部品合成による漢字スケルトンフォントの作成

田中哲朗

(1)

部品合成による
漢字スケルトンフォントの作成

田中哲朗

目次

第 1 章

はじめに

1.1 研究の目的

ワードプロセッサの普及, 清書システムの実用化によって印刷機としての計算機の使用がますます増えてきた. 当初は編集可能なタイプライタとして使われるだけだったが, 出力装置の発達と共に, より高品質の文書の作成が行なわれるようになった. そのため, 多種類の書体をさまざまな大きさで使いたいという要求が生じている.

英文用のフォントに関してはこの要求はかなり応えられている. アウトラインフォントのフォーマットが何種類か提案され, それぞれのフォーマットに対応した多くの書体が無料で提供されている. 更に, 個人用のフォントを作成するためのツールもいくつか作られている. このように英文用のフォントの扱いが容易なのは, 1 種類のフォントセットに含まれる文字がアルファベットの大文字小文字, 記号合わせて 100 文字以下しかないためである.

一方, 和文用のフォントを定義する場合, 仮名文字だけならばアルファベットとそう変わらないが, 漢字も含めると一般的に使われる JIS X0208 第 1● 第 2 水準だけでも 6000 余りの文字をデザインする必要がある. そのため, 漢字フォントのデザインは, もっぱら専門家の手にまかされ, 一般ユーザは, 彼らの製品を買って使うだけの状態になっている. また, JIS で定義されていない字を使うには, ユーザが, 使っている書体の他の文字とのバランスを崩さないように文字を定義する必要があるが, これも現状では難しい.

このような現状を打開するために, 漢字フォントの表現法としてスケルトン形式を導入し, ユーザが最小限の手間をかけるだけで新たな書体を作成したり, 文字を定義することができるようなシステムを目指した研究をしてきた. この研究は, 肉付けアルゴ

リズム, 組合せアルゴリズム, 漢字組合せデータベースの作成, ひらがな, カタカナフォントの作成, 毛筆書体, ユーザによるフォント開発のためのツール作りなど, 漢字スケルトンフォントを実用化するための種々の分野にわたっている. 本論文に述べるように, この研究は実用的レベルに達していると考えられる.

1.2 本論文の構成

第2章では, 計算機で扱うフォント全般についての話題, および漢字スケルトンフォントに関する従来の研究について紹介する.

第3章では, 試作したシステムの設計方針や構成について述べる.

第4章ではスケルトンフォントの特徴を生かすための自由度の大きい肉付けの実現について論じ, 明朝体の肉付けにその自由度の大きさをどのように生かしたかを説明する.

第5章ではシステムの最大の特徴である部品組合せの実現について述べる.

第6章ではユーザが書体を作成する際に必要な事項を述べ, ゴシック体, 丸ゴシック体の定義にどの程度の作業が必要になるかを説明する.

第7章ではひらがな, カタカナのスケルトンフォントの実現について述べる.

第8章では, アウトラインからドットイメージに変換する際の量子化の問題について考察し, それらを考慮に入れたアウトラインの生成法について述べる.

第9章では, フォントを利用するために採用したフォントサーバについて説明する.

第10章では, フォントの肉付けアルゴリズム, 組合せアルゴリズムをチェックしたり, データを入力したりするための開発環境について述べる.

第11章では, 本研究の成果をまとめ, 残された課題を列挙する.

第 2 章

フォント関連技術の現状

2.1 フォントの表現法

計算機で現在広く使われているフォントの表現法を大きく分けると，次の 3 種類に分類できる．

- ドットイメージ

  図 2.1のように，2 次元の格子上の正方形領域に分け，それらの黒白によって，フォントを表現する方法．濃淡出力が可能な出力装置用に，1 ドットを複数のビットで表現する場合もある．

- アウトライン

  図 2.2のように，黒白領域の境の曲線によってフォントを表現する方法．

- スケルトン

  図 2.3のように，ストローク単位で文字の骨格を与え，それに肉付けをする方法．

それぞれの特徴，および関連する研究を次に示す．

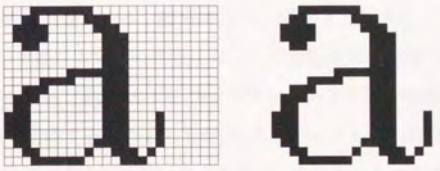


図 2.1: ドットイメージフォント

図 2.2: アウトラインフォント

図 2.3: スケルトンフォント

2.1.1 ドットイメージフォント

大部分の出力装置がラスタ型なので，出力装置と解像度が一致する場合はドットイメージフォントは高速な出力が可能になる．

出力装置と解像度が一致しない時は，拡大 / 縮小が必要になる．ドットイメージを拡大 / 縮小する一般的な手法を用いると線の太さが均一にならないことや，線が消えてしまうことがあるので，フォントの拡大 / 縮小では工夫が必要になる．[1] では，24×24 ドットの明朝体フォントを，書体に関する知識を利用して前処理を行なった後で，拡大 / 縮小を実行する方法を取っている．

ただ，このような工夫は，線幅が 1 ドット程度のフォントにしか適用できなかったり，拡大 / 縮小の範囲が限られているという制限があるし，また，制限をクリアしても，はじめからその大きさでデザインされたものと比較すると美しくない．

美しい出力を得るには文字のサイズに応じたフォントを用意する必要があり，記憶容量と，デザインの手間が増大する．縦 m ドット，横 n ドットのドットイメージフォントを定義するには，単純に 1 ピクセルを 1 ビットに対応させる方法では，$m \times n$ ビットの記憶容量が必要となる．TEX の PK フォーマットのように 2 値画像の圧縮手法を用いて圧縮した状態で記憶するものもあるが，大きなフォントや，高解像度の出力装置用のフォントの記憶容量が大きいことはドットイメージフォントの弱点になっている．

2.1.2 アウトラインフォント

低解像度の出力装置しかなく，とりあえず認識可能なフォントだけが求められている段階ではドットイメージフォントで十分であった．しかし，前述のように高解像度の出力装置では，ドットイメージフォントは記憶容量の点で問題がある．またデザイン上の観点からデザインの材料として文字を使いたいという要求が出てきて，サイズの変更ができるアウトラインフォントが注目されるようになった．

ここ数年でアウトラインフォントはさまざまなシステムで使われるようになった．アウトラインフォントの優れた点が認められたためだが，マイクロプロセッサの普及によって，ドットイメージへの変換に要する計算コストがそれほど問題にならなくなったことと，ページ記述言語の事実上の標準となった PostScript に採用されたことなど，外的要因も味方している．

アウトラインフォントの製作には，いくつかの方法がある．まず考えられるのが，計算機上のエディタを用いて人手でフォントデザインを行なう方法である．紙の上でデザインするのと比較すると，編集が容易になるかもしれないが，数千文字すべてについてデザインする必要があるので，作業量は膨大なものになる．

フォントデザインの手間を軽減するために，従来のフォントをドットイメージで取り込み，計算機で特徴点を抽出して曲線補間を行なおうという研究も，古くからなされている [2, 3]．ただ，計算機による曲線あてはめだけで十分な結果が得られることは少なく，人手による修正を行なうことが多い．

アウトラインフォントの特徴はサイズ変更が容易であることだが，縮小して出力装置の解像度が線の太さと同程度になると，量子化によって見苦しくなることがある．これを避けるためにヒント情報をつけて対処する必要がある．ヒント情報の付加はデザイナが行なうのが普通だが，これを計算機にやらせようという研究もなされている [4]．

アウトラインフォントの弱点は，太さを変更したフォントの生成が容易でないことだが，ゴシック体に関しては，同じファミリーの太さの異なるフォントの補間により任意の太さのフォントを得る研究がなされている [5]．

2.1.3 スケルトンフォント

スケルトンフォントは拡大 / 縮小が可能であるというアウトラインフォントと共通した性質を持っているが，更に，文字の太さを変更することが可能になっている．その

ため，アウトラインフォントにおける複数の書体を一つの書体で表現することができる．また，アウトラインフォントよりも少ない点の指定でフォントを定義できるので，記憶容量を大幅に減らすことができる．

スケルトンフォントの発想はそう新しいものではない．ラスタ型出力装置ではなくベクトル型出力装置（プロッタ等）では，フォントを線分の集合として定義するが，これもスケルトンフォントの一種である．また，Courier のようにストロークの太さが均一な書体を，スケルトン形式で定義している PostScript のバージョンもある．

ただ，これらはすべて，中心線に対して均一の太さで塗りつぶすという単純な肉付けを用いているので，適用できる書体は限られている．

2.2 漢字フォント

文字種が英文フォントと比較して桁違いに大きいため，漢字フォントの発達は遅れた．また漢字コードが JIS の規格として定まってから，約 15 年しかたっていない．

ここ十年ほどで日本語ワードプロセッサが発達したが，英文と違ってタイプライタが普及していなかった和文では，印字品質をタイプライタと比較されることがなく，当初は 16×16 ドット，あるいは 24×24 ドットのドットイメージフォントのフォントを 1 種類だけ用意する製品が多かった．

このような製品では，レイアウト上の要請から大きい文字を使う必要がある時は，元のドットイメージフォントを縦横方向に整数倍に拡大したいわゆる倍角文字を用いる．角が目立たないように近隣のドットから補間する処理を行なうと単純に格子を整数倍したものほど汚くないが，出力装置の解像度が増すと対応できない．

そのような中で，漢字アウトラインフォントも次第に使われるようになってきた．アウトラインフォントの特徴は，フォントの記憶容量が小さく，さまざまな書体を持つことができることだが，フォントの製作コストがかかるために，商品化されている書体はまだまだ少なく，また高価であるため，多種類のフォントを使える環境はまだ身近なものにはなっていない．

このように，漢字フォントの作成ではデザインの手間が問題になるので，太さの変更が可能なスケルトンフォントは漢字と相性がいいと考えられる．ここ数年，活字書体として主に使われる明朝体やゴシック体の漢字スケルトンフォントを作る研究がいくつか行なわれている．これらの研究を分類すると次のようになる．

- エレメント方式と非エレメント方式

  スケルトンフォントでは，骨格情報をストロークとその連結情報によって表すが，ストロークに縦棒，横棒といった情報を付加して定義するのがエレメント方式で，付加しないのが非エレメント方式である．

  複雑な肉付けを行なう明朝体では，エレメント方式が自然であるし，ゴシック体でもエレメント方式を使う場合が多いが，[6] では，非エレメント方式を用いている．

- 肉付けの方法

  非エレメント方式である [6] は，対象をゴシック体に限っているため，中心線上にある制御点を曲線で結んで太さ一定の肉付けを行なっている．

  一方，[7] では同じゴシック体を扱っているが，エレメントの種類によって，制御点付近での太さや，飾りを指定できるようになっている．この飾りの形はエレメントの種類に応じてあらかじめ決められている．

  明朝体を扱った [8, 9] では，始点終点の形を変形することによって，飾りの形を作っている．

- エレメントの種類

  エレメントの種類が多いと，デザインの際にエレメントの選択に関してデザイナの負担が大きくなる．[10] のように 150 種のエレメントを使った例もあるが，一般には数十種程度である．[9] ではデザイナの負担を減らすために，デザインの時点では 21 種で入力し，内部では座標データから 33 種に分類して肉付けをするという工夫がなされている．

活字書体でない毛筆書体についても，ストロークに分解して，計算機で扱うという研究が [12] で行なわれている．

漢字フォントではないが，ひらがな，カタカナをスケルトン形式で表現する研究が [11] でなされている．非エレメント方式を採用し，中心線を定義するいくつかの特徴点と，その点における太さでフォントを表現している．第 7章に述べるひらがな，カタカナのフォント表現方式は，おおよそ [11] に従っている．

第 3 章

研究の方針

第2章で述べた漢字スケルトンフォントに関する従来の研究の多くは，試験的にいくつかの文字を作っただけで終わったものが多く，実用的なフォントを作成する場合に遭遇する種々の問題は避けて通っていた．

本研究では，実用的なシステムを構築することを目指して，以下の方針で臨むことにした．

- 書体の種類

  漢字の書体には，はじめから活字用にデザインされた明朝体，ゴシック体，宋朝体のほかに，楷書体，行書体，隷書体，相撲文字などの毛筆書体がある．本研究では活字書体だけを扱う．

- 表現方法

  第2章で述べたように，漢字スケルトンフォントの表現方法としては，エレメント方式と非エレメント方式がある．本研究では，明朝体のような複雑な肉付けが必要な書体を考慮してエレメント方式を採用する．

- 部品組合せ

  漢字文字数が他の言語と比較して桁違いに多いのは，偏や旁を組み合わせて新たな文字を作成する漢字の造字能力による．偏や旁などの基本的な部品の数は漢字の数よりずっと少ないので，デザインの手間を軽減し記憶容量を減らすために，基本的な部品だけを定義しておいて，漢字をそれらの組合せとして表現する．

  その際，各部品の大きさを固定して組み合わせるのではなくて，アルゴリズムに

よって各部品を縦横方向に拡大 / 縮小して組み合わせる．スケルトンフォントを採用するので，部品の大きさを変えても同じ太さで肉付けすることができる．

方針のうちで，もっとも特徴的なのが部品組合せによって漢字を生成する点である．明治時代に偏と旁を別々の活字として，組み合わせて使うことによって活字の数を節約しようという試みが行なわれたが，すぐにすたれたという話がある．偏や旁別に活字を拾う手間がかかるという理由もあったが，偏や旁の幅が固定されているので，でき上がった文字が不自然であったからだという．

本研究では，部品を組合せの種類や組み合わせる相手によって拡大 / 縮小してから組み合わせるので，この点は改良されている．

3.1 漢字の階層化

本研究のシステムでは，漢字を次のように階層化して扱う (図 3.1)．

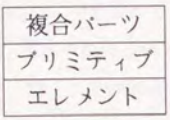


図 3.1: 漢字の階層化

- エレメント

  縦棒，横棒，左払い，右払いなどスケルトンフォントの最小単位のことである (図 3.2)．エレメントのタイプと制御点の座標によって定義する．

図 3.2: エレメントの例

- プリミティブ

  エレメントの集合に，エレメント間の連結情報を加えたものである (図 3.3)．エレメントの種類，制御点の座標データの他に組み合わせの際のヒントも定義することができる．

図 3.3: プリミティブの例

- 複合パーツ

  プリミティブや複合パーツ (この二つを合わせて部品と呼ぶ) を組み合わせたものである (図 3.4)．組合せは，部品の制御点にアフィン変換を施すことによって行なうが，例外的に「発」や「祭」の下の部品のようにアフィン変換以外の変形を施すこともある．

図 3.4: 複合パーツの例

「亜」のようにプリミティブとして定義される漢字もあるし，「挨」のように複合パーツとして定義される漢字もある．すべての漢字をプリミティブとして扱えば，漢字のデザイン上の自由度は上がるが，デザインに要するコストは大きくなる．

本研究ではデザインのコストを重視し，なるべくプリミティブを少なくするという方針で漢字の部品への分解を行なった．その結果，JIS 第 1 第 2 水準の漢字すべてを定

表 3.1: 階層化の効果

	非階層化	階層化
総プリミティブ数	6355	956
総エレメント数	87246	5861
平均エレメント数	13.7	6.1
平均部品数	1	3.01

義するのに必要なプリミティブを 956 個におさえることができた. JIS 第 1 第 2 水準の漢字 6355 文字のうち漢字自体がプリミティブになっているものは 360 文字だけで, 残りは複合パーツとして定義する.

階層化によるデザインのコストの節約を表現したのが, 表 3.1である. デザインの手間は, 総エレメント数にほぼ比例するので, 階層化によってデザインのコストを大幅に節約することができたことがわかる.

3.2 使用言語

システムは UtiLisp/C([19, 20, 21]) で記述した. UtiLisp を使ったのは, 以下の理由による.

1. 対話型言語なので, プログラムの変更が容易

   プログラムを少しずつ変えて実験をする必要があるので, C のようにコンパイルに時間のかかる言語は不利である.

2. データ構造の変更が容易

   アルゴリズムの変更などで, 扱うデータ構造を変更することが頻繁にあるが, すべてのデータが S 式という扱いやすい形をしているので, データの変更プログラムが簡単に書ける.

3. システムの中身が分かっている.

   内部に手を入れることのできるので, X-Window を扱う関数を書くというような言語とのインタフェースをとるのも容易であり, 必要な機能を実現するために

ヒープの自動拡張の機能を取り入れたりというUtiLisp/C自体の改造で対応することもできる.

C言語では最初から用意されているX-windowとのインタフェースの制作など,余分な作業が必要になったが,環境が整ってくると,1の特徴が生かされ,思った以上に能率が上がった.更に,UtiLisp/C自体のバグ取りや,機能拡張を促すという効果もあった.

3.3 システム全体の構成

システムは次のような部分から構成されている(図3.5).

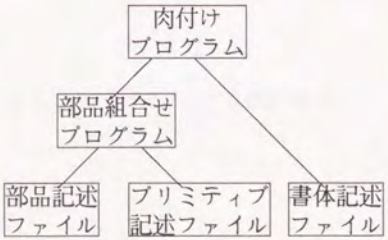


図3.5: システムの構成

- プリミティブ記述ファイル

  プリミティブを,含まれるエレメントの種類と接続関係,制御点の座標,組合せの際のヒントとなる属性の形で定義する.

