Wert*, gibt Auskunft darüber, wie stark Objekte gestaucht bzw. gedehnt werden müssen. Je weniger ein Objekt von seiner gewünschten

Breite abweichen muß, um die geforderte Zeilenbreite zu erreichen, umso kleiner ist der Badness-Wert. Der Algorithmus wählt nun nacheinander stets diejenigen möglichen Zeilen aus, die den kleinsten Badness-Wert besitzen. Ein Beispiel für eine bewährte Berechnung eines solchen Wertes wird weiter unten im Abschnitt *Beispielimplementierung* besprochen.

4.2.3 Total-Fit

Als eine Erweiterung des Best-Fit Algorithmus zieht der Total-Fit Algorithmus den kompletten Absatz in die Berechnung mit ein und sucht nach den Umbrüchen, die alle Zeilen möglichst miteinander harmonieren lassen. Er berücksichtigt dabei, ob aufeinanderfolgende Zeilen optisch kompatibel sind, damit einer dicht gesetzten Zeile keine lose folgt. Dazu wird das Umbruchproblem auf das Problem des kürzesten Weges transformiert. Ähnlich dem Best-Fit Algorithmus erzeugt er für jede zu setzende Zeile eine oder mehrere virtuelle Zeilen mit unterschiedlichen Umbruchstellen. Für jede dieser Zeilen wird ein Strafwert errechnet. Dieser ist komplizierter als beim Best-Fit und wird als *Demerits* bezeichnet. Der optimal umgebrochene Absatz ist der, bei dem die Summe dieser Strafen minimal wird. Dabei lassen sich über verschiedene Parameter u. a. Witwenwörter (alleinstehende Wörter in der letzten Zeile eines Absatzes) vermeiden oder auch, in einem gewissen Maße, die Zeilenanzahl steuern. Der komplette Absatz wird dann entweder lockerer oder dichter umgebrochen. Der Total-Fit Algorithmus kommt in TEX zum Einsatz, aber auch in anderen Anwendungen wie Adobe InDesign. Interessant ist, daß TEX beim Umbrechen standardmäßig dreistufig arbeitet. Knuth hat festgestellt, daß bei langen Zeilen in 90% der Fälle überhaupt keine Trennung notwendig ist. Daher wird nur dann ein Lösung mit Trennung gesucht, wenn der erste Durchlauf kein zufriedenstellendes Resultat ergibt, was performanter ist. Der dritte und letzte Lauf wird nur dann ausgeführt, wenn auch mit Trennung keine ausreichenden Ergebnisse erzielt werden konnten. Dabei werden allerdings nur die Anforderungen gelockert, TEX gibt sich dann schlicht mit einem schlechteren Absatz zufrieden. Der Wert von 90% gilt dabei für englischen Text. Im Deutschen muß wesentlich häufiger getrennt werden, da die Sprache reich an langen Wörtern ist und häufig Wörter zusammengesetzt werden.

Zum Vergleich zeigt Abbildung 4.1 die Ergebnisse der drei vorgestellten Algorithmen. Der Absatz stammt aus der englischen Version des Märchens *Der Froschkönig* der Gebrüder Grimm und wurde gewählt, um einen Vergleich mit [6] zu erlauben. Dort wird er mit diesen Umbrüchen als Beispiel

First-Fit

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Best-Fit

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Total-Fit

In olden times when wishing still helped one, there lived a king whose daughters were all beautiful; and the youngest was so beautiful that the sun itself, which has seen so much, was astonished whenever it shone in her face. Close by the king's castle lay a great dark forest, and under an old lime-tree in the forest was a well, and when the day was very warm, the king's child went out into the forest and sat down by the side of the cool fountain; and when she was bored she took a golden ball, and threw it up on high and caught it; and this ball was her favorite plaything.

Abbildung 4.1: First-, Best- und Total-Fit Algorithmus im Vergleich

verwendet. Damit die Unterschiede besser zur Geltung kommen, wurde der Text hier linksbündig ausgerichtet. Die Qualität der einzelnen Algorithmen spiegelt sich direkt an der Gleichmäßigkeit des rechten Randes wider. Zu erkennen ist, daß eben dieser von oben nach unten wie erwartet immer ruhiger wird. Gerade die ersten drei Zeilen sind bei First- und Best-Fit Algorithmus unterschiedlich lang. Die Zeilen sind zueinander inkompatibel. Bei Blocksatz werden die Leerräume in Zeile zwei weit gedehnt, während der Text in den Zeilen eins und drei gestaucht werden muß. Das Beispiel des Total-Fit Algorithmus illustriert, daß ein global optimierender Algorithmus zu Blocksatz neigt, auch wenn die endgültige Austreibung (das Anpassen der Leerräume) nicht stattfindet. In ausgerichtetem Text, sei es links, rechts oder zentriert, erzielt er gleichmäßigere Ergebnisse.

4.3 UTAs Umbruchmodell

Soviel zunächst zu den populärsten Umbruchalgorithmen. Der wiederkehrende Begriff des Objekts läßt schon den Kern des Umbruchmodells erahnen. UTA setzt ein sehr einfaches Modell ein. Es dürfte ähnlich dem Kerf-Modell sein, welches Knuth im Addendum von *Breaking Paragraphs Into Lines* (vgl. [6, S. 154]) erwähnt (ausführlich erklärt in *Document Preparation Systems*, Jurg Nievergelt, Giovanni Coray, Jean-Daniel Nicoud, Alan C. Shaw; Amsterdam, 1982; S. 221-242 – dieses Buch stand mir leider nicht zur Verfügung). Das Kerf-Modell wird dabei als Vereinfachung und Generalisierung von TEX's Box/Glue/Penalty Modell bezeichnet. Eine Box ist ein Element fester Breite, Glue eines variabler Breite und Penalty bezeichnet einen Strafwert für den Umbruch nach dem Element, welches dem Strafwert vorangeht. In diesem Modell werden alle drei Einzelelemente zu einem sog. Kerf zusammengefasst. Ähnlich dem kennt UTA lediglich das Item, das Einfluß auf den Umbruch hat. Dies ist das Objekt, von dem bisher die Rede war. Ein Umbruch ist nach jedem Item erlaubt und wie bei TEX mit einer optionalen Strafe belegt. Diese Penalty geht in die Demerits Berechnung mit ein. Ein hoher Wert senkt die Wahrscheinlichkeit eines Umbruchs nach diesem Item, ein niedriger erhöht sie. Ein Item hat, abhängig von der in der Zeile vorgesehenen Position, drei unterschiedliche Breiten: eine minimale, eine optimale/bevorzugte sowie eine maximale. Zusätzlich zu den vier möglichen Positionen, die ein Item innerhalb einer Zeile einnehmen kann, nämlich am Anfang bzw. Ende einer Zeile sowie umgeben von keinem Item bzw. zwei Items, ergeben sich daraus maxi-

mal 12 verschiedene Breiten. Wie das Item-Konzept in das Satzmodell eingebettet ist, wird im nächsten Kapitel besprochen.

4.3.1 Zusätzliche Anforderungen

An dieser Stelle soll auf weitere Anforderungen eingegangen werden, die ein Umbruchalgorithmus erfüllen muß. Auch die Wichtigkeit der Item-Position und der damit verbundenen Breite wurde nur teilweise begründet.

Trennung

Ein Hauptgrund für Notwendigkeit von Item-Breiten in Abhängigkeit von der Position in der Zeile ist die Silbentrennung. Sie kompliziert die Berechnung von Wortbreiten zusätzlich. Muß ein Wort getrennt werden verändert sich dadurch der umzubrechende Text und damit die Anzahl und Größe der zu berücksichtigenden Glyphen. Im einfachsten Fall wird das Wort an der entsprechenden Stelle getrennt und dem ersten Teil ein Trennzeichen angehängt. In komplizierten Fällen ändert sich auch die Schreibweise. Ein gern angeführtes Beispiel ist das Wort backen, welches nach der alten Rechtschreibung bei Trennung bak-ken geschrieben wird. Durch das Item-Modell wird es dem Umbruchalgorithmus ermöglicht eine solche Veränderung einzukalkulieren, unabhängig davon, ob getrennt wurde oder nicht.

Formsatz

Es macht Sinn, daß ein Umbruchalgorithmus mit variierenden Zeilenlängen umgehen kann. Dadurch wird Formsatz und das Umfliessen von Objekten möglich. Soll ein Objekt umflossen werden (vgl. Abbildung 4.2) kommt das dem Aufteilen des Absatzes in mehrere zusätzliche Zeilen gleich. Auf welche Weise die von ihm bestimmten Zeilen letztendlich vom Satzalgorithmus platziert werden, liegt außerhalb des Zuständigkeitsbereichs des Umbruchalgorithmus. Ebenso ist es für ihn nicht von Belang, auf welche Weise die Zeilenlängen errechnet werden, was bei unregelmäßigen Formen kompliziert werden kann. Daher macht eine Komponente Sinn, die eine Zeilenlänge für eine gewünschte Zeilennummer liefert.

Im nachfolgenden Kapitel wird näher auf das Umfließen von Objekten in Text eingegangen.

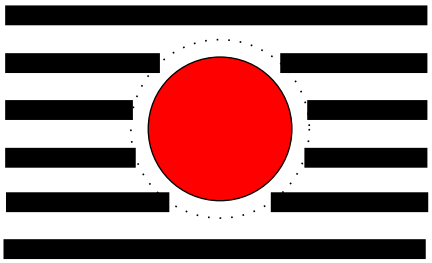


Abbildung 4.2: Der Text fließt um ein Objekt. Auf dem gepunkteten Kreis liegen die Längen der einzelnen Zeilen, wie sie vom Umbruchalgorithmus berücksichtigt wurden.

Parallelisierung von Satz und Umbruch

Ein UTA Umbruchalgorithmus beherrscht somit Formsatz und ist in der Lage, Breiten zu berücksichtigen, die in Abhängigkeit zu der Position in der Zeile stehen. Die Parallelisierung von Satz und Umbruch ist eine weitere sinnvolle Funktion, die hilft, den Verwaltungsaufwand gering zu halten. Um dies zu erlauben, werden stets einzelne Items an den Algorithmus übergeben und nicht eine Liste, über die dann iteriert wird. Nach Beendigung des Satzvorgangs, wenn keine weiteren Items mehr entstehen, können die optimalen Umbruchstellen abgerufen werden.

Qualitätsmanagement

Ein einfacher Algorithmus bezieht möglicherweise nur eine einzige Breite und eine einzige Positionierung des Items in die Berechnungen ein. Die First-Fit Beispielimplementierung ist ein solcher Fall, der nur die bevorzugte Breite eines Items bei der Position zwischen zwei anderen Items abfragt. Dazu später mehr. Beide Angaben können also als Qualitätskriterien Anwendung finden. Ein Umbrecher wird nur dann ein optimales Ergebnis liefern, wenn auch die generierten Items in sein Konzept passen. Ein höher angesiedelter Algorithmus muß Sorge tragen, daß sämtliche Komponenten zusammenpassen. Um diese Aufgabe zu erleichtern stellt UTA eine eigene Komponente zur Verfügung (s. Kapitel 6 *Qualitätsmanagement*).

Optischer Randausgleich

Des weiteren hat ein Item jeweils eine optische Dichte am Anfang und Ende. Diese Angabe ist für Algorithmen vorgesehen die einen optischen Randausgleich berücksichtigen wollen, worunter auch das Vermeiden von

direkt aufeinanderfolgenden Trennungen fällt. Ein kleiner Wert bezeichnet dabei eine kleine optische Dichte, ein solches Zeichen hat also viel Weiß (Fleisch) und wenig Schwarz. Daher stellt die optische Dichte eine Art Grauwert eines Glyphen dar. Ein Glyph mit geringem Grauwert ist in vielen Schriften das Trennzeichen. Diese Funktion beschreibt bereits eine typografische Feinheit, deren optischen Auswirkung ziemlich gering ist, aber einen hohen Implementierungsaufwand erfordert. UTA stellt deshalb keine konkrete Umsetzung zur Verfügung. Da es auch noch keine Definition gibt, wie Dichtewerte zu interpretieren sind, ist diese Funktion noch als experimentell zu betrachten.

Die Idee, Grauwerte zu berechnen soll es ermöglichen, den optischen Randausgleich zu generalisieren. Bisher werden nur bestimmte Glyphen wie das Trennzeichen „-“ berücksichtigt. Je nach Schriftart (z. B. der Times) ist das Trennzeichen aber sehr kompakt und ein Randausgleich ist nicht unbedingt notwendig.

4.3.2 Ablauf des Umbruchs

Nachdem Umbruchalgorithmus und die Einheiten, die einen Absatz ausmachen, ausführlich besprochen wurden, nun zum eigentlichen Ablauf.

Erzeugung der Items

Für die Erzeugung der Items sind die einzelnen Sprachmodule zuständig. Sie sind letztendlich auch diejenigen, die die einzelnen Glyphen setzen. Wir werden diese *Scripts* im nächsten Kapitel genauer kennenlernen. An dieser Stelle reicht es zu wissen, daß sie ein Schriftsystem repräsentieren und die Eingabe durch sie verarbeitet wird. Um legitime Umbruchstellen finden zu können, ist umfassendes Wissen über die dem Text zugrundeliegende Sprache nötig, das Kapitel *Herausforderung Schrift* hat das gezeigt. An dieser Stelle sei nur noch einmal an die gravierenden Unterschiede zwischen lateinischen Schriften und Thai erinnert, gerade im Hinblick auf das Vorhandensein von Worttrennern. Zudem sind Attribute wie verwendete Schriftarten, -größen und dergleichen relevant, die über die effektive Breite eines Schriftzeichens entscheiden. Dieses Wissen ist im jeweiligen Script gekapselt. Durch das Item-Modell sind die unterschiedlichsten sprachlichen Anforderungen realisierbar. So kann ein Script Leerraum fester oder variabler Breite erzeugen, sowie Elemente mit Breite Null. Das Modell sollte sogar soweit kompatibel zu TEX sein, daß sich (obwohl keine ursprüngliche Anforderung) zentrierter Text realisieren läßt, indem Leerraum eingefügt

wird, der sich am Zeilenanfang und -ende anders verhält als in der Zeilenmitte. Solcher Leerraum hätte am Anfang und am Ende einer Zeile eine große bevorzugte Breite, gepaart mit einer minimalen Breite von Null. In der Mitte einer Zeile wäre die Breite die eines normalen Leerraums.

Auf bereits erwähnte Weise (vgl. *Parallelisierung von Satz und Umbruch*) werden die erzeugten Items an den Umbruchalgorithmus übermittelt, der damit die notwendigen Berechnungen anstellt.

Ermittlung der Umbruchpunkte

Bei Bedarf wird ein endgültiger oder ein vorläufiger Umbruchpunkt ermittelt. Auf jeden Fall aber muß bei einem Aufruf der `getBreakpoints`-Methode eine Liste der endgültigen Umbruchpunkte zurückgegeben werden. Spätestens beim Aufruf dieser Methode beginnt somit die Suche nach den optimalen Umbruchstellen, welche gerade bei absatzbasierten Algorithmen meist sehr aufwendig ist. Dabei behilflich sind die Hilfsklassen zur Graphentheorie, siehe gleichnamigen Abschnitt weiter unten. Bei einem Aufruf von `getBreakpoints` hat sich der Umbruchalgorithmus in einen wiederverwendbaren Zustand zu versetzen und muß ggf. Ressourcen freigeben. Mit der Rückgabe der Liste, die die berechneten Umbruchpunkte enthält, ist für ihn der Vorgang beendet.

