Kapitel 1

Das System Collage-Two

Der praktische Teil dieser Arbeit besteht in der Entwicklung des Systems Collage-Two, das die Eingabe und Ableitung von Collagen-Grammatiken sowie die Darstellung der erzeugten Bilder erlaubt. Dieses Kapitel ist als Einführung in das System gedacht. Da Collage-Two in seinen Konfigurationsmöglichkeiten und Parametern mittlerweile recht komplex ist, wurden alle Informationen, die zur konkreten Anwendung notwendig sind, in einem separaten Referenzteil zusammengefaßt, der im nächsten Kapitel zu finden ist. Das übernächste Kapitel schließlich gibt Einblicke in die Implementierung des Systems.

1.1 Einleitung

> Es ist nicht genug zu wissen, man muß auch anwenden; es ist nicht genug, zu wollen, man muß auch tun.
>
> — *Joh. Wolfgang von Goethe*

Der Name des Systems legt die Annahme nahe, daß ein Vorgängersystem existiert, das wahrscheinlich Collage-One heißt ([?]). Collage-One ist der direkte Vorläufer zu Collage-Two und wurde von mir im Rahmen meiner Tätigkeit als studentische Hilfskraft im DFG-Projekt *Collagen-Grammatiken* entwickelt. Collage-One wiederum basiert in seinen Grundstrukturen auf einem von Stefan Taubenberger entwickelten Prototyp, von dem es unter anderem den Großteil der PostScript-Ausgabe geerbt hat. Collage-Two ist das bisherige Endprodukt dieser Entwicklungsserie. Es enthält alle Fähigkeiten seiner Vorgänger, ist abwärtskompatibel zu deren Eingabesyntax und erweitert sie in wichtigen Punkten, z.B. durch Bereitstellung von Farben, parametrisierbaren Eingabedateien, kontextsensitiven Collagen-Grammatiken und der Möglichkeit, Animationen zu erzeugen.

Mögliche Anwendungen von Collage-Two liegen hauptsächlich in den Bereichen Forschung und Lehre. Collage-Two ermöglicht die experimentelle Forschung auf dem Gebiet der Collagen-Grammatiken und bereitet durch seine PostScript-Ausgabe die Ergebnisse automatisch in einer publikationsfähigen Form auf. Viele neuere Artikel über Collagen-Grammatiken enthalten Abbildungen, die durch Collage-One generiert wurden. Alle Abbildungen von Collagen und Collagen-Grammatiken in dieser Arbeit wurden durch Collage-Two generiert. Durch die Möglichkeiten zur Herstellung von Animationen können zukünftig vielleicht auch neue Anwendungsfelder für das System erschlossen werden, die es für eine größere Anwendergruppe interessant machen.

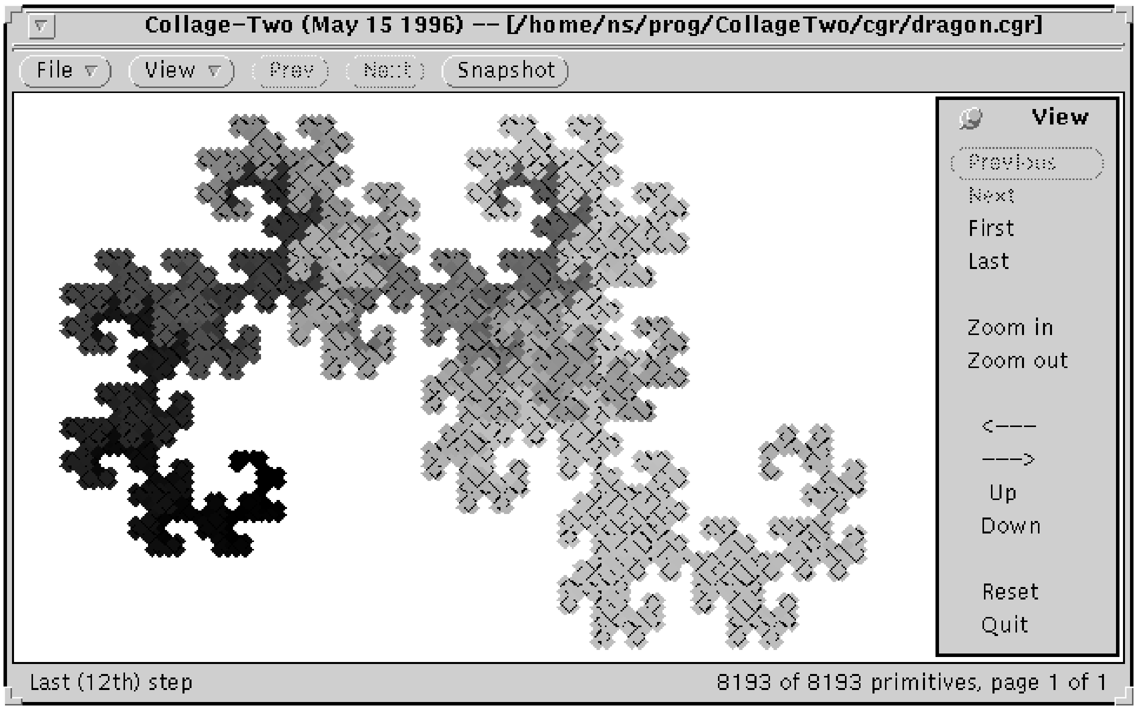


Figure 1: Anzeige einer abgeleiteten Collage (*Drachenkurve*) in Collage-Two

1.2 Funktioneller Überblick

Collage-Two ist ein interaktiv bedienbares System, das unter dem X-Windows Fenstersystem läuft ([?, ?]). Es wurde in C++ geschrieben ([?, ?]) und verwendet das von Sun Microsystems kostenlos erhältliche XView-Toolkit zur Oberflächensteuerung ([?, ?]). Da es ansonsten keine besonderen Ansprüche an die Zielplattform stellt, ist es sehr leicht portierbar. Bislang wurde Collage-Two auf Sun-Plattformen sowie auf PCs unter dem Betriebssystem Linux getestet.

1.2.1 Eingabedatei

Die Eingabe besteht aus einer Textdatei, die gemäß einer definierten Syntax eine Collagen-Grammatik inklusive ihrer Parts und Hyperkanten sowie eine Ableitung in dieser Grammatik beschreibt. Sie ist analog zum Quelltext in einer Programmiersprache zu sehen und kann entweder mit dem in Collage-Two eingebauten oder mit jedem anderen Editor bearbeitet werden.

1.2.2 Bilderzeugung

Collage-Two liest die Eingabedatei ein, analysiert sie auf syntaktische Korrektheit und gibt ggf. entsprechende Fehlermeldungen aus. Ist die Eingabe in Ordnung, so wird die definierte Ableitung ausgewertet. Ausgehend von der Start-Collage werden in jedem Ableitungsschritt gemäß der definierten Regeln Hyperkanten durch rechte Regelseiten ersetzt und ein zugehöriges Bild produziert. Die Ableitung endet, wenn die definierte Ableitungslänge erreicht wurde oder wenn keine Hyperkanten mehr in der Collage existieren. Das letzte Bild einer Sequenz repräsentiert

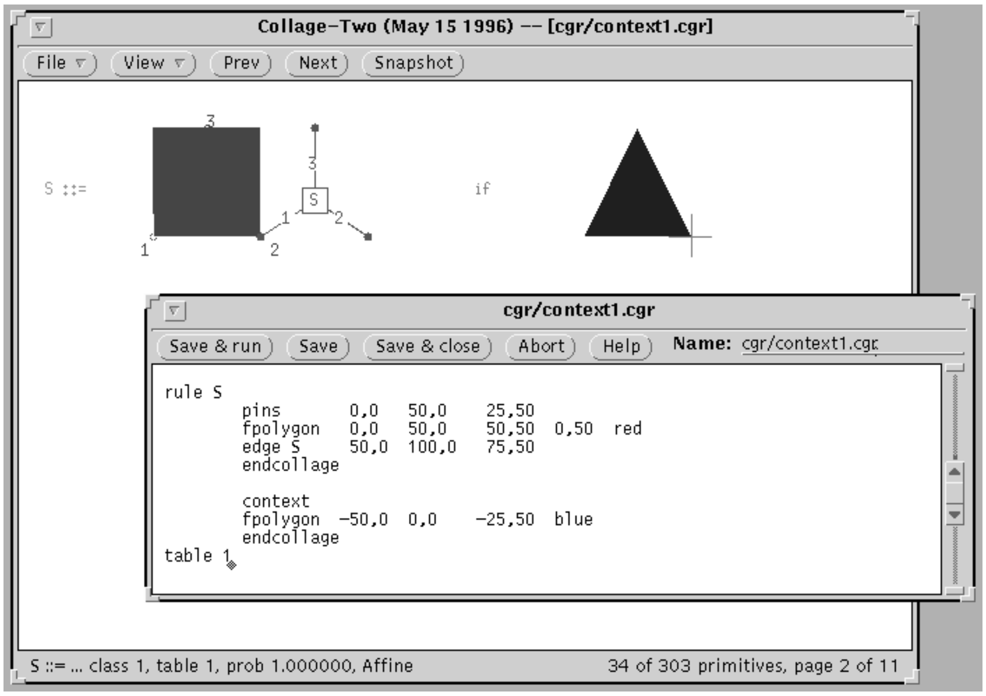


Abbildung 2: Graphische und textuelle Form einer Produktion in Collage-Two

oftmals eine terminale Collage (Abbildung 1).

Neben den zu jedem Ableitungsschritt erzeugten werden zusätzliche Bilder generiert, die die textuell definierte Collagen-Grammatik graphisch darstellen (Abbildung 2). Das System besitzt zwei unabhängige Ausgabekomponenten (*Backends*), eine für die interaktive Bildschirmansicht und eine für PostScript. Weitere Backends können bei Bedarf dem System hinzugefügt werden. Auf diese Weise wurde eine Einbindung des Vorgängers Collage-One in das System BIZARR-Studio ([?]) realisiert.

Das System zeigt nach Durchlaufen der Ableitung alle erzeugten Bilder seitenweise organisiert in einem Fenster an. Zwischen den Seiten kann vor- und zurückgeblättert und es kann in die Teile eines Bildes hineingezoomt werden. Mit einer Schnappschußfunktion kann der aktuelle Bildinhalt in eine externe Grafikdatei gesichert werden.

1.2.3 Performance

Mit jeder Änderung der Eingabedatei wird der gesamte Prozeß erneut durchlaufen. Die Turn-Around-Zeiten von der Änderung der Grammatik bis zur erneuten Ausgabe hängen stark von der Art der Grammatik und der vorgegebenen Ableitungslänge ab. Collagen-Grammatiken, die exponentiell viele Teile (Hyperkanten und Parts) erzeugen, zwingen jedes System recht schnell in die Knie, während bei linearen Grammatiken auch mehrere hundert Ableitungsschritte ohne spürbare Verzögerung berechnet werden können. Collage-Two unterstützt auch *Annotated Collage Grammars*, die die Berechnung beschleunigen und vor allem Speicherplatz einsparen können.

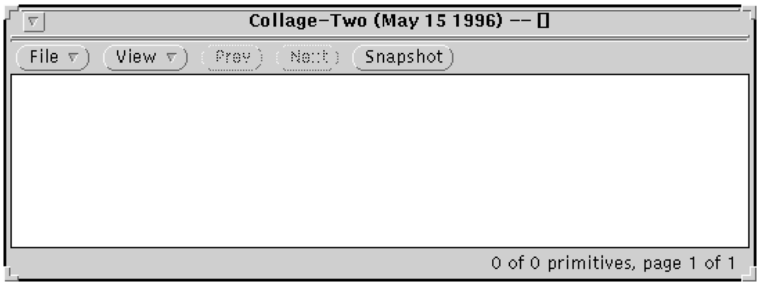


Abbildung 3: Collage-Two direkt nach dem Programmstart

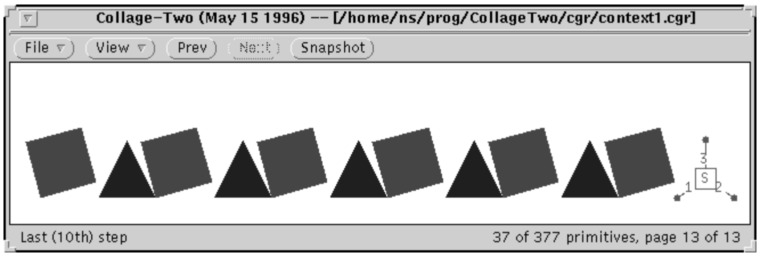


Abbildung 4: Letztes erzeugtes Bild für `context1.cgr`

1.3 Ein Rundgang

Nach dem Start des Systems zeigt sich zunächst das Hauptfenster mit leerem Anzeigebereich (Abbildung 3). Nach Auswahl des Menüpunktes `Load...` aus dem Klappmenü `File` kann man eine bereits erstellte Grammatikdatei (Dateiendung `.cgr`) einladen. Für dieses Beispiel dient die Datei `context1.cgr`. Collage-Two berechnet direkt nach dem Laden der Datei die gesamte Ableitung und zeigt das letzte erzeugte Bild an (Abbildung 4).

Durch Auswahl des Menüpunktes `Edit` aus dem Menü `File` wird der eingebaute Editor aufgerufen, der sofort die aktuelle Grammatikdatei zur Bearbeitung einlädt. Abbildung 5 zeigt den Editor und einen Teil des Inhaltes der Grammatikdatei. Der vollständige Inhalt dieser Datei wird im folgenden diskutiert.

1.3.1 Aufbau einer Grammatikdatei am Beispiel `context1.cgr`

```
%
% Collage-Two
% Context sensitive collage grammars
% EXAMPLE 1: Forcing the use of alternating productions for the same hyperedge.
% Nils Schwabe 2/96
%
#include "trans.csi"

% ----- Definitions -----
#define RED    rgb (200,0,0)
#define BLUE   rgb (0,0,200)
#define SQUARE fpolygon {ROTATE2D(rad(15))} (0,0) (50,0) (50,50) (0,50) {POP} RED
```

```
#define TRIANGLE    fpolygon (0,0) (50,0) (25,50)  BLUE
#define EDGETRI     (50,0) (100,0) (75,50)
#define PINTRI      (0,0)  (50,0)  (25,50)
#define DERIVATIONS 10

% ----- Grammar -----
grammar

start
    pins
        SQUARE
        edge S EDGETRI
    endcollage

rule S
    pins PINTRI
        SQUARE
        edge S EDGETRI
    endcollage

    context
        {TRANSLATE2D(-50,0)}
        TRIANGLE
        {POP}
    endcollage
table 1

rule S
    pins PINTRI
        TRIANGLE
        edge S EDGETRI
    endcollage

    context
        {TRANSLATE2D(-50,0)}
        SQUARE
        {POP}
    endcollage
table 1

% ----- Derivation -----
derivation
    until DERIVATIONS do 1

tableinfo 1
    exhaustprob 1
```

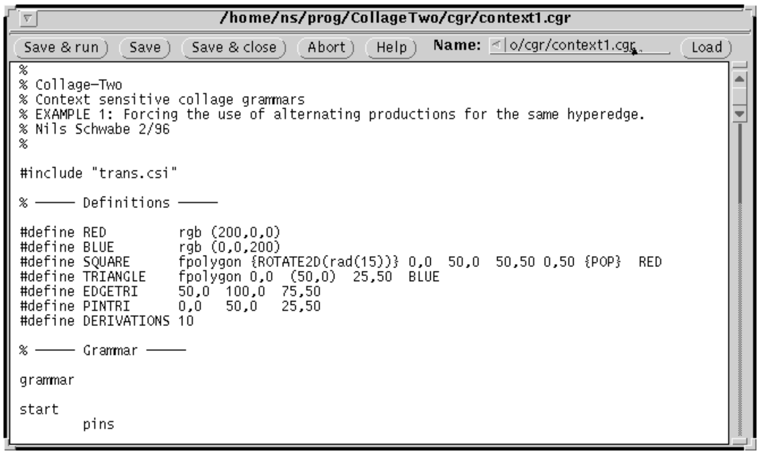


Abbildung 5: Collage-Two-Editor mit geladener Datei `context1.cgr`

Der Inhalt einer Grammatikdatei ist in drei Hauptteile gegliedert, die hier durch die Kommentare "Definitions", "Grammar" und "Derivation" hervorgehoben sind.

Definitionsteil

Im ersten Teil werden Optionen für das System eingestellt (im Beispiel nicht verwendet) und Definitionen für Symbole und Makros mit Hilfe von Präprozessor-Anweisungen (`#define`) vorgenommen. Der Präprozessor ersetzt rekursiv alle definierten Symbole durch die rechte Seite einer `#define`-Anweisung, bevor das Resultat an den Collage-Two-Parser übergeben wird. Es ist ebenfalls möglich, mittels einer `#include`-Anweisung den Inhalt anderer Dateien, z.B. Makro-Bibliotheken und partielle Grammatiken mit komplexen Part-Definitionen, an einer Stelle einzufügen. Über diesen einfachen Mechanismus wird das Schreiben modular aufgebauter Grammatiken ermöglicht und somit die Wiederverwendung von existierenden Bausteinen in mehreren Anwendungen unterstützt. Weiter unten wird noch erläutert, wie mit Collage-Two die geometrische Angleichung bestehender Teil-Grammatiken in eine neue Anwendung bewerkstelligt werden kann.

Grammatik- und Ableitungsteil

Im zweiten Teil (hinter `grammar`) wird die eigentliche Collagen-Grammatik definiert, bestehend aus Start-Collage, beliebig vielen Regeln bzw. Produktionen sowie der Ableitungsdefinition (`derivation` und `tableinfo`).