- 複合パーツ記述ファイル

  複合パーツを,含まれる部品と組合せの種類の形で定義する.

- 部品組合せプログラム

  組合せの種類の種類ごとに,組み合わせる部品を与えて,組み合わせた結果を得る関数を定義する.

- 書体記述ファイル

  各書体ごとに別々のファイルになっている．エレメントと飾りの肉付けアルゴリズムを記述する．書体の定義は，モデルの書体に対して変更部分を定義することによって行なうので，モデル書体との違いが激しい場合はプリミティブや複合パーツを定義し直すこともある．

- 肉付けプログラム

  組合せ後の漢字やプリミティブに対して肉付けを行ない，結果のアウトラインを返す関数を定義する．

各部分の詳細については，次章以降で説明する．

3.4 データの作成

漢字を生成するためには以下のデータを入力する必要がある．

- 組合せデータ
- プリミティブデータ

漢字をドットイメージからパターン認識によってエレメントに分割する方法が [17] で提案されている．エレメント分割を完全に自動化することができれば，漢字の部品への分割もある程度自動化できる．

しかし，[17] では，384 × 384 ドットという高品質のドットイメージフォントを用いたにもかかわらず，認識率は 77% に留まっている．かなりの文字について人手による修正が必要となるので，自動化によるメリットがほとんど得られない．本研究では自動化はあきらめ，手作業による漢字の部品への分解を行なった．JIS 第 1 第 2 水準の漢字すべてについての組合せの記述には 20 人 × 日ほどかかった．

プリミティブのデータは，第 10章で説明する UtiLisp で書かれた X-Window 上のスケルトンエディタを使って入力した．一つのプリミティブの入力に要する時間は 2 ～ 3 分程度である．

第 4 章

肉付けアルゴリズム

前章で述べたように，本研究では人手でフォントをデザインすることを想定している．一般ユーザによるフォント作成もありうるので，フォントデザインのコストを節約することが重要な課題となる．

フォントデザインのコストを節約するために，第10章で説明するように対話的なエディタなどの道具をそろえる必要があるが，それと共にエレメントの種類や，エレメントの制御点の数をなるべく少なくすることも必要である．

明朝体のような書体では生成しようとするアウトラインは複雑な形をしている．少種類のエレメント，少数の制御点で，複雑なアウトラインを生成するには，自由度の大きい肉付けアルゴリズムを採用しなければならない．

4.1 従来の肉付けアルゴリズム

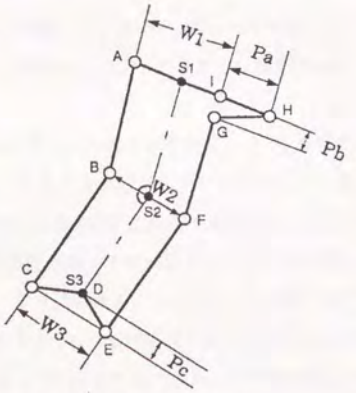
従来，漢字スケルトンフォントの試作に使われた肉付けアルゴリズムには以下の2種類がある．

1. 手本アウトラインの変形

   あらかじめ用意した手本アウトラインに座標変換をほどこすもの．[9] では明朝体でテストしている．

   座標変換による変形という自由度の小さい方法を使っているので，エレメントの長さや，制御点間の角度がある程度変化すると対応できない．そのため，エレメントの種類を増やさなくてはいけないという問題点がある．

   [9] では，全体で33種類のエレメントを使っているが，たとえば左払いなどは8種類用意されている．ただし，デザイナが入力する際には21種類のコードで入



1) ストローク種別：「左払い」
2) 骨格点：●
3) 形状情報：W1, W2, W3, Pa, Pb, Pc

図 4.1: 制御点付近での太さ指定 [7]

力して，計算機で細分することによってデザインの手間の増加を防いでいる．

2. 制御点付近での相対位置指定

図 4.1は [7] で採用しているエレメント指定法である．制御点 S1, S2, S3 付近での太さ W1, W2, W3, 及び S1 付近での形状を指定するパラメータ Pa, Pb と終点付近での形状を指定するパラメータ Pc から，輪郭特徴点(輪郭線が通る点)A-I を決定する．A-I を場所に応じて直線，Bezier 曲線，疑似スプライン曲線で結ぶことにより，アウトラインを得る．

実際に使っている方法は，この図ほど単純なものではなく，左払いではパラメータの数が 12 個に増えている．また，制御点の多いエレメントに関しては，制御点付近での太さではなく，制御点間の関係から補助的な制御点を求め，その付近での太さを指定している．

この方法は太さ変化の少ない書体に向いており，[7] で例示されたゴシック体フォントは実用上問題ない品質に達している．ただし，同じグループによる [8]

の明朝体フォントへの適用では，別の輪郭形状生成法を用いている．

この二つのアルゴリズムに共通しているのが，単一の単純なアルゴリズムを使い，理解しやすい少数のパラメータ指定によって，さまざまな種類のエレメントを表現する点である．

この特徴はエレメントのデザインの点では有利なのだが，自由度の点で問題があるため，アルゴリズム1のようにプリミティブのデザインのコストをあげることになったり，アルゴリズム2のように書体の変化に対応できなくなっていたりする．

肉付けアルゴリズムの自由度を高くする方法として，汎用で能力の高いアルゴリズムを用意し，多数のパラメータを調節することによって，必要な肉付けを得る方法が考えられる．しかし，どんな書体に対しても対応できるアルゴリズムを決定するのは難しく，複雑なアルゴリズムになって，指定するパラメータの数が増加するのも困る．そこで，本研究では各エレメントに対してパラメータを指定するかわりにプログラムを指定する方法をとった．

プログラムでフォントを表現すること自体はそれほど珍しいことではない．PostScript の組込みフォントに使われる TYPE I フォントフォーマットは，PostScript のごく小さなサブセットと対応したバイトコードでフォントを表現している [13]．大部分のフォントの記述はこれだけですむが，複雑な処理を行なうために，PostScript プログラムを呼ぶことができるようになっている．

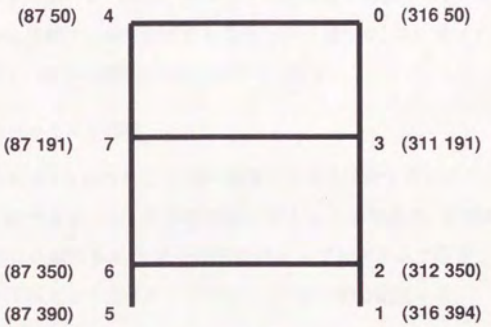
本研究では指定するプログラムは Lisp で書くが，肉付けプログラムで使うための関数を多数用意して，短いプログラムで複雑な肉付けを定義できるようにする．

4.2 肉付けの手順

複合パーツも，組み合わせた結果はプリミティブと同じ形をしているので，肉付けプログラムはプリミティブの肉付けだけを扱えばよい．プリミティブのデータ構造は以下の形をしている (図 4.2)．

1. 制御点の座標

   プリミティブ中のすべての制御点の X, Y 座標の対をリストとしたもの．X-window 上でエディットする都合上，X, Y 座標の範囲は 0 から 399 までになっている．




```
(defprimitive nil 日
   '(((316 50)(316 394)(312 350)(311 191) ; (1)
      (87 50)(87 390)(87 350)(87 191))
     ((tate (0 1) ; (2)
            (link 2 3)) ; (3)
      (tate (4 5) (link 6 7))
      (yoko (6 2))
      (yoko (7 3))
      (yoko (4 0)))
     (xunit . 200) ; (4)
))
```

図 4.2: プリミティブの表現

2. 各エレメントの種類と制御点との対応

   エレメントの種類は, tate, yoko といったシンボルで表す. エレメントは種類に応じていくつかの制御点を持つが, その点との対応を 1 の座標対リストの要素番号で表現する.

3. 各エレメントの属性リスト

   エレメントの肉付けの際にプログラムに渡される属性リスト. ただし, エレメントの途中に接続する点を記述する link という属性は, エレメントの肉付けの際には無視され, 飾りの肉付けの決定に用いられる.

4. 部品組合せのための情報

   部品組合せは, なるべく 1, 2, 3 の情報だけから決定するという方針ですすめる. 第5章で述べるように, 組合せの際に導入した基準高さ, 基準幅などのプリミティブ独自の属性も 1, 2, 3 の情報だけからプログラムで推定して求めるが, 推定結果が不満足なプリミティブではここに明示的に記述する.

   この部分は肉付けアルゴリズムでは参照されない.

始点, 終点における連結は以下の 3 種類に分類する. 種類に応じて始点, 終点付近の肉付けが変わる.

- 他のエレメントと関連を持たない場合

  図 4.2の点 1, 5 がこれにあたる.

- エレメントの始点または終点を他エレメントの始点または終点と共有

  点 0 のように, 複数のエレメントの制御点に含まれる場合.

- エレメントの始点または終点が他エレメントの中途に接続 点 2, 3, 6, 7 のように他のエレメントの属性リストの link 情報に含まれている場合.

従来の研究でも, 接続の種類に応じて肉付けを変えるために以下のような方法が提案されている.

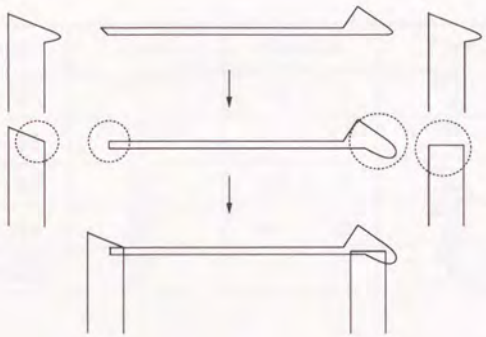


図 4.3: アウトラインの変形

1. エレメント変形による飾り定義

   [8] では接続によって生ずる飾りを図 4.3のように接続がない場合の始点や終点の飾りの変形として表現している. 線の太さや飾りの大きさを変更しても, 接続が不自然にならないように, 変形を表現するのが難しいという問題点がある.

2. 飾りを含んだエレメント定義

   [9] では図 4.4のように飾りを含んだエレメントを定義している. 始点と終点がそれぞれ別のエレメントに接続するエレメントの始点, 終点付近の飾りをこの方法で表現すると組合せの数のエレメントを用意する必要があるので, エレメント定義による飾りは図 4.4にとどめ, 他の飾りは 1 の方法で表現している.

3. 中途接続

   中途接続の場合, アウトラインの始点終点が接続するエレメントの内部にくるようにしなければならない. [8] では, アウトラインの始点終点を, 接続するエレメントのスケルトン (制御点を結んだ線分データ) との交点に置くことによって実現している. スケルトンが必ずしも肉付け結果の内側に来るとは限らないので, この方法では, はみ出してしまうことがありうる.

CScode	Form	CScode	Form
03 + 05	ｱ	03 + 13	ﾃ
03 + 06	ｲ	03 + 14	ﾄ
03 + 07	ｳ	03 + 18	ﾅ
03 + 08	ｴ	03 + 19	ﾆ
03 + 09	ｵ	03 + 20	ﾇ

図 4.4: 飾りを含んだエレメント定義 [9]

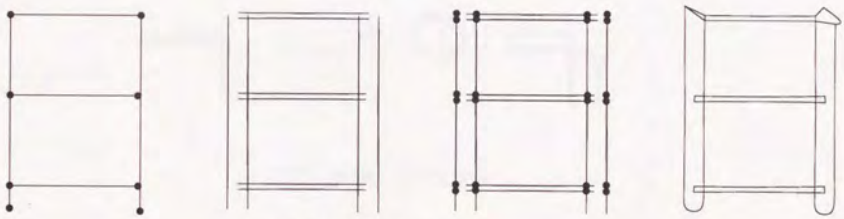


図 4.5: 肉付けの手順

このような過去の研究を参考にし，連結情報を意識した肉付けを以下の手順で行なうことにした．(図 4.5).

1. エレメント肉付け

   各エレメントを接続関係を無視して肉付けする (図 4.6). 肉付けした結果は，左右のアウトラインの二つのアウトラインデータのリストになる．

2. 接続の飾り肉付け

   エレメントの始点，終点同士が接続している時は，エレメントの種類に応じて飾りの肉付けを行ない，飾り部分のアウトラインデータを生成する．飾りの肉付け

図 4.6: スケルトンと肉付け

の際には，グローバル変数とエレメントのアウトライン同士の四つの交点を参照する．飾り部分のアウトラインデータの始点，終点と一致するようにエレメントのアウトラインの始点，終点を変更する (図 4.7)．

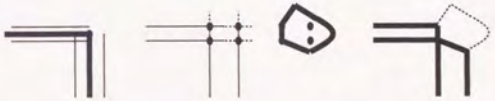


図 4.7: 接続の飾り肉付け

3. 中途接続

   始点，終点が他エレメントの中途から接続しているときは，アウトラインの始点終点を，他エレメントのアウトラインの内部に入るように修正する (図 4.8)．これは，アウトライン同士の四つの交点を求めて，その中点をとることによって行なう．

4. 孤立した始点，終点付近の飾り

   アウトライン上の 4 点を与えて，それを参照してアウトラインデータを生成するという接続の飾りと同じ方法を取る (図 4.9)．

以上の手順によって，エレメントのアウトライン，飾りのアウトラインが得られるので，これらを別々に塗りつぶして使う．

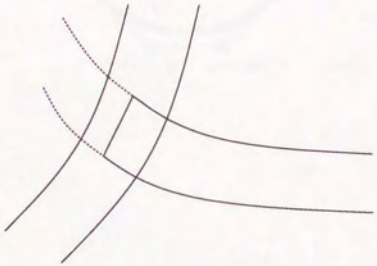


図 4.8: 中途接続

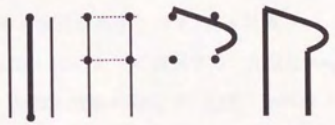
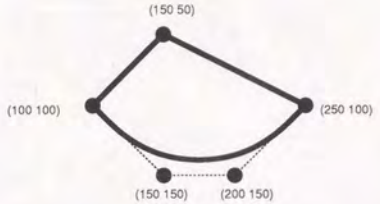


図 4.9: 孤立点の飾り




```
((angle 250 100)
 (angle 150 50)
 (angle 100 100)
 (bezier 150 150)
 (bezier 200 150))
```

図 4.10: アウトラインデータの表現

通常のアウトラインフォントは, 白抜き文字ができる形でアウトラインを定義するが, この方法で得られるフォントは内部に余計な線が出現する. 白抜き文字を使うには [16] で提案されている方法を適用すればよい.

4.3 アウトラインの表現法

生成するアウトラインは, 線分と 3 次の Bezier 曲線で表現する. アウトラインの表現として, 円弧も用いる例があるが, 実用上は円弧も複数の Bezier 曲線に分割して近似できるので, 表現力の点は問題がない. 線分を Bezier 曲線で表現することも可能だが, 線分の方が交点検出等の演算が高速なので, 別々に扱う.

アウトラインデータは図 4.10のように表現する. 先頭に angle というタグがついている点が, 線分あるいは Bezier 曲線の端点で, 先頭に bezier というタグがついている点が, Bezier 曲線の制御点である.

4.4 明朝体の肉付け

前節のような枠組の中で, 目的の書体を作り上げる方法を明朝体を例に説明する.

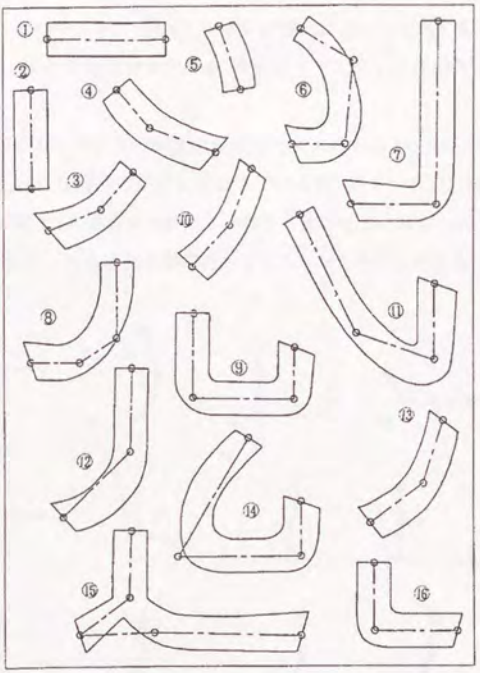


図 4.11: ゴシック体エレメント ([7])

4.4.1 エレメントの種類

ゴシック体のエレメントに関しては, [7] で図 4.11のような 16 種類が提案されている.

本研究では, 9 と 14 を合わせる一方, 2 を縦線と, 曲り縦に分けた 16 種類のエレメントを用意した. また, 15 のしんにょうはエレメントとして複雑過ぎるので, 三つのエレメントの接続として実現することにした. 第 1 第 2 水準の漢字を表現するのに用いたエレメントは, 下の 16 種類である (図 4.12).