Anwendung der Ergebnisse

Ein UTA Umbruchalgorithmus nimmt keine Modifikation an den einzelnen Items vor, er justiert also nicht automatisch Zwischenräume. Lediglich die aktuelle und bevorzugte Zeilenlänge steht bei jeder Zeile zur Verfügung. Die Anpassung bleibt einem höheren Algorithmus vorbehalten. Das vermeidet Probleme wenn keine stufenlose Anpassung möglich ist. So kann es nötig sein, Glyphen mit unterschiedlichen Weiten zu benutzen, die alle das gleiche Schriftzeichen darstellen. Das ist für anspruchsvollen Schriftsatz im Arabischen der Fall. Des weiteren wird in der OpenType-Spezifikation als Ausgleichsmöglichkeit das Aufteilen einer Ligatur in ihre Einzelteile erwähnt, die Einzelglyphen „f“, „f“, und „i“ beanspruchen marginal mehr Platz als eine ffi-Ligatur. Dabei ist eine stufenlose Dehnung nicht möglich. Ein zu kleiner Wert schiebt die einzelnen Glyphen übereinander, ein zu großer zu weit auseinander. Sinn macht letzteres nur in sehr schmalen Zeilen, bspw. wenn in einem mehrspaltigem Layout eine Grafik umflossen wird.

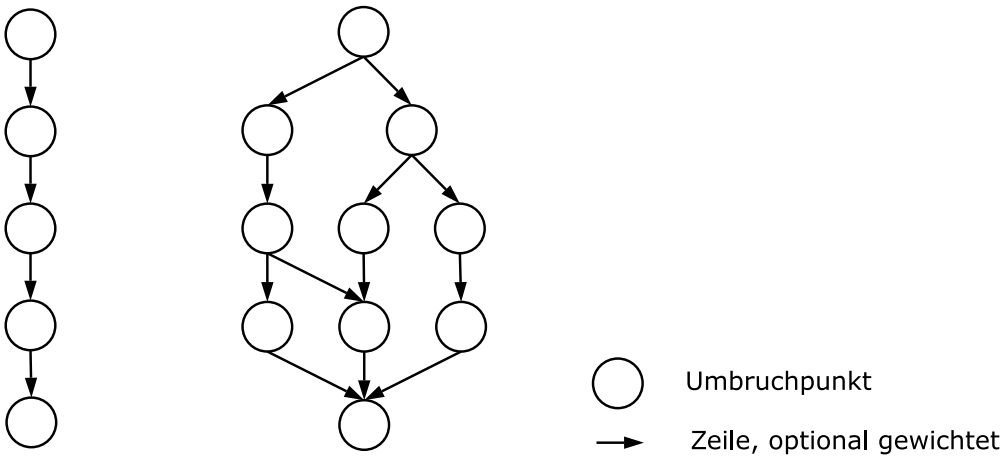


Abbildung 4.3: Unterschiedlich komplexe Umbruchalgorithmen und deren Zuordnung zu einem Graphentyp. Während der First-Fit Algorithmus einen Graphen in Form einer Liste (links) erzeugt, benötigt ein global optimierender Algorithmus wie er in TEX zum Einsatz kommt eine Baumstruktur (rechts). Im Baum stehen vier alternative Pfade zur Verfügung, in der Liste existiert nur einer.

4.3.3 Implementierung

Mit der Umbruchschnittstelle, die global optimierende Algorithmen unterstützt, ist eine weitere Teilkomponente verfügbar, wie sie zur Implementierung hochwertiger Umbruchalgorithmen nötig ist.

Hilfsklassen zur Graphentheorie

Um die Implementierung von global optimierenden Algorithmen zu erleichtern stellt UTA ein Helferpaket bereit, welches auf einfache Weise erlaubt, einen gewichteten Graphen bestehend aus möglichen Umbruchstellen zu erzeugen und dann den kürzesten Pfad daraus zu ermitteln.

Ein Graph in der Graphentheorie besteht aus Knoten und Kanten. Veranschaulicht ist damit eine Liste und ein Baum genauso ein Spezialfall eines Graphen wie ein Netz. Je nach Umbruchalgorithmus entsprechen die erzeugten Graphen diesen Sonderfällen, bzw. Zwischenformen. Alle Graphen sind dabei gerichtet, in den meisten Fällen auch gewichtet. Es gibt sowohl eine Richtung, in die von einem Ausgangsknoten gewandert werden muß, als auch einen Wert für die Länge der zurückzulegenden Strecke.

Bei einer Liste gibt es nur einen Pfad, der gleichzeitig den kürzesten darstellt. Bei einer Baumstruktur muß ein spezieller Algorithmus, wie der von Dijkstra, angewandt werden, um den kürzesten Pfad zu finden. Der kürzeste Pfad ist der, bei dem die Summe aller Weglängen minimal wird. Die Suche nach dem kürzesten Pfad ist ein Standardproblem der Informatik, weshalb hier nicht näher darauf eingegangen wird. Tatsächlich benutzt das erwähnte Helferpaket aus diesem Grund derzeit eine bereits existierende, frei verfügbare Graphen-Bibliothek. Um den Austausch dieser zu erleichtern, wird in UTA nicht direkt darauf zurückgegriffen, sondern über eine eigene, minimale Graphen-Schnittstelle. Um eine neue Bibliothek zu unterstützen, muß eine Implementierung des GraphAdapter Interfaces vorhanden sein. Derzeit existiert ein Adapter für die JDigraph[1] Bibliothek. Sie ist unter der sehr liberalen BSD Lizenz verfügbar und kann damit problemlos mit UTA ausgeliefert werden.

Strafwertberechnung

Natürlich liefern die gleichen Knoten mit gleicher Gewichtung die gleichen Ergebnisse wie in TEX. Die Qualität des Umbruchs hängt damit davon ab, ob es gelingt ähnlich gute Umbruchstellen und Gewichtungen zu finden. Da sich die Berechnung der Gewichtung der Kanten bewährt hat, bietet UTA eine (teilweise) Implementierung der Demeritsberechnung nach Knuth an.

Die grundlegenden Faktoren sind dabei

- das Verhältnis, wie sehr eine Zeile tatsächlich gestaucht bzw. gedehnt wird und wieviel sie maximal gestaucht/gedehnt werden darf,
- die aus diesem Wert berechnete Badness einer Zeile, sowie
- die aus Badness und Penalty Werten berechneten Demerits.

Die Badness b berechnet sich durch

$$b = 100 * r^3,$$

wobei das Adjustment-Ratio r das erwähnte Verhältnis von aktueller zu maximaler Stauchung/Dehnung ist. Die Demerits d ergeben sich durch die Formel

$$d = (p + b)^2 + sgn(b) * b^2.$$

[1] URL http://jdigraph.sourceforge.net

Die Line-Penalty p ist die Strafe die mit dem Erzeugen einer neuen Zeile einhergeht, die Break-Penalty b ist die Strafe für den Umbruch nach dem aktuellen Item. Diese Berechnung unterscheidet sich dadurch von TEX, daß sie zum einen mit Gleitkommazahlen statt Ganzzahlen arbeitet und zum anderen keine Unterscheidung zwischen Break-Penalty-Werten kleiner -10000 macht. Zudem bietet die derzeitige Implementierung keine Unterstützung zur Verhinderung von inkompatiblen Zeilen. In diesem Fall müßten weitere Strafwerte hinzuaddiert werden. Die derzeitige Implementierung der Strafwertberechnung befindet sich also auf dem Stand des oben besprochenen Best-Fit Algorithmus.

Beispielimplementierung

UTA stellt zwei Beispielimplementierungen des First-Fit Algorithmus zu Demonstrationszwecken zur Verfügung. Der Algorithmus ist dabei auf zwei unterschiedliche Weisen implementiert. Die einfache Variante fügt die erzeugten Umbruchpunkte in eine Liste ein und gibt diese unverändert zurück. Um auch die Benutzung der Graph-Bibliothek zu demonstrieren fügt die zweite Implementierung die Umbruchpunkte in einen Graphen ein und läßt sich zum Schluß den kürzesten Pfad zurückgeben. Der ist in diesem Fall natürlich identisch mit der Liste, die die einfache First-Fit Implementierung errechnet.

4.4 Wiederverwendbarkeit für verschiedene Ausgabemedien

Abschließend noch einige Worte zur Unabhängigkeit der Umbruchergebnisse von der Darstellung. Das Umbrechen eines Absatzes in einzelne Zeilen ist nur bis zu einem bestimmten Rahmen darstellungsunabhängig machbar. Ausschlaggebend ist dabei, ob das Ausgabemedium all die Funktionen bietet, die der Satzalgorithmus berücksichtigen muß. Letztendlich muß das Medium eine freie Platzierung von Objekten ermöglichen. Dieser Punkt spielt für die Anwendungen eine gewichtige Rolle, die Text auch in ganz einfachen ASCII-Textdateien formatieren wollen. Prinzipiell spricht nichts dagegen auch hier Trennung und Blocksatz anzuwenden, allerdings ist der Zeichensatz stark begrenzt, Ligaturen und Hoch-/Tiefstellung von Zeichen sind nicht möglich. Dies erfordert eine Vorverarbeitung des Textes für ein Medium bereits bevor es dem Umbruchalgorithmus übergeben wird. Fußnoten müssen ggf. umgeschrieben werden, bspw. zu einer in ecki-

gen Klammern gebetteten Ziffer [1] anstatt hochgestellt. Dadurch ändert sich natürlich auch die Länge des Absatzes, vergleichbar der Änderung bei Trennung. Meist wird ASCII-Text in diktengleicher Schrift dargestellt, die Fontmetrikdaten in einem solchen Fall sind entsprechend einfach. Jedes Glyph hat eine Breite von eins, ebenso das Leerzeichen, wodurch wiederum nur bedingt Spielraum für die Variation des Wortabstandes bleibt. Einmal errechnete Umbruchergebnisse können also nicht immer wiederverwendet werden.

5 Der Satzvorgang

Dieses Kapitel ist erheblich umfangreicher als das vorangehende, deshalb zunächst ein Blick auf die Elemente, die das Satzmodell von UTA ausmachen. Die zweite Hälfte des Kapitels beschäftigt sich dann mit dem Ablauf eines Satzvorgangs.

5.1 Einleitung

Beschreibt man den Schriftsatz in einem Satz, auf der niedrigsten Ebene, so ist er nichts anderes als das regelbasierte Platzieren von geometrischen Objekten. Diese Objekte können dabei extrem einfach (Rechteck), oder extrem komplex (Glyph) sein. Die tatsächliche Erscheinung, also Größe und Form der einzelnen Objekte, kann dabei von äußeren Faktoren abhängen. Schriftgröße oder Auflösung sind konkrete Beispiele dafür.

Nachdem wir im Kapitel *Herausforderung Schrift* lediglich die Probleme mit internationalem Text geschildert haben, hier nun die Lösung.

5.2 UTAs Satzmodell

UTAs Satzmodell orientiert sich an bestehenden Technologien wie XSL oder TEX. Zudem bietet es Konzepte, die in dieser Form in keiner der Technologien enthalten sind. In erster Linie macht UTA konkrete Aussagen wie multilingualer Text gehandhabt werden kann.

5.2.1 Koordinaten und Koordinatensysteme

Koordinatensysteme und Koordinaten spielen im Satzmodell eine wichtige Rolle. Deshalb wird in UTA zwischen *globalem* und *lokalem Koordinatensystem* sowie *Referenzkoordinatensystem* unterschieden. Jedes Koordinatensystem ist dabei kartesisch. Koordinaten können *absolut* oder *relativ* sein.

Referenzkoordinantensystem und globales Koordinatensystem

Um die Orientierung eines Koordinatensystems angeben zu können, ist eine Referenz nötig. Die positive x-Achse des Referenzkoordinatensystems orientiert sich nach rechts, die positive y-Achse nach oben. Das globale Koordinatensystem orientiert sich mit der positiven x-Achse nach rechts und der positiven y-Achse nach unten. Die Vektoren, die die Achsen in Relation zum Referenzsystem beschreiben sind damit:

$$\vec{x} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \text{ bzw. } \vec{y} = \begin{pmatrix} 0 \\ -1 \end{pmatrix}$$

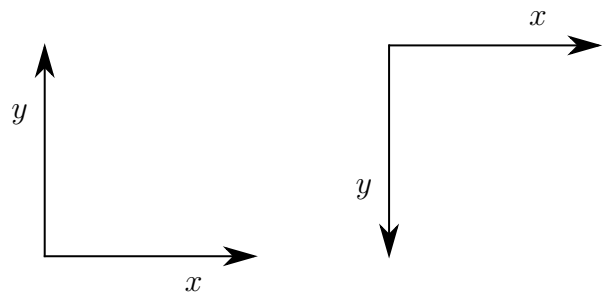


Abbildung 5.1: Referenz- und globales Koordinatensystem.

Diese Orientierung entspricht auch der in 2D-Schnittstellen üblichen, bei der die linke obere Ecke als Ursprung gilt. Ein Beispiel dafür ist die Java 2D API. Aufeinanderfolgende Zeilen haben dadurch positive Koordinaten, sollten die Zeilen von oben nach unten platziert werden.

Lokale Koordinatensysteme

In lokalen Koordinatensystemen werden die eigentlichen Glyphen gesetzt. Auf die Termini der Computergrafik reduziert, verwenden westliche und arabische Schriften unterschiedliche Koordinatensysteme (siehe Abbildung 5.2). Die Orientierung der x-Achse unterscheidet sich, bezeichnet man die Schreibrichtung in einer Zeile als solche. Die y-Achse ist „nach oben“(!) gerichtet und zeigt *nicht* die Richtung, in der Zeilen platziert werden.

Die Koordinatensysteme zeigen die *bevorzugte* Richtung. Arabisch und Hebräisch sind alleine bidirektional, da Zahlen von links nach rechts geschrieben werden. Bidirektionale Schriftsysteme benötigen daher *zwei* Koordinatensysteme.

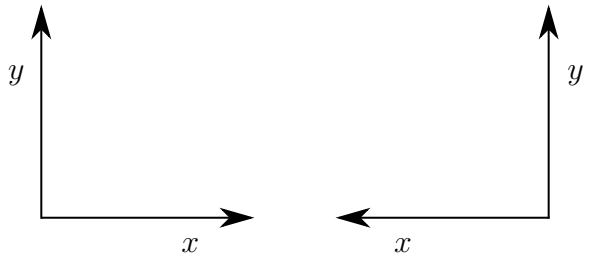


Abbildung 5.2: Lateinisches und arabisches Koordinatensystem im Vergleich. Die Schreibrichtungen sind links nach rechts, bzw. rechts nach links.

Die Unterscheidung zwischen den einzelnen Koordinatensystemen wurde getroffen, um die Implementierung der einzelnen Sprachmodule – wir lernen sie gleich kennen – möglichst zu vereinfachen. Jedes Modul kann sein eigenes Koordinatensystem verwenden und muß lediglich sicherstellen, daß das lokale in das globale Koordinatensystem transformiert werden kann.

Absolute und relative Koordinaten

Absolute Koordinaten beziehen sich auf den Ursprung eines Koordinatensystems. Relative Koordinaten beziehen sich auf einen beliebigen Punkt in einem Koordinatensystem. Absolute Koordinaten sind ein Spezialfall relativer Koordinaten, bei denen der Bezugspunkt identisch dem Ursprung ist. Punkte, die durch relative Koordinaten beschrieben werden, können ebenfalls relative Punkte als Bezugspunkte benutzen. Eine Umrechnung in absolute Koordinaten ist dann möglich, wenn in der Liste dieser Bezugspunkte ein absoluter Punkt ist.