Die beiden Regeln der Grammatik haben beide `S` als Markierung und sind zueinander symmetrisch. Die rechte Seite der ersten Regel enthält ein rotes Quadrat als Part (`SQUARE`) sowie eine `S`-Hyperkante (`edge S EDGETRI`). Bei Betrachtung der numerischen Definitionen für die Pinpunkte (`PINTRI`) und die Attach-Punkte (`EDGETRI`) stellt man fest, daß die Hyperkante um 50 Einheiten nach rechts verschoben ist. Die erste Regel enthält auch eine Kontext-Collage (`context`), die aus einem um 50 Einheiten nach links verschobenen blauen Dreieck besteht. Die zweite Regel ist analog zur ersten, nur daß Quadrat und Dreieck vertauscht sind. Der Effekt dieser Definitionen besteht darin, daß die Ersetzung einer `S`-Kante durch die eine Regel nur dann möglich ist, wenn im Schritt zuvor eine Ersetzung mittels der anderen Regel erfolgte, die den passenden Kontext produziert hat. Der Anfang dieser Kette wird durch die Start-Collage realisiert, die identisch mit der rechten Seite der ersten Regel ist.

Wie aus diesem Beispiel ersichtlich, kann man in Collage-Two durch Verwendung von Symbolen und Makros Grammatiken recht komfortabel und strukturiert entwickeln. Neben den hier gezeigten Möglichkeiten erlaubt der Collage-Two-Parser auch, jede numerische Angabe in Form eines Ausdruckes zu schreiben, wobei neben Grundrechenarten auch viele trigonometrische Funktionen eingebaut sind.

Stack für Punkt-Transformationen

Der Collage-Two-Parser verwaltet einen Stack von Transformationen, die (in Stack-Reihenfolge) auf jeden Punkt angewendet werden, der in der Grammatikdatei spezifiziert wird. Dadurch ist es einerseits bequem möglich, transformierte Teile zu konstruieren, wie im Beispiel das um 15 Grad gedrehte Quadrad `SQUARE`. Andererseits ist es erst durch den Transformations-Stack möglich, beliebige Teil-Grammatiken mittels `#include` in andere (Teil-)Grammatiken einzubetten. Zum Beispiel kann eine Teil-Grammatik durch folgende drei Zeilen eingebunden und um Faktor 2 proportional vergrößert werden:

```
{SCALE2D(2,2)}
#include "some_rules.cgr"
{POP}
```

1.4 Animationen

Collage-Two erlaubt die Erzeugung von Animationen auf der Grundlage parametrisierter Eingabedateien. Jeder numerische Ausdruck in der Eingabedatei kann Variablen enthalten, deren Werte außerhalb der Datei festgelegt werden. Mit dem Collage-Two-Commander können Programme der Skript-Sprache Tcl (Tool Command Language, [?]) erstellt werden, die auf spezielle Befehle zum Steuern von Collage-Two zugreifen können. Damit ist es möglich, die in einer Eingabedatei verwendeten Variablen durch ein Programm berechnen zu lassen. Um eine fließende Animation herzustellen, wird das Programm typischerweise kleine Änderungen an den Variablenwerten vornehmen und für jede Belegung der Variablen eine Ableitung in der Collagen-Grammatik anstoßen. Das Abspeichern erzeugter Bilder in eine Grafikdatei kann ebenfalls vom Programm aus geschehen. Mittels eines externen Tools können die Einzelbilder in eine Filmdatei im MPEG-Format konvertiert werden.

Animationen auf der Grundlage von Collagen-Grammatiken wurden bereits im System BIZARR-Studio ([?]) realisiert. Beim dort verfolgten Ansatz bestanden die Einzelbilder einer Animation aus den Schritten einer einzelnen Ableitung. Da viele Collagen-Grammatiken exponentiell viele Teile bezogen auf die Ableitungslänge erzeugen, können solche Animationen nur von kurzer Dauer sein. Außerdem entstehen durch die direkte Aneinanderreihung der Bilder einzelner Ableitungsschritte sehr sprunghafte Übergänge. Diesen Problemen wird durch das Berechnen von Zwischenbildern begegnet. Jede in einem Ableitungsschritt ersetzte Hyperkante wird von einer kleinen Anfangsgröße langsam auf die Originalgröße skaliert. Für jede Zwischengröße erfolgt der

Ersetzungsprozeß[1] und die Erzeugung eines Bildes. Auf diese Weise entsteht der Eindruck, daß die in einem Ableitungsschritt neu eingefügten Teile langsam heranwachsen. Der offensichtliche Nachteil ist, daß es sich um ein sehr willkürliches Verfahren handelt und viele andere denkbare Möglichkeiten zur Herstellung von Animationen nicht berücksichtigt werden.

Durch den in Collage-Two verfolgten Ansatz wird in sehr viel geringerem Maße festgelegt, worin eine auf Collagen-Grammatiken basierende Animation eigentlich besteht. Da man über die Möglicheit verfügt, alle numerischen Parameter der Eingabedatei durch ein Programm berechnen zu lassen, sind der Phantasie kaum Grenzen gesetzt. Die im BIZARR-Studio implementierte Animationstechnik ist ebenfalls realisierbar, indem einfach die Ableitungslänge parametrisiert wird. Die Zwischenstufen sind durch Berechnung der Attach-Punkte der Hyperkanten erzeugbar. Das Grundgerüst einer Animation muß bei Collage-Two nicht mehr von einer einzigen Ableitung gebildet werden. Stattdessen kann die Ableitungslänge auch konstant gehalten und dafür beispielsweise die Position oder Orientierung von Hyperkaten und Parts schrittweise verändert werden.

1.5 Zukünftige Entwicklungen

1.5.1 Anzahl der Dimensionen

Der Kern von Collage-Two arbeitet mit n-dimensionalen Koordinaten. In der Praxis ist diese Allgemeinheit durch die beiden Backends begrenzt, namentlich die Bildschirm- und PostScript-Ausgabe. Da die Bildschirmanzeige von 3-dimensionalen Objekten nur die ersten beiden Dimensionen anzeigt, ist für räumliche Anwendungen derzeit die Verwendung der in das System BIZARR-Studio ([?]) eingebundenen Version von Collage-One vorzuziehen. Da PostScript nicht direkt 3-dimensionale Objekte unterstützt, müßten diese erst von einem 3D-Rendering-Programm in möglichst hoher Auflösung berechnet werden und würden dann nur als Rasterbild zur Verfügung stehen. Die Verwendung des Grafikstandards OpenGL ([?]) erscheint für diesen Zweck vielversprechender, da es durch Auswertung der *Callbacks* der Rendering-Engine im Prinzip möglich sein sollte, eine direkte und vor allem auflösungsunabhängige 3D-PostScript-Ausgabe zu generieren.

1.5.2 Graphischer Editor

Derzeit existiert kein speziell auf die Möglichkeiten von Collage-Two abgestimmter graphischer Editor. Aufgrund der Möglichkeiten, die die textuelle Eingabe bietet, müßte ein solcher Editor mit gekoppelten Sichten arbeiten, so daß die textuelle Sicht stets erhalten und änderbar bleibt. Von Jutta Bildstein und Lutz Zegartowski wurde im Rahmen ihrer Diplomarbeit ein graphischer Editor für dreidimensionale Collagen-Grammatiken entwickelt, der einen Export in das von Collage-Two verwendete Syntaxformat erlaubt, so daß man bei Anwendungen, die dies erlauben, zumindest die Hyperkanten sowie einfache Parts und Regeln graphisch entwerfen kann. Sofern noch eine Weiterentwicklung von Collage-Two über diese Arbeit hinaus stattfinden sollte, wäre

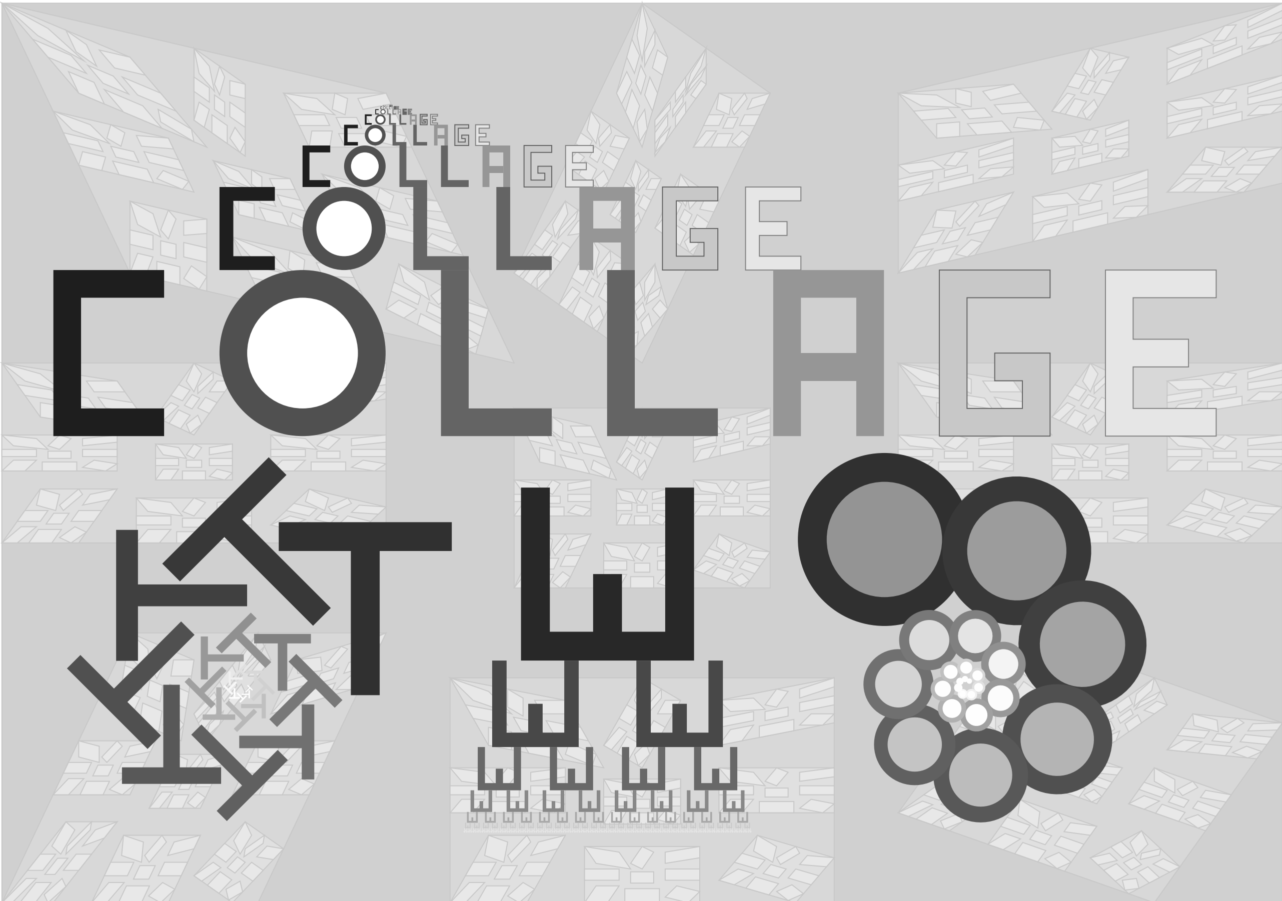
[1] Da affine Abbildungen verwendet werden, ist die Möglichkeit der Ersetzung einer Hyperkante unabhängig von deren Skalierung.

der nächste Schritt die Vereinheitlichung der Ausgabe sowie die Realisierung eines interaktiven 3D-Editors mit gekoppelter Textsicht unter Verwendung des OpenGL-Grafikstandards.

Kapitel 2

Collage-Two Referenz

In diesem Kapitel werden alle Funktionen von Collage-Two in Form eines Referenzhandbuches aufgeführt. Die Darstellung beginnt mit der Beschreibung der interaktiven Bedienung des Systems. Daran schließt sich die vollständige Angabe der Syntax und Semantik der Eingabedateien an. Den Abschluß bilden ein Abschnitt über den Collage-Two-Commander, der unter anderem die Erstellung von Animationen ermöglicht, sowie eine Aufstellung der PostScript-Attribute des Systems.



2.1 Bedienung von Collage-Two

Im folgenden wird die interaktive Bedienung von Collage-Two beschrieben und alle vorhandenen Funktionen dargestellt. Es wird vorausgesetzt, daß das System installiert und durch Eingabe von `cs` gestartet wurde. Ebenfalls vorausgesetzt werden grundlegende Kenntnisse im Umgang mit graphischen Benutzungsoberflächen im allgemeinen sowie mit *OpenWindows* im besonderen.

Maustasten

Collage-Two wird mit einer Drei-Tasten-Maus bedient. Gemäß der *OpenWindows*-Spezifikationen werden die drei Tasten mit `SELECT`, `ADJUST` und `MENU` bezeichnet. Die Abbildung dieser logischen auf physikalische Maus- oder andere Tasten kann vom Anwender geändert werden. Hierzu sei auf die Benutzungshandbücher zum *OpenWindows*-System verwiesen. Standardmäßig gelten folgende Zuordnungen: `SELECT` = linke Maustaste, `ADJUST` = mittlere Maustaste und `MENU` = rechte Maustaste.

Menüzeile des Hauptfensters

Abbildung 1: Menüzeile des Hauptfensters

Abbildung 1 zeigt die Menüzeile des Hauptfensters von Collage-Two. Werden die Knöpfe *File* und *View* mit der `MENU`-Taste angeklickt, so werden die in Abbildung 2 gezeigten Menüs sichtbar. Die Knöpfe *Prev*, *Next* und *Snapshot* lösen hingegen direkt eine Aktion aus, wenn sie mit der `SELECT`-Taste angeklickt werden:

- *Prev* bzw. *Next* zeigen die vorherige bzw. nächste Seite im Hauptfenster an. Diese Funktionen sind auch noch einmal im *View*-Menu vorhanden.
- *Snapshot* sichert den aktuell angezeigten Bildinhalt in eine Grafikdatei `ColSnap_`*n*`.ppm`. Diese Datei hat das Grafikformat *Portable Bitmap*. *n* ist ein Zähler, der mit jedem erneuten Aufruf dieser Funktion erhöht wird.

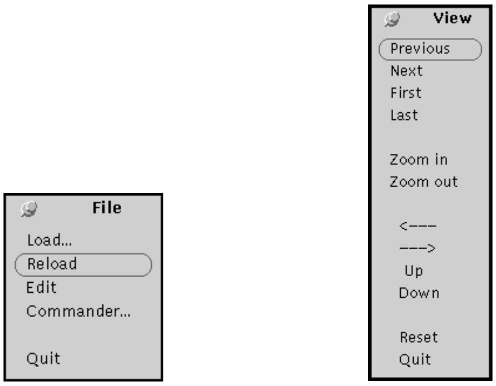


Abbildung 2: Ausgeklappte Menüs *File* und *View*

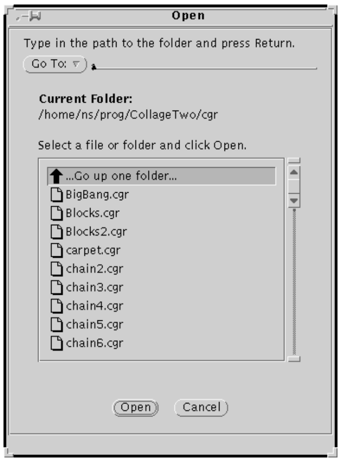


Abbildung 3: Dateiauswahl-Dialogbox

File-Menü

- `Load...`: Es wird eine Dateiauswahl-Dialogbox dargestellt (Abbildung 3), in der eine bereits erstellte Eingabedatei (Dateiendung `.cgr`) ausgewählt und durch Anklicken von *Open* geladen werden kann. Die Eingabedatei wird daraufhin geöffnet und die darin definierte Ableitung durchlaufen. Das letzte erzeugte Bild wird im Hauptfenster angezeigt.
- `Reload`: Die aktuelle Eingabedatei wird erneut geladen und die Ableitung wird neu berechnet. Diese Funktion ist insbesondere bei Grammatiken mit Wahrscheinlichkeiten nützlich, um schnell hintereinander einige der möglichen Resultate zu betrachten.
- `Edit`: Der Editor wird aufgerufen. Falls eine aktuelle Eingabedatei vorhanden ist, so wird diese automatisch in den Editor geladen.
- `Commander...`: Eine neue Instanz eines Collage-Two-Commanders wird erzeugt. Diese Funktion wird ausführlich in 2.10 beschrieben.
- `Quit`: Das Programm wird beendet.

View-Menü

- `Previous` / `Next`: Die vorhergehende bzw. nächste Seite wird angezeigt.
- `First` / `Last`: Die erste bzw. letzte Seite wird angezeigt.
- `Zoom in` / `Zoom out`: Es wird um ca. 10% in das Bild hinein- bzw. herausgezoomt.
- `<---` / `--->` / `Up` / `Down`: Der aktuelle Bildausschnitt wird nach links / rechts / oben / unten verschoben.
- `Reset`: Die vorgenommenen Änderungen des Bildausschnittes werden rückgängig gemacht und es wird der Ursprungszustand wiederhergestellt.
- `Quit`: Das Programm wird beendet.

Mausfunktionen im Hauptfenster

- *Zoom in:* Durch Drücken der `SELECT`-Taste und gleichzeitiges Bewegen der Maus kann ein rechteckiger Bereich aufgezogen werden (*Rubberbanding*), in den beim Loslassen der `SELECT`-Taste gezoomt wird. Der Bereich ist normalerweise proportional zum aktuellen Bildauschnitt, so daß sich durch das Zoomen keine Verzerrungen ergeben. Wird während des Aufziehens des Bereiches die `Control`-Taste gedrückt, so sind auch nicht-proportionale Zoom-Rechtecke definierbar. Durch Drücken der `MENU`-Taste kann die Zoomfunktion abgebrochen werden.
- *Zoom out:* Durch kurzes Drücken der `MENU`-Taste wird um ca. 10% aus dem aktuellen Bildausschnitt herausgezoomt.
- *Panning:* Durch Drücken der `ADJUST`-Taste und gleichzeitiges Bewegen der Maus kann der aktuelle Bildausschnitt unmittelbar verschoben werden. Von der Benutzung dieser Funktion sollte man absehen, wenn sehr viele Objekte dargestellt werden oder wenn der verwendete Computer sehr langsam ist, da das System dann unter Umständen nachläuft und während dieser Zeit nicht auf weitere Eingaben reagiert.