1. 点, 2. 縦棒, 3. 横棒, 4. 上払い, 5. 左払い, 6. 縦棒 + 左払い, 7. 右払い, 8. こざとへんの一部, 9. 縦棒 + はね, 10. つくりのはね, 11. さんずいの下, 12. 心の一部, 13. たす

き, 14. まがり縦棒, 15. かぎ, 16. しんにょう

各エレメントはなるべく少ない, 指定しやすい制御点によって定義する. 制御点の数を増やすと, デザイン上の自由度があがるが, その分プリミティブのデザインが面倒になる.

図 4.12は, 明朝体のエレメントの制御点を線分で結んだものと, 肉付け結果である. 制御点を結んだものはほぼ肉付け結果と重なっているが, 必ずしも中心線上にあるとは限らない. 漢字の横方向の組合せでは上下の端を一致させる必要があるるが, 中心線上に置くと, 太さを変更した時に上下端があわなくことがあるからである.

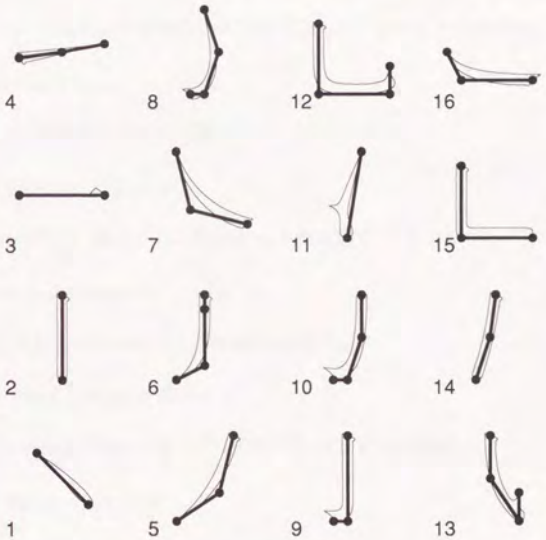


図 4.12: 明朝体のエレメント

4.4.2 座標計算のための関数

肉付けアルゴリズムの記述を容易にするために, 座標の計算を行なう次のような関数とマクロを用意する. 点及びベクトルは, 数字を要素とする長さ 2 のリストで表現する.

- length2 Vector

  ベクトル Vector の長さを返す.

- metric2 Point-1 Point-2

  2 点 Point-1, Point-2 間の距離を返す.

- norm2 Vector

  ベクトル Vector に平行な単位ベクトルを返す.

- normlen2 Len Vector

  ベクトル Vector に平行で長さが Len であるようなベクトルを返す.

- times2 Len Vector

  ベクトル Vector の Len 倍を返す.

- plus2 Vector-1 ... Vector-n

  n 個のベクトル Vector-1,...,Vector-n の和を返す (マクロ).

- diff2 Vector-1 Vector-2

  二つのベクトル Vecotr-1 と Vector-2 の差を返す.

- inter2 Point-1 Point-2 Rate

  2 点 Point-1 と Point-2 を 1-Rate 対 Rate に分ける点を返す.

- mul2 Vector-1 Vector-2

  ベクトル Vector-1 と Vector-2 の内積を返す.

- rot90 Vector

  ベクトル Vector を 90 度回転したベクトルを返す.

- rot270 Vector

  ベクトル Vector を 270 度回転したベクトルを返す.

- costheta Vector-1 Vector-2

  ベクトル Vector-1 を基準にしたベクトル Vector-2 の角度の cos を返す.

- sintheta Vector-1 Vector-2

  ベクトル Vector-1 を基準にしたベクトル Vector-2 の角度の sin を返す.

- theta Vector-1 Vector-2

  ベクトル Vector-1 を基準にしたベクトル Vector-2 の角度を返す.

- affine Point Affine

  点 Point をアフィン変換 Affine で変換した結果を返す. アフィン変換は大きさ 6 の配列で表現する.

- affine-outline Outline Affine

  アウトラインデータの各点をアフィン変換 Affine で変換した結果のアウトラインデータを返す.

- rmat Mat

  2 次行列 Mat の逆行列を返す.

- movex Xmove [Affine]

  アフィン変換 Affine(指定されなければ恒等変換) に平行移動 (Xmove,0) をかけたアフィン変換を返す.

- movey Ymove [Affine]

  アフィン変換 Affine(指定されなければ恒等変換) に平行移動 (0, Ymove) をかけたアフィン変換を返す.

- movexy Xmove Ymove [Affine]

  アフィン変換 Affine(指定されなければ恒等変換) に平行移動 (Xmove, Ymove) をかけたアフィン変換を返す.

- scalex Xscale [Affine]

  アフィン変換 Affine(指定されなければ恒等変換) に, 原点を中心にした X 方向の Xscale 倍拡大をかけたアフィン変換を返す.

- scaley Yscale [Affine]

  アフィン変換 Affine(指定されなければ恒等変換) に, 原点を中心にした Y 方向の Yscale 倍拡大をかけたアフィン変換を返す.

- scalexy Xscale Yscale [Affine]

  アフィン変換 Affine(指定されなければ恒等変換) に, 原点を中心にした X 方向に Xscale 倍, Y 方向に Yscale 倍の拡大をかけたアフィン変換を返す.

- rotate Theta [Affine]

  アフィン変換 Affine(指定されなければ恒等変換) に, 原点を中心の Theta 回転をかけたアフィン変換を返す.

明朝体の複雑な形状を表現するために, 更に高度なライブラリもいくつか用意する.

折れ線近似から曲線への変換

飾りや, はねなどの複雑な形状を表現する場合, 折れ線近似から曲線への変換が必要となる. このような部分では曲線の性質は気にしなくてよいことがある. このような用途のために, 以下の条件を満たす内挿スプライン曲線を生成する関数を定義する.

1. Bezier 曲線の切れ目

   折れ線の始点, 終点と途中の線分の中点をアウトライン特徴点とし, これらの間がそれぞれ一つの Bezier 曲線で表される.

2. 接線ベクトル

   アウトライン特徴点における接線ベクトルの方向が, その点での折れ線の方向と一致する.

3. 曲率の連続

   生成する Bezier 曲線が, 曲率連続で接続する.

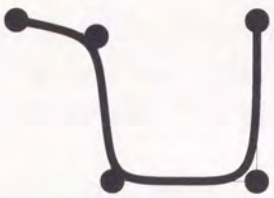


図 4.13: 折れ線近似から曲線への変換

4. 境界条件

両端での曲率が直線と接続することを考えて 0 となる.

これらの条件を満たす補間曲線は一意には決まらないが, その中でもっとも単純な, 接続する点で曲率が 0 になるような補間曲線の使用を試みる.

点 p_0, p_1, p_2, p_3 で指定される Bezier 曲線上の一点は次のような式で表される.

$$p(t) = (p_3 - 3p_2 + 3p_1 - p_0)t^3 + 3(p_2 - 2p_1 + p_0)t^2 + 3(p_1 - p_0)t + p_0$$

t に関する微分, 及び二階微分は次のようになる.

$$p'(t) = 3(p_3 - 3p_2 + 3p_1 - p_0)t^2 + 6(p_2 - 2p_1 + p_0)t + 3(p_1 - p_0)$$

$$p''(t) = 6(p_3 - 3p_2 + 3p_1 - p_0)t + 6(p_2 - 2p_1 + p_0)$$

$t = 0$ での曲率が 0 であるための条件は, $p''(0) = 0$ または, $p''(0) \parallel p'(0)$ である. $t = 0$ を代入すると, 前者の条件は, $p_2 - p_1 = p_1 - p_0$ 後者の条件は $p_2 - p_1 \parallel p_1 - p_0$ と書ける. 後者は前者を含んでいるので, $p_2 - p_1 \parallel p_1 - p_0$ すなわち, p_0, p_1, p_2 の 3 点が一直線上にあることが, $t = 0$ での曲率が 0 であるための条件となっている.

$t = 1$ での曲率が 0 であるための条件も同様に論ずることができ, この二つの条件を満たすのは, $p_1 = p_2$ の時であることがわかる. この時の p_1, p_2 はもとの折れ線近似の際の頂点になっている.

以上の方法によって生成される曲線は図 4.13のようになる.

太さ変化からアウトラインへの変換

いくつかのエレメントは中心線を決定して，それに左右均等にアウトラインを生成することによって実現できる．このようなエレメントを扱うために中心線を Bezier 曲線で与え，両端の太さおよび太さの微分値から，アウトラインを生成するいくつかの関数を用意する．

中心線が $0 \leq t \leq 1$ なる t の 3 次 Bezier 曲線で表され，太さが同様に t の 3 次式で表されていても，アウトラインは 3 次 Bezier 曲線では表すことができない．そこで，近似曲線を使うことになるが，どのように近似するかで，2 種類の方法が考えられる．

まず考えられるのが，アウトラインの両端での t に関する微分がベクトルの絶対値も含めて，正しい値になるように近似曲線を生成する方法である．アウトラインの中心線が $p(t)$，太さが $w(t)$ で与えられている時，アウトライン上の点 y について，$y, \dot{y}$ は次のように書ける．

$$
\begin{aligned}
y &= (p + \frac{w}{|\dot{p}|}(-\dot{p}_y, \dot{p}_x)) \\
\dot{y} &= \dot{p} + \frac{\dot{w}}{|\dot{p}|}(-\dot{p}_y, \dot{p}_x)) + \frac{w}{|\dot{p}|}(-\ddot{p}_y, \ddot{p}_x)) - \frac{w(\ddot{p} \cdot \dot{p})}{|\dot{p}|^3}(-\dot{p}_y, \dot{p}_x))
\end{aligned}
$$

両端での位置 y_0, y_1 と，t に関する微分 y_0', y_1' が求まると，これに適合する Bezier 曲線 $y_0, y_0 + \frac{1}{3}y_0', y_1 - \frac{1}{3}y_1', y_1$ が一意に決定する．$t = \frac{1}{2}$ という中間の値でのこの Bezier 曲線上の点が，中心線から太さ $w(\frac{1}{2})$ の点と十分近いことをチェックし，不十分の時は，中心線をこの点で二つの曲線に分けてから，再びこの操作を行なう．

この操作を行なう関数が，curve1 で，Bezier の制御点と両端での太さ，及び太さ変化を渡して，アウトラインのリストを得る．

2 つめの方法はアウトラインの Bezier 曲線の制御点の位置関係が中心線の Bezier 曲線の制御点の位置関係と同じになるような Bezier 曲線を生成する方法である．図 4.14のようにアウトラインを折れ線近似で求めて，それぞれの線分を中心線と同じ比率で分ける点を Bezer 曲線の制御点とする．第 1 の方法と比較すると高速で，多くのエレメントの肉付けはこちらで十分である．この操作を行なう関数 curve2 の引数および，返す値は curve1 と同一である．

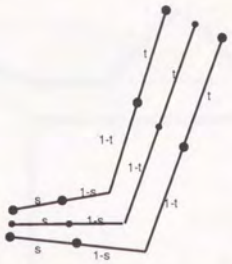


図 4.14: アウトラインの生成

4.4.3 肉付けを制御するグローバル変数

明朝体の肉付けの際に使われるグローバル変数は次のようなものである (図 4.15).

- minchowidth

  縦棒の中心からアウトラインへの太さを与える.

- tateyokoratio

  縦棒の太さに対する横棒の太さの比率を与える.

- kazariheight

  縦線と横線の連結によって生ずる飾りの高さの, 縦線の太さに対する比率を与える.

- tomeheight

  横棒の終点の飾りの高さの, 縦棒の太さに対する比率を与える.

これらの変数は独立に変化させることができるが, ある変数を変化させた時に, 他の変数をどう変化させるときれいなフォントを得ることができるかを会得するには多少の経験が必要である.

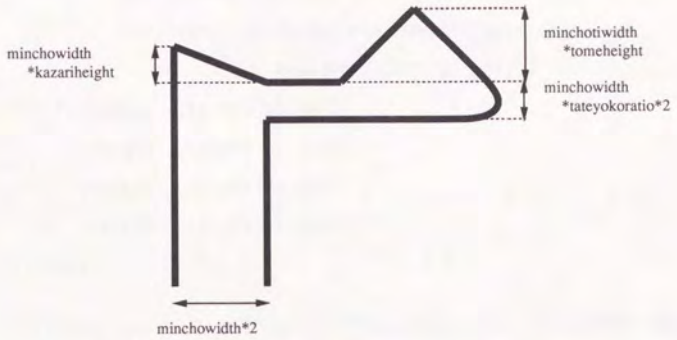


図 4.15: 明朝体肉付けで使われるグローバル変数

図 4.16: 横棒

4.4.4 エレメントの肉付け

エレメントの肉付けは，制御点のリストとエレメントの属性リストを引数とし，左右のアウトラインデータをリストにして返す関数として定義する．書体全体の太さなどのパラメータの受渡しは，グローバル変数を使用する．

縦棒，横棒のような単純な形からなるエレメントは割合簡単に記述できる．これらのエレメントは，中央がくびれて細くなっているフォントもあるが，単純な線分で図4.16のように表現する場合は，次のように書けばよい．

```
(putprop 'yoko
  (function
    (lambda (points (alist))
      (lets ((p0 (first points))
```

```
            (p1 (second points))
            (v0 (normlen2 (times minchowidth tateyokoratio)
                          (rot270 (diff2 p1 p0))))
      '(((angle .,(plus2 p0 v0))
         (angle .,(plus2 p1 v0)))
        ((angle .,(diff2 p0 v0))
         (angle .,(diff2 p1 v0))))))))
  'mincho)
```

このように, エレメントの肉付けプログラムの実体はエレメントの種類に対応するシンボルの属性リストに書体をタグとして収められている. こういう内部の構造を表面に表さないためにマクロを使い, 次のように書くことにする.

```
(defelement mincho tate
  (lets ((p0 (first points))
         (p1 (second points))
         (v0 (normlen2 (times minchowidth tateyokoratio)
                       (rot270 (diff2 p1 p0))))
    '(((angle .,(plus2 p0 v0))
       (angle .,(plus2 p1 v0)))
      ((angle .,(diff2 p0 v0))
       (angle .,(diff2 p1 v0)))))))
```

点の肉付けは見本のアウトラインをアフィン変換することによって実現する. 点は二つの制御点で指定する. アフィン変換は 6 自由度があるが, 見本のアウトラインにおける制御点が指定された制御点に移る変換とすることによって, 自由度は二つに減り, 更に等角変換とすることによって自由度は一つに減る. 残った 1 自由度は線の太さの指定に使う (図 4.17).

他の多くのプリミティブは, エレメントをいくつかの部分に分割して, それぞれの部分のアウトラインを関数 curve1, curve2 あるいは折れ線近似によって生成して, つなぎ合わせることによって生成している.

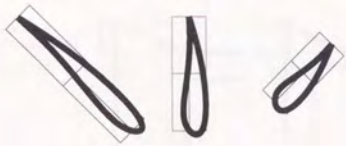


図 4.17: 点の肉付け

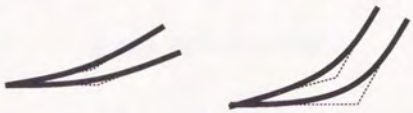


図 4.18: 左払い

curve1, curve2 を呼ぶ際の引数である太さ及び太さ変化は，大部分は定数で指定するが，左払いだけは図 4.18のように，始点付近と終点付近での角度に応じて，太さ変化が変わってくるので，角度に応じた関数として定義している．

4.4.5 飾りの肉付け

始点，終点付近の飾り

縦棒，左払い等の始点付近の飾りは図 4.19のように，エレメントの傾き方に比べると，傾き方が小さい．これを実現するために，始点の飾り付けプログラムは，エレメントの傾き θ を求めて飾りの傾き変化が，$0.4 \times \theta$ になるように調節している．

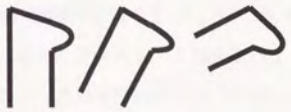


図 4.19: 始点の飾り

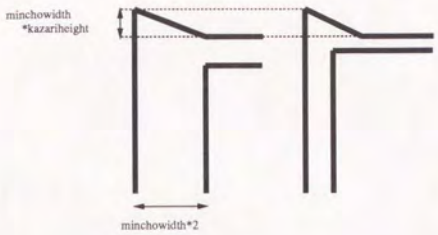


図 4.20: 横棒の始点の接続

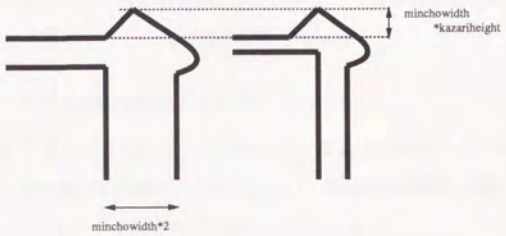


図 4.21: 横棒の終点の接続

接続による飾り

接続による飾りが生ずるのは，明朝体では，主に横棒の始点，終点を他のエレメントと共有する場合である．

横棒の始点と縦棒等の始点との接続はグローバル変数 kazariheight が小さい時は，図 4.20の左側のように縦棒の範囲内で飾りをつけるが，大きくなると鋭角になって醜くなるので，図の右のようにはみださせるという補正を行なう．横棒の終点と縦棒の始点の場合も，同様にしてパラメータの変化に応じたきめ細かい変形を実現している (図 4.21)．

第 5 章

組合せアルゴリズム

5.1 組合せの種類

本研究のシステムでは, 以下の基準でプリミティブにする漢字を決め, 第 1 第 2 水準すべての漢字に必要なプリミティブは 956 個となった.