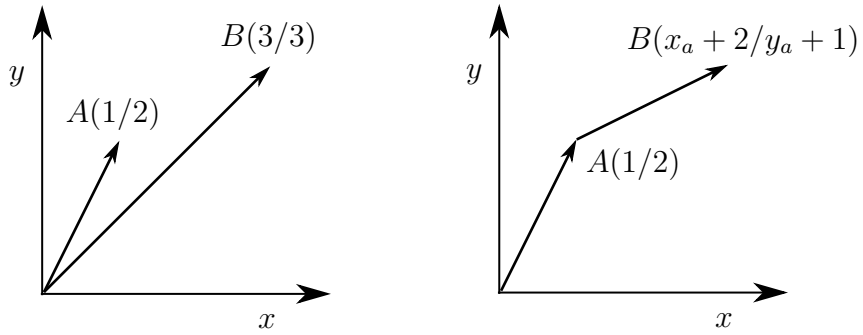


Abbildung 5.3: Absolute und relative Koordinaten

Der Vorteil von relativen Koordinaten ist, daß sich Transformationen des Bezugspunktes automatisch auf die referenzierenden Punkte auswirken. Wird in Abbildung 5.3 der Punkt A verschoben, ändert sich nur bei relativen Koordinaten die Position von B. Der Satzvorgang ist nach UTA das regelbasierte Platzieren von geometrischen Objekten. Dabei ist ein Punkt der Ursprung des Objekts. Wird dieser Punkt transformiert, wird auch das Objekt transformiert. Der Satzvorgang ist somit ebenfalls das regelbasierte Platzieren von Punkten.

5.2.2 Box

Das zentrale Element des Zeichensatz ist in UTA die Box. *Spezialfälle einer Box* sind Grafiken oder Glyphen. Eine Box kann eine beliebige Anzahl an Attributen besitzen. Diese werden in Form einer Schlüssel-Wert-Tabelle zur Verfügung gestellt. Jede Box hat eine Hülle, die mit einem Aufruf der Methode `getShape` erfragt werden kann.

Für die Platzierung ist die horizontale und vertikale Ausdehnung einer Box ausschlaggebend. Diese Ausdehnung kann durch ein Rechteck repräsentiert werden, daher ist im Rest des Kapitels auch von Rechtecken die Rede. Die Platzierung derartiger Rechtecke erfolgt durch das Erzeugen und Zuweisen von Transformationsmatrizen. Eine Transformation bezieht sich auf den Ursprung einer Box. Der Ursprung ist ein Punkt. Die Qualität eines Satzsystems ist damit zu einem großen Teil davon abhängig, welche optische Qualität die Gesamtheit der Transformationsmatrizen im Endeffekt bewirken.

5.2.3 Glyph

Ein Glyph ist ein Spezialfall einer Box und besitzt einen Index, der das Glyph innerhalb der verwendeten Schriftart eindeutig kennzeichnet. Ein Glyph repräsentiert ein oder mehrere Code Points die mit `getCharacters` abgerufen werden können. Da Java, wie im Abschnitt *Code Points* auf Seite 28 erwähnt, nur einen Datentyp für Schriftzeichen anbietet, der auf 65536 unterschiedliche Zeichen begrenzt ist, liefert diese Methode ein `int`-Array.

Ein Glyph kann auch durch eine Glyphsubstitution entstanden sein. In diesem Fall liefert die Methode `getSubstitutedGlyphs` eine Liste der ersetzten Glyphen.

5.2.4 Typesetter

Ein Typesetter übernimmt in UTA die Funktion eines Managers, der als Ansprechpartner für Fremdsysteme fungiert und den Ablauf des Satzvorgangs leitet. Er nimmt den zu setzenden Text entgegen und gibt das Ergebnis an das Fremdsystem zurück. Ein Typesetter verwaltet eine beliebige Anzahl von Schriftimplementierungen (s. Abschnitt *Script*), einen Umbruchalgorithmus (s. Kapitel *Zeilenumbruch*), sowie einen Austreibalgorithmus (s. Abschnitt *JustificationAlgorithm*).

Ein Typesetter arbeitet absatzbasiert. Der Text, der übergeben wird, ist ein Absatz. Desweiteren regelt er die Erzeugung von Embedding-Leveln, die wir gleich kennenlernen werden.

5.2.5 Script

Eine Box und deren Spezialfälle müssen meist nach bestimmten Regeln platziert werden. Diese Ansammlung von Regeln nennt man Schrift. Eine Schrift wird in UTA durch ein Script repräsentiert. Ein Script ist die *Implementierung einer Schrift*. So gibt es für jedes Schriftsystem mindestens ein Script. Im Typesetter werden Scripts einem Locale zugeordnet. Daher ist es möglich, für verschiedene Dialekte, die das gleiche Schriftsystem benutzen, ein eigenes Script zu registrieren. Umgekehrt kann auch ein Script zuständig für mehrere Locales sein.

Ein Locale besteht aus maximal drei Angaben: Der Sprache, dem Land und einer Sprachvariante. Wobei die beiden letzteren Angaben optional sind. Ein Locale dient dazu, eine Zuordnung zwischen einer geografischen, kulturellen oder politischen Region und einer Sprache zu schaffen. Ein Beispiel ist de-DE. Es bezeichnet die Sprache Deutsch in Deutschland. de-AT bezeichnet die Sprache Deutsch in Österreich. Die zu verwendenden Kürzel sind dabei ISO Language[1] bzw. Country Codes[2].

Ein Script kapselt sämtliche Informationen, die für den Satz von Glyphen im jeweiligen Schriftsystem notwendig sind. Konkret ist dies das Wissen um

- erlaubte Umbruchstellen,
- das korrekte Auswählen und Platzieren von Glyphen,
- das Ausrichten des Textes um Blocksatz zu erzeugen.

[1]URL http://www.w3.org/WAI/ER/IG/ert/iso639.htm
[2]URL http://www.iso.org/iso/en/prods-services/iso3166ma/index.html

Während die ersten beiden Forderungen verpflichtend sind, ist letztere optional. Wobei optionale Komponenten immer dann verpflichtend werden, wenn sie von einem Schriftsystem verlangt werden, um geringsten typografischen Ansprüchen zu genügen.

Das Vorhandensein dieses Wissens ermöglicht keine Aussage über dessen Umfang. Ein Script muß in jedem Fall korrekte Ergebnisse erzeugen. In Bezug auf Umbruchstellen dürfen keine falschen ermittelt werden. Nicht alle korrekten Umbruchstellen zu kennen/finden ist hingegen konform. Das betrifft in erster Linie Schriftsysteme mit Silbentrennung. Unterstützt ein Script Trennung nicht, liefert es eine gültige Untermenge aller möglichen Umbruchstellen.

Die korrekte Auswahl und Platzierung von Glyphen kann die Substitution von Glyphen und die Platzierung von diakritischen Markierungen einschließen. Die Anforderungen in diesem Bereich variieren stark von Schrift zu Schrift. Auf jeden Fall müssen die Glyphen in eine dem Schriftsystem angemessene Reihenfolge gebracht werden, d. h. Berücksichtigung der Schreibrichtung und von Glyphzwischenräumen. Typografische Feinheiten sind optional, im gerade erwähnten Sinn. In lateinischen Schriften wäre eine solche optionale Feinheit das Unterschneiden von Glyphen.

5.2.6 JustificationAlgorithm

Mit Justification, zu deutsch Austreibung, wird der Vorgang bezeichnet, bei dem Wortzwischenräume innerhalb einer Zeile in ihrer Ausdehnung angepasst werden, um eine vorgegebene Zeilenbreite vollständig auszufüllen. Dadurch wird Blocksatz erreicht. Der JustificationAlgorithm nimmt diese Anpassung vor.

5.2.7 Justifiable

Kern des Blocksatzes ist das Justifiable. Es repräsentiert variable Breite und daher meist Wortzwischenraum, aber auch die anderen im Abschnitt *Anwendung der Ergebnisse* auf Seite 58 angesprochenen Fälle.

Justifiables werden vom jeweiligen Script erzeugt, da sie schriftspezifisch sind. Sie werden nur vom Austreibalgorithmus benötigt und sind daher keine Spezialfälle einer Box sondern ein eigenes Interface. Allerdings spricht auch nichts dagegen eine Klasse zu erstellen, die gleichzeitig Box und Justifiable ist.

Blocksatz in TEX und UTA

Um Blocksatz realisieren zu können, ist, wie wir inzwischen wissen, eine Breitenanpassung von Wortzwischenräumen (Inter-Word-Spacing), ggf. auch von Glyphzwischenräumen (Inter-Letter-Spacing) oder gar Glyphen an sich notwendig. TEX verwendet dazu sogenannten Glue. Glue hat eine bevorzugte Breite und einen Wert, um welchen er gedehnt oder gestaucht werden kann. UTA übernimmt weitestgehend dieses Modell, allerdings mit ein paar Abwandlungen. Um diese verstehen zu können, ist ein tieferer Blick in die Verwendung von Glue innerhalb von TEX notwendig.

Die Breitenanpassung kann erst erfolgen, wenn der Zeilenumbruch abgeschlossen ist. Wortzwischenraum wird in Abhängigkeit von seiner Stauch- bzw. Dehnbarkeit variiert. Muß eine Zeile um 4pt erweitert werden, und es stehen drei Glue-Objekte mit den Stretchability-Werten 2pt, 2pt und 4pt zur Verfügung, beträgt die gesamte Dehnbarkeit der Zeile 8pt. Die ersten beiden Glue-Objekte werden um jeweils einen Punkt gedehnt, das Dritte um zwei.

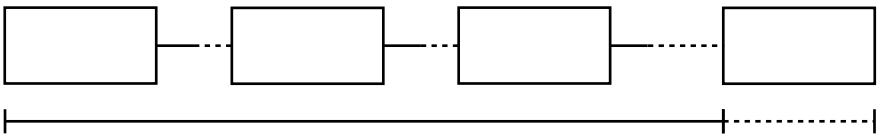


Abbildung 5.4: Anpassung von Wortzwischenraum. Die Linien zwischen den Kästchen symbolisieren Glue, der gestrichelte Teil zeigt die notwendige Dehnung, um von der optimalen Zeilenlänge (Linie darunter) auf die gewünschte Zeilenlänge (gestrichelter Teil) zu kommen.

Mathematisch ausgedrückt errechnet sich die Dehnung eines einzelnen Glue-Objektes δ_i aus n Elementen wie folgt. Die Dehnbarkeit des jeweiligen Glues d_i wird mit dem Bruch multipliziert, der sich ergibt, wenn man die notwendige Dehnung d_z der Zeile durch die Summe aller Dehnwerte $\sum_{j=0}^{n} d_j$ der n Elemente teilt.

$$\delta_i = d_i * \left(\frac{d_z}{\sum_{j=0}^{n} d_j} \right)$$

Daraus errechnet sich beispielhaft obiger Wert für δ_3:

$$\delta_3 = 4 * \left(\frac{4}{8} \right) = 2$$

Muß eine Zeile gestaucht, und nicht gedehnt, werden, wird die gleiche Rechnung durchgeführt, nur mit der Stauchbarkeit (auch Shrinkability genannt) des Glues und der Zeile. Die Berechnung dieser Wert erfolgt in UTA identisch zu TEX.

In TEX gibt es die Befehle `\hfil` und `\hfill`, bzw. die vertikalen Gegenstücke `\vfil` und `\vfill`. Diese fil-Befehle stellen speziellen Glue dar, der sich unendlich dehnen läßt. Die fill-Varianten, mit zwei *l*, lassen sich „noch unendlicher“ dehnen. Knuth sagt dazu:

> „You can think of it as if `\vfil` has one mile of stretchability, while `\vfill` has a trillion miles.“ [2, S. 71]

Dieser unendlich dehnbare Glue ist ein wichtiges Element im Satzsystem TEX, das überhaupt erst viele Funktionen ermöglicht, die TEX bietet. Dadurch, daß Wortzwischenraum nicht gleichmäßig angepasst wird, sondern proportional zur Stauch- bzw. Dehnbarkeit, lassen sich mit diesen Befehlen links- wie

rechtsbündiger, aber auch

zentrierter Text erzeugen.

Letzteres ist möglich, wenn links und rechts vom Text ein `\hfil` eingefügt wird. Wird in der Mitte des Textes ein `\hfil` eingefügt, entsteht dort eine Lücke; eine nützliche Funktion, wenn es darum geht am Ende eines Artikels den Autornamen rechtsbündig in der letzten Zeile zu setzen.

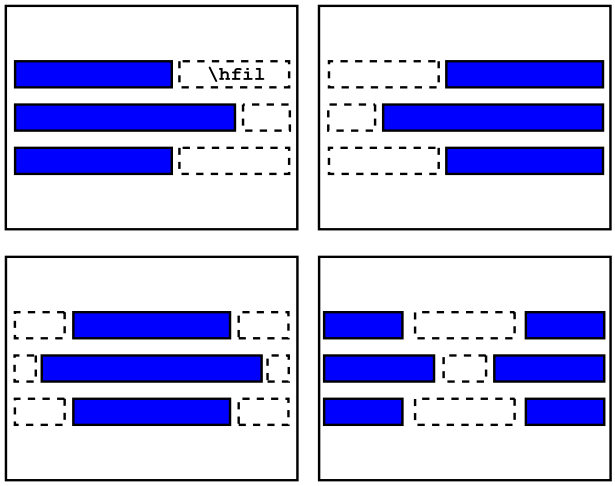


Abbildung 5.5: Einsatzmöglichkeiten von unendlich dehnbarem Wortzwischenraum

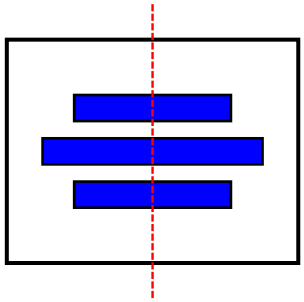


Abbildung 5.6: Ohne Glue zentrierter Text

Abbildung 5.5 zeigt die erwähnten Möglichkeiten noch einmal, wobei ersichtlich ist, daß immer eine feste Zeilenlänge aufgefüllt wird. Das entspricht dem Bild, nachdem TEX die Arbeit eines Schriftsetzers imitiert, der Zeilen in seinem Satzkasten zurechtrückt. In UTA spielt dieser Mechanismus eine wesentlich geringere Rolle. Der Grund liegt darin, daß sich UTA mehr an Techniken anlehnt, wie sie in Vektorzeichenprogrammen zum Einsatz kommen. So läßt sich zentrierter Text auch dadurch erreichen, daß die Mitte der Zeile an der Mittelachse des „Satzkastens“ ausgerichtet wird. Ähnliches gilt für links- und rechtsbündigen Text. Die Ausrichtung von Text wird in UTA als Layoutaufgabe betrachtet, die ein übergeordneter Algorithmus erledigen muß.

In TEX kann Glue negative Stretchability-Werte annehmen. Dadurch lassen sich leicht zwei Glyphen übereinander platzieren. Dazu werden sie in einer 0pt breiten Box gesetzt, der Glue nimmt einen negativen Wert an, der die positive Ausdehnung der beiden Glyphen kompensiert. Dies führt letztendlich zur Verschiebung des zweiten Glyphen nach links. Die Ergebnisse sind jedoch nur in wenigen Fällen zufriedenstellend, zudem enthalten die meisten Schriftarten Symbole wie das von Knuth in [2, Kap. 12] als Beipiel genannte Nicht-Gleich-Zeichen ($\neq$).

Unterschiede

Ein Justifiable kann in UTA keine negativen Stretch- oder Shrinkability-Werte annehmen. Das Übereinanderlegen von Glyphen muß rechnerisch erledigt werden. Ebenfalls gibt es in UTA keine Befehle. Der Effekt, den `\hfil` verursacht, ergibt sich durch einen sehr großen Stretchability-Wert (bspw. 10^6). TEX kennt negative und positive `\hfil`-Werte und addiert diese auf. So ist es möglich den `\hfil`-Befehl zu eliminieren, wenn der

gleiche negative `\hfil` Wert in der gleichen Zeile verwendet wird. Dies ist in UTA nicht möglich.

UTA kennt im Gegensatz zu TEX variable Elemente, deren Breite nur eine endliche Anzahl von Werten annehmen kann. In diesem Zusammenhang sprechen wir von *gestuft anpassbar* im Gegensatz zu *stufenlos anpassbar*.