Menüzeile des Editors

Abbildung 4: Menüzeile des Editors

- `Save & run`: Die Eingabedatei wird abgespeichert und die Übersetzung und Ableitung gestartet.
- `Save`: Die Eingabedatei wird abgespeichert, ohne daß die Ableitung neu berechnet wird.
- `Save & close`: Die Eingabedatei wird abgespeichert, die Übersetzung und Ableitung gestartet und der Editor geschlossen.
- `Abort`: Der Editor wird geschlossen, ohne daß die Eingabedatei abgespeichert wird.
- `Name`: Dieses Eingabefeld enthält den Pfadnamen der Eingabedatei. Durch Ändern dieses Eintrages kann die aktuelle Eingabedatei unter einem anderen Namen gesichert werden.
- `Load`: Die Datei mit dem unter `Name:` angegebenen Pfadnamen wird in den Editor geladen.

Weitere Editor-Funktionen

Der Collage-Two-Editor ist eine eingebundene Version des *OpenWindows*-Editors `textedit`. Durch Drücken der `MENU`-Taste im Textbereich des Editors erscheint ein Popup-Menü, das alle Editor-Funktionen bereithält. Zur detaillierten Beschreibung dieser Funktionen sei wiederum auf die *OpenWindows*-Dokumentation verwiesen.

Graphische Anzeige von Produktionen

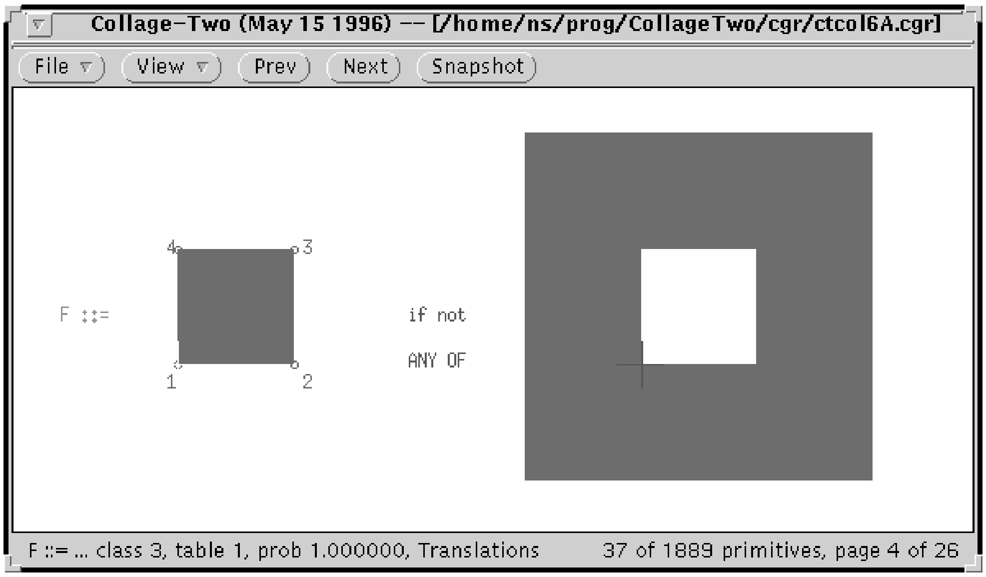


Abbildung 5: Anzeige einer kontextsensitiven Produktion im Hauptfenster

Collage-Two zeigt die textuell definierten Produktionen der Grammatik im Hauptfenster graphisch an. Abbildung 5 zeigt eine kontextsensitive Produktion für Hyperkanten mit Label `F`. Die Produktion hat in diesem Beispiel einen negativen Kontext und verwendet die *Any Of*-Option (siehe 2.4.8), die einer Zusammenfassung von mehreren Produktionen entspricht.

Am unteren Fensterrand werden links weitere Informationen zur Produktion dargestellt. Die Zahl hinter `class` gibt die Äquivalenzklasse der Produktion an. Collage-Two normalisiert intern die Grammatik und ordnet dabei jede Produktion einer Äquivalenzklasse zu. Es ist möglich, daß zwei Produktionen für ein gleiches Label verschiedenen Äquivalenzklassen zugeordnet werden, wenn sich die Pin-Punkte der rechten Regelseiten nicht mittels der zugelassenen Transformationen aufeinander abbilden lassen. Hinter `table` wird die Nummer der Tabelle angegeben und hinter `prob` die Wahrscheinlichkeit der Produktion. Dahinter folgt die Angabe der Transformationsklasse, die für diese Produktion verwendet werden darf. Standardwert hierfür ist `Affine`; im Beispiel sind zur Anwendung dieser Produktion nur Translationen zugelassen.

Unten rechts wird die Anzahl der verwendeten Graphik-Primitiva für die aktuelle Seite sowie deren Gesamtanzahl angezeigt. Dahinter folgt noch die aktuelle Seitennummer und die Anzahl der Seiten.

2.2 Syntax der Eingabedatei

Das Folgende ist eine Darstellung der vollständigen Syntax für Collage-Two-Eingabedateien in Backus-Naur-Form (BNF). In normaler Schrift geschriebene Worte sind Nichtterminale, alle **fett** geschriebenen Worte und Zeichen sind Terminalzeichen. Die verwendeten Metazeichen sind '(' und ')' zum Gruppieren von Termen und ' | ' zur Darstellung von Alternativen. Runde Klammern werden auch als Terminalzeichen verwendet. Da sie leicht mit den Metasymbolen verwechselt werden können, werden sie als '(' bzw. ')' notiert. Mit empty wird eine leere Eingabe gekennzeichnet.

Grammarfile	::=	Options TbGlobal Grammarpart Derivationpart
Options	::=	Options Option | empty
Option	::=	OptionF | OptionA | OptionCCP | OptionDPC | OptionDEC
		| OptionD | OptionR | OptionLO | OptionSource
		| OptionEval | OptionCC | OptionMP | OptionBBox
OptionF	::=	**/f** Filenamepattern
OptionA	::=	**/Annotated**
OptionCCP	::=	**/ColorChangeParts**
OptionDPC	::=	**/DefaultPartColor** Num
OptionDEC	::=	**/DefaultEdgeColor** Num
OptionD	::=	**/Dimensions** Num
OptionR	::=	**/RandomInit** Num
OptionLO	::=	**/LastOnly**
OptionSource	::=	**/Source** String
OptionEval	::=	**/Eval** String
OptionCC	::=	**/ColorContext**
OptionMP	::=	**/MergeParts**
OptionBBox	::=	(**BBox** | **XBBox** | **YBBox**) TheOptionBBox
TheOptionBBox	::=	SelectedPics (**Min** | **Max** | **MinMax** | **MaxMin** | **Avg**) OperandPics
SelectedPics	::=	SelectedPics PictListItem | empty
OperandPics	::=	OperandPics XPictListItem | empty
PictListItem	::=	String XNumberList
XPictListItem	::=	PictListItem
		| '(' Num **,** Num **,** Num **,** Num ')'
XNumberList	::=	XNumberList **,** XNumListItem | XNumListItem
XNumListItem	::=	XNumber | XNumber **..** XNumber
XNumber	::=	Num | **L**
TbGlobal	::=	Textblock

Grammarpart	::=	PtOperations **Grammar** OptTransforms TbGrammar StartCollage Rules
TbGrammar	::=	Textblock
StartCollage	::=	**Start** TbStart Collage
TbStart	::=	Textblock
Rules	::=	Rules Rule | empty
Rule	::=	**Rule** Label RuleOptions TbRule Collage Context NContext Tablenumber
RuleOptions	::=	Transformations OptRuleProb
		| RuleProb OptTransforms
		| empty
OptRuleProb	::=	RuleProb | empty
RuleProb	::=	**Prob** Num
TbRule	::=	Textblock
Tablenumber	::=	**Table** Num | **Global**
Context	::=	**Context** OptAnyOf Collage | empty
NContext	::=	**Negative Context** OptAnyOf Collage | empty
OptAnyOf	::=	**Any Of** | empty
Collage	::=	PtOperations CollageContent
CollageContent	::=	Pins CollageContent PtOperations
		| OneEdge CollageContent PtOperations
		| OnePart CollageContent PtOperations
		| PtOperations **EndCollage**
Pins	::=	**Pins** Points
OneEdge	::=	**Edge** Label Points OptEdgeColor
Label	::=	String
OptEdgeColor	::=	**Color** Num | empty
OnePart	::=	Part OptPartColor OptColorProc
OptPartColor	::=	**Color** Num | **RGB** ‘(’ Num **,** Num **,** Num ‘)’ | empty
OptColorProc	::=	**ColorProc** String | empty
Part	::=	Dot | Line | DashLine | Polyline | Polygon | Circle | Ellipse
Dot	::=	**Dot** Point
Line	::=	**Line** Point Point
DashLine	::=	**DashLine** Point Point
Polyline	::=	**Polyline** Point Point Points
Polygon	::=	(**Polygon** | **FilledPolygon**) Point Point Points
Circle	::=	(**Circle** | **FilledCircle**) Point Point
Ellipse	::=	(**Ellipse** | **FilledEllipse**) Point Point Point Point Num **,** Num

Points ::= Point Points | PtOperations
Point ::= PtOperations TransPoint
TransPoint ::= '(' Num , Num ')' | '(' Num , Num , Num ')'

OptTransforms ::= Transformations | empty
Transformations ::= **Transformations** Transformation
Transformation ::= **Affine** | **Translations** | **ScaleTranslations**

PtOperations ::= { SomePtOperations } | empty
SomePtOperations ::= PtOperation SomePtOperations | empty
PtOperation ::= **PtPush** NewPointTrans | **PtPop**
NewPointTrans ::= **Map Affine** IPoints **To** IPoints
IPoints ::= InternalPoint IPoints | empty
InternalPoint ::= '(' Num , Num ')' | '(' Num , Num , Num ')'

DerivationPart ::= **Derivation** TbDerivation TbDerivStart DerivEntries TableInfos
TbDerivation ::= Textblock
TbDerivStart ::= Textblock
DerivEntries ::= DerivEntries DerivEntry
DerivEntry ::= **Until** Num **Do** Num TbTableEntry
| **Until** Num **Repeat** TableElemEntries TbTableEntry
TableElemEntries ::= TabElemEntry | TableElemEntries , TabElemEntry
TabElemEntry ::= Num **Times** Num
TbTableEntry ::= Textblock
TableInfos ::= TableInfo TableInfos | empty
TableInfo ::= **TableInfo** Num TableInfoEntries
TableInfoEntries ::= TableInfoEntry TableInfoEntries | empty
TableInfoEntry ::= **Sequential** Num | **ExhaustProb** Num

Num ::= Product | Product + Product | Product - Product
Product ::= Term | Term * Term | Term / Term
Term ::= AtomicNum | - Term | '(' Num ')' | $ String
| **sqrt** '(' Num ')'
| **exp** '(' Num ')' | **log** '(' Num ')' | **log10** '(' Num ')'
| **sin** '(' Num ')' | **cos** '(' Num ')' | **tan** '(' Num ')'
| **asin** '(' Num ')' | **acos** '(' Num ')' | **atan** '(' Num ')'
| **sinh** '(' Num ')' | **cosh** '(' Num ')' | **tanh** '(' Num ')'
AtomicNum :: Digits [. Digits] [**e** (+ | -) Digits]
Digits ::= Digit Digits | Digit
Digit ::= (**0** | ... | **9**)

<table>
<tr><td>Textblock</td><td>::=</td><td>BeginAttributes Strings EndAttributes | empty</td></tr>
<tr><td>Filenamepattern</td><td>::=</td><td>Characters % Characters .ps</td></tr>
<tr><td>Strings</td><td>::=</td><td>String Strings | empty</td></tr>
<tr><td>String</td><td>::=</td><td>Character | “ Characters ”</td></tr>
<tr><td>Characters</td><td>::=</td><td>Character Characters | empty</td></tr>
<tr><td>Character</td><td>::=</td><td>(A | ... | Z | a | ... | z | _ | | Digit)</td></tr>
</table>

2.3 Optionen

2.3.1 *OptionF* – PostScript-Ausgabedateien

Mit dieser Option werden die Namen der PostScript-Ausgabedateien festgelegt. Fehlt die Angabe der Option, so erfolgt keine PostScript-Ausgabe. Die hinter dem Optionszeichen **/f** ohne Anführungszeichen anzugebende Zeichenkette sollte an einer beliebigen Stelle das Prozentzeichen % enthalten. Dieses dient als Platzhalter für den variablen Teil des Namens. Es werden folgende Dateien erzeugt:

G001	Start-Collage im Grammatikteil
Gn	Rechte Regelseite für Produktion $n-1$
Gn**_C**	Kontext-Collage für Produktion $n-1$, falls vorhanden
Gn**_N**	Negative Kontext-Collage für Produktion $n-1$, falls vorhanden
D001	Start-Collage im Ableitungsteil
Dn	Ableitungsschritte

Beispiel: /f MyFiles%.ps

Hinweis 1: Durch *Textblöcke* kann der Inhalt aller Dateien durch eigenen PostScript-Code ergänzt werden (siehe 2.8).

Hinweis 2: Fehlt das Prozentzeichen in der Zeichenkette, so wird lediglich die letzte Ausgabedatei ohne variablen Namensteil erzeugt.

2.3.2 *OptionA* – Annotated Collage Grammars

Durch Angabe dieser Option schaltet das System auf die Verwendung von *Annotated Collage Grammars* um. Das bedeutet, daß nach einer automatischen Normalisierung der Grammatik im voraus berechnet wird, welche Transformationen an welchen Hyperkanten benötigt werden. Hyperkanten im eigentlichen Sinne werden dann nicht mehr verwendet. Für Anwender des Systems ergibt sich daraus einerseits, daß keine Hyperkanten mehr sichtbar sind und zum anderen eine höhere Ablaufgeschwindigkeit und verminderter Speicherplatzbedarf.

Siehe auch: *OptionLO*.

2.3.3 *OptionCCP* – Farbänderungen von Parts

Diese Option ist für das grundsätzliche Aktivieren von Farbänderungen in Parts zuständig.

Siehe auch: *OptionSource*.

2.3.4 *OptionDPC* – Standardmäßige Part-Farbe

Diese Option legt die standardmäßige Farbe von Parts in Form eines Farbindexes fest, dessen Wertebereich vom Grafiksystem abhängt. Typische Werte liegen im Bereich von 0 bis 255.

Anmerkung: Diese Option ist obsolet. Es sollte in neuen Grammatiken lieber mit RGB-Farben gearbeitet werden.

2.3.5 *OptionDEC* – Standardmäßige Hyperkanten-Farbe

Diese Option legt die standardmäßige Farbe von Hyperkanten in Form eines Farbindexes fest, dessen Wertebereich vom Grafiksystem abhängt. Typische Werte liegen im Bereich von 0 bis 255.

2.3.6 *OptionD* – Anzahl der Dimensionen

Legt die Anzahl der Dimensionen des euklidischen Raums fest, mit dem Collage-Two arbeitet. Derzeit erlaubt sind die Werte 2 und 3, wobei 2 der Standardwert ist. Wird in drei Dimensionen gearbeitet, so werden zweidimensionale Punkte auf drei Dimensionen erweitert, indem eine 0 als dritte Koordinate verwendet wird.

2.3.7 *OptionR* – Startwert des Zufallszahlen-Generators

Hiermit wird der Startwert des Zufallszahlengenerators festgelegt. Dies ist für Grammatiken mit nicht-deterministischen Regeln und Wahrscheinlichkeitsangaben von Bedeutung. Ein gleicher Startwert garantiert reproduzierbare Ergebisse. Fehlt die Angabe der Option, so ermittelt Collage-Two einen "zufälligen" Startwert, der aus der Systemzeit abgeleitet wird.

Tip: Bei jedem Durchlauf wird der Startwert auf die Standard-Ausgabe geschrieben (z.B. das Terminalfenster, von dem aus Collage-Two gestartet wurde). Bei Bedarf kann dieser Wert einfach per Cut&Paste in den Collage-Two-Editor kopiert werden.

2.3.8 *OptionLO* – Nur das letzte Bild erzeugen

Durch Angabe dieser Option wird nur das letzte Bild erzeugt. Dies kann sinnvoll sein, um Arbeitsspeicher (Bildschirmausgabe) und/oder Plattenplatz (PostScript-Ausgabe) zu sparen.

2.3.9 *OptionSource* – Tcl-Script ausführen

Der dieser Option folgende String wird als Dateiname eines Tcl-Scripts aufgefaßt, das unmittelbar zur Ausführung gebracht wird. Derzeit liegt die Hauptanwendung darin, im Tcl-Script *Farbprozeduren* zu definieren, die später bei Part-Definitionen angegeben werden können.

Beispiel: /source "MyColorProcs.tcl"

Hinweis: Für Informationen zu Tcl (Tool Command Language) siehe [?].

2.3.10 *OptionEval* – Tcl-Code ausführen

Der dieser Option folgende String wird als Tcl-Code aufgefaßt, der unmittelbar zur Ausführung gebracht wird. In Zusammenspiel mit *OptionSource* liegt die Hauptanwendung darin, Parameter der Farbprozeduren zu definieren oder Defaultwerte zu überschreiben.

Beispiel: /eval "set redIncr 0.24; set blueIncr 1.2"

Hinweis: Für Informationen zu Tcl (Tool Command Language) siehe [?].

2.3.11 *OptionCC* – Farbkontext aktivieren

Wird diese Option angegeben, so wird bei kontextsensitiven Grammatiken die Farbe der Parts als zusätzliches Unterscheidungskriterium herangezogen. Normalerweise spielt die Farbe der Parts keine Rolle.

2.3.12 *OptionMP* – Parts zusammenfassen

Durch Angabe dieser Option eliminiert Collage-Two automatisch alle identischen Parts. Bei kontextsensitiven Grammatiken kann dies jedoch unerwünschte Effekte mit sich bringen. Der hier durchgeführte Identitätstest wird durch *OptionCC* beeinflußt.

Siehe auch: *OptionCC*

2.3.13 *OptionBBox* – PostScript Bounding-Boxes

Mit dieser komplexen Option können die für die PostScript-Ausgabe (Encapsulated PostScript) wichtigen Begrenzungs-Rechtecke (Bounding-Boxes) auf vielfältige Weise beeinflußt werden.