1. 部品間の交叉や接続が存在するもの

   「見」を「目」と「にんにょう(禿の下)」を組み合わせた文字であると定義すると, 二つの部品間の接続が生じてしまう. 部品間の接続も考慮した組合せは難しいので, このような漢字はプリミティブとする.

2. 部品の再利用ができないもの

   「以」は三つの部品に分けることができるが, 分けた部品を使う漢字が他にみつからないのでプリミティブとする.

3. 組合せで実現しにくいもの

   「れんが(然の下)」, 「さんずい」のように点を微妙に配置する必要がある部品, 「三」のように横棒の幅を変える必要がある部品は, 組合せにより生成するのが難しいので, プリミティブとする.

3 の制限によって, 組合せは以下の 3 種類に絞られる.

- 横方向の組合せ

  「波」, 「街」のような組合せ (図 5.1)).

図 5.1: 横方向の組合せ

図 5.2: 縦方向の組合せ

- 縦方向の組合せ

  「栗」,「志」 のような組合せ (図 5.2).

- その他の組合せ

  「国」,「聞」 などの「構えと旁」や,「雁」,「虎」 のような「たれ」,「道」,「処」 のような「にょうと旁」などの (図 5.3), 片方の部品の中の空白部分に他方の部品が入るような組合せ.

複合パーツの記述は以下のように行なう.

```
(defjoint nil 鯵 '(yoko うおへん 参))
```

図 5.3: その他の組合せ

```
(defjoint nil うおへん '(tate くのじ田 四点))
(defjoint nil 超 '(nyou そうにょう 召))
```

5.2 組合せアルゴリズム

組合せアルゴリズムは，利用者が抽象的に記述した組合せから，具体的な組合せを求めるアルゴリズムである．組合せアルゴリズムを決定する際に，以下の原則を与えた．

- 肉付けパラメータによって組合せが変わらない

  スケルトンフォントの利点の一つは，太さなどのパラメータを容易に変更できることなので，組合せの結果がパラメータによって影響を受けないようにする．

- 一度に一つのレベルの組合せだけ扱う

  一度組み合わせた部品（例「合」）を別の部品（例「塔」）の組合せに使う際に再組合せを行なわずに，まとまった１つの部品として扱う．

- エレメントの種類を考慮する

  エレメントの種類を考慮せずにエレメントの制御点を結んだ線分列の幾何データだけから組合せを行なうと，線の太さや飾りの有無が反映されないので，うまくいかない．

組合せアルゴリズムは，次のような二つの部分からなる．

1. 基準幅，基準高さの算出

   組み合わせる部品の大きさの調節が狂うと，バランスのとれない字ができてしまう．バランスをとるために，部品の空間的な広がりを定義する基準幅，基準高さという概念を導入し，組合せの際にこれを使って，部品の大きさを決定する．

   プリミティブ一つ一つに人手で基準幅，基準高さを与えるのは避け，プログラムによって自動的に算出する．

2. 食い込み限界の算出

   組み合わせる部品の大きさがよくても，部品の配置の際の部品間の間隔が広過ぎると間の抜けた字になるし，狭過ぎると窮屈な字になってしまう．１で求めた基準幅，基準高さを元に部品間の間隔をプログラムによって決定する．

5.2.1 基準高さ，基準幅

組合せプログラムは，組み合わせる部品を拡大 / 縮小した後，移動を行なう．部品の大きさの決定に使うのが部品の基準高さ，基準幅である．

部品組合せの実験で最初に試したのは，組み合わせる部品の面積あたりの線長を揃える方法である (図 5.4の上)．この方法には斜めの線を含む部品が小さくなるという欠点があるので，線長として $\sqrt{x^2+y^2}$ ではなく $|x|+|y|$ を使うことである程度改善されたが (図 5.4の下)，まだまだ手による修正を必要とする文字が多かった [23]．

図 5.4: 高さの比率

そこで別の方法を採用した．横棒の間隔を均一にするというのは書道の基本だが，フォントデザインでも縦棒や横棒が主な構成要素になっている文字では，位置や結合関係によって多少の変動はあるものの，間隔がほぼ均等になるようにデザインする．この考えを組合せプログラムにあてはめ，縦方向の組合せの場合はそれぞれの部品の横棒の間隔を，横方向では縦棒の間隔を揃えるように拡大 / 縮小してから組み合わせる (図 5.5)．

このままでは縦棒や横棒だけからなる部品にしか適用できない．そこで，縦棒や横棒を含まない部品についても横棒や縦棒があるとしたら間隔がどの位になるかを求める．これを基準高さ，基準幅と呼ぶ．

1000 個近くのプリミティブすべてに基準高さや基準幅を与えるのは大変なので，なるべくプログラムで求め，その結果修正が必要なものだけ利用者が与える．

基準高さと基準幅を求める方法は 2 種類ある．一つめはパターンマッチングによる方法である．図 5.6は [18] 中にあるものだが，このようなパターンと高さの表を作っておく．これらを含むプリミティブ中の対応部分の高さをパターンの高さで割ると基準高さが得られる．

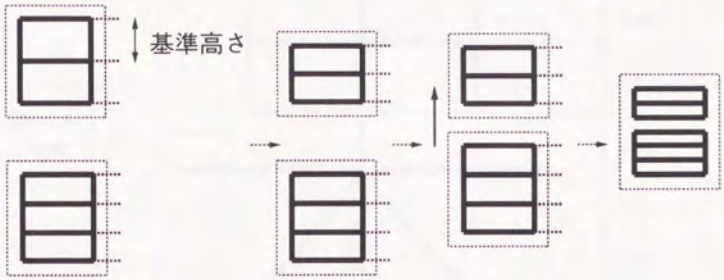


図 5.5: 横棒の間隔

図 5.6: パターンの高さ [18]

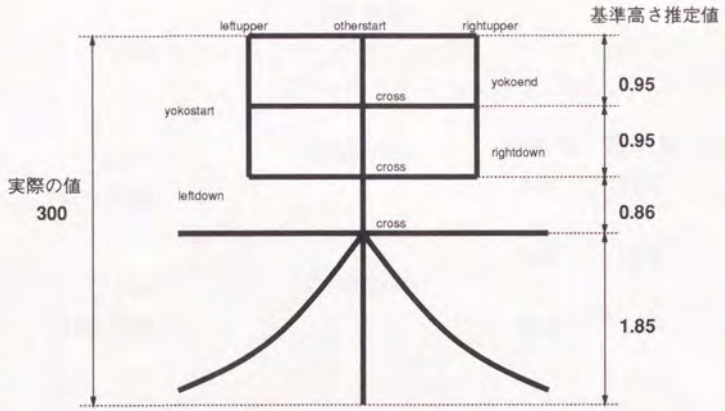


図 5.7: パターンマッチによる高さ推定

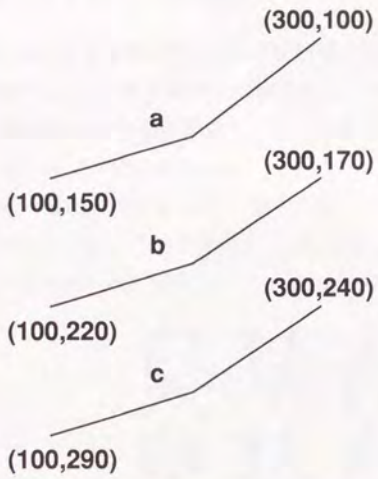
以下のリストのように横棒と横棒, 横棒と上端, 横棒と下端の間にはさまれるエレメントとの接続関係, 位置関係によってパターンを指定し, 対応部分の基準高さを定義する.

```
(((tate leftupper leftdown)
  (tate rightupper rightdown))
  . 1.0) ; 口
(((tate otherstart otherend))
  . 0.86) ; 工
...)
```

プリミティブに対して複数パターンがマッチする場合には対応部分の高さの和をパターンの高さの和で割る. 図 5.7では 4 つのパターンとマッチングするので, 実際の高さ 300 を $0.95 + 0.95 + 0.86 + 1.85 = 4.61$ で割った値 46.1 が基準高さになる.

これは, パターンマッチングの高速化と記述の容易さを狙ったためだが反面, 「平」と「平」の 2 本の横棒の間のパターンの区別がつかないという欠点もある.

この方法でかなりの数のプリミティブの基準高さが求まるが, 横棒を含まないプリ



組合せ	推定値
a-b	100
a-c	200
b-c	100

図 5.8: エレメント組合せによる推定

ミティブには適用できないし，基準幅には基準とできる適当なパターンが見つからないので使えない．

1番目の方法で決定できないプリミティブの基準高さや，基準幅の決定では二つのエレメントの位置関係から推定を行ない，それら総合して決定する．

n 個のエレメントからなるプリミティブの場合，二つのエレメントの組合せは，$n(n-1)/2$ 個ある．このうち接続，交叉がない組合せすべてについて基準幅，基準高さの推定を行なう．二つのエレメントの種類に応じて別々の関数を呼ぶ．縦棒と横棒の組合せのように基準幅，基準高さの推定を行なわないこともある．

図5.8のプリミティブの場合には，a-b, a-c, b-c の三つの組合せによって基準高さについて三つの推定値が出るが，a-c のように，どちらのエレメントも別のあるエレメントとの組合せでの推定値が存在してもともとの推定値よりも小さい組合せによる推定は捨て，残りの二つの推定値の平均を取ってプリミティブの基準高さとする．

5.2.2 プリミティブの移動

部品の大きさを変更した後, それぞれを移動する. 縦方向の組合せでは中心合わせや幅の決定を行なった後で縦方向の移動を行ない, 横方向の組合せでは上下の調節を行なった後で横方向の移動を行なう. これらの組合せについては移動量はパラメータ一つで表せるが, その決定は難しい.

部品間の接続, 交叉がないように漢字を部品に分けたのだから, 移動の結果スケルトンの交叉が生じないという条件が成り立つ. ただ, 単なる交叉が生じないようになるべく近付ける戦略では図5.9の上のような不自然な字が生じてしまう.

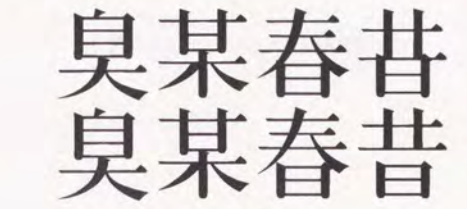


図 5.9: 食い込み

そこで, エレメントの種類を考慮して二つのエレメント間の位置関係の制約条件を宣言する方法を用いることにした. 位置関係の制約条件は次のように宣言する.

```
(deflimit mincho (yoko yoko)
  ((or (<= x00 x10 x01)(<= x00 x11 x01)(<= x10 x00 x11)(<= x10 x01 x11))
   (>= (diffabs (+ y00 y01) (+ y10 y11)) (* '(1.4 . 1.4) yunit))))
```

この例は, 二つの部品に含まれる横棒がX方向に重なりを持っているなら, Y座標の差が基準高さの 0.7 倍よりも大きくなければいけないということを宣言している (図 5.10).

組み合わせる際には二つの部品の制御点を $at + b$ の形にして, すべてのエレメント対について, 許される位置関係にある t の区間を計算し, その範囲でもっとも望ましい t の値によって組合せを行なう (図 5.11). この方法で移動だけでなく一点を中心とする拡大についても位置関係の制約を満たす t を求めることができる.

区間は次のようなリストで表現する.

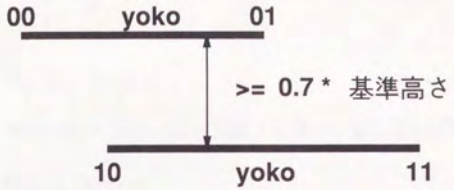


図 5.10: エレメント間の制約条件

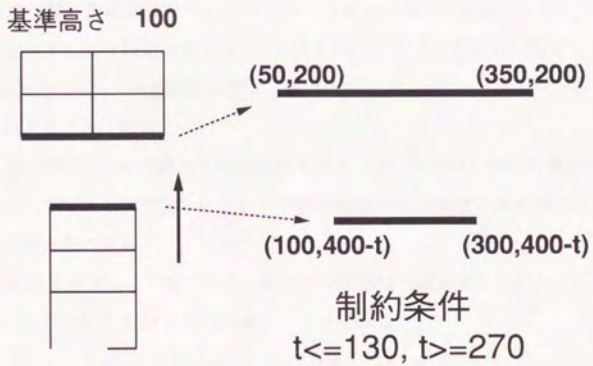


図 5.11: 制約条件の利用

```
((nil . 1)((> . 2) . 3) (4 . (< . 5)) (6 . t))
```

これは, $t \leq 1, 2 < t \leq 3, 4 \leq t < 5, 6 < t$ という区間を表現している. 全区間は ((nil . t)) で, 空区間は nil である. これらの区間の間の演算を行なうために, 以下のような関数を用意する.

- andsection Section1 Section2

  二つの区間 Section1 と Section2 の共通して含まれる区間を返す.

- orsection Section1 Section2

  二つの区間 Section1 と Section2 のどちらかに含まれる区間を返す.

- notsection Section

  区間 Section に含まれない区間を返す.

条件判断をともなう規則も, この区間の演算によって実現できる. 「A ならば B」という規則を満たす区間は, A を満たさない区間と B を満たす区間を算出して, その orsection を取れば求めることができる.

各条件は一般に四則演算を含む不等式として書かれるが, 結果的に t の 3 次以上の不等式になってしまう規則は許さない. 3 次 4 次の不等式は代数的に解くことができ, また, それ以上であっても数値的に解くことは可能だが, 計算のコストがかかるので, 2 次までに抑えることにする.

たとえば, 絶対値のの比較を 2 乗の比較で行なうと, 片方が 2 次式の場合は, 全体が 4 次式になってしまう. 次数をおさえるために中身が正の場合と負の場合の両方の区間を求めて組み合わせる.

規則が記述されていない場合にも, 組合せの結果交叉点を生じないという暗黙の条件がある. 線分 AB と CD に交点があるという条件は,

$$0 \leq \frac{(A_x - C_x) * (B_y - A_y) - (A_y - C_y) * (B_x - A_x)}{(C_x - D_x) * (B_y - A_y) - (C_y - D_y) * (B_x - A_x)} \leq 1$$

$$0 \leq \frac{(C_x - A_x) * (D_y - C_y) - (C_y - A_y) * (D_x - C_x)}{(D_x - C_x) * (A_y - B_y) - (D_y - C_y) * (A_x - B_x)} \leq 1$$

と書ける. A_x 等が t の 1 次式であるため, これらは 2 次の有理式の形で表され, 場合分けをすることによって, 2 次の不等式として解くことができる.

5.2.3 縦方向の組合せ

縦方向の組合せは，以下の手順によって行なう．

1. 両方の部品の中心を合わせる
2. 部品の幅を決める
3. 部品の高さの比率を決定する
4. 上下の部品の食い込みを決定する

3は各部品の基準高さを揃えるだけで実現できる．4は位置関係の制約条件を満たす範囲でなるべく近付けるような上下移動で実現できる．以下では縦方向の組合せに固有の1, 2の問題について述べる．

中心あわせ

縦方向に組み合わせることができる部品は左右対称に近い形をしたものが多く，対称な部品同士を組み合わせる場合には対称の中心を合わせる必要がある．対称の中心を求めずに単純に重心を中央にして組み合わせると，図5.12の上のようになってしまう．

図 5.12: 中心合わせ

すべての対称なプリミティブについて部品の中心を定義するのは手間がかかるので，プリミティブのデータからプログラムによって対称の中心を求める．まず，対称となりうるエレメント及びエレメントの対を与えてやる (図5.13)．

プリミティブの中心を求めるために，まずプリミティブのエレメントを制御点を結ぶ線で近似した時の重心を求める．これを仮の中心とし，プリミティブのすべてのエレ

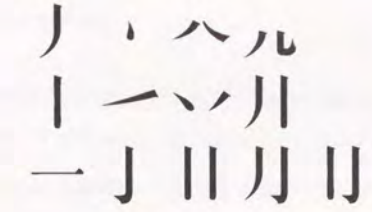


図 5.13: 対称とみなすエレメント

メント，エレメント対についてそれが対称とみなせて，その対称の中心と仮の中心とのずれがある程度以下のものを集める．

対称なエレメント，エレメント対が求まったら，それらの対称の中心の平均をそのプリミティブ全体の中心とする．ただし，中心付近に縦棒，縦棒＋左払い，縦棒＋はねがあるときはそのエレメントを中心にする．

幅の決定

縦方向の組合せでは，中心合わせと同様に幅の決定も重要である．図 5.14を見ると幅の調節の必要性がよく分かるであろう．

図 5.14: 幅の調節の有無

幅の決定のために xlimit という仮想的なエレメントを用い，xlimit と実際のエレメントとの位置関係の制限を宣言しておく．縦方向の組合せをする際は各部品について左右から xlimit を限界まで近付けておいて，縦方向の移動を行なう前に各部品の左右の xlimit が縦に並ぶように横方向の移動を行なう．

xlimit と実際のエレメントの位置関係は以下のように定義する．

```
(deflimit mincho ((tate tatehidari kokoro tatehane tsukurihane
```

```
kagi tasuki magaritate)
  xlimit)
  ((or (<= y00 y10 y01)(<= y00 y11 y01)(<= y10 y00 y11)(<= y10 y01 y11))
   (>= (diffabs (+ x00 x01) (+ x10 x11)) xunit)))
```

上の表現では xlimit と縦棒との間隔が基準幅の半分になるように宣言している. 一般に縦棒の数が多い部品の幅を広くすると, バランスのよい文字になるが, 縦棒の数が多い部品は基準幅が小さいので, 上のように宣言すると, xlimit がより近くまで近付き, 部品の幅が広くなるという効果が得られる.