Ein Austreibalgorithmus passt zunächst stufenlos anpassbare Justifiable-Objekte an. Sollte es dadurch nicht möglich sein, die komplette Zeilenlänge zu erreichen, werden die gestuft anpassbaren Objekte berücksichtigt.

5.2.8 Anchor

Ein Anchor [23], bzw. Anker, dient in erster Linie als Platzierungshilfe für Glyphen. Anchors sind nicht mit den Ankern zu verwechseln, die in gängigen Textverarbeitungsprogrammen vorhanden sind und dort dazu dienen, Gleitobjekte an eine bestimmte Textpassage zu binden (auch wenn sie für ähnliche Zwecke verwendet werden können). Vielmehr kann ein Glyph beliebig viele Anker unterschiedlicher Anker-Klassen besitzen. In Thai könnte für jede mögliche Position eines Vokals eine extra Anker-Klasse definiert werden.

Ein Anker beschreibt einen Punkt relativ zum Ursprung des Glyphen. Diese Position kann auch dynamisch, in Abhängigkeit von externen Faktoren, berechnet werden. Das Ziel ist die Ankerpunkte zweier Glyphen übereinander zu legen um Glyphen korrekt zueinander ausrichten zu können. Dadurch lassen sich dynamisch zusammengesetzte Glyphen erzeugen.

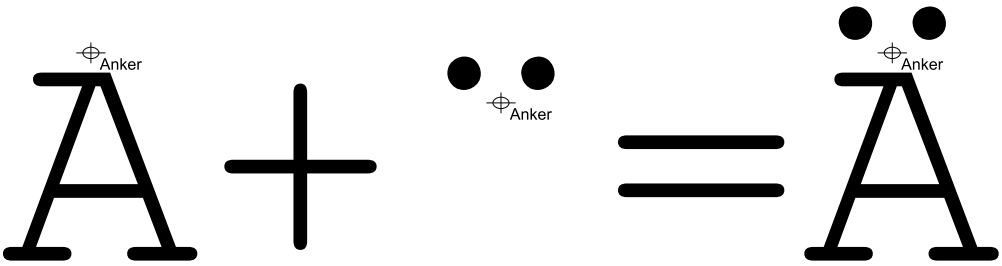


Abbildung 5.7: Positionierung durch Anker. Viele Schriftarten bieten vorgefertigte Glyphen für häufig benutzte Zeichen wie ein „Ä“. Ein alternativer Weg ist die Benutzung von Ankern.

Das Ankerkonzept kann nicht nur für Glyphen sondern auch in größerem Zusammenhang mit Grafiken verwendet werden. Zum Beispiel läßt sich die Platzierung von Bildbeschreibungen damit regeln. Beim mathematischen

Schriftsatz können sie ebenfalls hilfreich sein, wenn komplexe Ausdrücke auf Summenzeichen und dergleichen ausgerichtet werden müssen. Darum kann in UTA jede Box Ankerpunkte besitzen.

5.2.9 Item

Das Item ist bereits aus vorangehendem Kapitel als zentrales Element des Umbruchs bekannt. Hier erfährt es eine funktionale Erweiterung. Vor dem Umbruch besitzt ein Item mehrere mögliche Zustände gleichzeitig, die endgültige Position innerhalb einer Zeile steht erst nach dem Umbruch fest.

Ein Item repräsentiert eine beliebige Anzahl von Box-Objekten, darunter möglicherweise Embedding-Level (s. nächster Abschnitt) oder solche variabler Breite. Ein Item weiß, welche dieser Elemente bei welcher Position innerhalb einer Zeile überflüssig sind. In erster Linie ist damit Leerraum am Anfang/Ende gemeint. So wird ein Leerzeichen am Ende eines Items überflüssig, wenn die endgültige Position am Zeilenende ist. In diesem Fall darf dieser Leerraum nicht in die Austreibung einbezogen werden (außer derartiges Verhalten ist erwünscht).

5.2.10 Embedding-Level

Embedding-Level sind der Kern des multilingualen Schriftsatzes in UTA. Ein solches Level, im Rest des Textes wird synonym dafür der Begriff *Ebene* gebraucht, ist ein Spezialfall einer Box und kann Items enthalten. Eine Ebene kann nicht sich selbst enthalten. Als Box hat ein Embedding-Level einen Ursprung, Ankerpunkte und eine Ausdehnung. Embedding-Level können relativ zueinander platziert werden, ohne daß dabei die exakte Ausdehnung der einzelnen Ebenen bekannt sein muß.

Ein Beispiel verdeutlicht das Konzept der relativen Platzierung und der Embedding-Level. Gesetzt werden soll `Arabisch123Text`, wobei die Ziffern als ɥɔsqɐɹ zu betrachten sind. Um das Referenzieren einzelner Zeichen zu vereinfachen wurde auf Leerzeichen verzichtet.

ɥɔsqɐɹA ist bidirektional, die Buchstaben müssen von rechts nach links und die Zahl 123 von links nach rechts platziert werden. Da der komplette Text ɥɔsqɐɹ ist, wird er auch von dem gleichen Script gesetzt. Das Resultat soll so aussehen:

ɥɔsqɐɹA12
txeT3

Die Zahlen markieren die Embedding-Level, wie sie durch den Unicode Bidi-Algorithmus ermittelt wurden.

```
Arabischer123Text
1111111112221111
```

Das Embedding-Level 1 beeinhaltet sämtliche Buchstaben. Das (lokale) Koordinatensystem ist so ausgerichtet, daß die x-Achse nach links zeigt. Das Embedding-Level 2 beeinhaltet die Zahl 123. Die x-Achse des Koordinatensystems zeigt nach rechts.

Zum Ablauf: Wird über den Text Arabischer123Text in logischer Reihenfolge iteriert, kann das Script zunächst ein Glyph hinter dem anderen platzieren. Diese Platzierung geschieht mit relativen Koordinaten, lediglich das erste Glyph „A“ muß absolut zum lokalen Koordinatensystem des Embedding-Level 1 ausgerichtet werden. Trifft der Typesetter auf den Beginn des Embedding-Level 2, stellt er einen Wechsel in der Leserichtung fest. Folglich muß er ein neues Embedding-Level-Objekt erzeugen, das sich in der Orientierung des Koordinatensystems unterscheidet. Die genauen Regeln zur Erzeugung von Embedding-Leveln folgen später.

Die Zahlen „1“, „2“ und „3“ werden im Embedding-Level 2 gesetzt. Die „1“ relativ zum Ursprung, die „2“ relativ zur „1“, die „3“ relativ zur „2“. Damit kann ein Level höher mit der Platzierung des restlichen Textes fortgefahren werden. Das Glyph „T“ wird relativ zum Embedding-Level 2 an sich gesetzt. Und zwar links davon, wie es der Orientierung der x-Achse des ersten Levels entspricht. Abbildung 5.8 zeigt das Ergebnis. Es ist zu beachten, daß die Breite b_{EL} des Embedding-Level 2 dynamisch berechnet werden muß, d. h. hier kann keine einfache Translation mit festen Werten angegeben werden.

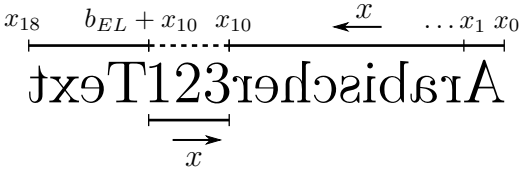
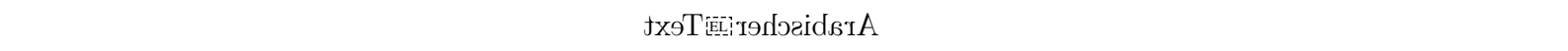


Abbildung 5.8: Platzierung von Glyphen in unterschiedlichen Embedding-Leveln. Die Position $b_{EL} + x_{10}$ muß dynamisch bestimmt werden. Die kleinen Pfeile geben die Orientierung der x-Achse im jeweiligen Embedding-Level (1 oben, 2 unten) an.

Vom Embedding-Level 1 aus betrachtet sieht das Resultat wie folgt aus. Das Kästchen symbolisiert das Embedding-Level 2.

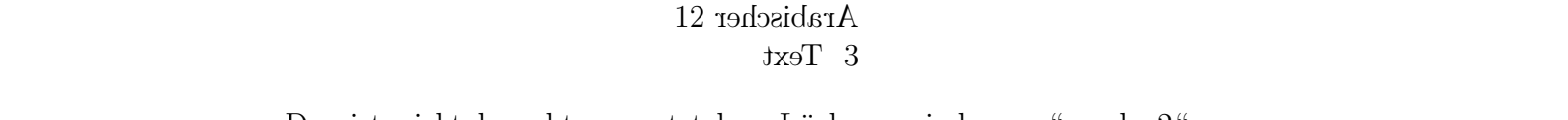


Werden die Embedding-Level von unterschiedlichen Script-Modulen gesetzt ist das Vorgehen identisch.

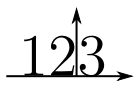
Damit ist der Satzvorgang vorerst abgeschlossen, die Glyphen befinden sich in der korrekten optischen Anordnung. Durch einfache Transformationen lassen sich ausgehend vom ersten gesetzten Glyph absolute Koordinaten für alle folgenden Glyphen ermitteln. Erst dann ist die absolute Breite untergeordneter Embedding-Level (Ebene 2 in obigem Beispiel) notwendig.

In den meisten Fällen ist für die relative Platzierung eines Rechtecks hinter einem Embedding-Level die Breite des Levels ausschlaggebend. So wie in obigem Beispiel. Bislang unberücksichtigt blieb der Zeilenumbruch. Hierzu muß das Modell etwas verfeinert werden. Erlauben wir aus Demonstrationszwecken den ansonsten unsinnigen Umbruch zwischen „2“ und „3“ (die Stelle ist mit einem kleinen Dreieck markiert).

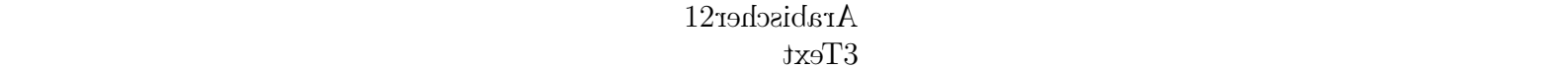
Unter bisheriger Annahme, daß die Breite eines Levels bei relativer Platzierung maßgeblich ist, erhalten wir:



Das ist nicht korrekt, es entstehen Lücken zwischen „1“ und „2“ sowie zwischen „3“ und „T“. Das Problem läßt sich beheben. Der richtige Wert ergibt sich durch Berücksichtigung des Umbruchpunktes. In diesem Fall fungiert der Umbruchpunkt als virtuelle y-Achse. Die Breite der Glyphen in negativer x-Achse ist die Breite, die es vor dem Zeilenumbruch zu berücksichtigen gilt. Die Breite in positiver x-Richtung ist die für den Beginn der neuen Zeile relevante.



Dies führt zum korrekten Ergebnis.



Ähnlich vorzugehen ist, wenn in einem Embedding-Level mehr als nur ein Umbruch vorkommt. Dann ist der erste und der letzte Umbruch zu berücksichtigen. Im Grunde ist *genau ein* Umbruch nur ein Spezialfall, bei dem erster und letzter zusammenfallen.

Kritisch sind nur Zeilen, die über die Grenze zwischen zwei Ebenen verlaufen. Bei Zeilen, die sich komplett innerhalb einer Ebene befinden, reicht es, die erste Box der neuen Zeile an den Zeilenanfang zu setzen. Das eigentliche Umbrechen besteht damit aus einer Translation eines Box-Ursprungs und einer evtl. notwendigen Berechnung von Teilbreiten eines Embedding-Levels.

5.3 Ablauf

Die Liste zeigt den Ablauf eines kompletten Satzvorgangs. Dieser Ablauf ist allgemein und unabhängig von UTA.

1. Vorbereitung: Einlesen und Auszeichnen der Daten.
2. Parallel:
   a) Setzen der Glyphen unter Berücksichtigung von Metrikdaten und Auszeichnungen des Textes wie Schriftgröße, sowie Besonderheiten der jeweiligen Schrift.
   b) Finden aller *möglichen* Umbruchstellen
   c) Finden aller in der Breite variablen Elemente
3. Umbrechen, evtl. Rücksprung zu 2. wegen Trennung; Finden *geeigneter* Umbruchstellen durch Berechnungen zur Zeilenbreite
4. Blocksatz; Anpassung der in Schritt 2c gefundenen Elemente.
5. Ausrichten des Textes an einem Pfad
6. Ausgabe

Variationen dieses Ablaufs sind nur sehr begrenzt möglich und mit dem Risiko von Funktionseinschränkungen verbunden. Der Umbruch, Satzvorgang und Blocksatz sind eng miteinander verknüpft. So muß der Text zwingend gesetzt sein um überhaupt umgebrochen werden zu können. Die Austreibung wiederum kann erst erfolgen, wenn die endgültigen Umbruchstellen festgelegt sind. Gleichzeitig wirkt sie sich auf den ursprünglichen

Satz aus. UTA verfolgt dennoch eine weitestgehende Trennung der einzelnen Schritte, da die Lösung jedes Teilproblems aufwendig ist und die Probleme sehr unterschiedlich strukturiert sind.

Optional kann der ursprüngliche Text nach Satz und Umbruch in die optische Ordnung gebracht werden, wie im Unicode Bidi-Algorithmus beschrieben. Diese Funktion ist nützlich für interaktive Systeme, bei denen Cursorpositionierung notwendig ist. Man denke an die Cursortasten und bidirektionalen Text. Hier läßt sich unterscheiden, ob ein Druck auf die Rechtstaste einen Vorschub in optischer oder logischer Richtung bewirkt. Da diese Funktion sich auf interaktive Systeme beschränkt und auch keinen Einfluß auf den Schriftsatz hat, ist sie nicht Bestandteil von UTA.

UTA kommt im allgemeinen Ablauf, wie er oben beschrieben wird, nach dem Einlesen und Auszeichnen der Daten ins Spiel. Die Rückgabe des Ergebnisses an das Fremdsystem erfolgt nach Schritt 4, der Austreibung. Hier nun was dazwischen, innerhalb von UTA, geschieht.

1. Umwandlung des Textes in *uninitialisierte* Glyphen
2. Zuteilung der Glyphen nach Locale zu einem Script
3. Verarbeitung durch die Script-Module:
   a) Initialisierung & Satz der Glyphen
   b) Erzeugung von Items und Meldung dieser an den Umbruchalgorithmus
4. Finden der optimalen Umbruchstellen durch den Umbruchalgorithmus
5. Parallel:
   a) Bestimmung der endgültigen Items und Justifiables
   b) Austreibung
   c) Rückgabe an das Fremdsystem

5.3.1 Eingabe aus dem Fremdsystem

Kein Satzsystem kommt ohne Zusatzinformationen im Datenstrom aus. Der Datenstrom besteht dabei aus einem attributierten Unicode-String. D. h. jedes einzelne Zeichen kann beliebig viele Attribute besitzen. Viele, wenn auch nicht alle, müssen beim Satz berücksicht werden. Bspw. ist in

einem Wort eine fi-Ligatur dann nicht möglich, wenn die beiden Buchstaben „f“ und „i“ in unterschiedlicher Schriftgröße dargestellt werden sollen (fi). Auch eine unterschiedliche Farbe reicht dafür aus. Die Auszeichnung der Daten wird nicht von UTA übernommen, sondern vom Fremdsystem.