Collage-Two berechnet normalerweise die Bounding-Boxes aller erzeugten Ausgabebilder automatisch. Jedes Bild erhält dabei die kleinstmögliche Begrenzung, die gerade noch alle erzeugten Objekte umfaßt. Durch dieses Verfahren erhalten aufeinanderfolgende Bilder einer Ableitung in der Regel unterschiedliche Bounding-Boxes. Möchte man nun eine Ableitungssequenz in Bildern gleicher Größe maßstabsgetreu wiedergeben, so müssen die Bounding-Boxes dieser Bilder auf einen einheitlichen Wert gebracht werden. Mit *OptionBBox* können derartige einheitliche Bounding-Boxes in Abhängigkeit von den normalerweise verwendeten Bounding-Boxes berechnet werden.

Auswahl der Bilder

SelectedPics definiert die Teilmenge der Ausgabebilder, deren Bounding-Boxes von der Berechnung der Option betroffen sind. *OperandPics* definiert die Teilmenge der Ausgabebilder zuzüglich einer Menge konstanter Bounding-Boxes, die als Eingabe für die Berechnung dienen. Wird für *SelectedPics* bzw. *OperandPics* nichts angegeben, so ist dies identisch mit der Angabe aller erzeugten Bilder. Bei Verwendung der **XBBox**- bzw. **YBBox**-Variante ist jeweils nur die X- bzw. Y-Koordinate von der Berechnung betroffen.

Bildnamen und Intervalle

Bilder werden durch ihren Gruppenbuchstaben (**G** für den Grammatikteil und **D** für den Ableitungsteil) gefolgt von der laufenden Nummer (1 bis n) benannt, z.B. **G1** für die Start-Collage im Grammatikteil. Dies entspricht dem variablen Teil (% in der *OptionF*-Option) des Dateinamens der Postscriptdateien. Bildintervallangaben bestehen aus zwei Bildnamen, die durch **..** getrennt sind, z.B. **D1..D5**. Das letzte Bild kann durch die spezielle Nummer **L** benannt werden, z.B. **DL** für das letzte Bild der Ableitung. *SelectedPics* und *OperandPics* können jeweils beliebig viele solcher Angaben enthalten, wobei innerhalb einer Gruppe die Angaben durch Kommata getrennt werden, während verschiedene Gruppen durch das wechselnde Gruppensymbol (**G** oder **D**) erkannt werden. In *OperandPics* können zusätzlich noch konstante Bounding-Boxes in der Form (llx, lly, urx, ury) angegeben werden. Dabei bedeuten llx = lower left x, lly = lower left y, urx = upper right x sowie ury = upper right y. Es wird von einem kartesischen Koordinatensystem ausgegangen mit dem Ursprung bei $(0, 0)$, sowie positiven Koordinaten im ersten Quadranten.

Operationen

Seien $x_1, x_2, \ldots, x_k$ die Operanden der Option.

Min berechnet die Schnittmenge der angegebenen Bounding-Boxes.

Max berechnet die Hülle der angegebenen Bounding-Boxes.

MinMax: wie **Min**, nur wird die letzte Bounding-Box nicht mit einbezogen, so daß das Ergebnis Max (Min($x_1..x_k - 1$), x_k) ist. Dies ist nützlich, um eine Minimum-Berechnung nicht unter eine vorgegebene Schranke fallen zu lassen, d.h. um eine Mindestgröße festzulegen. Aus diesem Grund wird oftmals eine konstante Bounding-Box als letzter Operand angegeben.

MaxMin: wie **Max**, nur wird die letzte Bounding-Box nicht mit einbezogen, so daß das Ergebnis Min (Max($x_1..x_k - 1$), x_k) ist. Analog zu **MinMax** kann man hiermit eine Maximalgröße festlegen.

Avg berechnet den Mittelwert der angegebenen Bounding-Boxes.

Beispiel 1

/BBox D1..3, 5 G2 Min D2 G4..L (0,0,200,200)

Die Bounding-Boxes der Bilder 1, 2, 3 und 5 des Ableitungsteils sowie die des Bildes 2 des Grammatikteils werden aus einer Minimum- Berechnung gewonnen, die das zweite Bild der Ableitung, alle Grammatik-Bilder ab dem vierten sowie eine konstante Bounding-Box $(0, 0, 200, 200)$ als Eingabe verwendet.

Beispiel 2

/BBox Max

Alle Ausgabebilder erhalten die gleiche, maximale Boundingbox, so daß sichergestellt ist, daß alle Bilder gleich skaliert sind und jede Bildinformation dargestellt wird.

Beispiel 3

/XBBox MaxMin (0,0,1000,0)
Wie Beispiel 2, es wird aber nur die X-Ausdehnung der Bounding-Boxes auf diese Weise berechnet. Die Y-Ausdehnung kann dagegen bei jedem Bild verschieden sein. Außerdem wird durch die konstante Bounding-Box eine Maximalausdehnung in X-Richtung festgelegt (die Angabe der dritten 0 ist also ohne Wirkung; ein beliebiger Wert muß aber aus syntaktischen Gründen angegeben werden).

Beispiel 4

/XBBox MaxMin (0,0,1000,0)
/YBBox D1..L Avg
Wie Beispiel 3, nur wird nun für die Y-Ausdehnung der Ableitungsbilder der Mittelwert aller Bounding-Boxes herangezogen, in dem eine weitere Bounding-Box-Option verwendet wird.

2.4 Grammatikteil

2.4.1 *Grammarpart* – Einleitung des Grammatikteils

Nach dem Optionsteil wird mit dem Schlüsselwort **Grammar** der Grammatikteil eingeleitet. Dieser besteht im wesentlichen aus vier aufeinanderfolgenden und teilweise optionalen Teilen, die im folgenden beschrieben werden.

Hinweis: *PtOperations*, die vor der Angabe von **Grammar** und an vielen anderen Stellen erfolgen können, sind unter 2.6 beschrieben. Textblöcke (*Tb**) werden zusammenhängend unter 2.8 beschrieben.

2.4.2 *OptTransforms* – Klasse erlaubter Transformationen

Durch diesen optionalen Teil kann die globale Voreinstellung der Klasse der erlaubten Transformationen geändert werden. Hinter **Transformations** wird **Affine** (Affine Transformationen), **Translations** (nur Translationen) oder **ScaleTranslations** (nur Skalierungen und Translationen) erwartet. Die Voreinstellung ist **Affine**.

Diese globale Voreinstellung kann in jeder einzelnen Regel überschrieben werden (siehe 2.4.4).

2.4.3 *StartCollage* – Start-Collage der Grammatik

Hinter **Start** wird, nach dem optionalen Textblock *TbStart*, die Angabe einer Collage (siehe 2.4.10) erwartet, die als Start-Collage der Grammatik verwendet wird.

2.4.4 *Rules* – Produktionen der Grammatik

Nach der Start-Collage können beliebig viele Produktionen der Grammatik definiert werden. In Collage-Two werden Produktionen auch als "Regeln" bezeichnet. Jede Regel wird mit dem Schlüsselwort **Rule** eingeleitet. Durch die nachfolgende *Label*-Angabe wird die linke Seite der Regel definiert. *Label* kann ein einfaches Zeichen oder eine in Hochkommata eingeschlossene Zeichenkette sein.

2.4.5 *RuleOptions* – Regelspezifische Optionen

Jeder Regel kann eine eigene Klasse erlaubter Transformationen (siehe 2.4.2) sowie eine Wahrscheinlichkeit im Bereich 0 bis 1 zugeordnet werden. Die voreingestellte Klasse erlaubter Transformationen ergibt sich aus der *OptTransforms*-Option bzw. ist **Affine**, falls diese nicht angegeben wurde. Die voreingestellte Wahrscheinlichkeit ist 1.

2.4.6 *Context* – Kontext-Collage einer Regel

Hinter **Context** wird die Angabe einer Collage (siehe 2.4.10) erwartet, die als (positive) Kontext-Collage der Regel verwendet wird. Eine Hyperkante kann dann nur noch durch Anwendung dieser Regel ersetzt werden, falls die transformierte Kontext-Collage in der zu ersetzenden Collage vorhanden ist.

Hinweis 1: Die Angabe von Pin-Punkten bei Kontext-Collagen ist obsolet.

Hinweis 2: Die Anwendungsbedingung der Regel wird bei gleichzeitigem Vorhandensein einer negativen Kontext-Collage (s.u.) mit dieser UND-verknüpft. Werden mehrere bis auf Kontexte identische Regeln verwendet, dann kann jede denkbare Kombination von Vorhandensein bzw. Nicht-Vorhandensein von Parts und Hyperkanten im Kontext dargestellt werden. Hierzu ist in der *TableInfo* die Angabe **ExhaustProb 1** zu setzen (siehe 2.7). Die Bedingung für die Ersetzung einer Hyperkante durch eine Menge von Produktionen für diese Kante ist dann ein Prädikat folgender Form:

$$PC(rule_1) \wedge NC(rule_1) \vee PC(rule_2) \wedge NC(rule_2) \vee \cdots \vee PC(rule_n) \wedge NC(rule_n).$$

PC steht dabei für die Erfüllung der positiven und NC für die Erfüllung der negativen Kontextbedingung.

2.4.7 *NContext* – Negative Kontext-Collage einer Regel

Hinter **Negative Context** wird die Angabe einer Collage (siehe 2.4.10) erwartet, die als negative Kontext-Collage der Regel verwendet wird. Eine Hyperkante kann dann nur noch durch Anwendung dieser Regel ersetzt werden, falls die transformierte Kontext-Collage *nicht* in der zu ersetzenden Collage vorhanden ist.

Hinweise: Siehe *Context*.

2.4.8 *OptAnyOf* – Zusammenfassung von Kontext-Bedingungen

Bei der Defininition positiver oder negativer Kontext-Collagen kann die Bedeutung der einzelnen Parts und Hyperkanten durch Angabe von **Any Of** wie folgt geändert werden: Die jeweilige Kontext-Bedingung gilt als erfüllt, falls ein beliebiges Part oder eine beliebige Hyperkante der Kontext-Collage die Bedingung erfüllt. Durch diese Option werden generell keine neuen Möglichkeiten erschlossen; es handelt sich nur um eine Alternative zur Definition mehrerer Regeln (siehe Hinweis 2 bei *Context*).

2.4.9 *Tablenumber* – Zuordnung einer Regel zu einer Tabelle

Collage-Two arbeitet mit Tabellen von Regeln (siehe **??**), so daß jede Regel einer Tabelle zugeordnet werden muß. Diese Zuordnung ist gleichzeitig die Endemarkierung einer Regel und wird durch die Angabe von **Table** gefolgt von einer positiven ganzen Zahl vorgenommen. Alternativ dazu kann durch Angabe von **Global** die Regel allen Tabellen gleichzeitig zugeordnet werden.

Tip: Um eine Regel mehreren, jedoch nicht allen Tabellen zuzuordnen, muß die Definition für jede Tabelle getrennt erfolgen. Damit kein redundanter Code entsteht, definiert man die Regel mittels **#define** als Makro, das die Tabellennummer als Parameter erhält:

```
#define RULE_A(tn) Rule A  <right-hand-side, context, etc.> Table tn
...
RULE_A(1) RULE_A(2) RULE_A(3)
```

2.4.10 *Collage* – Definition einer Collage

Die Definition einer Collage beginnt je nach Zusammenhang (Start-Collage, Regeln, Kontext-Collagen) mit verschiedenen Schlüsselworten und endet stets mit **EndCollage**. Dazwischen sind beliebig viele Definitionen von Pin-Punkten (*Pins*), Parts (*OnePart*) und Hyperkanten (*OneEdge*) möglich. Vor und hinter jeder Definition können *PtOperations* (siehe 2.6) vorgenommen werden.

2.4.11 *Pins* – Pin-Punkte einer Collage

Hinter **Pins** wird eine Punktsequenz (siehe 2.5) erwartet, durch die die Pin-Punkte der Collage definiert werden. Nicht jede Collage benötigt Pin-Punkte; standardmäßig ist die Sequenz der Pin-Punkte leer.

Anmerkung: Die mehrfache Angabe von Pin-Punkten zu einer Collage ist syntaktisch möglich, wenn auch unsinnig. Die letzte Angabe erhält den Zuschlag.

2.4.12 *OneEdge* – Definition einer Hyperkante

Hinter **Edge** wird das *Label* der zu definierenden Hyperkante erwartet. Auf diese Angabe wird in der Definition der Regeln Bezug genommen (siehe 2.4.4). Danach folgt eine Punktsequenz (siehe 2.5), durch die die Attach-Punkte der Hyperkante festgelegt werden, sowie eine optionale Farbzuweisung (*OptEdgeColor*) in Form eines Farbindexes.

2.4.13 *OnePart* – Definition eines Parts

Eine Part-Definition beginnt mit dem Bezeichner für eine der möglichen Grundformen. Derzeit stehen folgende Grundformen zur Verfügung: **Dot** (einzelner Punkt), **Line** (Strecke zwischen zwei Punkten), **DashLine** (gestrichelte Linie zwischen zwei Punkten), **Polyline** ($n-1$ Strecken durch n Punkte), **Polygon** bzw. **FilledPolygon** (n Strecken durch n Punkte bzw. mit Farbe gefüllte Polygone), **Circle** bzw. **FilledCircle** (Kreisbogen bzw. mit Farbe gefüller Kreis) sowie **Ellipse** bzw. **FilledEllipse** (Ellipsenbogen bzw. mit Farbe gefüllte Ellipse).

Je nach Grundform werden verschiedene Parameter, meist in Form von Punkt-Angaben (siehe 2.5), erwartet. Die Bedeutung dieser Parameter wird im folgenden erläutert.

Dot *Point*: Angabe des zu setzenden Punktes.

Line / **DashLine** *Point Point*: Angabe des Anfangs- und Endpunktes.

Polyline *Point Point Points*: Angabe von zwei oder mehr Punkten, durch die die Strecken verlaufen sollen.

Polygon / **FilledPolygon** *Point Point Points*: Angabe von zwei oder mehr Punkten, durch die das Polygon aufgespannt wird.

Circle / **FilledCircle** *Point Point*: Angabe des Mittelpunktes und eines beliebigen Randpunktes.

Ellipse / **FilledEllipse** *Point Point Point Point Angle* **,** *Angle*: Angabe zweier Achsen (jeweils Anfangs- und Endpunkt) sowie Anfangs- und Endwinkel des gewünschten Ausschnittes in Grad (für eine volle Ellipse sind entsprechend 0 und 360 anzugeben).

2.4.14 *OptPartColor* – Festlegen der Partfarbe

Die Farbe eines Parts kann entweder durch **Color** gefolgt von einem Farbindex oder durch **RGB** gefolgt von einem 3-Tupel für die Intensitäten der Farbkanäle Rot, Grün und Blau festgelegt werden.

Der Wertebereich eines Farbindexes ist von der Grafikhardware des verwendeten Systems abhängig. Die resultierende Farbe hängt von der globalen Farbpalette des Systems ab. Diese Form der Farbzuweisung existiert nur noch, um Kompatibilität zu älteren Grammatiken sicherzustellen und sollte in Zukunft nicht mehr verwendet werden.

Der Wertebereich der drei Farbkanäle Rot, Grün und Blau geht stets von 0 bis 255, unabhängig von den tatsächlichen Möglichkeiten der Grafikhardware. Durch das RGB-Modell wird eine additive Farbmischung realisiert, d.h. der Wert $(255, 255, 255)$ steht für weiß und $(0, 0, 0)$ für schwarz.

Erfolgt keine Farbzuweisung für ein Part, so ist es schwarz. Dies kann auf einem TrueColor-System dazu führen, daß Parts nicht sichtbar sind, da dann der Hintergrund ebenfalls schwarz ist. Es wird empfohlen, die Farbe jedes Parts mittels einer **RGB**-Definition festzulegen.

2.4.15 *OptColorProc* – Farbprozeduren für Parts

Nach **ColorProc** wird ein *String* erwartet, der den Namen einer Tcl-Prozedur enthalten sollte, die zuvor mittels der Optionen *OptionSource* bzw. *OptionEval* definiert wurde. Wird einem

Part eine Farbprozedur zugewiesen und wurde die Option *OptionCCP* aktiviert, so wird in jedem Ableitungsschritt, in dem dieses Part (als Teil einer rechten Regelseite) in die aktuelle Collage eingefügt wird, die Farbprozedur aufgerufen, um die Farbe des Parts zu berechnen. Die Farbprozedur muß folgender Deklaration genügen:

```
proc MyColorProc {step redV greenV blueV} {
}
```

Die Prozedur erhält vier Parameter. Der erste (`step`) ist ein reiner Wertparameter und enthält die Nummer des Ableitungsschrittes. Diese Information kann von der Farbprozedur verwendet werden, um Farben in Abhängigkeit vom aktuellen Ableitungsschritt zu setzen. Die folgenden drei Parameter (`redV`, `greenV` und `blueV`) sind Variablen-Parameter. Diese Variablen müssen von der Farbprozedur jeweils mit Werten von 0 bis 255 belegt werden. Für den Zugriff auf die Variablen wird in Tcl das `upvar`-Kommando benötigt. Eine minimale Farbprozedur sieht zum Beispiel folgendermaßen aus:

```
proc MyColorProc {step redV greenV blueV} {

    upvar $redV red
    upvar $greenV green
    upvar $blueV blue

    set red 100
    set green 250
    set blue 50
}
```

Hinweis 1: Die Datei `colorprocs.tcl` enthält einige vorgefertigte Farbprozeduren, die Farbverläufe verschiedener Art realisieren.

Hinweis 2: Für Informationen zu Tcl (Tool Command Language) siehe [?].

2.5 Punkte und Punktsequenzen

Points – Punktsequenz

Eine Punktsequenz besteht aus beliebig vielen (auch Null) Punkten. Vor jedem Punkt und am Ende der Sequenz können *PtOperations* (siehe 2.6) erfolgen.