5.2.4 横方向の組合せ

横方向の組合せは以下の手順によって行なう. 上下対称の部品が少ないので, 縦方向の組合せのように中心合わせを行なう必要はなく, 縦方向の組合せと比較して簡単になっている.

1. 両方の部品の上下を合わせる
2. 部品の幅の比率を決定する
3. 食い込みを決定する

部品の上下合わせのために, 縦方向の組合せ同様 ylimit という仮想的なエレメントを導入して, 実際のエレメントとの位置関係の制約条件を定義しておく. ただし, 横方向の組合せでは部品の上下端が横棒の場合, 少数の例外を除くと部品の基準高さに無関係に横棒の位置を揃えることが多いので, ylimit と横棒の間隔は基準高さに比例して与えるのではなくて, 入れようとする枠に対する割合で与えるようにした.

ただ, 冂や旁の一部は例外的に上下を大きめに開ける必要がある. このような部品に関してはプリミティブの属性リストに ylimit という属性を手で与えることによって修正する (図 5.15).

5.2.5 その他の組合せ

横方向, 縦方向以外の組合せは, 片方の部品中の領域の一部に他方の部品を埋め込むことによって実現する. どの領域に埋め込むかは, 以下のようにプリミティブの定義の中でデザイナがあらかじめ与えておく.

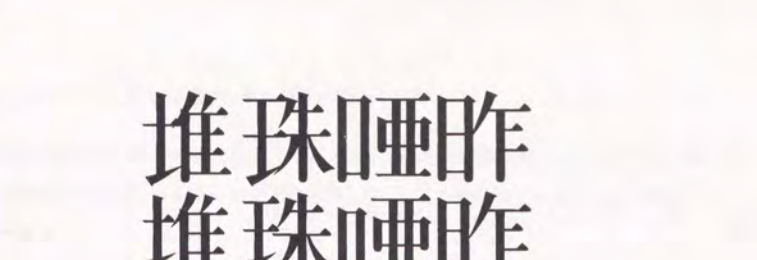


図 5.15: 上下合わせの補正

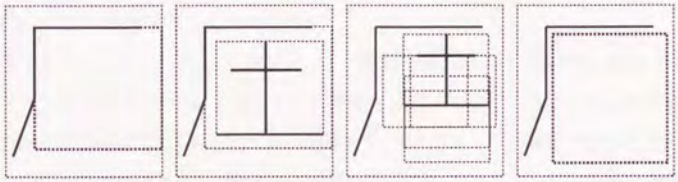


図 5.16: その他の組合せの手順

```
(setq くにがまえ
   '(((39 366)(361 366)(38 32)(38 392)(361 32)(361 392))
     ((yoko (0 1))(tate (2 3)(link 0)(tate (4 5)(link 1)(yoko (2 4)))
     (kamae 39 32 361 366)))
```

この領域に他方の部品を位置関係の制限を満たすようになるべく大きく配置すればよいのだが, 縦横方向の配置と違って自由度が 4 あるため, 簡単ではない.

現在は, 以下のアルゴリズムを採用している (図 5.16).

1. 空白部分を推定し (a), その中心に埋め込む部品を置く.
2. 埋め込んだ部品に X, Y 等倍の拡大 / 縮小を施して制約をみたす限界を求める (b).
3. 限界の 0.7 倍程度に拡大し, X 方向, Y 方向にどの程度平行移動できるかチェックする (c).
4. それに応じて, 空白部分を X, Y 方向独立に拡大し, 中心位置をずらす (d).

5. これが1と近ければ終了，それ以外の時は1に戻る．

縦方向，横方向の組合せと比較して部品の移動，拡大の回数が多いので，組合せに要する時間は多い．手による修正が必要な文字も多く，この組合せアルゴリズムは改良の余地がある．

5.3 プリミティブの変形

同じ漢字の要素であっても，組合せの種類，組合せの相手によって，エレメントの種類が変わったり，変形することがありうる．

「木」と「きへん」のように，エレメントの種類が変わるような要素は，別のプリミティブとすることにしたが，「巻」の下の要素と「己」のようにちょっとした座標の変化があるものは，同じプリミティブで定義して，プリミティブの変形で対応する．

プリミティブが変形すると，組合せの結果も変化する．プリミティブの変形の指定は次のようにして行なう．

- 組合せの定義時に指定

  ```
  (setq 祭 '(tate 祭がしら (kashira 示)))
  ```

  のように，組合せる相手によって，変形の有無が決まるようなプリミティブは組合せの定義の時点で明示的な変形を宣言しておく．

- プリミティブ独自の組合せアルゴリズム

  次節で述べるが，プリミティブの hook を利用して，hook の中で組合せを実行することができる．これを変形と組み合わせると，組合せとプリミティブ変形を任意の順序で何度でも呼ぶことができる．

これらの指示によるプリミティブの変形法は，あらかじめプリミティブ中に以下のように定義する．

```
(setq 土 '(((200 10) (200 367) (11 368) (388 368) (41 160) (355 160))
           ((tate (0 1)) (yoko (2 3) (link 1)) (yoko (4 5)))
  (trans (0 100 400 200) (-0.4 0 0 0 80 0))))
```

trans という属性リストの中身が，変形の指示である．プリミティブ内の矩形領域に含まれる全ての点に対してアフィン変換を施すことによって変形を行なう．変形の程度をパラメータとして与えることができるようになっている．上の例では，$0 \leq x \leq 400, 100 \leq y \leq 200$ なる制御点に対して，$\left(I + \begin{bmatrix} -0.4 & 0 \\ 0 & 0 \end{bmatrix} t\right)\vec{x} + \begin{bmatrix} 80 \\ 0 \end{bmatrix} t$ というアフィン変換をほどこすという意味になっている．

5.4 組合せ時の hook

基本的な組合せアルゴリズムで扱えないプリミティブの特殊な性質を扱うために，組合せの際に，組合せの種類や種々の情報に基づき，プリミティブの属性を変化させたり，独自の組合せプログラムを起動することができるようにする．

- 基準高さ，基準幅の変更

  「杏」と「呆」は，「口」と「木」を上下逆に組み合わせた文字だが，「口」と「木」の高さの比率が，異なっている．一般に縦方向の組合せでは，一番下に来る部品を大きめにするとバランスが良いが，「口」の場合は特にその傾向が強い．

  そこで，「口」というプリミティブに縦方向の組合せの組合せの一番下になった時には，基準高さを 0.6 倍するという hook をつけてやると，バランスの良い字を作ることができる．具体的な定義は以下のように記述する．

```
(setq 口
  '(((329 63) (329 357) (328 334) (71 63) (71 365) (73 334))
    ((tate (0 1) (link 2))
     (tate (3 4) (link 5))
     (yoko (5 2))
     (yoko (3 0)))
     (ylimit 0 400)
    (primhook (lambda (type i n)
                (and (eq type 'tate)
                     (equal i (1- n))))
```

図 5.17: 組合せとプリミティブ変形

図 5.18: 組合せによるプリミティブ変形

```
lambda (prim type i n)
    (add-unit prim '(1.0 . 0.6)))))
```

- 組合せとプリミティブ変形

  縦横以外の組合せの際,「門がまえ」や「虎がまえ」は中に入る部品と基準高さが等しくなるように変形する. 中に入る部品の基準高さは組合せの結果決まるので, 組合せ前にプリミティブ変形の程度を決定することはできない.

  このようなプリミティブには, 一度デフォルトの値で組合せを試行し, その後プリミティブの変形をどの程度行なうかを決定して, 再び組合せを実行するような関数を hook として与える (図 5.17).

  図 5.18は,「門がまえ」が組合せる部品によって変形する様子を表している.

5.5 漢字全体の配置

欧文では, 組版の際に文字の幅が可変であるようなフォントも扱うし, 前の文字と次の文字の形の関係によって間隔をつめたりすることがある.

和文でも，レタリングや書道では，漢字の種類によって漢字の幅（横組）や高さ（縦組）を変更するのが普通だが，印刷用のフォントの世界では，漢字をフォント全体に共通の長方形（一般には正方形）の枠に収まるようにデザインし，固定幅で配置するのが一般的である．

また，仮名や記号を含まない漢字だけについていえば，縦組や横組の専用書体は本文用の書体として頻繁に使われる細明朝体を除いてはあまり作られていない．ここでも縦組，横組を意識せず，正方形の枠にうまく配置することを考える．

本研究で採用したのは，組合せアルゴリズムの時に使ったのと同じ，仮想的な xlimit, ylimit というエレメントを周囲から近付ける方法である．xlimit を左と右から近付け，ylimit を上と下から近付けて枠を作り，それがデザイン枠に変換されるように漢字をアフィン変換する．

このような単純な方法でも一部の漢字を除くと問題なく配置される．問題のある漢字には手で xlimit, ylimit という属性を与えて解決する．

第 6 章

書体の変更

6.1 書体の階層構造

スケルトンフォントには，肉付けアルゴリズムを変更して別の書体を生成できるという特徴がある．本章では細明朝体と太明朝体のように少数のパラメータの変更で実現できる書体変更ではなくて，明朝体とゴシック体のような変更点の多い書体変更について論ずる．

書体の変更は，以下のものを変更することによって実現される．

1. エレメントの種類
2. 肉付けアルゴリズム
3. プリミティブデータ

明朝体の飾りを一部変更するだけなら 2 の変更で十分であるが，明朝体からゴシック体への変更では，2, 3 の変更が必要になる．その場合，すべてのプリミティブのデータを再定義する必要はなく，かなりのプリミティブをそのまま使うことができる．

既に定義されたものの一部を再定義し直して使うオブジェクト指向におけるクラス定義を真似て書体を親子関係による階層構造で扱う (図 6.1)．

ある書体について，2, 3 を使う時は，その書体で定義があるかどうか調べ，なければ親の書体の定義を順に調べていき，書体 nil に至る．書体 nil は肉付けアルゴリズムは持っていないが，明朝体のプリミティブデータを持っている．

フォント gothic のサブフォント mini-gothic は以下のように宣言する．

```
(subfont mini-gothic gothic)
```

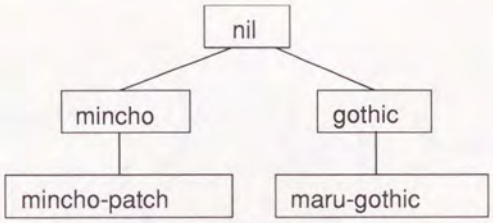


図 6.1: 書体の階層構造

上の宣言は内部的には, mini-gothic という Lisp のシンボルに parent というタグで gothic というシンボルを putprop することで実現される.

活字書体として明朝体と共によく使われるゴシック体の定義を例に書体変更に必要な定義について述べる.

6.2 ゴシック体のエレメント

6.2.1 エレメント

明朝体のエレメントを決定する際に参考にした [7] では, ゴシック体を表現するために 16 種類のエレメントを使用しているが, 本研究のシステムでは肉付けアルゴリズムの表現力を増したことにより, エレメントの種類を減らすことができると考えられる.

しかし, 明朝体とプリミティブの一部を共有する都合上, エレメントの種類を減らすことができないので, 明朝体と同じ種類のエレメントを使う. 各エレメントをゴシック体で肉付けしたものが, 図 6.2である.

ゴシック体の肉付けは, ほとんど太さに変化がない. そこで, 明朝体と比べるとデザインの自由度が小さくなり, curve1, curve2 を太さ, 太さ変化を定数として与えて呼ん

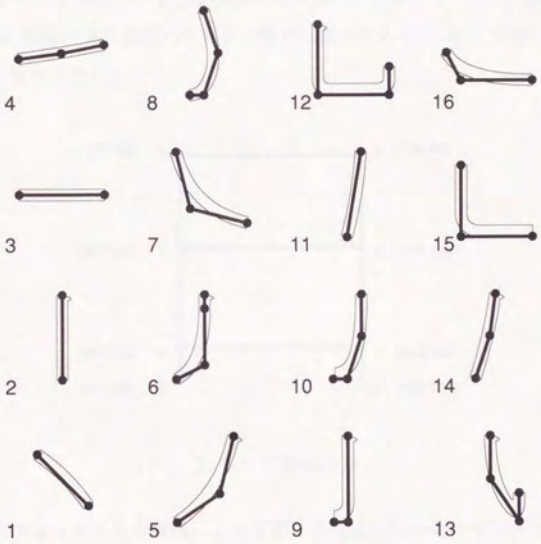


図 6.2: ゴシック体のエレメント

だ結果のアウトラインをつなぎあわせるだけで実現できる.

6.3 ゴシック体のプリミティブ

ゴシック体のプリミティブの大部分は明朝体のプリミティブを流用できる. ただし, 以下のものは再定義が必要となる.

1. 接続が変化するもの

   明朝体では図 6.3のように, 縦棒の終点へ横棒を接続させずに, 中途接続とした. これは, 縦棒の突出部分の大きさが微妙に変化するので, 飾りで表現するのが困難だったからである.

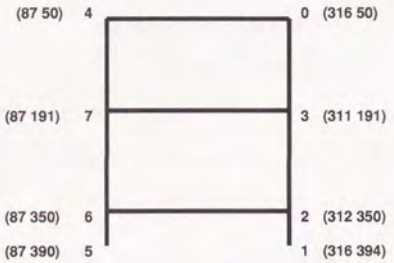


図 6.3: 明朝体の日

   本研究ではこのような接続によって飾りを生成しないようなゴシック体を製作したので, ゴシック体では制御点を共有する接続をするプリミティブを検出してゴシック体用に変換するプログラムを作って, 変換プログラムの出力ファイルを読み込んで使う (図 6.4).

2. エレメントの種類が変わるもの

   図 6.5の「うかんむり」のように, エレメントの種類が変わるプリミティブがある. これはプログラムで検出するのが困難なので, 人手で再定義する.