Der Text muß NFKC-normalisiert vorliegen. Dies ist die kleinste Untermenge des Unicode-Zeichensatzes ohne jegliche Kompatibilitätsformen. Dieses Vorgehen impliziert, daß unter Umständen zusätzliche Attribute notwendig sind, wenn seitens des Anwenders ausdrücklich bestimmte Effekte erwünscht sind. UTA vertritt die Ansicht, daß ein Script alle Aspekte einer Schrift handhabt und der Anwender möglichst keinen Einfluß auf den Satzvorgang zu nehmen braucht. Ligaturen sind hier einmal mehr ein gutes Beispiel. UTA honoriert es nicht, wenn ein Anwender sich die Arbeit macht Code Points für verschiedene Ligaturen auszuwählen. Diese Code Points werden durch die NFKC-Normalisierung in ihre Einzelzeichen aufgeteilt. Zusätzliche, den neu entstandenen Zeichen zugeordnete, Attribute können die Information, daß es sich um eine Ligatur handelt, erhalten. Allerdings muß ein Script diese Attribute auch interpretieren. Auf diese Weise ist das Eingabeformat für alle Implementierungen einer Schrift identisch. Die Tatsache, daß es sich um die kleinste Untermenge des Unicode-Zeichensatzes handelt, soll helfen, Implementierungen zu erleichtern. Zeichen, die aus Kompatibilitätsgründen in Unicode aufgenommen wurden, brauchen in einem Script nicht berücksichtigt werden (vgl. Abbildung 5.9).

Die `typeset`-Methode des Typesetters ist der Eintrittspunkt für das Fremdsystem. Sie bekommt den attributierten String übergeben und liefert das Ergebnis in Form eines Embedding-Level zurück.

5.3.2 Verarbeitung

Zunächst wird der ausgezeichnete String in eine Liste von uninitialisierten Glyphen umgewandelt, der internen Repräsentation der Daten. Zu diesem Zeitpunkt sind die Daten noch uninterpretiert, die erzeugten Glyphen haben keinen Index oder dergleichen zugewiesen bekommen. Lediglich Attribute und der ursprüngliche Code Point sind abrufbar – Daten, die durch den ganzen Satzvorgang erhalten werden. Attribute können hinzugefügt oder geändert werden.

Die uninitialisierten Glyphen werden daraufhin dem Script zugeteilt, welches das gleiche Locale besitzt. Welchem Locale ein Glyph, bzw. das entsprechende Schriftzeichen, zuzuordnen ist, bleibt dem Typesetter überlassen. Allerdings gibt es zwei grundlegende Verfahrensweisen. Zum einen sind die Code Points in Blöcke unterteilt, die einem Schriftsystem zuor-

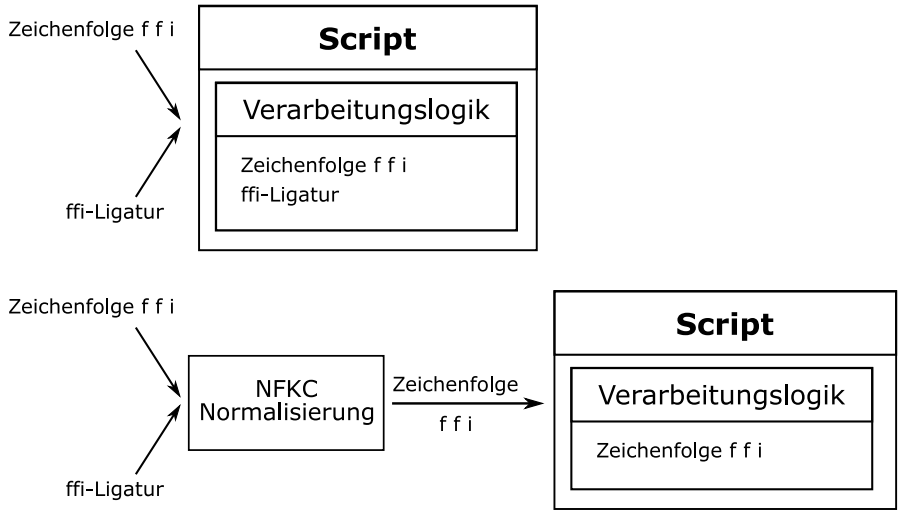


Abbildung 5.9: Vereinfachung der Implementierung eines Scripts durch Normalisierung. Ohne (oben) und mit vorangehender Normalisierung (unten), wodurch die notwendige Verarbeitungslogik in einem Script verringert wird.

denbar sind, zum anderen kann das Fremdsystem UTA das Locale per Attribut mitteilen.

Werden zwei Glyphen vom Typesetter an die schriftspezifische Schicht weitergeleitet, kann es zu folgenden Situationen kommen:

1. Locale und Ordnung des Embedding-Levels sind identisch (gleiche Sprache, gleiche Schreibrichtung)

2. Locale ist identisch, Ordnung unterschiedlich (Wechsel in der Schreibrichtung; bidirektionale Schrift)

3. Locale ist unterschiedlich, Ordnung identisch (gleiches Schriftsystem, andere Sprache)

4. Locale und Ordnung unterschiedlich (anderes Schriftsystem, andere Sprache)

Der Typesetter ist dafür verantwortlich, daß dem Script das korrekte Embedding-Level mitgeteilt wird. Es gibt stets ein aktives Embedding-Level und beliebig viele inaktive. Sie liegen auf einem Stack. Erhöht sich die Ordnung des Embedding-Levels von einem Glyph zum nächsten, entspricht dies dem Erzeugen eines neuen Embedding-Levels, das auch gleichzeitig das aktive wird. Wird die Ordnung kleiner, ist zumindest das bis da-

hin aktive Embedding-Level abgeschlossen und wird vom Stack entfernt. Es sei darauf hingewiesen, daß es nötig sein kann, weitere Embedding-Level abzuschließen, sollte der Sprung mehr als eine Ordnung überschreiten.

Vor einem Scriptwechsel (unterschiedliches Locale) wird das derzeit aktive Script informiert, daß es das nächste Glyph nicht mehr zu verarbeiten hat. Gleichzeitig wird das nächste Script „aufgeweckt“ und das aktive Embedding-Level mitgeteilt. Stellt der Typesetter fest, daß ein neues Embedding-Level erzeugt werden muß, erfragt er dies vom Script, das aufgeweckt wird. Bei der Abfrage wird dem Script die Ordnung der neuen Ebene mitgeteilt, da nur das Script weiß, welche Orientierung das lokale Koordinatensystem der Ebene besitzen muß. Daraufhin können solange Glyphen übergeben werden bis erneut einer der oben genannten Fälle eintritt.

Wir befinden uns nun in der schriftspezifischen Schicht, in der die Glyphen gesetzt werden. Das Script interpretiert dazu die mit dem Glyph verbundenen Attribute wie Schriftart, -größe etc. Satz heißt dabei nicht allein die Berechnung der Position unter Berücksichtigung des Koordinatensystems des Embedding-Levels, sondern auch Initialisierung des Glyphs mit einem Index. Glyphsubstitutionen werden innerhalb eines Scripts auf Glyphebene durchgeführt. Das heißt, ein Script erzeugt ggf. ein neues Glyph, welches Referenzen auf die ersetzen Glyphen besitzen muß. Die Entstehungsgeschichte kann somit jederzeit zurückverfolgt werden. Ein Script muß u. U. mehrere Glyphen zwischenspeichern um Substitutionen etc. durchführen zu können (kontextabhängige Verarbeitung).

Zu diesem Zeitpunkt werden auch die Items durch das Script gebildet und an den Umbruchalgorithmus weitergeleitet. Die Items werden zusätzlich dem aktiven Embedding-Level hinzugefügt, welches als Container fungiert. Auch der aktiven Ebene untergeordnete Embedding-Level sind stets Bestandteil eines Items und damit Teil der Ebene.

An dieser Stelle ein kleiner Hinweis zu möglichen Implementierungen: gerade das Erzeugen von Objekten ist in Programmiersprachen meist mit hohem Zeitverlust verbunden. Daher macht es Sinn Glyphen nach jedem Satzvorgang wiederzuverwenden.

Nachdem alle Glyphen gesetzt sind, können die optimalen Umbruchstellen bestimmt werden, wie im vorangehenden Kapitel ausführlich besprochen. Danach kommt es zur (optionalen) Austreibung und der Rückgabe an das Fremdsystem.

5.3.3 Austreibung und Ausgabe

Wie wir wissen hat ein Item vor dem Umbruch vier potentielle Zustände, charakterisiert durch die Position innerhalb einer Zeile. Nach dem Umbruch erhält es einen dieser Zustände, womit auch die Justifiables innerhalb des Items feststehen, die für die Austreibung verwendet werden können. Daher wird mit der Festlegung der Position eine Liste der Justifiables zurückgegeben. Der Austreibalgorithmus korrigiert die Breite der einzelnen Justifiables durch einen Aufruf der `setWidth`-Methode. Sind alle Positionen gesetzt und alle Justifiables angeglichen, ist der Satzvorgang beendet. Die `typeset`-Methode des Typesetters ist abgearbeitet, das Fremdsystem erhält das Embedding-Level mit der niedrigsten Ordnung als Satzergebnis, welches sämtliche Glyphen, Grafiken und Embedding-Level höherer Ordnung enthält.

5.4 Parallelen zu anderen Technologien

Auch hier sind, wie beim Umbruch, gewisse Parallelen zu anderen Technologien vorhanden. So weist ein Embedding-Level Ähnlichkeit mit einer Gruppe auf. Gruppierung spielt bei Vektorformaten und -anwendungen eine wichtige Rolle. Gruppen sind eine Zusammenfassung von Einzelelementen, auf die eine Transformation angewendet werden kann.

In Javas Bibliothek für grafische Benutzeroberflächen, Swing, werden einzelne Komponenten (Schaltflächen, Eingabefelder, usw.) in Containern platziert. Wird ein solcher Behälter nach seiner Ausdehnung befragt, errechnet er diese, indem er die Ausdehnung seiner Kindelemente abruft. Wie in einem Embedding-Level werden absolute Maßangaben erst gebildet, wenn sie benötigt werden.

Im Bezug auf XSL sind die Ähnlichkeiten noch gravierender. Eine ungefähre Übereinstimmung läßt sich zwischen Box und Area, Glyph und Glyph-Area und Embedding-Level und Inline-Areas feststellen. In Abschnitt 5.8 *Unicode BIDI Processing* des XSL-Standards ist folgender Satz zu finden:

> „[...]a sub-sequence of Arabic characters in an otherwise English paragraph would cause the creation of an inline formatting object with the Arabic characters as its content[...]“ [12]

Die Erzeugung von FO-Objekten spiegelt sich als solche auch bei der Erzeugung der Areas wider. Das Vorgehen entspricht dem Satz von bidi-

rektionalem Text in UTA. In zukünftigen UTA Versionen ist zu untersuchen, wie weit sich eine Annäherung an XSL-Struktur und Begrifflichkeit verwirklichen läßt. Wo derzeit noch eine Konvertierung zwischen Modellen stattfinden muß, könnte diese später entfallen.

Pango[3] ist das Textmodul von Gnome[4]. Gnome ist wiederum eine Sammlung von Bibliotheken zur Erstellung grafischer Oberflächen unter GNU/Linux. Pango verfolgt ähnliche Ziele wie UTA und will auf lange Sicht nicht nur Unterstützung für alle Schriftsysteme bieten, sondern auch in der Lage sein, diese in hoher Qualität zu setzen. In Pango läßt sich die Unterstützung für ein neues Schriftsystem durch ein System hinzufügen, das dem Script-Ansatz in UTA ähnelt. Pango schaltet sog. Engines zu, die schriftspezifische Aufgaben übernehmen.

5.5 Anmerkungen

Die folgenden Anmerkungen gehen auf Probleme beim Satz von mehrsprachigem Text ein. Die Aussagen sind nicht verpflichtend sondern präsentieren lediglich mögliche Lösungen.

5.5.1 Bidi, Anführungen & Umbruch

Der Unicode Bidi-Algorithmus verändert die Zuordnung der Anführungszeichen nicht. In [6, Kap. 4] widmen sich Knuth und MacKay diesem Thema, als Lösung schlagen sie vor, die Anführungen dem eingebetteten Text zuzuordnen. Die Anführungen sind dann die der eingebetteten Sprache. Hier die zwei Alternativen im Vergleich:

Er sagte „-ıu ǝuıǝ ʇsı ɔɥsıqɐɹA ǝɥɔɐɹdS ǝʇʇǝssɐɹǝʇ“, jedoch muß man sie erst lernen!

Er sagte -ıu ǝuıǝ ʇsı ɔɥsıqɐɹA„ ”ǝɥɔɐɹdS ǝʇʇǝssɐɹǝʇ, jedoch muß man sie erst lernen!

Hier sind die Grenzen dessen erreicht, was sich reglementieren läßt. Welche Zuordnung bevorzugt wird, mag an dieser Stelle ein wenig Geschmackssache sein. Korrekter erscheint das Vorgehen von Unicode. Die Anführungen rahmen den eingebetteten Text ein, was dem Lesefluß förderlich ist.

Bei der Unicode-Lösung muß das Satzsystem darauf achten, daß kein Umbruch zwischen Anführung und eingebettetem Text erfolgt, da dann die Anführung allein am Ende einer Zeile stehen kann, dieses Problem entfällt bei Knuth' Lösung.

[3] URL http://www.pango.org
[4] URL http://www.gnome.org

5.5.2 Vertikaler Satz

Der vertikale Satz unterscheidet sich prinzipiell wenig vom horizontalen. Nur spielt bei der Platzierung der Glyphen nicht die Breite, sondern die Höhe die entscheidende Rolle beim Umbruch. Die Breite kommt bei den Spaltenabständen eine ähnliche Bedeutung zu, wie der Höhe bei den Zeilenabständen. Im Groben handelt es sich dabei um 90° gedrehten horizontalen Satz mit ein paar Eigenheiten. Zu beachten ist, daß einige Zeichen, wie Klammern und Bindestriche tatsächlich gedreht werden müssen, während normale Zeichen ihre Orientierung behalten.

Um keinen eigenen Verarbeitungszweig für vertikale Schrift einführen zu müssen, findet in ATSUI (vgl. [24, S. 90]) eine spezielle Methode Anwendung. Dabei werden vorgefertige, um 90° gegen den Uhrzeigersinn rotierte, Versionen der Glyphen aus Schriftarten genommen (oder zumindest deren Metrikdaten), horizontal gesetzt und dann um 90° im Uhrzeigersinn gedreht. Da nicht nur Apple, sondern auch Microsoft in seiner Unicode-Schnittstelle Uniscribe Text rotiert, sind dafür entsprechende Tabellen in TrueType und OpenType vorhanden.

Prinzipiell ist derartiges Vorgehen in UTA zwar möglich und die konkrete Umsetzung liegt im Ermessen des Scripts, der intuitivere Weg dürfte allerdings sein, die Glyphen direkt vertikal zu platzieren.

Nicht jeder beliebige Text kann in einem Durchgang gesetzt werden. Gemeint ist damit, daß ein Typesetter (mit zugewiesenem Umbruchalgorithmus) oder besser, *ein* Zustand des Typesetters, nicht ausreichend ist. Dies ist bspw. der Fall, wenn größere Mengen vertikalen Textes in horizontalen eingebettet werden sollen. Eine einzelne Spalte bereitet keine Probleme. Muß hingegen vertikaler Text umgebrochen werden, so kann dies nicht ohne weiteres vom gleichen Umbruchalgorithmus erledigt werden. Die Ausdehnung der Items beschreiben in horizontalen Schreibrichtungen die Breite, in vertikalen die Höhe von Objekten.