TransPoint – Einzelner Punkt

Ein Punkt wird definiert, indem die einzelnen Koordinaten (derzeit 2 oder 3) durch Kommata voneinander getrennt werden. Die Punktangabe wird in runde Klammern eingeschlossen. Jede Koordinate kann ein komplexer Ausdruck sein (*Num*, siehe 2.9), der vom Collage-Two-Parser ausgewertet wird. Auf Punkte, die durch *TransPoint* definiert werden, wirken die Transformationen des Punkt-Transformations-Stack.

2.6 Punkt-Transformations-Stack (PTS)

Der Collage-Two-Parser verwaltet einen Stack von Transformationen, die auf alle Punktangaben wirken. Standardmäßig ist dieser Stack leer und alle Punktangaben werden unverändert übernommen. Befinden sich hingegen zu einem Zeitpunkt während des parsens n Transformationen auf dem Stack, so wird ein zu diesem Zeitpunkt definierter Punkt p wie folgt transformiert:

$$p := t_1(t_2(\dots t_n(p)\dots)).$$

t_n ist dabei diejenige Transformation, die als letztes auf den Stack gelegt wurde.

Während des parsens kann der PTS durch spezielle Befehle modifiziert werden, indem Transformationen auf den Stack gelegt (**PtPush**) oder die oberste Transformation entfernt wird (**PtPop**). Hierdurch bekommt die Reihenfolge, in der syntaktische Einheiten in der Eingabedatei verarbeitet werden, eine wichtige Bedeutung. Diese Reihenfolge ist jedoch identisch mit der Reihenfolge der syntaktischen Einheiten in der Textdatei, so daß sich hieraus keine besonderen Probleme ergeben.

PtOperations – PTS modifizieren

PTS-modifizierende Operationen sind an vielen Stellen der Syntax erlaubt und stets optional. *PtOperations* werden durch geschweifte Klammern {} eingeschlossen und können aus mehreren **PtPush**- bzw. **PtPop**-Operationen bestehen.

- **PtPush** legt eine neue Transformation auf den PTS. Derzeit hat die Angabe dieser Transformation (*NewPointTrans*) nur eine mögliche Ausprägung: **Map Affine** *IPoints* **To** *IPoints*. Die Transformation wird hierbei indirekt spezifiziert, indem zwei Punktsequenzen angegeben werden, die vermittels einer affinen Transformation t aufeinander abgebildet werden. Existiert ein eindeutiges t für zwei gegebene Punktsequenzen, so wird es von Collage-Two automatisch ermittelt und als neue Transformation verwendet. Es sollte beachtet werden, daß in zwei Dimensionen die Sequenzen aus jeweils drei paarweise linear unabhängigen Punkten bestehen müssen, um eine affine Abbildung eindeutig zu definieren. In drei Dimensionen sind entsprechend vier paarweise linear unabhängige Punkte erforderlich.
- **PtPop** entfernt die oberste Transformation vom PTS.

Hinweis 1: Wie der Definition von *NewPointTrans* zu entnehmen ist, werden die Punktsequenzen hier durch *IPoints* statt durch *Points* definiert. Der einzige Unterschied liegt darin, daß diese Punktangaben (*InternalPoint*) nicht durch den PTS modifiziert werden.

Hinweis 2: Die Datei `trans.csi` enthält Makro-Definitionen, die **PtPush**-Operationen für viele typische Anwendungsfälle auf einfach anwendbare Kommandos abbilden, z.B. `ROTATE2D`, `TRANSLATE2D`, `SCALE2D` und andere.

2.7 Ableitungsteil

Nach dem Grammatikteil wird durch das Schlüsselwort **Derivation** die Definition der Ableitung eingeleitet. Neben einigen optionalen Textblöcken (siehe 2.8) besteht diese Definiton aus zwei Hauptteilen: Ableitungseinträge (*DerivEntries*) sowie Zusatzinformationen zu den einzelnen Tabellen (*TableInfos*).

DerivEntries – Ableitungseinträge

Es können beliebig viele Ableitungseinträge definiert werden. Jeder einzelne Eintrag (*DerivEntry*) hat eine von zwei Formen:

1. **Until** *Step* **Do** *Table*: Bis einschließlich Ableitungsschritt *Step* $(1 \ldots n)$ wird Tabelle *Table* verwendet. Beispiel:

   ```
   Until 3 Do 1
   ```

   Bis einschließlich zum 3. Ableitungsschritt wird Tabelle 1 verwendet.

2. **Until** *Step* **Repeat** *TableElemEntries*: Bis einschließlich Ableitungsschritt *Step* $(1 \ldots n)$ wird die durch *TableElemEntries* spezifizierte Folge von Tabellen verwendet. Mehrere Element-Einträge (*TableElemEntry*) werden durch Kommata voneinander getrennt. Ein Element-Eintrag hat die Form *Anzahl* **Times** *Table*: Die Tabelle *Table* wird *Anzahl* mal hintereinander angewendet. Beispiel:

   ```
   Until 20 Repeat 3 Times 1, 1 Times 2, 4 Times 3
   ```

   Bis einschließlich zum 20. Ableitungsschritt wird folgende Sequenz von Tabellen zyklisch verwendet: 3 mal Tabelle 1, 1 mal Tabelle 2 und 4 mal Tabelle 3.

TableInfos – Tabellen-Parameter

Jede durch die Ableitungseinträge spezifizierte Tabelle verfügt über einen Satz von Parametern, die Ableitungen mit dieser Tabelle beeinflußen. Jeder dieser Werte hat eine Voreinstellung, so daß die Angabe einer *TableInfo* für eine Tabelle nur notwendig ist, wenn einer oder mehrere Parameter geändert werden müssen.

Nach **TableInfo** wird die Nummer der Tabelle erwartet, deren Parameter geändert werden sollen. Danach folgt eine Folge von Informationseinträgen (*TableInfoEntry*). Derzeit existieren lediglich zwei änderbare Parameter:

- **Sequential** *Num*: Die Voreinstellung ist 0. Wird ein Wert ungleich 0 angegeben, so erfolgen in dieser Tabelle lediglich sequentielle Ableitungsschritte, d.h. es wird pro Schritt maximal eine Hyperkante ersetzt.
- **ExhaustProb** *Num*: Die Voreinstellung ist 0. Wird ein Wert ungleich 0 angegeben, so erfolgt die Auflösung nicht-deterministischer Regeln unter Verwendung "ausschöpfender Wahrscheinlichkeiten". Bei diesem Verfahren wird, falls eine zunächst per Pseudo-Zufall ausgewählte Regel nicht anwendbar ist, solange versucht, unter den verbleibenden Regeln (mit gleichem Label) eine Alternative zu finden, bis entweder eine Regel anwendbar ist oder keine passenden Regeln mehr existieren. Normalerweise wird der Versuch einer Kantenersetzung abgebrochen, wenn eine per Pseudo-Zufall ausgewählte Regel nicht anwendbar ist.

Hinweis 1: Bei nicht-deterministischen kontextsensitiven Regeln ohne Wahrscheinlichkeitsangabe wird **ExhaustProb 1** immer die richtige Wahl sein, da ansonsten nur die erste angegebene

Regel überhaupt die Chance einer Anwendung erhält.

Hinweis 2: Das Setzen von **ExhaustProb 1** ist bei Regeln mit Wahrscheinlichkeitsangaben kein Ersatz für eine Gleichverteilung der Wahrscheinlichkeiten. Werden beispielsweise die Wahrscheinlichkeitsangaben weggelassen, so erhalten alle Regeln die Wahrscheinlichkeit 1. Existieren nun z.B. drei Regeln für ein Label `S`, so erhält die erste Regel eine deutlich höhere effektive Wahrscheinlichkeit als die beiden anderen. Genauer gesagt: sofern die erste Regel immer anwendbar ist, kommen die beiden anderen niemals zum Zuge. Richtig wäre es in diesem Fall, die Regeln jeweils mit einer Wahrscheinlichkeit von 1/3 zu versehen.

2.8 Textblöcke

Textblöcke werden verwendet, um die PostScript-Ausgabe von Collage-Two zu modifizieren oder zu erweitern. Für den größten Teil der bisherigen Anwendungen ist die Verwendung von Textblöcken nicht notwendig. Werden die erzeugten PostScript-Dateien in LaTeX-Dokumenten verwendet, so ist es in der Regel vorzuziehen, Änderungen der PostScript-Parameter von dort aus vorzunehmen.

Syntaktisch ist ein Textblock eine Sequenz von Zeichenketten (*Strings*), die durch die Schlüsselwörter **BeginAttributes** und **EndAttributes** eingeschlossen wird. Der Inhalt des Textblockes wird ohne Änderung und ohne Überprüfungen in die jeweilige PostScript-Datei geschrieben, d.h. es muß sich um korrekten PostScript-Code handeln, sonst können bei der Ausgabe der PostScript-Datei Fehlermeldungen des PostScript-Interpreters auftreten. Die PostScript-Attribute von Collage-Two sind in 2.11 beschrieben.

Wie der Syntax zu entnehmen ist, können Textblöcke (*Tb**) an verschiedenen Stellen der Eingabedatei definiert werden. Dabei gibt es drei Ebenen von Lokalität: Global, Ableitungs- oder Grammatikteil und auf bestimmte Bilder bezogen. Bei Ausnutzung aller Möglichkeiten werden in jede Datei daher drei Textblöcke in der oben genannten Reihenfolge kopiert.

Im einzelnen gibt es folgende Textblöcke:

Globaler Textblock

Der Textblock *TbGlobal* wird in jede generierte PostScript-Datei kopiert. Er sollte daher Einstellungen oder Änderungen sehr allgemeiner Natur vornehmen, die für Grammatik- wie Ableitungsbilder gleichermaßen gelten. Erzeugbar durch: *Grammarfile*.

TbGrammar und *TbDerivation*

Der Textblock *TbGrammar* wird in jede Eingabedatei (`G`n) kopiert, d.h. der PostScript-Code dieses Blocks wirkt sich auf die Darstellung der Start-Collage und der Regeln aus. Erzeugbar durch: *Grammarpart*.

TbDerivation wird in jede Ableitungsdatei (`D`n) kopiert, d.h. er wirkt sich auf die Darstellung der nullten und der ersten bis zur letzten Ableitung aus. Erzeugbar durch: *DerivationPart*.

Bildbezogene Textblöcke

- *TbStart* wird ausschließlich in die Datei `G001` kopiert und bezieht sich somit nur auf die Darstellung der Start-Collage im Grammatikteil. Erzeugbar durch: *StartCollage*.
- *TbRule* kann für jede einzelne Regel angegeben werden und bezieht sich dann auf die zur Regel erzeugte Datei (`G002` bis `G`n). Erzeugbar durch: *Rule*.
- *TbDerivStart* bezieht sich auf die Start-Collage im Ableitungskontext, d.h. der Textblock wird in die Datei `D001` kopiert. Erzeugbar durch: *DerivationPart*.
- *TbTableEntry* bezieht sich auf alle Ableitungsschritte und die dafür erzeugten Dateien, die durch den jeweiligen Ableitungs-Eintrag abgedeckt werden. Erzeugbar durch: *DerivEntry*.

2.9 Numerische Ausdrücke

Alle numerischen Angaben in der Eingabedatei werden letzlich auf die syntaktische Kategorie *Num* zurückgeführt. Aus der Syntaxdefinition ist bereits zu erkennen, daß das Bilden von arithmetischen Ausdrücken und die Verwendung einiger mathematischer Funktionen möglich sind. Im Einzelnen beherrscht der Parser die Grundrechenarten, die Klammerung von Ausdrücken, unäres Minus, Aufruf vordefinierter Funktionen sowie eine Variablenersetzung. Die beiden zuletzt genannten Eigenschaften werden im folgenden näher erläutert.

Vordefinierte Funktionen

Folgende Funktionen sind in den Collage-Two-Parser eingebaut. Jede Funktion erwartet in Klammern ein numerisches Argument, das wiederum durch *Num* aufgebaut werden kann.

- **sqrt**: Wurzelfunktion. Das Argument muß positiv sein.
- **sin**, **cos**, **tan**: Sinus, Kosinus und Tangens. Gradangaben werden grundsätzlich im Bogenmaß erwartet ($rad(deg) = deg*\pi/180$). (Die Datei `trans.csi` definiert ein entsprechendes Umrechnungsmakro.)
- **asin**, **acos**, **atan**: Arcus-Sinus, Arcus-Kosinus und Arcus-Tangens
- **sinh**, **cosh**, **tanh**: Sinus-Hyperbolicus, Kosinus-Hyperbolicus und Tangens-Hyperbolicus
- **exp**: Exponentialfunktion (e^x)
- **log**: Natürlicher Logarithmus (zur Basis e).
- **log10**: Logarithmus zur Basis 10.

Variablenersetzung

In einem *Term* kann nach dem Zeichen **$** eine Zeichenkette (*String*) angegeben werden, die als Name einer numerischen Tcl-Variablen interpretiert wird. Der *Term* wird dann durch den aktuellen Wert dieser Variablen substituiert.

Beispiel: $ "redIncr"

Hinweis: Für Informationen zu Tcl (Tool Command Language) siehe [**?**].

2.10 Collage-Two-Commander

Nach Auswahl des Menüpunktes `Commander...` des Menüs `File` wird eine neue neue Instanz eines Collage-Two-Commanders erzeugt und in einem neuen Fenster dargestellt (Abbildung 6). Der Commander ist analog zum Grammatik-Editor zu bedienen. Im Gegensatz zu diesem werden im Commander Tcl-Scripte bearbeitet und ausgeführt. Jede Commander-Instanz besitzt einen eigenen Tcl-Interpreter, der spezielle Kommandos enthält, um die wichtigsten Funktionen von Collage-Two "fernzusteuern". Die Hauptanwendung des Commanders liegt in der Erstellung von Animationen, die auf parametrisierten Grammatiken basieren.

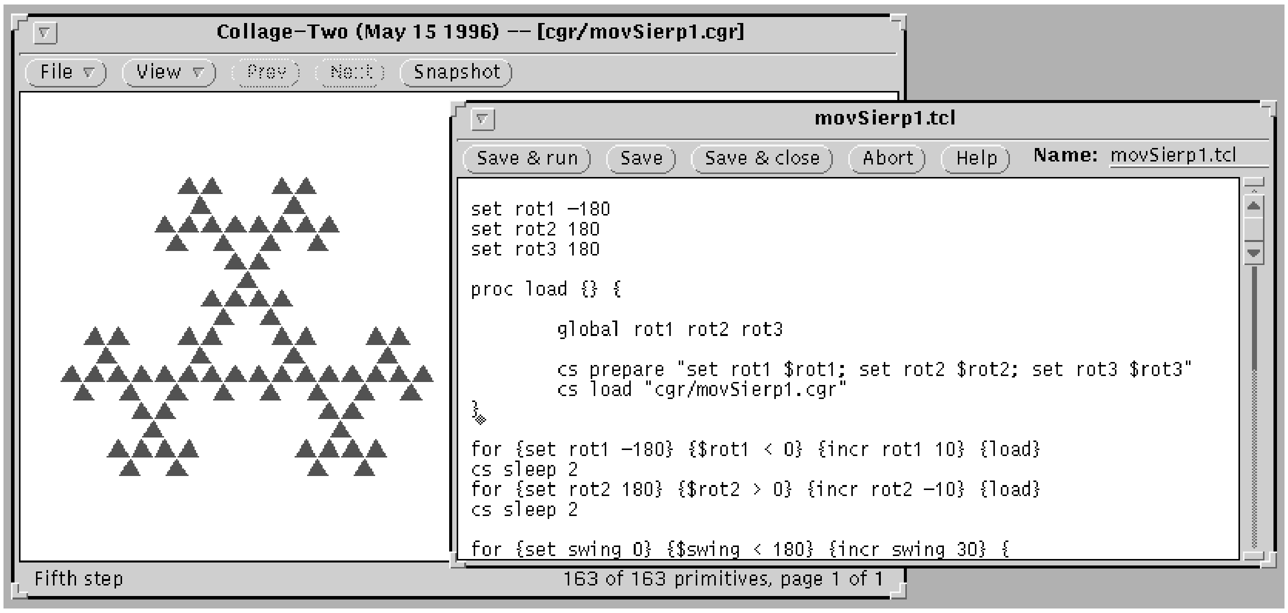


Abbildung 6: Collage-Two-Commander mit geladener Datei `movSierp1.cgr`

Neben den standardmäßig in jedem Tcl-Interpreter vorhandenen Kommandos ist ein spezielles **cs**-Kommando (für Collage-System) vorhanden, das als ersten Parameter die auszuführende Funktion erwartet. Folgende Funktionen stehen zur Verfügung:

- **cs load** *Filename*: Lädt die durch *Filename* spezifizierte Grammatikdatei ein und führt die darin definierte Ableitung aus. Dieses Kommando ist analog zur Auswahl des Menüpunktes `Load...` des Menüs `File`.
- **cs reload**: Lädt die aktuelle Grammatikdatei erneut ein. Dieses Kommando ist analog zur Auswahl des Menüpunktes `Reload` des Menüs `File`.
- **cs firstpage** / **cs lastpage** / **cs prevpage** / **cs nextpage**: Zeigt die erste / letzte / vorherige / nächste erzeugte Seite an. Diese Kommandos sind analog zur Auswahl der Menüpunkte `First` / `Last` / `Prev` / `Next` des Menüs `View`.
- **cs scrollleft** / **cs scrollright** / **cs scrollup** / **cs scrolldown** [*Num*]: Verschiebt den Bildinhalt nach links / rechts / oben / unten. Der optionale numerische Parameter gibt die Anzahl der Verschiebungen an. Diese Kommandos sind wiederum analog zur (mehrmaligen) Auswahl der entsprechenden Menüpunkte des `View`-Menüs.
- **cs zoomin** / **cs zoomout** [*Num*]: Zoomt in das Bild hinein oder aus dem Bild heraus. Der optionale numerische Parameter gibt die Anzahl der Wiederholungen an. Diese Komman-

dos sind analog zur (mehrmaligen) Auswahl der Menüpunkte `Zoom In` bzw. `Zoom Out` des Menüs `View`.