3. 座標が変わるもの

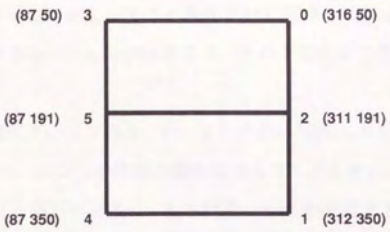


図 6.4: ゴシック体の日

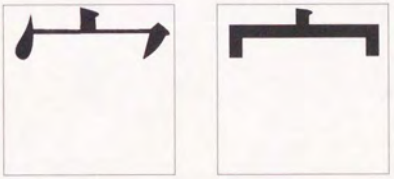


図 6.5: 書体によるエレメントの種類の変化

なるべく明朝体とプリミティブを共用するように肉付けアルゴリズムを決定するわけだが，座標を変更が必要になるものもある．

6.4 細丸ゴシック

明朝体のバリエーションが主に太さの変化だけに留まっているのに対し，ゴシック体にはいくつかのバリエーションが存在する．その１つとして細丸ゴシック体を製作する．

ふところの広い細丸ゴシック体は，ナールを中心に雑誌の本文用としてもよく使われるようになっている．ゴシック体用の組合せアルゴリズムを使い，肉付けだけを変更するだけで，図6.6の一番下のように，ある程度のものが実現できる．より丸ゴシックらしさを出すにはプリミティブの再定義が必要になるであろう．

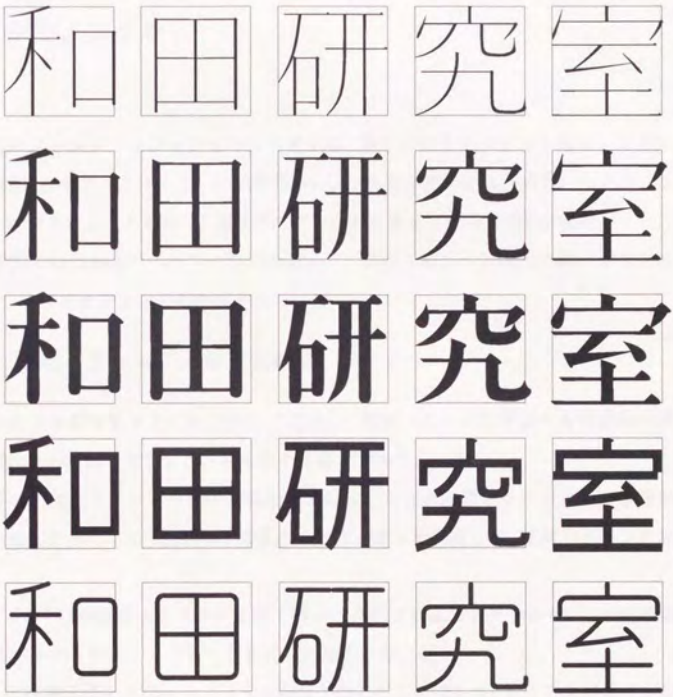


図 6.6: 書体の変更例

第 7 章

ひらがな，カタカナ

印刷用のフォントではひらがなやカタカナは，漢字とは別のフォントセットになっていて，独立にデザインされる．計算機用のフォントでは漢字とまとめた一つのフォントセットに含まれることが多く，清書プログラムも区別せずに扱う場合が多い．

本研究では既存のツールでの使用を考えて，漢字フォントと同様の扱いができるひらがな，カタカナフォントも用意する．

7.1 筆触パターンによるひらがな，カタカナ

[11] はひらがなをスケルトン形式で定義し，筆触パターンと呼ばれる仮想的な筆との接触面を多数描いてフォントを生成する研究である．

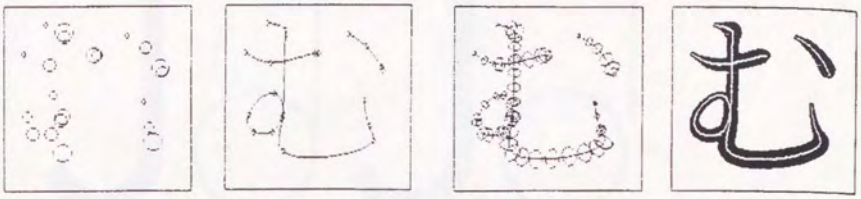
文字の定義はストロークの中心線が通る特徴的な点の座標とその点における筆触パターンの大きさによって指定し，特徴点の座標と太さを補間して，筆触パターンを置いていく．

図 7.1の (a) が制御点をその太さを半径とする円で表したものである．この制御点を結んだ 3 本のストロークがデータとして定義されている．

(b) は各制御点を 3 次のスプライン曲線で結んだものである．3 次のスプライン曲線といってもいろいろな定義があるが，この研究では区間距離パラメータ T を区間距離 d にする方法ではなくて d の 2/3 乗にする方法を取っている．

この 2/3 乗という数字は感覚実験によって決めたものだが，具体的には制御点が密集している付近での曲率を大きくする効果を持つ．そのため，大きな角変化がある図形も制御点をそれほど増やさずにデザインすることが可能になっている (図 7.2)．

(b) で計算された中心線に沿って筆の動きをシミュレーションして目的のドットイ



(a) Control point data (b) A spline that passes through the control points (c) Spots that give line thickness to the skeleton (d) Completed letterform with the skeleton

図 7.1: 筆触パターンによる生成過程

(a) proposed, $T = d^{2/3}$ (b) usual, $T = d$

図 7.2: 骨格の補間

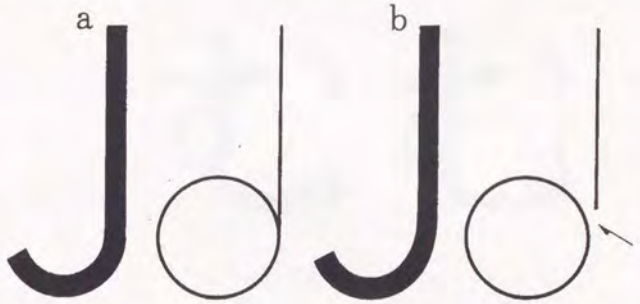


図 7.3: 直線から円周への接続

メージ (d) を得る. (c) は時間を離散化した時の各時点での筆と紙との接触部分 (筆跡パターン) を表している. 筆跡パターンは進行方向につぶれた (偏平率 0.8 程度) 楕円または八角形を用いる.

各制御点での太さが与えられているが, その補間は両端での曲率が 0 という境界条件のもとで 3 次スプラインを求めた後で, その一次導関数に主観係数 0.2 を掛け, 各区間内ではそれを 3 次で補間するという方法を採用している.

0.2 を掛けることによって, 太さの 2 次導関数は連続でなくなる. このため, 中心線が曲率連続であっても肉付けの結果得られるアウトラインは曲率連続ではない. 目は曲率の不連続には敏感なので (図 7.3) 避けたいが, 避けるために 0.2 を掛けるのをやめると図 7.4の左側のように補間し過ぎてしまう.

7.2 アウトラインへの変換

筆触パターンによるフォントの生成はプログラムの自由度が大きい点, 簡単なプログラムでいろいろな実験ができる点で有利である. しかし, 解像度に応じて生成する筆触の数を変化させる必要があるという問題点がある. また, 十分なめらかに見えるように筆触の数を増やすとアウトラインデータが大きくなってしまう. そこで, 中心線の座標と線の太さの補間法だけを [11] にならって, 筆触パターンと同様の効果を与えるア

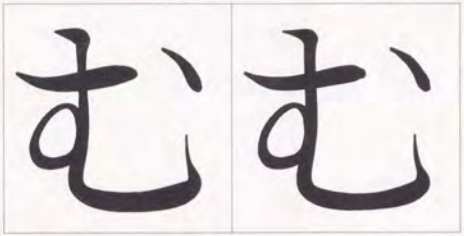


図 7.4: 太さの補間

ウトラインへ変換する方法をとる.

本研究のシステムではひらがなのストロークを制御点とその太さで表現するために, hira-long というエレメントを使う. hira-long は制御点が不定個で, エレメントの属性として, 制御点での太さを指定する hirawidth を与える必要がある.

hira-long を使った文字の定義は以下のように行なう.

```
(setq い '(((83 33)(96 48)(110 63)(102 75)(60 188)(66 264)(126 326)
           (130 316)(142 264)(173 205)(248 135)(288 130)(331 182)(335 267))
          ((hira-long (0 1 2) (hirawidth 4 12 16))
           (hira-long (3 4 5 6) (hirawidth 24 18 16 24))
           (hira-long (7 8 9) (hirawidth 22 8 0))
           (hira-long (10 11 12 13) (hirawidth 0 8 20 24)))))
```

筆触パターンと同様の効果を上げるには筆触パターンを並べた包絡線を求めることができればよいが, 楕円の包絡線を求めるのは難しいので, 楕円の厚みを 0 に近付けた極限, すなわち筆跡方向に垂直な線分を筆触として用いることにする. これは中心線に対する太さ変化であるから, 4.4.2で明朝体の肉付けに使った curve1 を使えば実現できる.

これを, 筆触パターンと重ねたのが図 7.5である. 太過ぎる場合を除くと, よい近似になっている.

7.3 濁音半濁音

ひらがな, カタカナのフォント見本を見ると, 濁音がつく文字については濁音がついた字(例「が」)だけがのっていて, 濁音がない文字(例「か」)はのっていないフォン

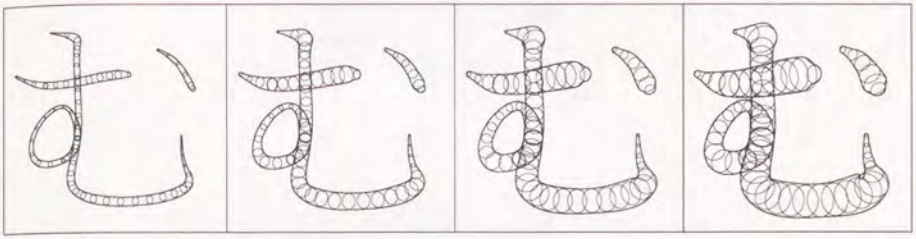


図 7.5: アウトラインへの変換

トセットがある．「か」は「が」から「゛」を除いて用いればよいからである．

本研究のシステムでは，濁音つき文字から濁音なし文字を作るのではなく，濁音なし文字と，濁音，半濁音のプリミティブを組合せて濁音つき文字を表現する．実際のフォントでは濁音の形が文字によって微妙に変化するが，形を固定してもそれほど不自然にはならない．

濁音半濁音は文字の右上に置くという規則があるものの，組み合わせる相手の形によって位置が微妙に変化する．組合せアルゴリズムで対応するのは難しいので，第10章で説明するスケルトンエディタを使った組合せを行なった (図 7.6)．したがって組合せは，直接変換を指定し，以下のような形で表す．

```
(setq が
  '(kana-joint '(#(1 0 0 1 0 0) #(1 0 0 1 269 43.5))
               '(か ゛)))
```

7.4 小さい文字の処理

濁音半濁音と同様に「ぁ」などの小さい文字はフォント見本に入っていない．これらの文字は大きな文字を縮小して用いることができるからである．

本研究のシステムでも，同様のことが実現できる．フォントの大きさ変化と太さ変化を独立に変えられるというスケルトンフォントの特徴を生かし，よりデザインの自由度を上げている．

小さい文字は，

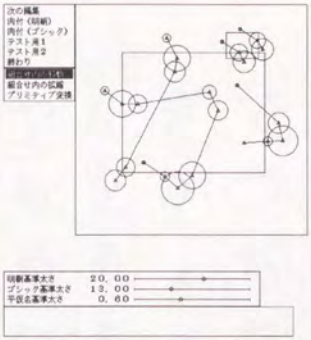


図 7.6: 濁音半濁音を含む字の編集

```
(setq ぁ '(smallkana あ))
```

のように定義するが, 関数 smallkana は,

```
(setq smallhira-affine
      (movexy 200 267
       (scalexy 0.65 0.65
        (movexy -200 -200))))
(setq smallhira-width 0.75)
```

という二つのシンボルの値に従い, 制御点の座標変換とエレメントの太さ変化を独立に扱う.

7.5 フォントの試作

この方法によって試作したひらがなのフォントを図 7.7に, カタカナのフォントを図 7.8に示す.

試作したフォント中のプリミティブについて, 統計的データを取ると, 表 7.1,7.2のようになっており, 少数の制御点によってフォントがデザインされていることが分かる.

図 7.7: 試作したひらがなフォント

図 7.8: 試作したカタカナフォント

表 7.1: ひらがなフォントの統計的データ

	最大	最小	平均
エレメント数	4 (わ, ほ, た, お)	1 (く他 11)	2.2
制御点数	28 (ゑ)	6 (へ)	16.3

表 7.2: カタカナフォントの統計的データ

	最大	最小	平均
エレメント数	4(キ, ホ, ネ)	1(レ, ヘ, フ, ノ)	2.4
制御点数	21(ホ, カ)	5(ノ)	12.8

第 8 章

出力装置を考慮したフォント生成

アウトラインをドットイメージに変換する際は量子化の影響を無視できない. プリンタの高解像度化によってある程度改善されるが, 量子化を意識せずにドットイメージ化するにはまだまだプリンタの解像度が不足している. たとえば, 400dpi のプリンタで 10 ポイントの文字を枠いっぱいに作る場合, ドット数は 55×55 ドットとなる. 細明朝体の横棒の太さは全体の枠の $\frac{1}{40}$ 程度だから, 単純にドットに変換すると場所によって太さが 1 ドットになる場合とと, 2 ドットになる場合がある. これではかなり不自然に見える.

スケルトンフォントで量子化の影響を回避するにははいくつかの方法がある. 第一はスケルトンフォントからヒントつきのアウトラインフォントに変換して, アウトラインフォントからドットイメージに変換する際, そのヒントを利用して基準線あわせ等を行なう方法である. この方法は, 異なる大きさのフォントに対して同じアウトラインフォントを使えるという利点がある.

第二は, 肉付けアルゴリズムに手を加え, 解像度を特定したアウトラインフォントを生成する方法である. この方法では, 異なる大きさのフォントでアウトラインフォントを共有できないという欠点があるが, アウトラインフォントのフォーマットによらない, きめ細かな処理を行なうことができる. 本研究ではこの方法を採用する.

8.1 アウトラインからドットイメージへの変換

アウトラインからドットイメージに変換するにはいくつかの方法がある.

1. ドットイメージで描画し内部塗りつぶし

図 8.1(a) のように，1 ドットの幅でアウトラインを描画し，その内部を塗りつぶす．1 ドット幅の線が必ず 2 ドットになるという欠点がある．

2. ドット中の黒白の面積比

   ドットイメージを拡大 / 縮小する際に用いられる方法をアウトラインに拡張したものである．黒い部分の面積の小さいフォントではしきい値を 0 にする，すなわち黒い部分が少しでもあれば黒くするという戦略が用いられることもある．

   図 8.1(b) のように，しきい値を 0 にすると線がどんなに細くなっても切れてしまうことはないが，黒のドットの割合が必ず大きめになってしまうという欠点がある．白黒のドット比率が元のアウトライン表現と平均的に同じになるようにするには，しきい値を 0.5 にする必要がある．

3. ドットの中心点の黒白

   図 8.1(c) のように，ドットの中心点がアウトラインの内部かどうかの判定を行ない，内部の場合はそのドットを黒にする．ドットの中心点がアウトラインの内部かどうかの判定は，Bezier 曲線を陰関数表現に直せば簡単に行なえるし，直線近似によって行なうとしても面積を求めるよりも高速に行なえる点で 2 の方法よりも有利である．また，黒白のドット比率の期待値がアウトライン表現と一致する点で 1 の方法よりもすぐれている．

量子化の影響を考慮したアウトライン生成を行なう場合，アウトラインの塗りつぶしアルゴリズムを知る必要があるが，アウトラインフォントを扱うシステムの多くは塗りつぶしアルゴリズムを公開していない．たとえば，PostScript インタプリタがどのアルゴリズムを採用するかは，PostScript の言語仕様としては規定されていないし，実際のインプリメントも異なっている．GNU の Ghostscript では幅が 1 ドットよりも細い線は表示されないことがあるが，OKI のプリンタおよび LW(Laser writer) では幅が 0 の線でも表示される．

以後は 3 のアルゴリズムを想定するが，2 のアルゴリズムもしきい値を $\frac{1}{2}$ とすると図 8.2のような場合を除いて塗りつぶし結果がほとんど変わらないので，そのまま適用することができる．1 のアルゴリズムの場合は線が太くなり過ぎないようにアウトラインを内側に半ドット分ずらす操作を加えれば，適用可能である．

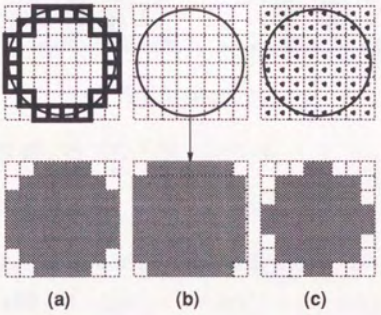


図 8.1: アウトラインからドットイメージへの変換

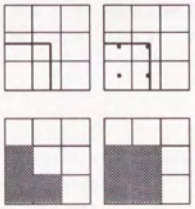


図 8.2: 面積比と中心点

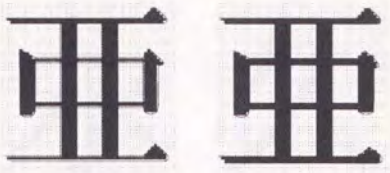


図 8.3: 線の太さ

8.2 出力装置を考慮したアウトライン生成

8.2.1 エレメント肉付け

エレメントの肉付けの際に最も考慮する必要があるのは，線の太さを均一にすることである．明朝体では横棒を除くとほとんどのエレメントの太さの最大値が，一致している．エレメントの太さが場所によって違うと図 8.3の左側のように不自然になるので，揃える必要がある．

これを実現するため，まず横棒や縦棒の太さをドットの大きさの整数倍に揃える．これだけで，かなりの確率で同じ太さになるが，更にアウトラインからの変換アルゴリズムに依存しないようにするために，更にアウトラインの境界がドットの境界に一致するよう，制御点を移動する．線の太さがドットの大きさの奇数倍か偶数倍かにより，制御点をドットの中心にそろえるのかドットの境界に揃えるかがきまる．

また，横棒や縦棒は直線でなく，中心部で細くなった Bezier 曲線で表すが，以下のように太さの変化が 1 ドットの幅よりも小さい時は直線を用いる．

```
;
; 縦棒の定義
;
(defelement mincho tate
    (lets ((dotsize (meshwidth (times minchowidth 0.9)))
           (w (times meshsize 0.5 dotsize))
           (x (grid (car points) dotsize))
           (y (grid (cadr points) dotsize)))
      (cond ((lessp (times 0.08 w) meshsize)
             (line2 x y w))
            (t
             (niku2 x y 0.4 0.4 w (times w 0.92)(times w 0.92) w)))))
```

図 8.4: 飾り

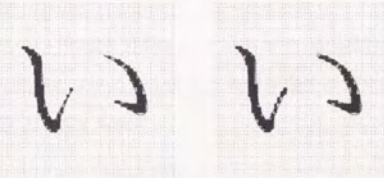


図 8.5: ひらがなカタカナ

8.2.2 飾り肉付け

制御点がドットの中心あるいは境界にそろえられていれば，飾りの形が場所によって異なる心配はない．ただし飾りの角の部分が丸くなってしまって，特徴がなくなる場合がある (図 8.4)．角の部分についてはアウトライン上の点を格子の特定の場所に来るよう補正し，角の特徴が出るようにする．

8.2.3 ひらがなカタカナ

ひらがなカタカナの場合は，太さを一定に揃えるという制約はない．ただし，面積比や中心点の黒白によってアウトラインからドットイメージへの変換する場合は，太さがドットの大きさ以下にならないような補正 (図 8.5) を行なう．

第 9 章

フォントサーバ

本研究では実用的なフォント作成を目的としているので, 試験的に表示するだけでなく, 清書システムなどの既存のツールから使うための手段を用意する必要がある. そのためには以下のような方法が考えられる.