Abbildung 5.10 zeigt das gewünschte Ergebnis. Der rote Strich in der ersten Spalte zeigt den berechneten Umbruchpunkt, wenn die Schreibrichtung nicht berücksichtigt wird. Die Länge des vertikalen Textes würde der Breite des horizontalen zugeschlagen, der gestrichelte Text würde in einer neuen Zeile platziert. Der schwarze Strich zeigt den korrekt berechneten Umbruchpunkt. Dies ist sicherlich ein eher theoretisches Problem, da große vertikale Textbereiche innerhalb eines horizontalen und umgekehrt nicht viel Sinn machen. Es zeigt aber, wo eine potentielle Fehlerquelle liegen kann. Eine mögliche Lösung ist, daß ein Script, das mehrere Spalten Text

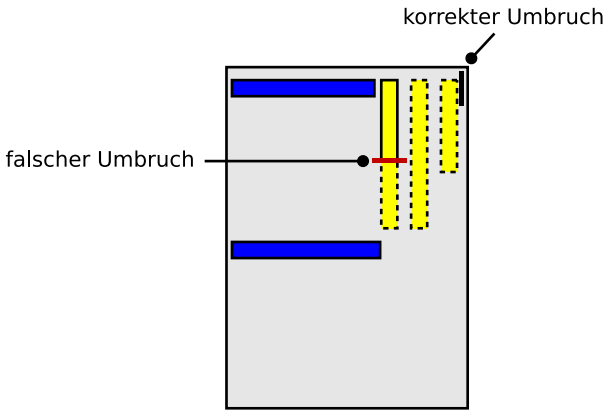


Abbildung 5.10: Falsche Umbruchpunktberechnung durch Missachtung der Schreibrichtung

vertikal zu setzen hat, eine eigene Instanz des Umbruchalgorithmus verwendet.

6 Qualitätsmanagement

UTAs erklärtes Ziel ist es, einfache wie komplexe Implementierungen von Satzsystemen zu ermöglichen. Zudem sollen unabhängige Teile austauschbar sein, um schrittweise Verbesserungen zu erlauben, ohne jedesmal, mit fortschreitendem typografischem Bedarf/Wissen, den kompletten Code umarbeiten zu müssen.

Der Anspruch, allen Qualitätsbedürfnissen gerecht zu werden und die Austauschbarkeit von einzelnen Komponenten kann gewisse Komplikationen mit sich bringen. Nehmen wir an in einer ersten Ausbaustufe wird ein einfaches Script für lateinische Sprachen eingesetzt, das keinerlei Rücksicht darauf nimmt, ob Leerraum dehn- oder stauchbar ist. Jeder Leerraum bekommt die gleiche, unabänderliche Breite; einzig am Zeilenanfang und -ende wird er zu Null. Ebenso eingeschränkt ist der Umbruchalgorithmus. Er berücksichtigt variierende Breiten nur am Zeilenanfang und -ende. Beide Algorithmen sind aufeinander abgestimmt. Wird nun das Script durch eine komplexere Variante ersetzt, welches Items variabler Breite erzeugt, entsteht eine Inkompatibilität. Der Umbruchalgorithmus wertet die hinzugekommene Information nicht aus.

6.1 Funktions- und Referenzlisten

Um in solchen Fällen den Überblick zu behalten gibt es die Qualitätsmanagement-Komponente (QM) in UTA. Zentraler Bestandteil ist die Klasse QManageable. Folgende Komponenten implementieren das Interface und stellen eine Liste der unterstützten Funktionen, die sogenannte FeatureList, zur Verfügung:

- LinebreakAlgorithm
- JustificationAlgorithm
- Typesetter
- Script

Die FeatureList besteht aus einzelnen Feature-Objekten. Ein Feature hat einen eindeutigen Schlüssel und wird über ein Singleton-Pattern erzeugt. Durch einen Vergleich von Funktionslisten wird das Problem von inkompatiblen Komponenten behoben. So würde in obigem Fall die Funktionsliste des Scripts mit der des Umbruchalgorithmus abgeglichen, wobei nur die Funktionen berücksichtigt werden, die in diesem Zusammenhang auch sinnvoll sind.

Ein anderes Problem ist indes noch nicht gelöst. Oft kann nicht gesagt werden, was für eine Satzqualität von einer Anwendung zu erwarten ist. Reicht sie für den benötigten Zweck aus? Lassen sich damit professionelle Dokumente erzeugen? Bietet das System ausreichende Unterstützung für die benötigten Sprachen? So könnte eine europäische Implementierung sehr gute arabische Resultate erzielen, allerdings keine Unterstützung für Nastaliq mitbringen. Wäre es in diesem Fall möglich ein externes Nastaliq-Modul einzubinden?

Um die typografische Qualität abschätzen zu können, die von einem Satzsystem zu erwarten ist, reicht es nicht aus, lediglich Funktionslisten von Teilkomponenten miteinander zu vergleichen. Daher kommt zum Konzept der Funktionsliste das der Referenzliste hinzu. Sie legt fest, welche Funktionen eine Komponente bieten muß, um alle notwendigen typografischen Grundfunktionen zu unterstützen, bzw. welche benötigt werden, um auch höchsten Ansprüchen gerecht zu werden. Der Funktionsumfang einer Komponente kann dann mit der Referenzliste abgeglichen werden indem eine FeatureList mit der ReferenceList verglichen wird. Je nachdem wie viele der Referenzfunktionen unterstützt werden, ergeben sich sechs sog. Support-Level:

`UNKOWN_SUPPORT` Es ist nicht möglich das Support-Level zu berechnen.

`NO_SUPPORT` Keine der unterstützten Funktionen befindet sich in der Referenzliste.

`MINIMAL_SUPPORT` Einige Grundfunktionen werden unterstützt, aber nicht alle. Zusätzliche Funktionen, die keine Grundfunktionen sind, werden nicht honoriert.

`BASIC_SUPPORT` Es werden genau die Grundfunktionen unterstützt.

`ADVANCED_SUPPORT` Alle Grundfunktionen und zumindest eine weitere werden unterstützt.

`FULL_SUPPORT` Das höchste Support-Level. Alle Funktionen sind vorhanden, die für hochqualitativen Satz notwendig sind (evtl. auch Funktionen darüber hinaus).

Zur Veranschaulichung des Konzepts vergleicht Tabelle 6.1 einige Anforderungen der deutschen und arabischen Sprache. Mit *ja* gekennzeichnete Felder sind zwingend notwendige Grundfunktionen, *nein* kennzeichnet Funktionen die nie benötigt werden, *optional* für qualitativ hochwertigen Schriftsatz notwendige Eigenschaften.

	Deutsch	Arabisch
Ligaturen	optional	ja
Diakritische Zeichen	ja	ja
Trennung	optional	nein
Bidirektionalität	nein	ja
Schreibschrift	optional	ja

Tabelle 6.1: Anforderungen verschiedener Schriftsysteme im Vergleich

Nicht nur für die Sprache, auch für die anderen Systemkomponenten definieren entsprechende Referenzlisten die Anforderung für höchste Qualität. Ein hochqualitativer Umbruchalgorithmus muß zumindest folgende Funktionen bieten:

- Berücksichtigung unterschiedlicher Breitenangaben
- Berücksichtigung der Position eines Items innerhalb der Zeile
- Berücksichtigung von Strafwerten bei Umbruch
- Globale Optimierung was kompatible Zeilen einschließt

Eine darüber hinaus gehende Funktion wäre die Möglichkeit die „Lockerheit“ eines Absatzes zu beeinflussen, wie es der TEX-Befehl `\looseness` erlaubt. Dabei wird absatzweit der Leerraum variiert. Ein langer Absatz kann damit ohne größere optische Einbußen um eine Zeile verkürzt oder verlängert werden.

6.2 Abrufen der Information

Der QualityManager sammelt diese Informationen und stellt sie in Form eines Berichts zur Verfügung. Dieser QualityReport kann bspw. in einheit-

lichen grafischen Konfigurations- und Informationskomponenten verwendet werden. Auf diese Weise wird das Leistungsvermögen einer Satzmaschine bewertbar. Das wirkt sich auch positiv auf die Einarbeitungszeit von Entwicklern aus, die ein System erweitern wollen. Es versetzt sie in die Lage, einen Blick in das Innere des Satzsystems zu werfen.

Der QualityManager verwaltet die Referenzlisten der unter Abschnitt 6.1 aufgeführten Komponenten. Dazu muß es entsprechende Getter- und Settermethoden geben. Die `qualityOf`-Methode des Managers bekommt eine Typesetter-Instanz übergeben und liefert den erwähnten QualityReport.

An dieser Stelle sei darauf aufmerksam gemacht, daß das QM-Paket eine *Einschätzung* der Qualität ermöglicht. Eine absolute Aussage über die Qualität der endgültigen Ausgabe kann es nicht geben. Ein Algorithmus mag zwar in der Lage sein, einzelne Glyphen miteinander zu verbinden, um eine Schreibschrift zu erzeugen (Cursive Linking), allerdings nützt diese Funktion nichts, wenn die verwendete Schriftart keine Unterstützung dafür bietet. Ebenso gibt es einen Unterschied zwischen der Berücksichtigung von einzelnen Optionen und der Qualität der Umsetzung. Ein Umbruchalgorithmus, der zwar alle Funktionen implementiert, kann trotzdem schlechte Ergebnisse liefern. Der Qualitätsbericht sagt aus, ob entsprechende Bemühungen unternommen wurden, nicht in wie weit sie von Erfolg gekrönt waren.

7 Ausblick

Die zum 18.8.2004 aktuelle UTA-Version wird als Milestone 1 festgelegt. Der Grund dafür ist der Abschluß des ersten großen Entwicklungsschritts.

In Version M1 bietet UTA ein Framework für die grundlegenden Dienste eines Satzsystems. Folgeversionen von UTA werden mehr Dienste zur Verfügung stellen. Es folgen Anmerkungen zu derzeitigen Einschränkungen von UTA und möglichen Ergänzungen.

7.1 Seitenumbruch

Ähnlich dem Zeilenumbruch gestaltet sich das Umbrechen von Seiten und Spalten. Die Anforderungen ähneln sich. Kompatible Spalten sind solche, deren Zeilenanzahl möglichst gleich ist. Wichtig sind dabei vor allem die korrekte Ausrichtung der ersten und letzten Zeile, Zeilenabstände sollen möglichst gleichmäßig sein. Wie beim Zeilenumbruch Witwenwörter eine Rolle spielen, sind dies beim Spaltenumbruch Schusterjungen und Hurenkinder. Bei diesen gilt es zu vermeiden, daß ein Absatz mit einer einzelnen Zeile am Ende einer Spalte/Seite beginnt bzw. am Anfang einer neuen endet.

Aufgrund dieser Verwandtschaft braucht es für den Spaltenumbruch keine grundsätzlich neuen Algorithmen. Wie in einer Zeile kann ein Objekt, sei es ein Absatz, Grafik oder Tabelle, vier verschiedene Positionen in einer Spalte einnehmen: Zu Beginn oder Ende, umgeben von anderen Objekte oder allein in einer Spalte. Variationsmöglichkeiten zum Stauchen und Dehnen einer Spalte ergeben sich durch den Platz zwischen einzelnen Objekten oder innerhalb von ihnen.

Ganz ohne Modifikation kommt ein derartiger Algorithmus dennoch nicht aus. Der Seitenumbruch hat eine zusätzliche Herausforderung zu meistern: Fußnoten. Deren Platzierung erweist sich als kompliziert, weil es zu gewährleisten gilt, daß die entsprechende Referenz im Text auf der selben Seite steht wie die Fußnote. Eine lange Fußnote muß unter Umständen auch noch umgebrochen werden, wobei die gleichen typografischen Regeln

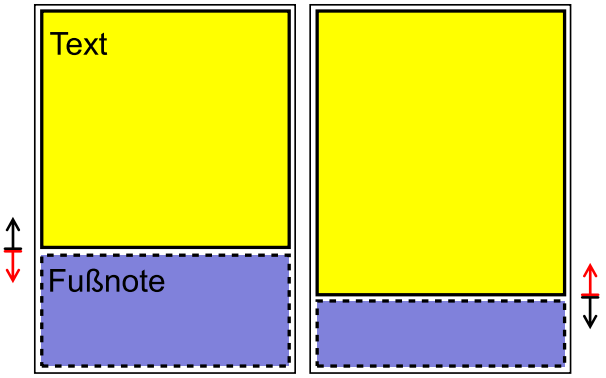


Abbildung 7.1: Abhängigkeit zwischen Text und Fußnote. Umso mehr Platz für den Text auf der ersten Seite benötigt wird, umso mehr Platz benötigt die umgebrochene Fußnote auf der folgenden, und umgekehrt.

gelten wie für normalen Text. Text und Fußnote beeinflußen sich dabei gegenseitig, wie Abbildung 7.1 zeigt.

TEX implementiert beim Seitenumbruch aus Speichergründen einen lokal optimierenden Algorithmus, es werden nur eine begrenzte Anzahl von Seiten herangezogen und der Umbruch nicht über ein komplettes Dokument optimiert. Der Seitenumbruchalgorithmus ist daher eine Art First-Fit Algorithmus.

7.2 Text an einem Pfad

Zum Platzieren von Text an einem Pfad macht die SVG Spezifikation klare Aussagen. Der Kern ist dabei, daß der Pfad zunächst zu einer horizontalen bzw. vertikalen Linie gestreckt wird. Auf dieser Linie wird der Text normal gesetzt und erst danach die endgültige Ausrichtung der einzelnen Glyphen durchgeführt. Typografische Feinheiten, wie Ligaturen, machen dabei nur noch bedingt Sinn. In lateinischen Schriften führen sie dazu, daß mehrere Zeichen dicht beieinander kleben, während die restlichen durch Rotation und Translation weit auseinandergezerrt werden.

Ein zusätzliches Problem stellen Schreibschriften dar, bei denen eine Verbindung der einzelnen Glyphen zwingend notwendig ist. Dazu sind komplexe Manipulationen an der Glyphform notwendig.

7.3 Trennung

Silbentrennung ist in vielen lateinischen Sprachen üblich. Im Kapitel *Technischer Hintergrund* wurde bereits die bahnbrechende Arbeit von Frank Liang in diesem Bereich erwähnt. In seiner Doktorarbeit beschäftigte er sich zwei Jahre mit der stochastischen Analyse von Wörtern und wie sie getrennt werden. Das Ergebnis ist ein System, das auf Trennmustern beruht. Sie werden für jede Sprache separat erstellt und finden nicht alle, dafür aber ausschließlich gültige Trennstellen. Die Trennmuster werden neben TEX bspw. auch in OpenOffice, KOffice und Scribus benutzt. FOP hat ebenfalls den notwendigen Quellcode um eine Silbentrennung nach Liang durchzuführen.

Das System hat sich mit kleineren Einschränkungen bewährt und benutzt stets den gleichen Algorithmus, nur die Trennmuster variieren von Sprache zu Sprache. Ein gemeinsames Modul ist daher sinnvoll. Implementierungen sind reichlich vorhanden, der Bedarf an weiteren ist gering. Für UTA macht es Sinn, auf bestehende Lösungen zurückzugreifen. Aussichtsreicher Kandidat ist die FOP Implementierung. Sollte es sich als lizenzrechtlich unproblematisch erweisen diese Klassen zu übernehmen und zu verändern, könnte es direkt als Trennmodul Verwendung finden.

Die Einschränkungen des Liang-Algorithmus beziehen sich darauf, daß die Trennung zwar korrekt, aber nicht immer sinnentsprechend sein muß. Unter dem Namen SiSiSi[1] (Sichere sinnentsprechende Silbentrennung) wurde an der TU-Wien ein Trennverfahren entwickelt, das diese Schwäche nicht hat. Als Beispiel nennt die SiSiSi *Wach-stube* und *Wachs-tube*, wo durch die Trennung vollkommen unterschiedliche Begriffe entstehen.

7.4 Font-Management

Die Handhabung verschiedener Schriftformate ist in sich ein aufwendiges Thema. Die Probleme stehen nicht nur im Zusammenhang mit den unterschiedlichen Schriftformaten, sondern auch mit den Ausgabemedien und der Rechnerkonfiguration beim Endanwender. Nicht alle Ausgabemedien unterstützen alle Schriftformate und nicht alle in einem Dokument geforderten Schriftarten sind notwendigerweise auf einem Rechner installiert – oder nur in einem Format, das die textverarbeitende Anwendung nicht lesen kann. Wird ein Dokument auf einen anderen Rechner übertragen, fehlen dort u. U. die auf dem Quellrechner installierten Schriftarten.