- **cs reset** setzt die Parameter für die Bildschirmansicht zurück, analog zur entsprechenden Funktion des `View`-Menüs.
- **cs zoomrect** *x y width height*: Zoomt in den angegebenen Bildausschnitt. x und y sind die Bildschirm-Koordinaten (Koordinatenursprung oben links, alle Angaben in Pixel) des Zoom-Rechteckes, *width* und *height* geben die Breite und Höhe des Rechteckes an.
- **cs sleep** *seconds*: Hält die Ausführung des Systems für *seconds* Sekunden lang an. Während dieser Zeit wird auch nicht auf Benutzereingaben reagiert.
- **cs snapshot** *Filename*: Erstellt einen Schnappschuß der aktuellen Bildschirmansicht und sichert diesen in die angegebene Bildatei. Die Bilder werden im sogenannten *Portable Bitmap Format* abgespeichert, das mit vielen gängigen Grafikprogrammen weiterverarbeitet werden kann.
- **cs prepare** *Script*: Dieses Kommando realisiert die Kommunikation zwischen Commander und der aktuellen Grammatikdatei. *Script* kann beliebigen Tcl-Code enthalten, der im Interpreter der aktuellen Grammatikdatei zur Ausführung gebracht wird. Typischerweise werden hierdurch Variablen gesetzt, auf die in der Grammatik Bezug genommen wird.

2.11 PostScript-Attribute

Die von Stefan Taubenberger geschriebene PostScript-Ausgabe von Collage-Two kann auf vielfältige Weise beeinflußt werden. Zunächst soll das Grundprinzip dargestellt werden.

PostScript-Moduldatei `collagen.ps`

Mit der *OptionF*-Option (siehe 2.3.1) wird die PostScript-Ausgabe aktiviert und gleichzeitig die Namen der Ausgabedateien festgelegt. Für jedes Bild wird genau eine Datei erzeugt. Der PostScript-Code einer Datei ist jedoch nicht ohne weiteres interpretierbar, da er Kommandos verwendet, die in einem von allen Dateien gemeinsam benutzten Modul definiert sind. Diese Modul-Datei heißt `collagen.ps`, wobei n die Version der PostScript-Ausgabe wiederspiegelt und zur Zeit den Wert 1 hat. `collagen.ps` ist eine komprimierte Version einer kommentierten Quelltextdatei namens `colla1-n.ps` und geht aus dieser durch Anwendung des Programmes `squeeze` hervor, das im `dvips`-Paket enthalten ist.

Um eine erzeugte Bilddatei, z.B. `Test_D001.ps`, auf einem PostScript-Gerät auszugeben, muß diese mit `collagen.ps` zusammengehängt werden. Auf einem UNIX-System kann dies beispielsweise durch den `cat`-Befehl bewerkstelligt werden: `cat collagen.ps Test_D001.ps > output.ps`. Die Datei `output.ps` kann dann direkt auf einem PostScript-Drucker ausgegeben werden.

Das Problem bei direkter Ausgabe ist jedoch, daß das Bild wahrscheinlich nicht korrekt skaliert ist und sich zudem mit großer Sicherheit an der falschen Position befindet. Diesem Problem kann auf zwei Arten begegnet werden. Entweder man verwendet von vornherein ausschließlich positive Koordinaten in "vernünftigen" Bereichen, so daß keine Änderungen an der Ausgabe notwendig sind, oder aber man fügt zusätzlichen PostScript-Code ein, der das Bild skaliert und an die richtige Stelle verschiebt (PostScript-Befehle `scale` bzw. `translate`). Einfacher ist jedoch sicherlich die Einbindung der Bilder in ein LaTeX-Dokument, was im folgenden beschrieben wird.

Einbindung in LaTeX

Für die Einbindung der von Collage-Two erzeugten PostScript-Bilder in LaTeX-Dokumente existiert eine kleine Style-Datei namens `collfign.sty`. Wie schon bei `collagen.ps` gibt das n die Versionsnummer der Datei an (derzeit 1). Es werden drei Kommandos definiert, die in LaTeX verwendet werden können:

- `\beginattributes`: Nachfolgend können beliebige Attribute der PostScript-Ausgabe geändert werden. Die einzelnen Attribute werden im nächsten Abschnitt beschrieben.
- `\endattributes`: Gegenstück zu `\beginattributes`, um die Änderung der Attribute abzuschließen.
- `\collfig{}`: Gibt eine von Collage-Two erzeugte PostScript-Datei aus. Die Parameter sind analog zu denen des `\epsfig`-Befehls aus dem gleichnamigen LaTeX-Paket. Da `\collfig` auf `\epsfig` abgebildet wird, ist die Verwendung dieses Paketes obligatorisch. Soll ein anderes Paket zur Einbindung von PostScript-Dateien verwendet werden, so kann die notwendige Änderung in der Datei `collfign.sty` vorgenommen werden.

Änderung von PostScript-Attributen

Durch Änderung von Attributen kann Einfluß auf die Darstellung einzelner Elemente genommen werden. Beispielsweise können die Größe der Pin-Punkte, die verwendeten Schriftarten und Größen sowie Linien- und Füllgrade eingestellt werden. Die Änderung von Attributen kann auf zwei Arten geschehen:

- In der Grammatikdatei innerhalb der Textblöcke (siehe 2.8)
- Im LaTeX-Dokument zwischen `\beginattributes` und `\endattributes`.

Innerhalb der Änderungsblöcke wird jeweils reiner PostScript-Code erwartet. Obwohl im Prinzip beliebige PostScript-Anweisungen an diesen Stellen eingefügt werden können, wird man sich in den meisten Fällen auf die Änderung von Attributen beschränken. Eine Attributänderung hat folgende syntaktische Form:

$$\{\ \textit{Wert Attributname}\ \}\ \textit{Kategorie}$$

Attributänderungen in einer Grammatikdatei haben Vorang vor denen, die in einem LaTeX-Dokument vorgenommen werden. Daher sollten in Grammatikdateien nur solche Attribute gesetzt werden, die für alle Verwendungen in LaTeX-Dokumenten gleich bleiben.

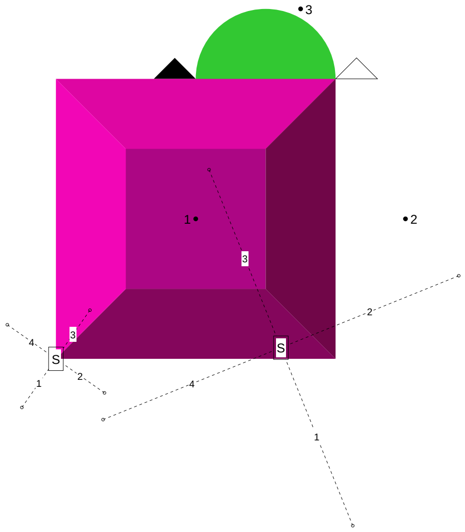


Abbildung 7: Collage *Test* mit standardmäßigen PostScript-Attributen

Attributnamen sind generisch und erlangen erst mit Angabe der Kategorie Eindeutigkeit. Es existieren folgende Kategorien:

Kürzel	Kategorie	Bedeutung
A	`Attachs`	Die Attach-Punkte der Hyperkanten
B	`Boxes`	Die Kästen der Hyperkanten, in denen das Label steht
E	`Edges`	Die Hyperkanten (als Ganzes)
I	`Pins`	Die Pin-Punkte
P	`Parts`	Die Parts einer Collage
T	`Tentacles`	Die Tentakeln ("Ärmchen") der Hyperkanten

Die folgende Tabelle listet alle vorhandenen Attribute auf. Zu jedem Attribut wird neben einer Erläuterung der Wertebereich sowie die Menge der Kategorien angegeben, in denen das Attribut Gültigkeit hat.

Attribut	Wertebereich	Kategorien	Bedeutung
`ClearBackground?`	`true`, `false`	B,E,I,T	(Text-) Hintergrund löschen ?
`Draw?`	`true`, `false`	A,E,I,P	Diese Kategorie zeichnen ?
`DrawBoundaries?`	`true`, `false`	P	Den Rand von Flächen zeichnen ?
`DrawToCenter?`	`true`, `false`	E,T	Tentakeln auf das Zentrum ausrichten ?
`Fill?`	`true`, `false`	A,B,E,I,P	Objekte dieser Kategorie füllen ?
`FillColour`	$0 \ldots 1$	A,B,E,I,P	Füllgrad: 0=schwarz, 1=weiß
`Font`	`/`*Fontname*	global	Font wählen: `Helvetica`, `Times` etc.
`FontSize`	$1 \cdots \sim 50$	B,I,P	Fontgröße
`Height`	$0 \ldots$	B,E	Höhe der Hyperkantenbox
`Horizontal?`	`true`, `false`	E,I,T	Horizontale Textausrichtung?
`LineColour`	$0 \ldots 1$	global	Linienhelligkeit (obsolet bei Parts)
`LineWidth`	$0 \ldots$	global	Linienbreite
`LineDash`	$[n\ m\ \ldots]$	global	Linienschraffur (numer. PostScript-Array)
`NumberOfOutputs`	$-1, 0, 1, \ldots$	E	$\geq 0 \rightarrow$ Einschränkung der Pin- und Attach-P.
`Offset`	$0 \ldots$	B,I,T	Abstand zu Box, Pinpunkten und Tentakeln
`Radius`	$0 \ldots$	A,E,I,P	Radius von Kreisen
`Width`	$0 \ldots$	B,E	Breite der Hyperkantenbox
`Write?`	`true`, `false`	B,E,I,T	Text bzw. Label ausgeben ?
`WriteSingleNr?`	`true`, `false`	E	Einzelne Pinpunkte/Tentakeln beschriften ?

Beispiel

Abbildung 7 zeigt eine Test-Collage, die ohne Änderung von PostScript-Attributen ausgegeben wurde. Die dazu notwendigen LaTeX-Kommandos beschränken sich auf eine Zeile:

```
\collfig{file=Test.ps,width=6cm}
```

Abbildung 8 zeigt die gleiche Collage, allerdings mit geänderten PostScript-Attributen. Diese Abbildung wurde mit folgendem LaTeX-Code realisiert:

```
\beginattributes
  {true Fill?} Attachs
  {.4 FillColour} Attachs
  {8 Radius} Attachs
  {false ClearBackground?} Boxes
  {16 Width} Boxes
```

```
  {16 Height} Boxes
  {true Fill?} Edges
  {.8 FillColour} Edges
  {true Fill?} Pins
  {.9 FillColour} Pins
  {40 FontSize} Pins
  {/Times Font} Pins
  {25 Offset} Pins
  {10 Radius} Pins
  {false Fill?} Parts
  {[5 3] LineDash} Parts
  {1.5 LineWidth} Parts
  {15 FontSize} Tentacles
  {[] LineDash} Tentacles
\endattributes
\collfig{file=Test.ps,width=10cm}
```

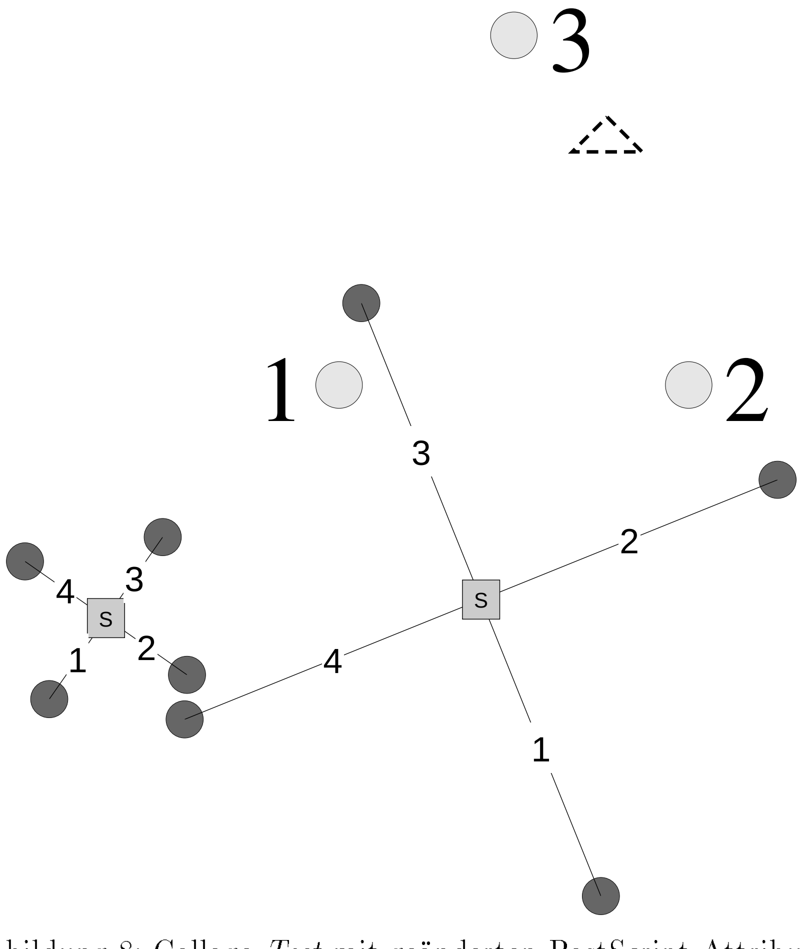


Abbildung 8: Collage *Test* mit geänderten PostScript-Attributen

Kapitel 3

Implementierung von Collage-Two

> The process of preparing programs for a digital computer is especially attractive, not only because it can be economically and scientifically rewarding, but also because it can be an aesthetic experience much like composing poetry or music.
>
> — *Donald E. Knuth, in [?]*

In diesem Kapitel werden die Grundzüge der Implementierung von Collage-Two dargestellt. Dabei werden die direkt mit Collagen-Grammatiken zusammenhängenden Aspekte in den Vordergrund gestellt, während andere Bereiche, wie z.B. der Parser, die Oberflächenprogrammierung und die Einbindung in das Betriebssystem, nicht behandelt werden.

Im Text erfolgen des öfteren Verweise auf bestimmte Quelltextdateien. Am Ende des Kapitels sind einige dieser Dateien, die wichtige Klassendeklarationen enthalten, auszugsweise abgedruckt. Für ein vollständiges Studium der Quelltexte ist es jedoch notwendig, das System zu installieren und mit den Original-Dateien zu arbeiten.

3.1 Modellierung von Collagen-Grammatiken

3.1.1 Die Implementierungssprache C++

> Perfection is achieved only on the point of collapse.
>
> *C.N. Parkison, zit. in [?]*

Als Implementierungssprache für Collage-Two wurde C++ gewählt ([?, ?]). Diese Sprache weist zunächst einmal einige bedeutende Nachteile auf. Durch ihre Abwärtskompatibilität zur Sprache C erbt sie deren wohlbekannte Schwachpunkte. Beispielsweise erlaubt C Pointerarithmetik und dadurch den unkontrollierten Zugriff auf beliebige Speicheradressen. Es kommt hinzu, daß der Zugriff auf Array-Elemente in der gleichen Weise erfolgt und somit nicht zur Laufzeit überprüfbar ist. Als letztes seien noch die weitreichenden und oftmals implizit erfolgenden Typumwandlungen genannt. Die Vorteile von C++ liegen vor allem in der Möglichkeit, objektorientiert zu programmieren. Wenn man sich der von C ausgehenden Gefahren bewußt ist und gefährliche oder unüberschaubare Konstrukte vermeidet, ist mit C++ der Entwurf gut strukturierter und wiederverwendbarer Klassen möglich. Ein weiterer Vorteil liegt, in ambivalenter Weise, wiederum in der Abwärtskompatibilität zu C, die die Einbindung in UNIX- und X-Windows-basierte Systeme erleichtert.

3.1.2 Entwurfsgrundsätze

Collagen-Grammatiken sind ein vollständig formalisiertes theoretisches Konstrukt. Für die Softwareentwicklung sind dies paradiesische Voraussetzungen, da in der zu modellierenden Anwendungsdomäne bereits mathematische Begriffe verwendet werden, die sich sehr direkt in einen Programm- bzw. Klassenentwurf umsetzen lassen. Das bei Softwareprojekten für die "reale Welt" häufig auftretende Problem der "semantic gap", d.h. das Vorhandensein einer Kluft zwischen den Begriffswelten der Anwender und den Entwicklern eines Systems, ist hier praktisch nicht gegeben.

Da der Entwurf objektorientiert erfolgen sollte, war es notwendig, die in Collagen-Grammatiken vorkommenden Objekte zu identifizieren und in Klassendefinitionen umzusetzen. Dabei stellte sich heraus, daß sich viele der in Collagen-Grammatiken definierten Begriffe direkt in Form von Klassen ausdrücken lassen. Dadurch ergibt sich eine weitgehende strukturelle Übereinstimmung des Modells mit dem Modellierten. Um einen ersten Eindruck zu vermitteln, wie weit diese Übereinstimmung geht, seien einige Namen der definierten Klassen genannt: `Coordinate`, `Point`, `PointSequence`, `Transformation`, `Part`, `PartSet`, `Hyperedge`, `PinSequence`, `Collage`, `CollageRule`, `CollageRuleSet`, `Base`, `CollageGrammar`, `Table`.

Im folgenden Abschnitt werden diese und weitere Klassen kurz vorgestellt.

3.2 Die C++-Klassen von Collage-Two

3.2.1 Koordinaten, Punkte und Punktsequenzen

Eine wichtige Grundlage für alles, was in Collagen und Collagen-Grammatiken passieren kann, stellen Koordinaten, Punkte sowie Punktmengen bzw. Punktsequenzen dar. In Collage-Two sind diese in Form der Typen bzw. Klassen `Coordinate`, `Point` sowie `PointSequence` realisiert.

`Coordinate` (Datei `basic.h`): Eine Koordinate ist eine reelle Zahl. Sie wird modelliert durch den C++-Datentyp `double`, der im Gegensatz zu reellen Zahlen sowohl eine eingeschränkte Anzahl von Nachkommastellen als auch einen begrenzten Wertebereich insgesamt hat.

`Point` (Datei `point.h`): Ein Punkt im d-dimensionalen Raum ist ein Element des $\mathbb{R}^d$ und wird als d-Tupel reeller Zahlen geschrieben. In Collage-Two werden Punkte repräsentiert durch die Klasse `Point`. Ein `Point`-Objekt ist eine Sequenz von `Coordinate`-Werten, deren feste Länge bei der Erzeugung des `Point`-Objektes angegeben wird.

