漢字スケルトンフォントの生成支援システム
Kanji Skelton Font Creation Support System

東京大学 工学部
田中 哲朗, 石井 裕一郎, 長橋賢児,
TANAKA Tetsurou, ISHII Yuuichirou, NAGAHASHI Kenji,
竹内 幹雄, 岩崎 英哉, 和田 英一
TAKEUCHI Mikio, IWASAKI Hideya, WADA Eiiti

概要

漢字フォントの表現にスケルトン形式を使うと, 変形や書体の変更が容易に行なえる. 問題となるのは, スケルトン形式にどう肉付けすると見ばえのするフォントが得られるかである. 接続点に飾りのつく明朝体は特に難しい.

我々は UtiLisp を使ってスケルトン形式から直線と 3 次の Bezier 曲線からなる肉付けデータを作る実験を行なっている. Lisp を用いたことによって自由度の高いシステムを作ることができ, 従来困難とされてきた明朝体に対しても質の高い肉付けデータを作ることができた.

また, スケルトン形式の特徴を生かして漢字を偏や旁などの部品の合成として定義し, 少数の部品から多数の漢字を生成する実験も行なっている.

1 はじめに

計算機の印刷機としての役割はますます大きな比重をしめている. 漢字の印刷には, 当初ドット数の少ないドットイメージフォントが使われた. 大きさは固定で書体も一種類にとどまっていたが, 高解像度プリンタの普及によりさまざまな大きさ, 複数の書体への要求が高まってきた. これを, ドットイメージフォントの形で表現すると, 莫大な記憶容量が必要になる.

そこで, プリンタにアウトラインフォントを内蔵したり, 計算機上のアウトラインフォントを印刷時に計算機上でドットイメージに変換する製品も出てきた. アウトラインフォントは拡大縮小, 斜体化を容易に行なうことができるので, 大きさの違うフォントを別々に用意する必要がなくなり, 記憶容量を減らすことができる.

扱う書体は明朝体とゴシック体の二種類であることが多いが, 最近はもっと多くの書体を含んだ製品も出まわっている. ただ, これもアルファベットのフォントの豊富な種類に比べると見劣りがする. また, 多数の書体を扱うとフォントの記憶に大量のディスクが必要になる.

アウトラインフォントの最大の欠点はフォントの製作にかかる労力が大きいことである. アルファベットのフォントなら数十文字だが, 漢字のフォントは JIS 第一水準の漢字だけで 2965 文字になるので, フォント製作の労力は軽視できない.

アウトラインフォントの製作には通常, 紙に打ち出された文字をイメージスキャナで取り込み, それをなぞってアウトラインの制御点を選んでいく方法が用いられる. アウトラインの制御点の選択はある程度までは自動的に行なうことができるが [1], うまくいかない場合も多く, 人間が修正を加える必要がある.

また, すでにあるアウトラインフォントに対して, 太さや肉付けの方法を変更することはできない. 変更するには新たにフォントを作り直す必要がある.

このような問題はスケルトンフォントを用いれば解決することができる. スケルトンフォントは文字をストロークに分け, それぞれのストロークをいくつかの制御点によって定義するものである. 表示の際には, ストロークの属性や連結の具合を考慮して肉付けデータを作り, それを表示する.

ゴシック体に対してスケルトンフォントを作る試みは, [3] などさまざまな所でなされている. ゴシック体は太さの変化がそれほどなく, 連結の際に飾りがつかないので, あまり工夫しなくてもかなりのレベルのフォントが得られる.

一方, 漢字フォントの中でもっとも使用頻度が高い明朝体のスケルトンフォントの試作も, [2] で試みられている. これを見るといくつかの文字についてよい字ができているが, 続報がないのですべての漢字をこの方法でうまく表現できるかどうかわからない.

我々が試作しているシステムでは, 肉付けに大きな

自由度をもたせることにより, さまざまな書体に対応できるようにしている. 現在は主として明朝体のフォント生成を実験している.