- 既存のフォント形式へのバッチ式の変換

  既存のツールにフォーマットを合わせてアウトラインフォントあるいはドットイメージフォントをバッチ式に生成する方法. この方法では, 太さの変更や, 低解像度への対応といったスケルトンフォントの長所を殺してしまうが, 新たなツールを作成する必要がない点が有利である.

- Lisp による印刷ドライバ

  TEX を PostScript プリンタに出力する dvi2ps のように, 他の言語で書かれていた印刷ドライバを Lisp で書き直す方法. 太さの変更, 低解像度への対応, 部品合成などシステムの能力を最大限に引き出すことができるが, システム起動時に時間がかかる点, 一から印刷ドライバを書き直す必要がある点が問題である.

- Lisp によるフォントサーバ

  ネットワークを通じて, 漢字コードからアウトラインフォントへの変換のサービスを行なうサーバを Lisp で書く方法. そのサーバに他の言語で書かれた印刷ドライバがクライアントとしてフォント生成を依頼する.

  立ち上がりのオーバーヘッドは 1 回だけで, クライアントは待たされなくてすむし, 同じパラメータによる変換はキャッシュを使うことにより省略することもできる. ドライバの書き直しも少ない.

本研究では3にしたがって，フォントサーバを試作し，TEXの出力用にdvi2psクライアントを作成した．

9.1 サーバ

9.1.1 実現

UtiLisp/Cにはsyscallという組込み関数がある．これを使って，socketやselectといったソケット関係のシステムコールを呼び出し，ソケットを通常のLispストリームとして扱い，print, readなどのLisp関数を使えるようにした．

クライアントとサーバ間の通信は，拡張性を考えて，クライアントからサーバに送られた1個のS式をサーバがLispのS式としてそのまま実行して，その結果を一行のテキストとして返す方法をとった．このため下の例のように，telnetを使ってクライアントプログラムの代わりに直接サーバと通信することが可能になる．

```
moe[1] telnet moe 2002
Trying 130.69.168.19 ...
Connected to moe.wadalab.t.u-tokyo.ac.jp.
Escape character is '^]'.
(setfont Wadalab-mincho-20)
Wadalab-mincho-20
(gettype1 亜 15.0)
<10bf317079ca388f179c5c9a49ea3.. 途中略..0f96be>
(gettype1 行 15.0)
<10bf317079ca388f197959240ddf2.. 途中略..af9f71>
(gettype1 行 20.0)
<10bf317079ca388f19644a0f37007.. 途中略..1addea>
(car '(1 2))
1
(car 'a)
```

9.1.2 クライアントごとの環境

肉付けアルゴリズムの中で, minchowidth のようなグローバルなシンボルの値を参照している所がある. クライアント間のコンテクストスイッチは Lisp の S 式を一つ評価する間は起きないから, 毎回,

```
(progn (setq minchowidth 15.0) ... ( gettype1 行 20.0))
```

といったものを送ることにすると, 他のクライアントの影響を受けない.

しかし, このようにパラメータを毎回送るのは通信量が問題となる. そこで, 各クライアントごとの環境を用意して, 一度送った環境は通信の間覚えておく.

ユーザごとの環境を実現する際の原則は, 値の参照を通常のグローバルなシンボルとして実行できるようにすることである. つまり, 値を設定する際には, (setq minchowidth 15.0) の代わりに (defvar minchowidth 15.0) のように書かなくてはいけないが, 値の参照は minchowidth と書けるようにする. この原則にしたがって, 以下の方法でクライアントごとの環境を実現した.

サーバはそれぞれのクライアントとの接続が開始した時に, 次のようなリストを作る.

```
(let
    nil ; (1)
    nil ; (2)
)
```

変数を設定するマクロは, defvar はこのリストの (1) の部分を書き換える. たとえば, 初期状態に対して (defvar minchowidth 15.0) を実行した時のリストは次のように変わる.

```
(let
    ((minchowidth 15.0))
    nil
)
```

クライアントから送られたS式を評価する時は，(2) の部分を評価する式で置き換えて，リスト全体を評価する．すると，(1) の部分のシンボルが束縛された環境での評価になる．

ただし，次のような式の評価はうまくいかない．

```
(progn (defvar minchowidth 15.0)(times minchowidth 0.5))
```

結果が，クライアントの環境ではなくてサーバの環境の minchowidth に 0.5 を掛けたものとなるからである．一回の通信で変数の設定と利用を行ないたい時は，次のように書く必要がある．

```
(progn (defvar minchowidth 15.0)
       (let ((minchowidth 15.0))(times minchowidth 0.5)))
```

9.1.3 プリミティブの圧縮

スケルトンデータはイメージデータやアウトラインデータと比較すると，使用メモリ量が少ないので，基本的な書体についてJISの第1第2水準の漢字すべてを作成するのに必要なデータを最初に一度にメモリ上に読み込んでおく．

プリミティブの表現形式は，利用者がエディタで修正することも考えて，理解しやすい形式(標準形式)にしているが，そのために冗長な表現になっている．そこで，プリミティブについては，圧縮した形式で記憶しておき必要に応じて標準形式に変換して組合せ，肉付けを行なうことにした．プリミティブの標準形式は，以下の構造をしている．

```
( ((100 100)(200 100)(300 100 (link-ok t))) ; 点の座標指定
  ((yoko (2 3)) ; 制御点だけ
   (tate (0 1) (link 2)) ; 連結情報
   (hira-long (4 5 6)(hirawidth 8 9 10))) ; 連結情報以外の属性
  (xunit . 200) ; プリミティブの属性
)
```

各部分の圧縮方法について述べる．

エレメントの種類	制御点の数	中途連結の数	制御点	中途連結
tate	2	3	(2 4)	(3 5 6)
0x01	0x02	0x03	0x02 0x04	0x03 0x05 0x06

(tate (2 4)(link 3 5 6))

図 9.1: エレメントの圧縮

1. 点の座標

   点の座標は，多くの場合X，Y共に 0 から 399 までの整数になっている．したがって，3 バイトあれば座標を表現できるので，長さが 3 の文字列に変換する．点には属性をつけることができるが，そのうち link-ok という属性はスケルトンエディタで編集中に使うものなので無視するが，それ以外の属性がついている点は圧縮しない．

   すべての点が圧縮可能の時には文字列をつなげたものをリストの代わりに用いる．文字列は長さが長いほど文字当たりの記憶容量が小さくて済むからである．

2. エレメント

   エレメントは通常はエレメントの種類，制御点のリスト，連結情報からなる．エレメントの種類は，制御点の数は 256 以下なので，それぞれ 1 バイトで表す．そこで，図 9.1のようになる．

   かなのエレメントのように，連結情報以外の属性を持つエレメントは圧縮しない．すべて圧縮可能な時は，圧縮して得られた文字列をつなげたものを使う．エレメントの中に圧縮可能なものと圧縮できないものが混在している時は，圧縮可能なものを圧縮して得られる文字列をつなげたものと，それ以外のエレメントをつなげたリストを使う．

3. 属性リスト

   属性リストは圧縮しない．

点の座標とエレメントの両方とも文字列に圧縮できた時は，これらの文字列をつなげて用いる．属性リストがない場合は，プリミティブ全体が一つの文字列で表される．

以上の圧縮を行なったところ，JIS 第 1 第 2 水準で使われるプリミティブ，ひらがなカタカナすべての 1377 個のプリミティブについて，圧縮前は 837516 バイトだったメ

モリ消費が204648バイトに改善された．圧縮率はほぼ $\frac{1}{4}$ である．

9.1.4 キャッシング

フォントの生成には時間がかかるので，一度生成したフォントに関する情報を覚えておくことにより，次回の生成の速度を上げることは意味があることである．

バッチ式に毎回動かすのならば，UtiLisp/C のヒープ拡張機能を使って，無制限に覚えても問題ない．しかし，サーバクライアント方式ではサーバにデータが溜る一方なので，ヒープを増やすと容易にシステムの限界に達してしまう．

そこで，キャッシングに制限をもうけて，メモリの利用状況によって制限を変化させるという方法をとる．

データのうちで次のものは，削除しない．

- 書体 nil のプリミティブ，組合せ
- 個人個人のユーザスペース

また，loadfont によって得られたファイルの内容は，そのファイルを使っているユーザがいる限り削除されない．ただし，ユーザがいない時も，しばらくは残しておき，メモリに余裕がなくなった時に削除する．

次のものは，キャッシュの対象になる．gc-hook によって gc の発生時にメモリの使用状況をチェックしキャッシュのサイズを動的に変える．

- 組合せ後のデータ
- 肉付け後のデータ

9.2 dvi2ps クライアント

普通に TEX で使うフォントを変更なしに使う限りでは，サーバクライアント方式のメリットは余りない．TEX のフォントに対応したアウトラインフォントをバッチ式にあらかじめ作ってファイルにしておき，dvi フィルタがそのファイルを使えばよいからである．

そこで，TEX の special にサーバに対するコマンドを埋め込み，フォントのさまざまな変更を行なえるようにする．現在の dvi2ps で使われている special は PostScript

ファイルの埋め込みに使われているが，これと区別するために special の中身の先頭の 7 文字が utilisp で始まる場合は special の終わりまでサーバに対するコマンドとして解釈することにする．

しかし，dvi2ps の special 解釈の部分を書き換えてみても変更したつもりが，次のページになってはじめて現れたりするという現象が出た．これは dvi フィルタは普通はファイルの先頭，あるいはページの先頭で使われるフォントを先読みして定義するのに対して，special は普通に処理するようになっていたためである．そこで，special の内容もフォントの先読み時にチェックして，サーバへの命令の時にはその時点で処理するようにした．

現在扱える機能は次のようなものである．

- 書体の変更
- 組合せによる漢字生成
- プリミティブ指定による漢字生成

普通使われない文字を使うには次のように書く．

```
\special{utilisp (defvar gaijione
 '(((15 143)(380 143)(56 58)(261 58)(128 215)
   (323 215)(125 291)(319 291)(128 365)
   (323 365)(161 13)(161 143)(316 26)
   (177 212)(7 302)(249 152)(281 207))
  ((yoko (0 1) (link 11))(yoko (2 3))
   (yoko (4 5))(yoko (6 7))(yoko (8 9))
   (tate (10 11))(tate (4 8) (link 6))
   (tate (5 9) (link 7))(hidari (12 13 14))
   (ten (15 16)))))}
\special{utilisp (defvar gaijitwo '(tate 此 魚))}
```

者の旧字体は JIS 第 1 第 2 水準には入っていないが，
プリミティブを定義すると「\gaijione 」のように
使うことができる．また，「此」と「魚」を組み合わ

せて第3水準漢字「\gaijitwo 」も使うことができる.

このTeXファイルを処理して得られる出力は以下のようになる.

> 者の旧字体はJIS第1第2水準には入っていないが, プリミティブを定義すると「者」のように使うことができる. また,「此」と「魚」を組み合わせて第3水準漢字「鮓」も使うことができる.

第 10 章

開発環境

システム設計者用の開発環境と，ユーザがフォントを製作する際の開発環境の 2 種類が考えられる．

10.1 システム開発環境

肉付けアルゴリズムや組合せアルゴリズムを変更した時に，変更による生成されるフォントの変化を確認したいという要求がある．変更を即座に確認するために，X-window 上に表示するプログラムと，PostScript プリンタに印刷するプログラムを用意した．

10.1.1 X-window 表示プログラム

UtiLisp/C では，C で Lisp の関数を書いておき，そのオブジェクトファイルをダイナミックロードによって読み込んで使う手段が用意されている．この機能を使って，Xlib にあるすべての C の関数に対応する Lisp 関数を作れば X-window の機能をすべて使うことができるが，必要とする機能はそれほど多くないので，機能を限った Lisp 関数を記述することにした．記述したのは次のような関数である．

1. init_window width height

   X-server との connection を確立し，幅が width ドットで高さが height ドットのウィンドウを一つだけ作る．

2. close_window

   X-server との connection を切る．

3. drawlines pointlist

   点(carにX座標, cdrにY座標をfixnumで入れたもの)のリストを渡して, それらの点を順に結んだ線を引く.

4. fillpolygon pointlist

   点のリストを渡して, それらの点によって生成される多角形の内部を塗りつぶす.

5. checkevent

   ウィンドウ上でキーボードやマウスなどのイベントが発生した時に, イベントの種類と付加情報をリストの形で返す.

X-windowの標準サーバにはBezier曲線を描画する機能がない. サーバの機能を拡張することによってこの機能を加えることができるが, 速度はそれほど必要ないので, Lispで線分列に分解して描画することにした.

表示させたいのは, 線分とBezier曲線からなる閉曲線のリストである. Bezier曲線は図10.1のように再帰的に分割することができる. ドットの大きさに対してある程度の大きさになったところで, 線分で近似して曲線全体を線分列に直す.

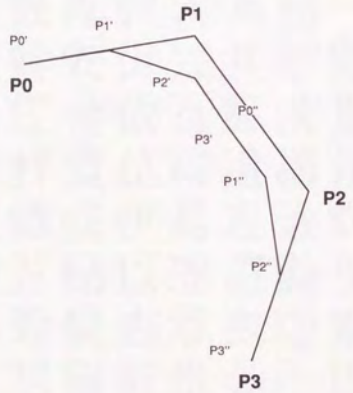
Bezier曲線の部分を線分列に直して, 多角形となった各閉曲線を塗りつぶせばフォントを表示することができる.

Xサーバの制約から線分の両端の座標が格子点上になければいけないため, 曲線が不自然になる部分があるが, おおむね問題ない.

10.1.2 PostScriptプリンタへの出力

X-window上の出力でも解像度としては十分なのだが, データの入力誤りのチェックなどのために, 漢字を一度に大量に生成して, 比較しながら見るという用途のため, プリンタへの出力も必要である.