`PointSequence`: Sequenzen und Mengen von Punkten werden durch die Klasse `PointSequence` dargestellt. Eine `PointSequence` ist eine Sequenz von `Point`-Objekten, deren Länge sich zur Laufzeit durch das Einfügen und Löschen von Objekten ändern kann.

3.2.2 Matrizen

In Collage-Two bilden Matrizen die Grundlage für die Bestimmung und Anwendung affiner und anderer Transformationen. Sie werden repräsentiert durch die Klasse `Matrix` (Datei `matrix.h`). Ein Objekt dieser Klasse wird konstruiert unter Angabe der Zeilen- und Spaltenanzahl. Die Elemente der Matrix sind vom Typ `Coordinate`. Einzelne Elemente können gesetzt und ihr Wert erfragt werden. Ein Matrixobjekt kann mit einem weiteren Matrixobjekt oder mit einem

`Point` (aufgefaßt als Vektor) multipliziert werden, wobei ein neues `Matrix`- bzw. ein neues `Point`-Objekt als Ergebnis erzeugt wird. Weiterhin existiert eine Funktion zum Invertieren einer Matrix.

Als Teil der Implementierung der Funktion zum Invertieren einer Matrix wird eine Funktion verwendet, die eine geeignete Pivotspalten-Permutation berechnet. Die Implementierung dieser Funktion (Datei `matrix.cc`) liegt in zwei alternativen Versionen vor. Die eine verwendet Backtracking, während die andere, effizientere Version die Implementierung eines Algorithmus von D. E. Knuth ist ([?]).

3.2.3 Transformationen

Unter dem Begriff Transformation wird bei Collagen-Grammatiken eine Abbildung $t : \mathbb{R}^d \to \mathbb{R}^d$ verstanden.

`Transformation` (Datei `trans.h`): In Collage-Two werden Transformationen durch diese Klasse repräsentiert. Es handelt sich um eine virtuelle Basisklasse, d.h. es gibt für sie keine Implementierung, sondern sie dient lediglich der Definition einer einheitlichen Schnittstelle, die von allen abgeleiteten Klassen unterstützt werden muß. Verlangt wird von diesen im wesentlichen die Bereitstellung einer `Transform`-Funktion, die einen einzelnen Punkt transformiert. Derzeit existieren zwei abgeleitete Klassen: `MatrixTrans` und `TranslationTrans`. Erstere wird für alle Transformationen verwendet, die auf einer Matrixmultiplikation basieren (z.B. affine Transformationen), während letztere lediglich die Translation eines Punktes um einen Vektor vornehmen kann.

`TransFinder`: Dies ist eine reine Implementierungsklasse, in dem Sinne, daß es keine direkte Entsprechung dafür auf der Seite der Collagen-Grammatiken gibt. `TransFinder` ist ebenfalls eine virtuelle Basisklasse. Ein Objekt einer daraus abgeleiteten Klasse dient dazu, bei Eingabe zweier Punktsequenzen aus diesen eine Transformation zu berechnen, die die eine auf die andere Punktsequenz abbildet, und diese in Form eines `Transformation`-Objektes auszugeben. Die derzeit vorhandenen Implementierungen dieser Klasse sind `AffineFinder`, `TranslationFinder` sowie `ScaleTranslationFinder`. Sie korrespondieren mit den drei möglichen Transformationsklassen, die in Collage-Two einstellbar sind (vgl. 2.4.2).

In der Klasse `AffineFinder` (Datei `trans.cc`) werden unter anderem die Funktionen implementiert, die eine beliebige Überspezifikation von Attach- und Pin-Punkten erlauben, d.h. die Angabe von mehr als $d+1$ Punkten in einem d-dimensionalen Raum, indem eine linear unabhängige Teilsequenz aus den angegebenen Punkten bestimmt wird, aus der dann mittels Matrix-Invertierung die affine Transformation berechnet wird.

3.2.4 Parts

Die Parts einer Collage werden durch die virtuelle Basisklasse `Part` und durch daraus abgeleitete Klassen repräsentiert (Datei `parts.h`). Konkretisierungen der `Part`-Klasse sind derzeit die Klassen `Dot`, `Line`, `DashLine`, `PolyLine`, `Polygon` und `Ellipse`.

Die von `Part` definierte Schnittstelle umfaßt Funktionen zum Zeichnen des Parts, zum Anwenden einer Transformation auf das Part, zur Berechnung des kleinstmöglichen d-dimensionalen Würfels, der das Part vollständig einschließt (Bounding-Box), sowie zum Setzen und Erfragen der Farbe.

Die Klasse `PartSet` implementiert die Komponente *PART* einer Collage. Ein `PartSet`-Objekt enthält eine Sequenz von beliebigen `Part`-Objekten.

3.2.5 Hyperkanten

Die Hyperkanten einer Collage werden in CollageTwo zunächst durch die virtuelle Basisklasse `Edge` (Datei `hyper.h`) abstrahiert. Die Schnittstelle von `Edge` definiert neben Funktionen zum Zeichnen sowie zum Setzen und Erfragen des Labels und der Äquivalenzklasse vor allem zwei für den Ersetzungsprozeß wichtige Funktionen: Zum einen kann mittels `Transform` jedes `Edge`-Objekt, wie ein `Part`, transformiert werden. Zum anderen verfügt es über eine Funktion `FindTrans`, die für eine gegebene Punktsequenz und ein vorgegebenes `TransFinder`-Objekt eine Transformation zurückliefert, die die Punktsequenz auf die Kante selbst abbildet.

Als Konkretisierungen der `Edge`-Klasse existieren derzeit die Klassen `Hyperedge` und `AnnotatedEdge`. `Hyperedge` realisiert die Funktion `FindTrans` in der zu erwartenden Weise, indem sie die übergebene Punktsequenz zusammen mit der eigenen Attach-Punkt-Sequenz dem `TransFinder`-Objekt übergibt, um die gesuchte Transformation zu ermitteln. Eine `AnnotatedEdge` hingegen ignoriert einfach die übergebene Punktsequenz und liefert immer eine bereits zuvor ermittelte Transformation zurück. Die `Transform`-Funktion wird verwendet, wenn ein `Edge`-Objekt als Teil einer rechten Regelseite in eine Collage eingefügt wird. Handelt es sich um ein `Hyperedge`-Objekt, so transformiert dieses seine Attach-Punkte mit der übergebenen Transformation. Ein `AnnotatedEdge`-Objekt hingegen verkettet die gespeicherte mit der übergebenen Transformation.

Die Klasse `EdgeSet` implementiert eine Sequenz von `Edge`-Objekten und korrespondiert somit mit der geordneten Menge *EDGE* einer dekorierten Collage.

3.2.6 Pin-Punkte

Die Pin-Punkt-Sequenz einer Collage wird durch die Klasse `PinSequence` (Datei `collage.h`) realisiert, die von der Klasse `PointSequence` abgeleitet ist und zusätzlich Funktionen zum Zeichnen der Punkte sowie zum Anwenden einer Transformation definiert.

3.2.7 Collagen

Die für die Repräsentation von Collagen zuständige Klasse heißt erwartungsgemäß `Collage` (Datei `collage.h`). Datenmäßig betrachtet ist sie direkt dem Tupel-Vorbild nachempfunden, indem sie die einzelnen Komponenten einer Collage in Form eines `PartSet`-, `PinSequence`- sowie `EdgeSet`-Objektes beinhaltet. Die Komponenten *lab* und *att* einer Collage haben hier keine Entsprechung, da diese funktional definierten Attribute der Hyperkanten durch die objektorientierte Sichtweise direkt den `Edge`-Objekten zugeordnet wurden. Gleiches gilt für die *col*-Komponente, die Teil eines jeden `Part`-Objektes ist.

Neben Funktionen zum Setzen und Erfragen der einzelnen Komponenten existieren folgende wichtige Operationen auf `Collage`-Objekten:

`Transform` wendet eine Transformation auf alle Komponenten der Collage an ist somit die direkte Umsetzung der unter ?? definierten Transformation einer Collage.

`ReplEdge` ersetzt eine einzelne Hyperkante e in der Collage durch eine Collage *repl* unter Verwendung einer Transformation t. `DeriveWithBase` erhält eine Ableitungsbasis in Form eines `Base`-Objektes, um unter Verwendung von `ReplEdge` alle definierten Ersetzungen in der Collage vorzunehmen. Dies ist die Realisierung der unter ?? definierten Hyperkantenersetzung.

`FindContext` stellt die Basisfunktion für positive wie negative Kontextbedingungen dar, indem überprüft wird, ob eine gegebene Collage in der eigenen enthalten ist. Dies ist die Implementierung der partbasierten Variante des *included*-Prädikats (??).

3.2.8 Produktionen

Die Klasse `CollageRule` repräsentiert eine einzelne Produktion. An Daten enthält ein `CollageRule`-Objekt das Label, die Äquivalenzklasse, die Collage der rechten Regelseite, ggf. eine positive und eine negative Kontext-Collage, die Nummer der Tabelle und die Wahrscheinlichkeit der Produktion. Außerdem enthält es noch ein `TransFinder`-Objekt, das für das Finden der für die Anwendung dieser Produktion erlaubten Transformationen zuständig ist.

Die Funktionalität einer `CollageRule` beschränkt sich neben der graphischen Ausgabe der Produktion auf das Lesen und Schreiben der genannten Komponenten sowie das Durchreichen der `FindTrans`-Funktion des gespeicherten `TransFinder`-Objektes.

Mengen von Produktionen werden durch die Klasse `CollageRuleSet` realisiert. Diese entspricht somit direkt der Produktionsmenge P einer Collagen-Grammatik. Ein `CollageRuleSet` enthält eine Sequenz von `CollageRule`-Objekten. Da die Tabellennummer bereits in jeder einzelnen Regel vermerkt wird, korrespondiert `CollageRuleSet` gleichzeitig mit Tabellen von Produktionen, d.h. mit der Menge der Tabellen $\mathcal{P}$ einer Collagen-Grammatik mit Tabellen (vgl. ??). `CollageRuleSet` stellt einige wichtige Funktionen zur Verfügung:

`Normalize` konvertiert die Produktionen derart, daß sie der unter ?? beschriebenen Normalform genügen. Bei diesem Prozeß werden Äquivalenzklassen generiert und jede Regel erhält eine solche zugewiesen. Die Äquivalenzklassen fungieren bei der Regelanwendung als die eigentlichen linken Seiten, d.h. die in der Grammatik angegebenen Label spielen während der Ableitung in Collage-Two keine Rolle mehr. Durch den Normalisierungsprozeß kann es vorkommen, daß zwei Produktionen für ein gleiches Label verschiedene Äquivalenzklassen zugewiesen bekommen. Dies ist allgemein dann der Fall, wenn sich die Pin-Punkte zweier rechter Regelseiten nicht mittels der erlaubten Transformationen aufeinander abbilden lassen. Da die Start-Collage nicht Teil der Produktionsmenge ist, muß nachfolgend für diese separat ein Aufruf der Funktion `ClassifyEdges` erfolgen, um die in dieser Collage enthaltenen Hyperkanten zu klassifizieren.

`AnnotateEdges` erwartet eine normalisierte Collage als Parameter und tauscht in dieser alle `Hyperedge`-Objekte gegen äquivalente `AnnotatedEdge`-Objekte aus. Zuvor müssen die Produktionen normalisiert worden sein.

3.2.9 Ableitungsbasis

Objekte der Klasse `Base` repräsentieren eine Basis für eine Ableitung in der definierten Collagen-Grammatik. Eine `Base` enthält eine Sequenz von `BaseElement`-Objekten. Jedes dieser Objekte beschreibt die Ersetzung eines `Edge`-Objektes durch ein `Collage`-Objekt unter Verwendung eines `Transformation`-Objektes und enthält daher als Daten genau diese Angaben.

Die einzige öffentlich zugängliche Funktion eines `Base`-Objektes ist `Build`. Diese erhält eine `Collage`, ein `CollageRuleSet` sowie eine `TableInfo` (s.u., Ableitungsbeschreibung) als Parameter und baut daraufhin eine Basis auf.

Eine `Base` stellt intern die für das Aufbauen der Basis benötigten Operationen bereit. `GetNormRandom` liefert einen systemunabhängigen normalisierten Pseudo-Zufallswert im Bereich von 0 bis 1. `ContextOk` testet für eine gegebene Collage und eine anzuwendende Regel die Kontextbedingungen. Dabei wird, falls jeweils vorhanden, sowohl der positive als auch der negative Kontext ausgewertet. `GetEdgeRuleNormal` und `GetEdgeRuleExhaust` sind zwei alternative Funktionen, um eine anzuwendende Regel auszuwählen. Die Alternative entspricht der `ExhaustProb`-Option (siehe 2.7).

3.2.10 Collagen-Grammatik

Collagen-Grammatiken finden ihr Gegenstück in der Klasse `CollageGrammar`. Ein Objekt dieser Klasse enthält lediglich zwei Komponenten: Ein `Collage`-Objekt als Start-Collage sowie ein `CollageRuleSet`-Objekt, das alle Produktionen der Grammatik enthält. Auf die explizite Repräsentation des Markierungsalphabetes wird verzichtet; alle in der Eingabedatei verwendeten Zeichen für Label werden implizit dem Alphabet hinzugefügt.

Neben der Zugriffsmöglichkeit auf die beiden Komponenten gibt es die Funktionen `Normalize` und `Annotate`, die die bereits beschriebenen Operationen ausführen und dabei die Start-Collage mit einbeziehen.

3.2.11 Ableitungsbeschreibung

Die Klasse `Table` repräsentiert die in der Eingabedatei definierte Ableitungsbeschreibung. Für diese gibt es keine formalisierte Entsprechung in Collagen-Grammatiken; im Prinzip handelt es sich aber lediglich um eine Sequenz von Tabellennummern. Daß die interne Repräsentation sich davon unterscheidet, liegt an den flexiblen Definitionsmöglichkeiten (siehe 2.7), durch die auch sehr lange Ableitungen mit nach einem bestimmten Schema wechselnden Tabellennummern spezifizierbar sind. Eine Vorausberechnung und explizite Abspeicherung aller Tabellennummern könnte für sehr lange Ableitungen ineffizient sein.

Die wesentliche Funktionalität eines `Table`-Objektes besteht darin, für jeden Ableitungsschritt die Nummer der als nächstes anzuwendenden Tabelle zurückzuliefern.

`TableInfo`-Objekte enthalten Informationen über Optionen, die einer bestimmten Tabelle zugeordet sind (vgl. 2.7).

3.3 Ausgewählte Header-Dateien

Auf den folgenden Seiten sind einige der erwähnten Header-Dateien, die die Klassendeklarationen enthalten, abgedruckt. Teilweise wurden die Dateien erheblich gekürzt, damit nicht vom wesentlichen abgelenkt wird und um den Umfang dieses Teils erträglich zu halten. Ein vollständiger Abdruck der gesamten Quelltexte würde den Rahmen sicherlich sprengen.

Im einzelnen sind folgende Dateien abgedruckt:

`point.h`	Seite ??
`matrix.h`	Seite ??
`trans.h`	Seite ??
`hyper.h`	Seite ??
`collage.h`	Seite ??

3.3.1 `point.h`

```
#ifndef __POINT_H__
#define __POINT_H__
//
// point.h  --  CollageTwo
//
// Point, PointSequence
//
// Author             : Nils Schwabe
// Date of creation   : Feb 94 ns
// Last modification  : 16 Nov 95 ns
//
// University of Bremen, Germany
//

#ifndef __BASIC_H__
#include "basic.h"
#endif

// ------ Point ---------------------------------------------------------------

//       A point within the n-dimensional euclidian space.

const cntEffDim=3;
          // number of dimensions that are handled 'efficiently', must be > 0.

const Coordinate pEpsilon = 0.03; // used in comparisons

class Point
{
public:
    Point (ushort dimensions);

        // Constructs n-D point with all coordinates = 0.

    Point (Coordinate x, Coordinate y);

        // Constructs 2-D point with specified coordinates.

    Point (const Point &);

        // Copies a point.

    void operator=(const Point &);

        // Assigns another point of equal dimension.
```

```
    ~Point ();

    Coordinate Get (uint index) const;

        // Gets the coordinate value for dimension 'index' (0..n).

    void Set (uint index, Coordinate newX);

        // Sets the coordinate value for dimension 'index' to 'newX'.

    ushort GetDim () const                                    {return dim;}

        // Returns the dimension of the point.

    int operator==(const Point &) const;

        // Returns non-zero if both points are equal.

    int operator!=(const Point &) const;

        // Returns non-zero if both points are not equal.

    void TestOut (const char *title) const;
    void PSOut () const;

private:
    short dim;
    Coordinate x[cntEffDim];
    Coordinate *_x;
};

ostream& operator<< (ostream &s, const Point& p);


// ------- PointSequence -----------------------------------------------------

//          A sequence (list) of points.
//          The points in the list have their own identity, i.e. they are
//          _not_ stored as references or pointers.

class BoundingBox;

typedef SList<Point> Pointlist;

class PointSequence : public Pointlist
{
public:
    PointSequence () : Pointlist() {}
    PointSequence (const Pointlist& copy) : Pointlist(copy) {}
    void PSOut () const;
    void CalcBoundingBox (BoundingBox &bb) const;

    Boolean CyclicIdentical (const PointSequence &cmp) const;
};

ostream& operator<< (ostream &s, const PointSequence& ps);


#ifndef __BOUNDBOX_H__
#include "boundbox.h"
#endif


#endif /* !defined(__POINT_H__) */
```