2　全体の構成

2.1　漢字の階層化

我々のシステムでは漢字を次のように階層化して扱う (図 1).

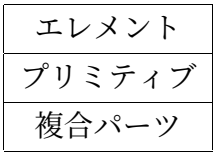

図 1: 漢字の階層化

- 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. エレメントのタイプと制御点の座標によって定義する.

- 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. 組み合わせて使うための情報を持つこともある.

- 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パーツと呼ぶ) を組み合わせたものである. 一般的にはプリミティブは縦横の拡大縮小, 移動をほどこすだけだが, 「発」, 「祭」 などのかしらの下のパーツのようにそれ以外の変形を伴う場合もある.

2.2　エレメント

現在使っているエレメントは, 下の 16 種類である.

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. しんにゅう

図 2に肉付けしたデータを示す.

第一水準の漢字を明朝体で表現するにはこれだけで足りる. 第二水準の漢字を表現する場合や, 他の書体の場合にはこれ以外のエレメントを必要とするかもしれないが, それほど種類を増やす必要はないと思われる.

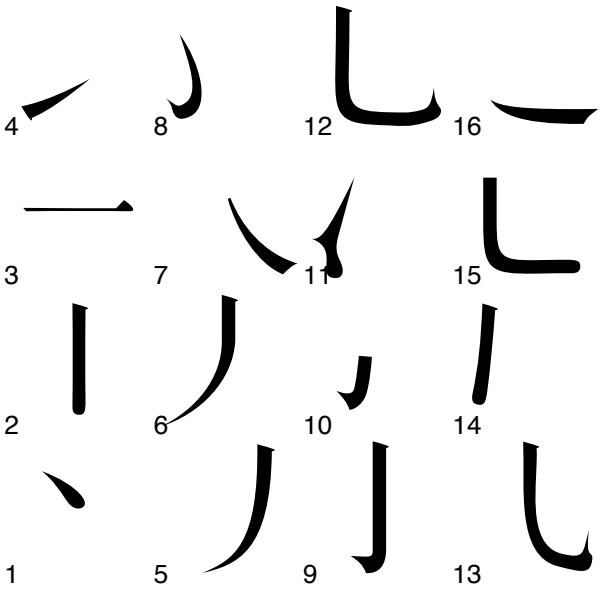

図 2: 明朝体のエレメント

2.3　プリミティブ

漢字はプリミティブである場合もあるし (例 「亜」), 複合パーツの場合もある (例 「挨」). 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.

そこで, 我々はなるべくプリミティブを少なくするという方針でのぞんだ. その結果, 第一水準の漢字すべてを定義するのに必要なプリミティブは 687 個におさえることができた. 第一水準の漢字 2965 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 272 文字だけで, 残りは複合パーツとして定義される.

まだすべての漢字について肉付けを試みていないので, この先プリミティブを増やす必要が生ずるかもしれない. ここにあげる数字はあくまでも目安である.

プリミティブの記述の例をあげる.

```
(setq 日 '(((316 50)
            (316 394)
            (312 350)
            (311 191)
            (87 50)
            (87 390)
            (87 350)
            (87 191))
           ((tate (0 1) (link 2 3))
            (tate (4 5) (link 6 7))
            (yoko (6 2))
            (yoko (7 3))
            (yoko (4 0)))))
```

このスケルトンと肉付けデータとの対応を図 3に示す.

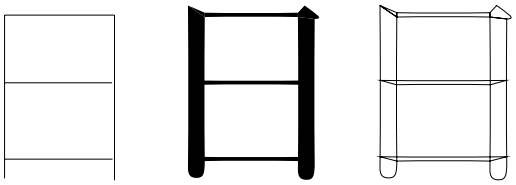

図 3: プリミティブの肉付け

プリミティブのデータは, 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.

2.4 パーツ

パーツの組合せを次のように分類した.

- 横方向の組合せ

  「波」 とか 「街」のような横方向の組合せによって生ずる漢字は非常に多い. 我々の分類では 1673 文字もあった.