PostScriptプリンタを使えば, Beizer曲線の描画が, 容易に実現できる. そこで, 開発当初からPostScript出力のプログラムを作り, ファイル名や日付も表示させるように改良され, 最終的には, 図10.2のような形式で出力するようにした.



$$P0' = P0, P1' = \frac{1}{2}P0+\frac{1}{2}P1, P2' = \frac{1}{4}P0+\frac{1}{2}P1+\frac{1}{4}P2, P3' = \frac{1}{8}P0+\frac{3}{8}P1+\frac{3}{8}P2+\frac{1}{8}P3$$

$$P0'' = P3', P1'' = \frac{1}{4}P1 + \frac{1}{2}P2 + \frac{1}{4}P3, P2'' = \frac{1}{2}P2 + \frac{1}{2}P3, P3'' = P3$$

図 10.1: 再帰による Bezier 曲線描画

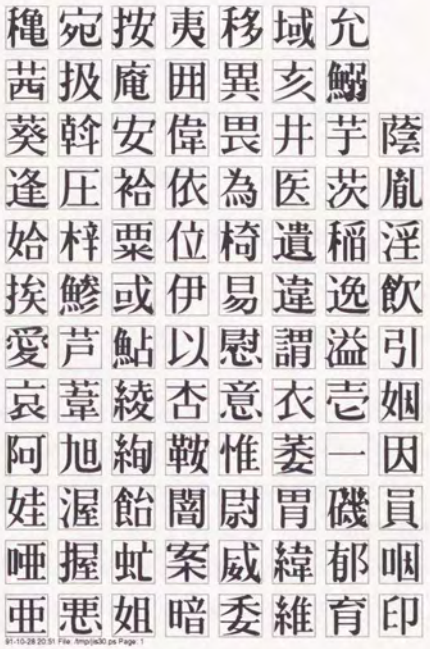


図 10.2: PostScript 出力の例

10.2 スケルトンエディタ

ユーザが次のような使い方をする場合は，既存のテキストエディタを使って，TEX の special を書くだけで実現できる．

- あらかじめ用意された書体を使う
- 組合せアルゴリズムにしたがって，文字を製作して使う
- 書体の作成，修正を行なう

しかし，次のようなことをする際には特別なツールが必要になる．

- プリミティブとして文字を定義する
- 組合せアルゴリズムによる文字の配置を修正する

当初，明朝体のプリミティブを入力するために簡単な X-window インターフェースを利用したスケルトンエディタを制作したが，ユーザが使うには不親切なもので，機能の拡張も難しいものだった．

そこで，Common Lisp 用の X-window インターフェース CLX を UtiLisp に移植し，スケルトンエディタを作り直すことになった[1]．スケルトンエディタは最初は，プリミティブの入力を行なうだけのものだったが，機能を拡張していくうちに，スケルトンエディタというよりはユーザによるフォント開発のための総合環境というにふさわしいものになった．スケルトンエディタの内部構造などについて説明しないが，システムの特徴であるユーザによる書体製作と切り離すことができないツールなので，使い方だけ述べる．

図 10.3が編集の対象を指定せずに立ちあげた場合のスケルトンエディタの画面である．画面の右上の 400 × 400 ドットの部分が，スケルトンや肉付け結果を表示するウィンドウで，左にならんでいるのがメニュー，下の部分がスライド方式のパラメータ指定ウィンドウである．

スケルトンエディタには，プリミティブ編集，漢字組合せ編集の二つのモードがある．指定された編集対象が漢字組合せの形をしている場合は自動的に漢字組合せ編集

[1]実際のプログラム制作は修士 2 年の石井裕一郎君が担当した [24]

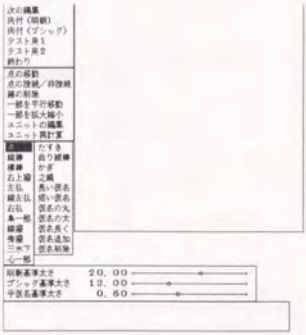


図 10.3: 初期画面

モードに入り，それ以外の場合はプリミティブ編集モードに入る．漢字組合せで実現していた漢字をプリミティブの形に変換することもでき，その時はプリミティブ編集モードに入る．

10.2.1 プリミティブ編集

プリミティブ編集モードは，図 10.4のような画面をしている．次のような機能が用意してある．

- 新たなエレメントの配置

  図 10.4の左下にエレメントの種類が並んでいるので，このうちの一つをクリックして，主ウィンドウ上に制御点をクリックしていく．制御点の数が不定なかなエレメントでは最後の制御点を指定した後で終了のために別のボタンをクリックする必要があるが，他のエレメントでは必要な点をクリックすると，自動的にこのモードを抜ける．

- 制御点の移動

  「点の移動」をクリックしてから，注目する点の近くでマウスの左ボタンを押し，そのまま移動先までマウスを動かしボタンを離すと，点が移動する．左ボタン

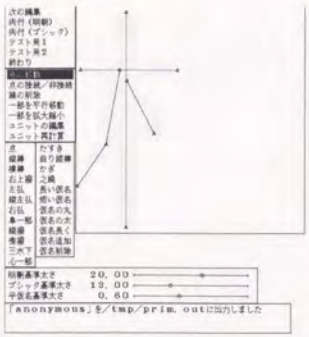


図 10.4: プリミティブ編集モード

の代わりに中ボタンを使うと，同じエレメントの隣の点とX座標，Y座標を揃える処理を行なう．これは，縦棒／横棒の指定に便利である．

- 部分の移動，拡大／縮小 「部分を移動」を選ぶと長方形領域内の制御点を一度に移動させることができる．また，「部分の拡大」を選ぶと，長方形領域内の制御点を長方形のいずれかの頂点を中心にX，Y方向独立に拡大／縮小させることができる．

- 接続情報の変更

  図10.4中の制御点を見ると，三角で表現されている点と四角で表現されている点があるのがわかる．四角で表現されている点が，他のエレメントの中途に接続している点である．スケルトンの内部表現ではどのエレメントの中途に接続するかという形で表現されるが，画面上で確認するのは難しいので，ある距離以下のエレメントすべてに中途接続していると見なす．

- 肉付け結果の確認

  図10.4の「肉付け結果の確認（明朝）」をクリックすると，明朝体で肉付けした結果が，図10.5のように表示される．下のスライド部分を変更して，太さを変更した肉付けを確認することもできる．

図 10.5: 肉付け結果の確認

現在のところ，明朝体をゴシック体しか用意されていないが，ユーザの作る書体も指定できるようにすることになっている．

- 基準高さ基準幅変更

  プリミティブに付加させることのできる情報はいろいろあるが，その中で組合せアルゴリズムのために特に重要な基準高さ，基準幅は，変更による組合せの変更も確認しながら画面上で編集できるようにしている．

  図 10.6がその様子である．基準高さ，基準幅は長方形で表している．長方形の高さ，幅を変更すると，右側のウィンドウ上に編集中のプリミティブを含む漢字が組合せを再計算して表示される．

- かなエレメントの太さ変更

  かなを表現するためのエレメントは，各制御点における肉付けの太さを指定する必要がある．そのため，それを編集する機能が用意してある．

  図 10.7が変更中の様子である．注目しているエレメント中の制御点の太さを，円で表している．その円の半径をマウスで指定することによって太さを指定することができる．

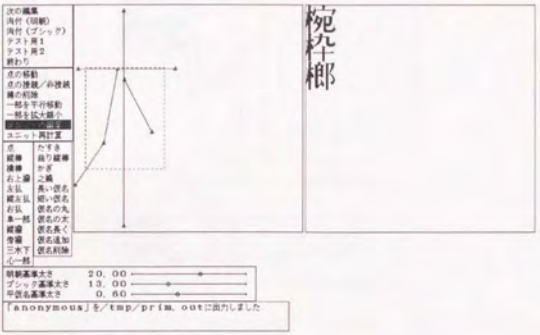


図 10.6: 基準高さ基準幅変更

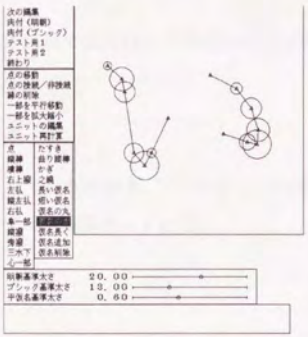


図 10.7: かなエレメントの太さ変更

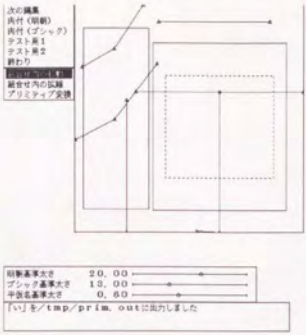


図 10.8: 組合せ編集モード

10.2.2 漢字組合せ編集

漢字組合せ編集は図 10.8のような画面をしている. 次のような機能が用意してある.

- 部品の移動

  組合せを構成する部品を, 大きさは変えないまま X, Y 方向に移動する.

- 部品の拡大 / 縮小

  部品の場所をおおまかに表す長方形の 1 頂点を中心に部品を X, Y 独立に拡大縮小する.

- プリミティブへの変換

  部品組合せで実現されていたものを, プリミティブに変換する. これを選ぶと変換後自動的にプリミティブ編集モードに入る.

第 11 章

結論

11.1 評価

本研究では，多様な書体に適用できる漢字スケルトンフォントを実現するために，肉付けをエレメントの肉付けと飾りの肉付けに分けて，それぞれをパラメータではなくプログラムで指定する方法を提案し，それに基づいて，明朝体とゴシック体の肉付けを行なった．自由度の大きな肉付けアルゴリズムを採用したため，エレメントの種類は16種類に抑えることができ，太さの変化もパラメータの調節で対応できた．

更に，漢字を偏や旁などの部品からなる階層構造で定義して，この定義から部品の配置をアルゴリズムによって決定した．階層構造を導入したことによって，JIS 第 1 第 2 水準の漢字の定義に必要な部品の定義を 6355 個から 956 個に，部品に含まれるエレメントの数を 87246 個から 5861 個に減らし，デザインのコストを大幅に削減できた．

ひらがなやカタカナのフォントも，過去の研究を参考にして漢字と同様に扱えるようにシステムに取り入れることができた．

スケルトンフォントの特徴を生かして，かつ現存するツールへの変更を最小限にとどめるために，Lisp 上にフォントサーバを動かし，クライアントプログラムがソケットを通じてサーバと通信してフォント情報をやりとりするという方法を採用した．この方法に基づいて，TEX の出力に用いられる dvi2ps を改造してクライアントとし，ユーザによる漢字合成，書体変更を行なえるようにした．本論文自体もこのシステムを用いて印刷されている．

フォントデザイナではない一般の利用者が容易にフォントをデザインできるようにするための道具として，多機能のスケルトンエディタを導入した．

11.2 今後の課題

本研究で製作したのは，明朝体とゴシック体だけだが，それ以外の書体についてもこの方法が適用できるかどうかを試す必要がある．毛筆書体への適用を 92 年度卒論生の飯尾淳君が実験中だが，部品組合せによって実行できるか，必要なエレメントの種類などはまだ分かっていない．

またスケルトンエディタでの修正が必要な組合せの率も減らしたい．現在のアルゴリズムは実行速度を意識した単純な方法を用いているが，基準高さ，基準幅と位置関係の制限という枠組の中でも，別の方法を試してみる余地がある．

システム及び作られたフォントを多くの人に使ってもらって，その批判を吸収してよりよいものを作っていきたい．

謝辞

本研究は様々な人の助力の上に成立した.

指導教官の和田英一先生からは, 研究テーマの決定に際に有効な助言を頂いた. その後も組合せアルゴリズム等, いろいろなアイデアを出して頂いた. 更には, JIS 第 2 水準の漢字の組合せデータの入力, 組合せ結果のチェックという大変手間のかかる作業をして頂いた. この協力がなかったら, 第 2 水準の漢字の作成は論文に間に合わなかったであろう.

助手の岩崎英哉氏からは, UtiLisp/C のバグ取りや, UtiLisp/C のコンパイラの製作, CLX の UtiLisp への移植といった助力を頂き, プログラム開発に大いに役立った.

大学院生の石井裕一郎君はプログラミングシンポジウムのデモと, フォントの実用化には欠かせないスケルトンエディタの製作を担当してくれた. こちらの都合で, スケルトンエディタで使うデータを変更したり, 仕様の変更を頼むことが何度もあったが, そのたびに素早く対応してくれた.

大学院生の竹内幹雄君は石井裕一郎君と一緒にデータ入力を手伝ってくれた.

元研究生の長橋賢児君は, CLX の UtiLisp 移植の中心となって作業し, プログラミングシンポジウムのデモに間に合わせてくれた.

91 年度卒論生の丹明彦君は, ひらがなカタカナの肉付けに関していろいろ実験して, 現在の肉付けの基礎を作ってくれた.

NTT の斎藤隆文氏からは, フォント全般に関して参考となる資料をいくつか頂いた.

以上の方々に心から感謝する.

参考文献

[1] 田中一男, 乙田清次, 岡田守：文字フォントの性質に着目した任意倍率文字サイズ変換法, 信学論 (D), J69-D, No. 3, pp.460-466 (1986).

[2] M. Plass, M. Stone : Curve-Fitting with Piecewise Parametric Cubics, SIGGRAPH '83 Conf. Proc., pp.229-239 (1983).

[3] THEO PAVLIDIS : Curve Fitting with Conic Splines, ACM Transactions on Graphics, Vol. 2, No. 1, pp.1-31 (1983).

[4] Roger D.Hersch, Claude Betrisey : Mode-based Matching and Hinting of Fonts, SIGGRAPH '91 Conf. Proc., pp. 71-80 (1991).

[5] 荻原ほか：内外挿を用いた高品位なゴシック体日本語超大形文字の自動生成, 信学論 (D), J72-D-II, No. 9, pp. 1388-1396 (1989).

[6] 荻原ほか：ゴシック体超大形日本語文字の自動生成, 信学論 (D), J72-D-II, No. 7, pp. 1048-1055 (1989).

[7] 上原徹三, 国西元英, 下位憲司, 鍵政秀子, 菊池純男：ストローク種別に基づく漢字形状生成方式, 情報処理学会論文誌, Vol. 31 No. 2, pp. 209-218 (1990).

[8] 菊池純男, 大山恵三子, 高橋栄：字体のパラメトリック基本エレメント貼付け方式による高品質文字形状生成方式, 情処学第 29 回全大, 3J-7 (1984).

[9] 陳和明, 小沢慎治：多様な明朝体文字の規則的な生成, 信学論 (D), J72-D-II, No. 9, pp. 1423-1431 (1989).

[10] 東潤一：日本語フォントクリエータ, 卓上出版シンポジウム報告集, pp. 91-96(1988).

[11] 坂元宗和, 高木幹雄：高品質明朝体ひらがな・カタカナフォントの計算機による生成, 信学論(D), J68-D, No. 4, pp. 702-709 (1984).

[12] 張憲栄, 真田英彦, 手塚慶一：漢字楷書毛筆字体の計算機による生成, 信学論(D), J67-D, No. 5, pp. 599-606 (1984).

[13] Adobe Systems Incorporated : Adobe Type 1 Font Format, Addison-Wesley Publishing Company, Inc(1990).

[14] 上原徹三：フォント関連技術の現状と課題, 情報処理, Vol. 31 No. 11, pp.1570-1580 (1990).

[15] Tung Yun Mei : LCCD, A LANGUAGE FOR CHINESE CHARACTER DESIGN, Stanford Department of Computer Science, Report No. STAN-CS-80-824 (1980).

[16] 下位 憲司, 上原 徹三：骨格ベクトル文字からアウトライン文字への変換方式, 情報処理学会論文誌, Vol. 30 No. 9, pp. 1111-1118(1989).

[17] 奥村彰二, 前田正弘：漢字画像から文字要素の自動抽出, 情報処理学会論文誌, Vol. 32 No. 1, pp. 50-61(1991).

[18] 佐藤 敬之輔：漢字(上, 下), 丸善(1973).

[19] 近山隆：Lisp 処理系の構成法に関する研究, 1981 年度東京大学工学部情報工学博士論文(1982).

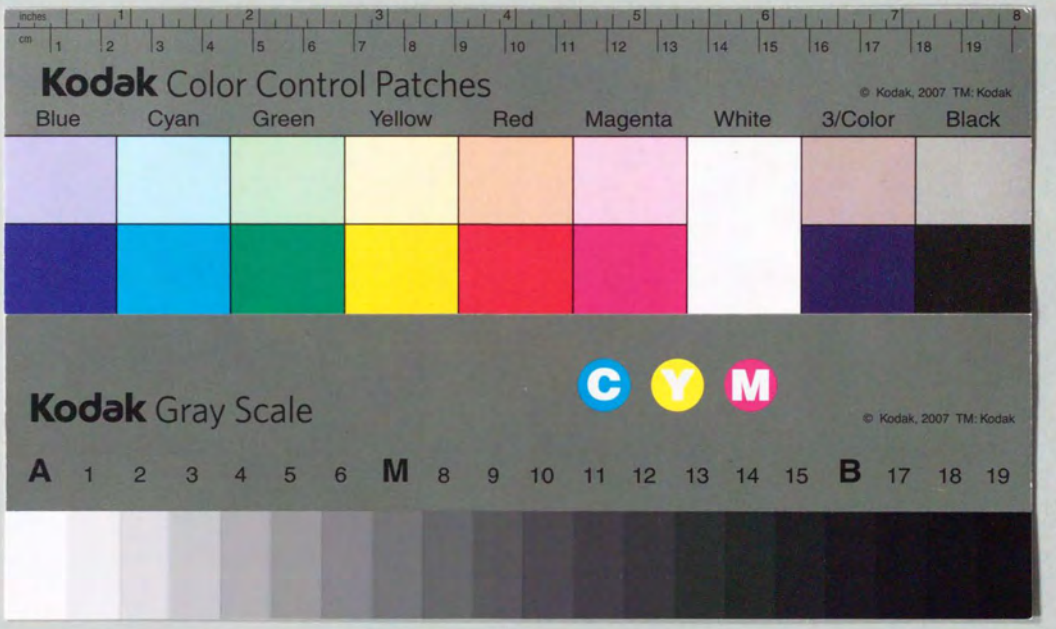
[20] 近山隆：UtiLisp システムの開発, 情報処理学会論文誌, Vol. 24, No. 5, pp. 599-604(1983).

[21] 田中哲朗：SPARC の特徴を生かした UtiLisp/C の実現法, 情報処理学会論文誌, Vol. 32, No. 5, pp. 684-690(1991).

[22] 丹明彦：ひらがなフォントの合成, 1990 年度東京大学工学部計数工学科卒業論文(1991).

[23] 田中哲朗, 石井裕一郎, 長橋賢児, 竹内幹雄, 岩崎英哉, 和田英一：漢字スケルトンフォントの生成支援システム, 第 32 回プログラミングシンポジウム報告集, pp. 1-8(1991).

[24] 石井 裕一郎：Lisp による漢字フォント作成支援ツールの開発, 情報処理学会記号処理研究会資料, SYM 61-4 (1991).


inches
cm
Kodak Color Control Patches
© Kodak, 2007 TM: Kodak
Blue
Cyan
Green
Yellow
Red
Magenta
White
3/Color
Black
Kodak Gray Scale
C
Y
M
© Kodak, 2007 TM: Kodak
A 1 2 3 4 5 6 M 8 9 10 11 12 13 14 15 B 17 18 19