3.3.2 matrix.h

```
#ifndef __MATRIX_H__
#define __MATRIX_H__
//
// matrix.h  --  Collage-Two
//
// Matrix class
//
// Author            : Nils Schwabe
// Date of creation  : Feb 94 ns
// Last modification : 20 Nov 95 ns
//
// University of Bremen, Germany
//

#ifndef __BASIC_H__
#include "basic.h"
#endif

#ifndef __POINT_H__
#include "point.h"
#endif

typedef Coordinate ME;    // Type of a matrix element

#include <math.h>
const ME mEpsilon = 0.01; // Intervall in which an element is treated as 0
                          // in some transformation tests.

class Matrix
{
public:

    Matrix (uint cntRows, uint cntColumns); // construct nxm matrix
    Matrix (ME,ME,ME, ME,ME,ME, ME,ME,ME);  // construct and init 3x3 matrix
    virtual ~Matrix();
    Matrix (const Matrix &);
    void operator=(const Matrix &);

    ME Get (uint rw, uint cl) const {return e[rw*_columns+cl];}
    void Set (uint rw, uint cl, ME value) {e[rw*_columns+cl] = value;}

    uint GetRows() const {return rows;}
    uint GetColumns() const {return columns;}
    Boolean Resize (uint cntRows, uint cntColumns);

    void Multiply (const Matrix &v, Matrix *result) const;
    void Multiply (const Point &p, Point *result) const;
    Boolean Invert ();
    void TestOut (char *header);

private:
    ME *e;
    uint rows, columns;
    uint _rows, _columns;
    void PivotStep (uint,uint);
    Boolean FindPivotColumns (uint, signed char *pcol);
};

#endif /* !defined(__MATRIX_H__) */
```

3.3.3 trans.h

```
#ifndef __TRANS_H__
#define __TRANS_H__
//
// trans.h  --  Collage-Two
//
// Transformation classes
//
// Author           : Nils Schwabe
// Date of creation  : Feb 94 ns
// Last modification : 20 Nov 95 ns
//
// University of Bremen, Germany
//

#ifndef __MATRIX_H__
#include "matrix.h"
#endif

class Transformation
//---------------------------------------------------------------------------
// A virtual base class that declares the transformation function that all
// transformation classes must support.
{
public:
    typedef enum
    {
        transMatrix,       // matrix multiplication used
        transTranslation   // translation transformation

    } TransInternal;

    virtual Transformation* Copy () = 0;

        // Virtual copy constructor.

    virtual ~Transformation() {}

    virtual void Transform (const Point &p, Point *res) const = 0;

        // Applies the transformation to a point: res := t(p).
        // Note: 'p' and 'res' may be the same variables.

    virtual Transformation *Chain (const Transformation *outer) const
                                   {return 0;}

        // Returns a pointer to a new transformation that represents
        // outer(this) or NULL if no such representation could be generated.

    virtual TransInternal InternalType () const = 0;

        // Returns the internal implementation type of the transformation.
};

class TransFinder
//---------------------------------------------------------------------------
// A virtual base class that declares a function that finds a transformation
// given two point lists such that the transformation maps the first point
// sequence to the second.
{
public:
    virtual ~TransFinder() {}
    virtual TransFinder* Copy () const = 0; // virtual copy constructor

    virtual Transformation *FindTrans (const PointSequence &pSeqSource,
                                       const PointSequence &pSeqTarget)
        const = 0;
```

```
    // Answer a pointer to a newly created instance of an appropriate
    // transformation or NULL if no such transformation could be found.

    virtual const char *GetName() const = 0;
    // Answer the (printable) name of the finder.
};

// Common transformation classes:
// ==============================

class MatrixTrans : public Matrix, public Transformation
//----------------------------------------------------------------------------
// A transformation based on a matrix that transforms points (i.e. vectors)
// by multiplying the matrix with the vector.
{
public:
    MatrixTrans (ushort r, ushort c) : Matrix(r,c) {}
    MatrixTrans (const MatrixTrans &copy) : Matrix(copy) {}
    Transformation *Copy () {return new MatrixTrans(*this);}
    void Transform (const Point &p, Point *res) const;
    // res := Matrix * p

    Transformation *Chain (const Transformation *outer) const;
    TransInternal InternalType () const {return transMatrix;}
};


class TranslationTrans : public Transformation
//----------------------------------------------------------------------------
// A transformation that adds a constant vector to a point.
{
            ...
};

// Common transformation finders:
// ==============================

class AffineFinder : public TransFinder
//----------------------------------------------------------------------------
// Supports affine transformations.
{
public:
    AffineFinder (ushort dimensions) {dim = dimensions;}
    ~AffineFinder () {}
    Transformation *FindTrans (const PointSequence &pSeqSource,
                               const PointSequence &pSeqTarget) const;
    TransFinder *Copy () const
        {TransFinder *n = new AffineFinder(dim); return n;}

    const char *GetName() const {return "Affine";}
private:
    ushort dim;
};

class TranslationFinder : public TransFinder
//----------------------------------------------------------------------------
// Supports translations.
{
            ...
};

class ScaleTranslationFinder : public TransFinder
//----------------------------------------------------------------------------
// Supports translations and scalings.
{
            ...
};

#endif /* !defined(__TRANS_H__) */
```

3.3.4 hyper.h

```
#ifndef __HYPER_H__
#define __HYPER_H__
//
// hyper.h  --  CollageTwo
//
// Classes for representing the hyperedges of a collage
//
// Authors           : Nils Schwabe
// Date of creation  : 06 Apr 94 ns
// Last modification : 16 Nov 95 ns
//
// University of Bremen, Germany
//

#ifndef __TRANS_H__
#include "trans.h"
#endif

// ------- Edge ---------------------------------------------------------------

typedef SString<char> EdgeText;
    // External label (input from and visible to the user)

typedef int EdgeClass;
    // Internal label (= equivalence class) that is used for assigning
    // rules to edges.

class Edge
{
public:
    Edge (const EdgeText& initLabel);
        // Construct edge with an initial label.

    Edge (const Edge& theEdge);
        // Construct edge from another edge.

    virtual ~Edge ()                                                        {}
        // Provides a virtual destructor.

    virtual Edge* Copy() = 0;
        // Virtual copy constructor.

    Edge &operator= (const Edge& theEdge);
        // Assignment.

    virtual int operator== (const Edge& compare) const;
        // Comparison. Returns nonzero value if both edges are equal.

    void SetLabel (const EdgeText& theLabel);
    const EdgeText& GetLabel() const                          {return label;}
        // Set or get the label of the edge.

    void SetClass (EdgeClass ec)                              {eqclass = ec;}
    EdgeClass GetClass () const                             {return eqclass;}
        // Set or get the class of the edge.

    virtual void Transform (const Transformation &T) = 0;
        // Transform the edge with transformation 'T'.

    virtual Transformation *FindTrans
        (const PointSequence *s, const TransFinder *finder) = 0;
        // Return a pointer to a transformation that maps 's' to the edge
        // using the 'finder' class or NULL if no such transformation exists.

    virtual void FreeTrans (Transformation *t) = 0;
        // Assuming that 't' is a transformation obtained by FindTrans(),
        // free this transformation if it had to be allocated.
```

```
    // Optional features that may or may not be supported by an edge:

    virtual void Draw () const                                            {}
    virtual void PSOut () const                                           {}
    virtual void CalcBoundingBox (BoundingBox &bb) const                  {}

private:
    EdgeText label;
    EdgeClass eqclass;
};

// ------- Hyperedge ---------------------------------------------------------

// A 'Hyperedge' is a labeled PointSequence. The points can be transformed and
// output. The Transform() method of this edge type applies the transformation
// to the point sequence.

typedef PointSequence AttachSequence;

class Hyperedge : public Edge, public AttachSequence
{
public:
    <...>
    virtual int operator== (const Edge& compare) const;

    void Transform (const Transformation &T);

    Transformation *FindTrans
        (const PointSequence *s, const TransFinder *finder);
    void FreeTrans (Transformation *t);
    <...>
private:
    ColorIndex color;
};

ostream& operator<< (ostream& s, const Hyperedge &h);

// ------- AnnotatedEdge -----------------------------------------------------

// An AnnotatedEdge is a labeled transformation. The Transform() method
// of this edge type replaces the stored transformation t such that it becomes
// t(T), where T is the parameter of Transform().

class AnnotatedEdge : public Edge
{
    <...>
};

// ------- EdgeSet -----------------------------------------------------------

// An EdgeSet is a container for edges with restricted access.

typedef SList<Edge *> EdgePList;

class EdgeSet : private EdgePList
{
public:
    <...>
    virtual void Include (Edge *newEdge);       // pointer is stored
    virtual void Exclude (Edge* theEdge);       // edge is also destroyed
    virtual void Add (const EdgeSet &theSet);   // makes deep copy

    Boolean Contains (const Edge &edge) const;  // uses operator==
    <...>
    void Transform (const Transformation &T);
    <...>
};
#endif /* !defined(__HYPER_H__) */
```

3.3.5 collage.h

```
#ifndef __COLLAGE_H__
#define __COLLAGE_H__
//
// collage.h  --  CollageTwo
//
// Classes for collages, rules and derivation bases
//
// Authors           : Nils Schwabe, Stefan Taubenberger
// Date of creation   : 19 Apr 94 ns
// Last modification  : 12 Feb 96 ns
//
// University of Bremen, Germany
//

#include "parts.h"
#include "hyper.h"
#include "trans.h"

// ------- PinSequence ------------------------------------------------------

// A PinSequence is a PointSequence that has additional Draw() and Transform()
// facilities and is used to represent the pin-points of a collage.

class PinSequence : public PointSequence  // (PointSequence defined in point.h)
{
public:
    <...>

    void Draw() const;
    void Transform (const Transformation &T);
};

// ------- Collage ----------------------------------------------------------

// Objects of this class represent both decorated and terminal collages.

class Base; // forward

class Collage
{
public:
    Collage (PinSequence *initPins,
             PartSet *initParts,
             EdgeSet *initEdges);
    Collage ();
    Collage (const Collage &theCollage);
    ~Collage ();

    Collage &operator= (const Collage &theCollage);

    void AppendPin (const Point &thePoint);
        // Appends 'thePoint' to the list of pin-points.

    const PinSequence *GetPins() {return pins;}
        // Returns a pointer to the pin-points sequence.

    void InsertPart (const Part &thePart);
        // Inserts 'thePart' into the collage. The part is copied.

    void InsertEdge (Edge *theEdge);
        // Inserts 'theEdge' into the collage. The pointer is stored.

    EdgeSet *GetEdgeSet() {return edges;}
        // Returns a pointer to the edge sequence.

    void SetEdgeSet (EdgeSet *set) {edges = set;}
        // Sets the edge sequence. The pointer 'set' is stored.
```

```
    void Draw () const;
        // Draws the collage into the current screen window.

    void PSOut (Textblock *t1, Textblock *t2, Textblock *t3) const;
        // Outputs the collage to the current PostScript context.

    void Transform (const Transformation &T);
        // Applies a transformation to the collage.

    void ReplEdge (Edge *edge,
                   const Collage &repl,
                   Transformation *t,
                   ColorIndex colorOffset
#if USETCL
                   ,Tcl_Interp *interp
#endif
                   );

    // Replaces 'edge' with 'repl' which will be transformed using 't'
    // which must be a transformation obtained by edge->FindTrans().
    // 'edge' and 't' are destroyed, 'repl' remains untouched.
    // The color indices of all new parts (and hyperedges, if any)
    // are set to the value of their source (from 'repl) plus the
    // value of 'colorOffset'.

    void DeriveWithBase (const Base &theBase,
                         ColorIndex colorOffset
#if USETCL
                         ,Tcl_Interp *interp
#endif
                         );

    // Carries out all edge replacements described in 'theBase'.

    void MakeBoundingBox (uint dim);
    const BoundingBox *GetBoundingBox() const {return bb;}

    Boolean FindContext (Collage &context, Boolean anyOf);

    // Returns True if 'context' is contained within this collage.
    // If 'anyOf' is True then True is returned if at least one element
    // (part or edge) is contained within this collage.

private:
    void Add (const Collage &theCollage);
    // Add the parts and edges of theCollage to this collage.
    // The pin points are ignored.

    Boolean FindParts (Collage &context, Boolean anyOf);
    Boolean FindEdges (Collage &context, Boolean anyOf);

    PartSet       *parts;   // PART
    PinSequence   *pins;    // pin
    EdgeSet       *edges;   // EDGE
    BoundingBox   *bb;
};

// ------- CollageRule(Set) ---------------------------------------------------

// Classes representing rules and sets of rules of a collage grammar.

class CollageRule
{
public:
    CollageRule (const EdgeText& left,
                 Collage *       right,
                 Collage *       context,
                 Collage *       ncontext,
                 Boolean         cAnyOf,
```

```
                 Boolean         ncAnyOf,
                 TransFinder *   transFinder,
                 int             table = 0,
                 double          prob = 1);
    ~CollageRule ();

    <...>    <...>    <...>

    void Draw () const;
    void PSOut (Textblock *first, Textblock *second) const;

    <...>

    Transformation *FindTrans (const PointSequence *source,
                               const PointSequence *target)
        {return transFinder->FindTrans (*source, *target);}

    <...>
private:
    EdgeText    leftHandText;   // external label
    EdgeClass   leftHandClass;  // internal label (equivalence class)
    Collage *   rightHandSide;  // right hand side collage
    Collage *   context;        // context collage (optional, may be NULL)
    Collage *   ncontext;       // negative context (dto.)
    Boolean     cAnyOf, ncAnyOf;// "any of" flag for both contexts
    int         tableNumber;    // table number
    double      prob;           // probability, 0.0 <= prob <= 1.0
    TransFinder *transFinder;   // allowed transformations
};

ostream& operator<< (ostream& s, const CollageRule &cr);

typedef SList<CollageRule *> CollageRulePList;

class CollageRuleSet : public CollageRulePList
{
public:
    void Insert (CollageRule *newElement);
        // Inserts a new rule into the collage rule set.

    void Draw () const;
    void PSOut (Textblock *first, Textblock *second) const;

    void Normalize (Boolean transRightSide);
        // Normalizes the rules, i.e. assigns equivalence classes to
        // every rule and every edge in the right-sides.
        // If 'transRightSide'==True then transforms all right-sides
        // of one equivalence class to identical pin-points.

    void ClassifyEdges (Collage *collage);
        // Assigns equivalence classes to all edges of 'collage'.
        // This is a part of Normalize() for all right-sides but must
        // be called explicitly for the start collage (see below).

    void AnnotateEdges (Collage *collage);
        // Assuming that all edges in 'collage' are 'Hyperedge's,
        // replaces them by corresponding 'AnnotatedEdge's. The
        // rules and edges have to be normalized and classified
        // for this to work correctly.

private:
    void ClassifyEdge (Hyperedge *edge);
};

//----- TableInfo ---------------------------------------------------------

// Contains settings information for a derivation table.
// 'Table'       : the number of the table
// 'Sequential'  : use sequential derivations in this table (edge by edge)
// 'ExhaustProb' : Exhausting probabilties mode (see documentation)
```

```
class TableInfo
{
public:
    <...>
private:
    unsigned int table;          // number of the table
    Boolean      sequential;     // sequential dervation
    Boolean      exhaustProb;    // 'exhausted probabilities' mode
};

class TableInfoList : public SList<TableInfo *>
{
public:
    ~TableInfoList ();
    TableInfo *Find (int table);
};

// ------- Base ---------------------------------------------------------------

// Derivation base.

class BaseElement
{
public:
    <...>
    Edge           *target;  // This edge should be replaced...
    Collage        *source;  // ...by this collage...
    Transformation *trans;   // ...using this transformation.
};

typedef SList<BaseElement> BaseElementList;

class Base : public BaseElementList
{
public:
    Boolean Build (Collage* theCollage, CollageRuleSet* theRules,
                   TableInfo *tableInfo);
private:
    void Insert (Edge *edge, Collage *repl, Transformation *t)
    <...>

    double GetNormRandom ();
        // Returns a normalized pseudo-random number ranging from
        // 0.0 to 1.0.

    Boolean ContextOk  (Collage *theCollage, CollageRule *curRule,
                        Transformation *trans);
        // Checks the context conditions.

    Boolean GetEdgeRuleNormal (Collage *theCollage, CollageRuleSet *theRules,
                               TableInfo *tableInfo, Edge *curEdge,
                    /* out: */ CollageRule *&curRule, Transformation *&trans);
        // Selects a rule using normal probabilities mode.

    Boolean GetEdgeRuleExhaust (Collage *theCollage, CollageRuleSet *theRules,
                               TableInfo *tableInfo, Edge *curEdge,
                    /* out: */ CollageRule *&curRule, Transformation *&trans);
        // Selects a rule using exhausting probabilities mode.
};

// ------- CollageGrammar -----------------------------------------------------

class CollageGrammar
{
public:
    <...>

    void InsertRule (CollageRule* rule) {rules->Insert (rule); }
```

```
    CollageRuleSet *GetRules() const {return rules;}

    void SetStart (Collage* s) {start = s;}
    Collage *GetStart () const {return start;}

    void PSOut () const;
    Textblock *psbGlobal;
    Textblock *psbGrammar;
    Textblock *psbStart;

    void Normalize (Boolean transRightSide);
        // Normalizes the collage grammar.

    void Annotate ();
        // Annotates the collage grammar (assuming an already normalized
        // grammar).

private:
    Collage        *start; // The start collage
    CollageRuleSet  *rules; // The productions of the grammar
};

// ------- Table --------------------------------------------------------------

// The following classes are used to store the derivation specification, i.e.
// which table to use in which derivation step.

class TableElemEntry
{
public:
    <...>
    unsigned int times, table;
};

class TableElement
{
public:
    <...>    <...>

    unsigned int          until;
    SList<TableElemEntry> entries;
    unsigned int          actEntry; // index in 'entries'
    unsigned int          actTimes; // 0 <= actTimes <= 'times' of actual entry
    Textblock             *psbTableElem;
};

typedef SList<TableElement *> TableElementList;

class Table : public TableElementList
{
public:
    <...>
    Boolean Insert (TableElement *te);
        // Inserts a new table element, if not already defined.
        // Answers True if successful (element did not exist).

    TableElement *NextTable (int until, int *table);
        // Gets the next table number and associated table element
        // with exact matched 'until' value. Returns NULL if no
        // such table element exists.

    Boolean GetMaxUntil (int *maxuntil) const;
    <...>
};

#endif /* !defined(__COLLAGE_H__) */
```

REALITY.SYS not found. Rebooting universe.