[1] URL http://www.ads.tuwien.ac.at/research/SiSiSi/

Diese Schwierigkeiten haben verschiedene Technologien nach sich gezogen. Viele Dokumentformate erlauben es inzwischen, Schriftarten in das Dokument selbst einzubetten um sicherzustellen, daß sie auf dem Zielrechner korrekt angezeigt werden können. Für den Fall, daß Schriftarten auch eingebettet nicht zur Verfügung stehen, wurde das Panose-System[2] entwickelt. Es definiert Eigenschaften, die eine Schriftart charakterisieren. Grundlegende Charaktereigenschaften sind die Metrikwerte der Glyphen oder bspw. ob die Schrift serifenbehaftet oder serifenlos ist. Die Charakterisierung geht soweit, daß auch die Form der Serifen berücksichtigt wird. Diese Kategorisierung erlaubt es, kompatible Schriftarten zu finden.

Die Problematik ist unabhängig von der Art der Anwendung oder der verwendeten Programmiersprache, folglich macht es Sinn eine allgemeine Lösung für das Font-Management zu suchen. Bereits erwähnt wurde FOray, bei dem ein derartiges System weit oben auf dem Wunschzettel steht. Hinzu kommt STSF[3] (Standard Type Services Framework), eine Entwicklung von Sun, welches ein ähnliches Ziel verfolgt. Nach dem Client-Server Modell aufgebaut, soll es typografische Dienste bereitstellen, darunter auch das Management von Schriftarten. UTA besitzt derzeit keine derartige Komponente. Hier lohnt es sich Entwicklungen abzuwarten. Entspringt FOray ein entsprechendes Modul, entfällt die Notwendigkeit seitens UTA ein solches selbst zu entwickeln.

7.5 Verbesserte Unterstützung von Formsatz

Die aktuelle Unterstützung für Formsatz kann noch weiter verbessert werden. Die Zeilenbreiten müssen derzeit im Voraus bekannt sein. Unberücksichtigt bleibt die Höhe der Zeile. Abbildung 7.2 zeigt einen problematischen Fall, in dem eine besonders hohe Box in der Zeile vorkommt.

Dadurch, daß die Zeilenhöhe nicht berücksichtigt wird, wird auch die vorgegebene Absatzform nicht mehr eingehalten. Die folgenden Zeilenlängen müssen korrigiert werden. Unter Umständen reicht diese Korrektur allein nicht aus. Dann nämlich, wenn die Box mit der größten Höhe nahe am Ende der Zeile steht. In diesem Fall muß auch die Länge dieser Zeile angepasst werden. Entweder rutscht die Box in die nächste Zeile, oder Wortzwischenraum kann so verkleinert werden, daß sie noch in die Zeile passt (vgl. Abbildung 7.3).

[2]URL http://www.w3.org/Fonts/Panose/pan2.html
[3]URL http://stsf.sourceforge.net

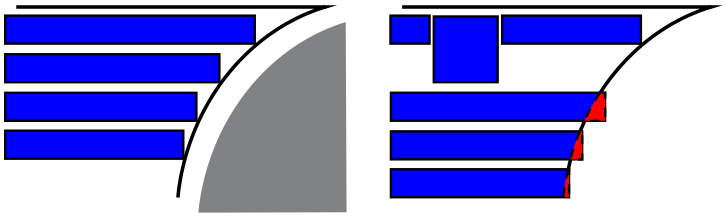


Abbildung 7.2: Probleme beim Formsatz mit einer übergroßen Box. Links ein korrektes Ergebnis, rechts die gleichen Zeilenlängen, aber mit einer hohen Box in der ersten Zeile. Wird die Zeilenhöhe nicht berücksichtigt, wird die vorgeschriebene Form nicht mehr eingehalten. Die schwarze Linie zeigt die gewünschte Form am rechten Rand des Absatzes. Die gestrichelten, roten Elemente markieren, wo die Zeile zu weit hinausragt.

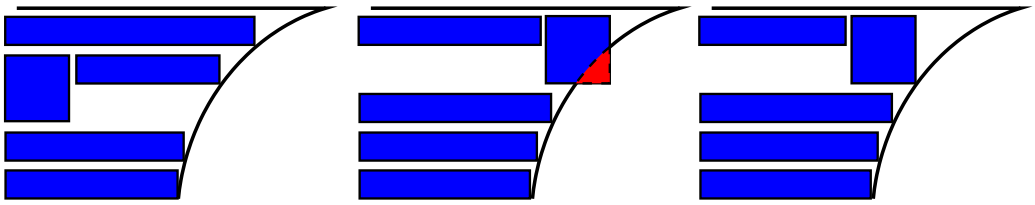


Abbildung 7.3: Zeilenlänge in Abhängigkeit von der Platzierung der übergroßen Box. Links ein unproblematischer Fall, bei dem die Boxhöhe berücksichtigt wird. In der Mitte wird die übergroße Box am Zeilenende platziert. Die Zeilenlänge ist dabei gleich der im linken Fall. Als Folge ragt die Box über die gewünschte Form hinaus. Im rechten Fall konnte der Wortzwischenraum so verkleinert werden, daß die Box in der Zeile Platz findet.

Um dieses Problem zu beseitigen muß dem LineWidthModel die Möglichkeit gegeben werden, die maximale Boxhöhe einer Zeile einzukalkulieren. Außerdem werden weitere Angaben wie der Zeilenabstand benötigt.

7.6 Umfließen von Objekten

Im vorangehenden Kapitel wurde bereits das Umfließen von Objekten angesprochen. Das Problem ist eine Layoutaufgabe und bleibt externen Algorithmen überlassen. Hier dennoch ein paar ergänzende Anmerkungen. Soll ein Objekt umflossen werden, werden immer zusätzliche „Zeilen“ benötigt. Sie werden erzeugt, in dem die ursprüngliche Zeile geteilt wird.

Abbildung 7.4 zeigt wie vier zusätzliche Zeilen entstanden sind. Der Umbruchalgorithmus generiert lediglich Zeilen unterschiedlicher Länge. Die Grafik zeigt, wie der gesetzte Text aussieht, wenn der Satzalgorithmus die einzelnen Zeilen einfach untereinander positioniert.

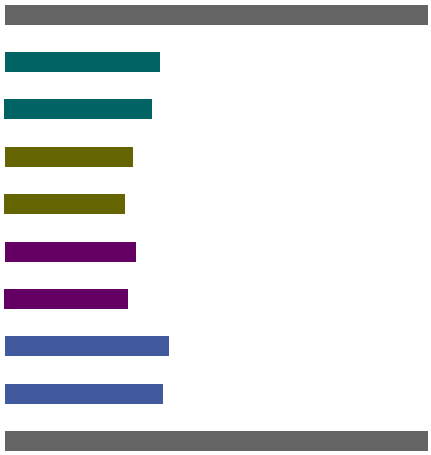


Abbildung 7.4: Die vom Umbruchalgorithmus erzeugten Zeilen untereinander angeordnet.

Das gewünschte Ergebnis wird erzielt, indem das Satzsystem die farbigen Zeilen korrekt zueinander ausrichtet. Die Differenz in x-Richtung ist gleich der Länge der ersten Teilzeile plus die Breite, die es auszulassen gilt (vgl. Abbildung 7.5).

Bei der Positionierung der zusammengehörenden Zeilen muß berücksichtigt werden, wie sich Grundlinien zueinander verhalten. Ist in der ersten Teilzeile eine lateinische Grundlinie ausschlaggebend, muß diese es auch in der zweiten Teilzeile sein, auch wenn sie ausschließlich aus Glyphen

Abbildung 7.5: Korrekt angeordnete Zeilen, die die gewünschte Form aus dem Absatz schneiden.

mit hängender Grundlinie besteht. XSL spricht in diesem Zusammenhang von der dominanten Grundlinie, die Aufschluß darüber gibt, wie Glyphen verschiedener Schriften zueinander platziert werden müssen.

Knuth geht in [6, Kap. 9] auf die gleiche Weise vor wie hier beschrieben. Wobei der Artikel Beispielcharakter hat und keine allgemeine, automatische Lösung darstellt, um beliebige Objekte umfließen zu können.

7.7 Weiterentwicklung des Satzmodells

Beim Satzmodell war bisher stets die Rede von Rechtecken, die anstatt der komplexen Glyphform gesetzt werden. Diese Sicht der Dinge könnte sich in Zukunft ändern. Gerd Neugebauer vom $\varepsilon\chi$TEX-Team möchte weg vom Rechteckmodell, hin zu einem Verfahren, bei dem die bisherige Vereinfachung wegfällt; ein Verfahren, bei dem die Glyphform in den Satzvorgang einbezogen wird.

Die Grundidee ist, ein Glyph nicht mehr mit einer einfachen, rechteckigen Fläche zu umgeben, sondern mit einer Hülle, die an den Umriß des Glyph angelehnt ist und damit beliebig komplex werden kann. Durch eine Art Kollisionserkennung könnten diese Hüllen aneinandergeschoben werden. Die Abbildung 7.6 zeigt den bisherigen und den neuen Ansatz. Links das heute übliche Bild von Glyphen. Beim Satz werden den Glyphen rechteckige Flächen zugeteilt. Unterschneidungsangaben erlauben es den Rechtecken sich zu überlappen. Rechts umgeben komplexeren Flächen die Glyphen. Sie können soweit zusammengeschoben werden, bis sich die Flächen berühren. Weißfläche wird entfernt, Unterschneidungen entstehen automatisch.

Ein grundlegendes Problem liegt darin, zu klären, wie diese Hülle zu erstellen ist. Eine Hülle, die in alle Richtungen gleich dick ist, wie rechts

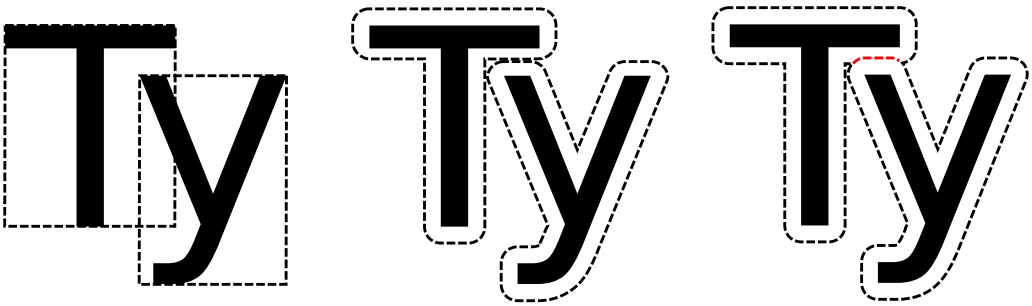


Abbildung 7.6: Die Abkehr vom Rechteck (links) zur komplexen Hülle (Mitte). Gleichmäßige Hüllen, d.h. gleicher Abstand um die Glyphen, führen zu Überlappungen (rechts).

in der Abbildung zu sehen, führt zu Überlappungen, die dem Prinzip des einfachen Zusammenschiebens widersprechen würden.

Gerd Neugebauer sieht hier vorerst die Schriftdesigner in der Pflicht, die nicht weiter Unterschneidungstabellen füllen würden, sondern für jedes Glyph eine entsprechende Hülle definieren müssten. Dieser Mehraufwand könnte sich als Hemmschuh erweisen. Daher gilt es die Frage zu klären, ob das Finden und Zusammenschieben der Hüllen nicht vollständig algorithmisch erledigt werden kann. So wäre es z. B. denkbar, daß die Hülle nicht als fest verstanden wird, sondern wie Schaumstoff, der sich in einem gewissen Rahmen quetschen läßt, was leichte Überlappungen wie in obigem Fall erlauben würde. Die Erstellung der Hülle würde dadurch vereinfacht, der „Schiebealgorithmus“ verkompliziert. Der andere Weg wäre, die Hülle aufwendiger zu berechnen. Da ein Schriftdesigner bei der Erstellung von Unterschneidungstabellen letztendlich die Form der Glyphen in Relation setzt, könnte dieser Vorgang durchaus automatisierbar sein.

Daran, daß der Ursprung des Glyph platziert werden muß, ändert sich freilich nichts. Das Hüllenmodell ändert daran etwas, wie die endgültige Position errechnet wird – und das kann große Auswirkungen auf Schriftformate und die Qualität des Schriftsatzes haben. So könnten Unterschneidungstabellen in Schriftarten nur noch eine geringe Rolle spielen und ein Algorithmus mit weniger externer Zusatzinformation, eben unter Verzicht auf die Unterschneidungstabellen, vergleichbare oder bessere Ergebnisse liefern. Aufeinanderfolgende Glyphen gleicher oder unterschiedlicher Schriftarten könnten einer Unterschneidung unterzogen werden, ohne daß eine entsprechende Tabelle vorhanden ist. Die Automatisierung hätte ohnehin zugenommen. In TEX noch notwendige manuelle Korrekturen, der

Feinschliff, den sich Knuth als Autor nicht entgehen lassen wollte, könnten dadurch verringert, vielleicht sogar ganz überflüssig werden.

Es gäbe auch noch ein anderes potentielles Anwendungsfeld. Die zuvor angesprochenen Probleme des Formsatzes und des Umfließens von Objekten sind eng miteinander verwandt. Bei beiden werden Zeilenlängen durch beliebig komplexe Hüllen vorgegeben. Ein Konzept, das dem hier angesprochenen gleicht. Denn in allen Fällen gilt es Formen derart zu platzieren, daß sie sich nicht schneiden – ein weiterer Vorteil der hier skizzierten Idee.

Da bei der Realisierung dieses Modells nicht mehr einmalgenerierte Metrikangaben die entscheidende Rolle spielen (vorausgesetzt es gelingt die Hüllen automatisch zu erzeugen), sondern die Kurvendefinitionen, die ein Glyph ausmachen, müsste auch ein nicht-grafisches System die Kontur eines Glyph berücksichtigen. UTA stellt in der Glyph-Klasse eine Methode bereit, die diese Kontur liefert. Da es UTA der jeweiligen Implementierung einer Schrift überläßt wie Glyphen platziert werden, ist die Realisierung seitens UTA möglich. Ist ein Weg gefunden, diese Idee konkret umzusetzen, kann UTA auch Helferklassen bereitstellen.

7.8 Mathematischer Schriftsatz

Die Verwendung von Embedding-Leveln hat bereits ein grundlegendes Konzept gezeigt. Die Möglichkeit Box-Objekte ineinander zu verschachteln. Gleiches wird in wesentlich intensiverer Form auch beim mathematischen Schriftsatz benötigt. Mit einem gewichtigen Unterschied. Während Embedding-Level bei Bedarf von einem Script generiert werden und sich daraus eine Struktur ergibt, die ursprünglich nicht vorhanden war, ist diese Struktur bei einer mathematischen Formel bereits vorgegeben. Die Information über diese Struktur muß irgendwie im eingehenden Datenstrom mitgeliefert werden. In TEX wird sie aus der Zeichenkette selbst extrahiert. Dazu ist ein Parsevorgang notwendig.

TEX bildet im mathematischen Modus zunächst eine spezielle horizontale Liste, eine *mlist*, die in einem weiteren Schritt durch einen komplexen Algorithmus in eine normale horizontale Liste umgewandelt wird. Beide Listen können dabei verschachtelt weitere Listen enthalten.

Ineinander geschachtelte Listen bilden eine Baumstruktur. Diese läßt sich mit Hilfe des Object Replacement Character und Attributen im UTA-Eingabeformat nachbilden. Des weiteren erfordert mathematischer Schriftsatz zusätzliche Funkionen. Als Beispiel sei hier die Matrix genannt.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & \frac{1}{10*\frac{1}{10*\frac{1}{10*\frac{1}{10}}}} & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Bei einfachem Text reicht es aus, eine Folge von Zeichen entgegenzunehmen. Bei mathematischem Satz kommen nun Layoutaufgaben hinzu. Obige Matrix gleicht einer Tabelle, die einzelnen Textbestandteile müssen in Relation zueinander positioniert werden. Dazu ist es notwendig die Elemente zu finden, die die größte horizontale und vertikale Ausdehnung haben, in diesem Fall der Bruch. Auch der Bruch selbst hat einige Besonderheiten. So muß bspw. die Schriftgröße angepasst, nämlich bis zu einem gewissen Grad verkleinert, werden, um so tiefer die Schachtelung ist. Hier muß ein hochqualitatives Mathematikmodul selbst über die Schriftgröße bestimmen, sollte dies nicht durch das Fremdsystem geschehen sein. Für die Baumstruktur ist daher ein rekursives Vorgehen nötig, bei dem zunächst alle untergeordneten Strukturen, die *Blätter*, gesetzt werden. Das Vorgehen ist ähnlich wie bei ineinandergeschachtelten Embedding-Ebenen. Dazu benutzt das Mathematikmodul einen eigenen Typesetter, dem gegenüber es als Fremdsystem fungiert.

8 Abschließende Bewertung

Die vorgestellte API ist möglichst abstrakt gehalten um Spielraum für die Realisierung unterschiedlicher Bedürfnisse zu lassen. Sie stellt die Schnittmenge der von allen Schriftsatzsystemen benötigten Funktionen dar, egal ob im Hintergrund ein aufwendiges XSL Area-Modell oder nur ein einfacher String in der Textkonsole steht. Mögliche Anwendungen für die erarbeiteten Ergebnisse gibt es genug. Die Allgegenwärtigkeit von Schrift und der aktuelle Stand der digitalen Typografie machen dies deutlich.

Wirtschaftliche Bestrebungen, deren Ziel es ist, besser internationalisierte Produkte zu ermöglichen, haben komplexe typografische und informationstechnische Probleme zu lösen – wie den Satz von bidirektionalem Text, der Handhabung von Ligaturen in arabischen Sprachen oder der wörterbuchbasierten Suche nach möglichen Umbruchstellen in Thai. Bestes Beispiel dafür ist sicherlich ICU. Mit überschaubarem Mehraufwand sollte es möglich sein, die in dieser Arbeit angestellten Überlegungen in ein solches Projekt zu integrieren, sowie weitere (mikro)typografische Feinheiten einzubauen und dabei auf eine einheitliche API zu achten. Während ICU und die Java 2D API auf interaktive Anwendungen mit hauptsächlich grafischen Benutzeroberflächen, wie Texteingabefelder und Editoren, zielen, können auch nicht interaktive Dokumentprozessoren wie FOP und Grafikframeworks wie Batik von einer einheitlichen API profitieren. Bisher herrschen bei beiden eigene, unvollständige Insellösungen vor, deren Funktionen sich teilweise überlappen, folglich redundant vorhanden sind. Zudem steht die Unterstützung von internationalem Text hinten an und wurde nicht von Anfang an in den Designprozess einbezogen. Hier bietet UTA mit den Sprachmodulen und den Embedding-Leveln ein tragfähiges Konzept.

Nicht zuletzt kann UTA ein Stück dazu beitragen zu realisieren, was der TEXnologie bis dato verwehrt blieb: Eine Weiterentwicklung zu einem voll funktionsfähigen, interaktiven, grafischen DTP-Programm. Programme wie LYX[1] erleichtern den Einstieg in TEX sicherlich erheblich, die

[1] URL http://www.lyx.org

Möglichkeiten zur freien Gestaltung wie bei Adobe InDesign oder Quark XPress bietet es hingegen nicht.

9 Glossar

Anchor Ein Anker ist ein Punkt, der bei der Platzierung von Objekten behilflich ist. Vergleiche Abschnitt 5.2.8 Anchor auf Seite 72.

Attachment Point Siehe Anchor.

ATSUI Apple Type Services for Unicode Imaging. Apples Bibliothek zur Darstellung von Unicode codiertem Text. Vergleiche Pango, Uniscribe.

Austreibung Bezeichnet den Vorgang der Anpassung der Wortzwischenräume bei Blocksatz.

Batik Java Bibliothek zur Erzeugung, Manipulation und Darstellung von SVG Daten. Batik bietet eine vollständige Implementierung der Java 2D API, wie sie von grafischen Bibliotheken wie Swing benutzt wird. Daher können alle Programme, die diese API nutzen ohne Modifikation in SVG gerendert werden.

Cross-Media-Publishing Cross-Media-Publishing verfolgt das Ziel ein Dokument in mehreren unterschiedlichen Medien publizieren zu können. Die beiden wichtigsten Medien sind dabei Druck und Internet. Elementarer Bestandteil ist dabei die Konvertierung des ursprünglichen Dokuments in verschiedene Dokumentformate. Für Druckmedien ist das wichtigste PDF, für das Internet HTML.

Cursive Linking Bei Schriftarten, die eine Schreibschrift imitieren, kann es notwendig sein, Kontrollpunkte so zu verschieben, daß Glyphen miteinander verbunden werden und damit direkt ineinander übergehen. Schriftarten bieten für diese Funktionen eigene Ankerpunkte. Den Vorgang bezeichnet man als Cursive Linking.

Diakritische Markierung/Zeichen Wikipedia definiert diakritische Zeichen als „[...] zu Buchstaben gehörige kleine Zeichen wie Punkte, Striche, Häkchen oder Kringel, die eine besondere Aussprache

oder Betonung markieren und unter oder über dem Buchstaben angebracht sind, in einigen Fällen auch durch den Buchstaben hindurch.“ [25]

DTP Unter dem Begriff Desktop-Publishing (DTP) wird die Erstellung von Drucksachen am PC verstanden. Obwohl auch Programme wie TEX in diesen Bereich fallen, wird der Begriff doch mehr mit der freien Gestaltung von Layouts verbunden.

Font-Management Bezeichnet in dieser Arbeit die Menge an Funktionen, die zum Einlesen verschiedener Schriftformate und dem Finden und Ersetzen von Schriftarten benötigt wird.

Graphite Das SIL-Projekt ermöglicht das Erstellen von Smart-Fonts. Graphite stellt eine eigene Sprache zur Formulierung von kontextabhängigen Verarbeitung bereit. Die Programme werden in eigenen Tabellen in TrueType Schriften eingebettet. Ein Graphitemodul kann diese Information auslesen und zur Textverarbeitung nutzen. Vergleiche Smart-Font, SIL International.

Justification Siehe Austreibung.

Kerning Siehe Unterschneidung.

Langzeichen Ursprüngliche, aufwendigere Form der chinesischen Schriftzeichen. Im Gegensatz zum vereinfachten Chinesisch (simplified Chinese).

LATEX Ein Makropaket für TEX. Layoutaufgaben wie mehrspaltiger Text werden dadurch gegenüber reinem TEX erheblich erleichtert.

Ligatur Verbindung von mehreren Schriftzeichen zu einem, um die Lesbarkeit zu erhöhen. Beispiele für im Deutschen gebräuchliche Ligaturen sind fi und ffi. Auch das „ß“ geht auf eine Ligatur der altdeutschen Buchstaben „langes s“ und „z“ zurück (ſ + ʒ = ß).

LYX Eine grafische Oberfläche zum Erstellen von LATEX-Dokumenten. LYX verfolgt das What You See Is What You Mean Konzept, bei dem nicht die optische Formatierung im Vordergrund steht, sondern das möglichst komfortable Erstellen des Dokuments.

Pango Schriftsatzsystem, das in Gnome zum Einsatz kommt. Hauptziele sind Unterstützung von internationalem Text und hochqualitativer Schriftsatz. Vergleiche ATSUI, Uniscribe.

Panose Wird in einem Dokument eine Schriftart verlangt, die auf dem System nicht vorhanden ist, bietet Panose eine Technologie, die eine optisch möglichst kompatible Schriftart aus den verfügbaren Schriftarten als Alternative anbietet.

Seitenbeschreibungssprache Beschreibt den gewünschten Aufbau und das Aussehen einer Seite, also die Platzierung von Text und Grafik. Die Beschreibung erfolgt dabei unabhängig vom Ausgabegerät.

SIL International Eine Organisation, die Forschungs-/Entwicklungsarbeit im Bereich internationaler Schriften leistet. Der Fokus liegt auf wirtschaftlich uninteressanten und daher vernachlässigten Sprachen und Schriftsystemen. SIL entwickelt in diesem Bereich u. a. Schriftarten und Software.

Smart-Font Die Bezeichnung für Schriftformate bzw. Schriftarten, die sich nicht auf eine einfache Sammlung von Glyphen beschränken, sondern zusätzliche Informationen bereithalten. TrueType, OpenType und METAFONT Schriftarten sind Smart-Fonts.

SVG Scaleable Vector Graphics. XML basiertes Vektorgrafikformat mit Animationsunterstützung, Funktionsumfang vergleichbar mit Macromedia Flash.

Swing Java-Bibliothek zur Erstellung grafischer Oberflächen. Bietet Unterstützung für bidirektionalen Text.

Transformation Eine Transformation im Sinne der Computergrafik bezeichnet eine grafische Operation auf ein zwei- oder dreidimensionales Objekt. Eine Transformation besteht dabei aus folgenden Komponenten: Rotation, Skalierung, Translation (Verschiebung) und Scherung.

Eine Transformation im Sinne von XSL beschreibt eine Umwandlung eines XML-Dokuments in ein Dokument anderer Struktur. Dies kann, muß aber nicht ein XML-Dokument sein.

Translation In der Computergrafik Teil einer Transformation, bezeichnet eine Verschiebung.

Uniscribe Schriftsatzsystem in Windows zur Handhabung von internationalem Text. Vergleiche ATSUI, Pango.

Unterschneidung Der deutsche Begriff für Kerning. Um zu große Weißflächen zu vermeiden, werden Glyphen näher zusammen geschoben. So wird in To das „o“ näher an das „T“ geschoben (To vs. To). Bei AW entsteht ohne Unterschneidung regelrecht ein weißer Balken (AW).

W3C Das World Wide Web Consortium ist eine der wichtigsten standardisierenden Instanzen in den Bereichen Internet und Hypertext. Das Konsortium ist für die Entwicklung von (X)HTML, XML, XSL, SVG, MathML, CSS, sowie vieler weiterer Technologien verantwortlich.

Kolophon

Diese Arbeit wurde mit ε-LaTeX gesetzt unter Verwendung von KOMA-Script für die Satzspiegelberechnung. Die Grafiken wurden mit Inkscape[1] erstellt. Die Schriftarten für ɑbcdɐɪA wurden wie in [6, Kap. 4] beschrieben erzeugt und mit mftrace in Type1-Schriftarten umgewandelt um die Darstellungsqualität auch im endgültigen PDF-Dokument gewährleisten zu können. Die Beispiele für andere Schriftsysteme in Kapitel 2 zeigen den jeweiligen Namen des Schriftsystems (Japanisch in Japanisch, etc.) und wurden durch Pango gesetzt.

[1] URL http://www.inkscape.org

Literaturverzeichnis

[1] Pratchett, Terry ; *Die volle Wahrheit :* Goldmann, 2000. – ISBN 3-442-45406-9

[2] Knuth, Donald E. ; *The TEXbook : Volume A of Computers & Typsetting :* Addison-Wesley, 1984.

[3] Knuth, Donald E. ; *TEX: The Program : Volume B of Computers & Typsetting :* Addison-Wesley, 1986.

[4] Wikipedia ; *Schrift.* – URL http://de.wikipedia.org/wiki/Schrift. – Version vom 6. Jul 2004, 14:39

[5] Wikipedia ; *Thai-Alphabet.* – URL http://de.wikipedia.org/wiki/Thai-Alphabet. – Version vom 14. Jun 2004, 08:08

[6] Knuth, Donald E. ; *Digital Typography :* Addison-Wesley, 1999. – ISBN 1-57586-010-4

[7] Davis, Mark ; *Unicode Standard Annex #9, The Bidirectional Algorithm :* Unicode Consortium, 2003. – URL http://www.unicode.org/reports/tr9/tr9-11.html

[8] Wikipedia ; *Chinesische Schrift.* – URL http://de.wikipedia.org/wiki/Chinesische_Schrift. – Version vom 11. Aug 2004, 17:54

[9] Wikipedia ; *Japanisches Schriftsystem.* – URL http://de.wikipedia.org/wiki/Japanisches_Schriftsystem. – Version vom 9. Aug 2004, 23:35

[10] Bauke, Heiko ; *Altdeutsche Schriften* – URL http://www-e.uni-magdeburg.de/bauke/latex/tipsutricks/oldgerman/ – Version vom 14. Sep 1999

[11] Dudenredaktion (Hrsg.) ; *Duden : Die Rechtschreibung :* Band 1. 19. Aufl. Mannheim : Dudenverlag, 1986.

[12] Adler, Sharon ; Berglund, Anders et al. ; *Extensible Stylesheet Language (XSL) Version 1.0 : W3C Recommendation 15 October 2001 :* W3 Consortium. – URL http://www.w3.org/TR/2001/REC-xsl-20011015

[13] Knuth, Donald E. ; *The METAFONTbook : Volume C of Computers & Typsetting :* Addison-Wesley, 1986.

[14] The Unicode Consortium ; *The Unicode Standard, Version 4.0.0, defined by: The Unicode Standard, Version 4.0.* – URL http://www.unicode.org/versions/Unicode4.0.0/

[15] Freytag, Asmus ; *Unicode Standard Annex #14, Line Breaking Properties :* Unicode Consortium, 2003. – URL http://www.unicode.org/unicode/reports/tr14/tr14-14

[16] Davis, Mark ; *Unicode Standard Annex #29, Text Boundaries :* Unicode Consortium, 2003. – URL http://www.unicode.org/reports/tr29/tr29-4.html

[17] Davis, Mark ; *Unicode Standard Annex #15, Unicode Normalization Forms :* Unicode Consortium, 2003. – URL http://www.unicode.org/reports/tr15/tr15-23.html

[18] Herman, Debra et al. ; *The TrueType Reference Manual :* Apple Computer Inc., 2002. – URL http://developer.apple.com/fonts/TTRefMan/

[19] Adobe ; *OpenType Specification v.1.4 :* Adobe, 2004. – URL http://partners.adobe.com/asn/tech/type/opentype/index.jsp

[20] Clark, James ; *XSL Transformations (XSLT) Version 1.0 : W3C Recommendation 16 November 1999 :* W3 Consortium. – URL http://www.w3.org/TR/1999/REC-xslt-19991116

[21] Clark, James ; DeRose, Steve ; *XML Path Language (XPath) Version 1.0 : W3C Recommendation 16 November 1999 :* W3 Consortium. – URL http://www.w3.org/TR/1999/REC-xpath-19991116

[22] Andersson, Ola et al. ; Ferraiolo, Jon (Hrsg.) ; Jackson, Dean (Hrsg.) ; *Scalable Vector Graphics (SVG) 1.1 Specification : W3C Recommendation 14 January 2003 :* W3 Consortium. – URL http://www.w3.org/TR/2003/REC-SVG11-20030114/

[23] Williams, George ; *FontForge : An Outline Font Editor*. – URL http://fontforge.sourceforge.net/overview.html

[24] Apple Inc. ; *Rendering Unicode Text With ATSUI :* Apple Inc., 2002. – URL http://developer.apple.com/documentation/Carbon/Conceptual/ATSUI_Concepts/atsui.pdf

[25] Wikipedia ; *Diakritisches Zeichen*. – URL http://de.wikipedia.org/wiki/Diakritisches_Zeichen. – Version vom 1. Jul 2004, 13:32

Index