- 縦方向の組合せ

  「栗」 ,「志」 のようなもの.

- その他の組合せ

  「国」, 「間」 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. これらは片方のパーツの中の空白部分に他方のパーツを配置するという点で統一的に扱うことができるであろう. これはまだ実験していない.

複合パーツの記述は次のように行なう.

```
(setq 鯵 '(yoko2 うおへん 参))
(setq うおへん '(tate2 くのじ田 四点))
```

JIS 第一水準の漢字すべてについての複合パーツの記述には人力で 10 日ほどかかった.

2.5 実現

漢字スケルトンフォントの実験システムは, UtiLisp で書かれている. UtiLisp を使ったのは, 次のような理由による.

- 対話型言語なので, プログラムの変更が容易

  C のような言語を使うと変更した後のコンパイルに時間がかかるので, プログラムを少しずつ変えて実験をするのには不向きである.

- 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たびたびあるが, すべてが 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.

- 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人間がいるので, X-Window を扱う関数を書いたり, ヒープの自動拡張の機能を取り入れたりするのが容易に行なえる.

システム全体の構成は図 4のようになっている.

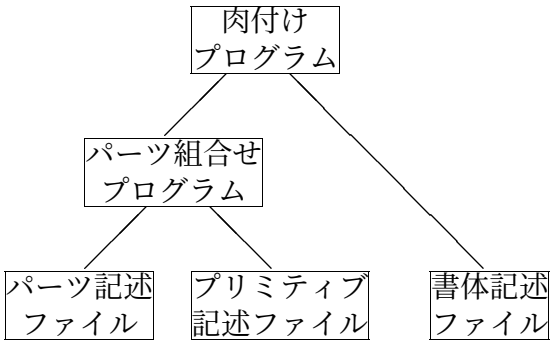

図 4: システムの構成

それぞれのファイルは次のようなものである.

- 肉付けプログラム

  漢字を書体を与えると, その漢字の構成と書体の記述から肉付けデータを返す関数と, その下請け関数の定義.

- パーツ組合せプログラム

  複数のパーツから, それらを組み合わせたパーツを作る関数.

- パーツ記述ファイル

  パーツの名前と, そのパーツがどういったパーツをどう組み合わせて作るかの記述.

- プリミティブ記述ファイル

  各プリミティブについて, 含まれるエレメントの種類と接続関係, あとデザインされた制御点の記述.

- 書体記述ファイル

  エレメントと飾りの肉付けアルゴリズムの記述. プリミティブやパーツを定義し直すこともある. また, 別の書体記述ファイルを読み込んで, 一部だけを変更することもある.

肉付けの結果得られる肉付けデータは直線と Bezier 曲線からなる閉曲線のリストの形をしている. このデータを X-Window 上に表示したり PostScript に変換する簡単なプログラムは別に用意してある.

3　肉付けプログラム

3.1　エレメントの肉付け

エレメントの肉付けは書体ファイルの中でいくつかの制御点から二本の曲線を求める関数として定義する.

```
;
; 右はらいの定義
;
(defprop migi
  (lambda (points alist)
    (let ((x (car points))
          (y (cadr points))
          (z (caddr points))
     (w minchowidth))
    (niku3 x y z 0.3 0.3
           (*$ w 0.2)
           (*$ w 0.4)
           (*$ w 0.8)
           (*$ w 1.0))))
  mincho)
; 実行例
> (funcall (get 'migi 'mincho)
>          '((50 50)(150 250)(350 350))
>          nil)
(((angle +0.4821115^+02 +0.5089443^+02)
  (bezier +0.1069589^+03 +0.1728622^+03)
  (bezier +0.2253490^+03 +0.2966188^+03)
  (angle +0.3455279^+03 +0.3589443^+03))
 ((angle +0.5178885^+02 +0.4910557^+02)
  (bezier +0.1130411^+03 +0.1671378^+03)
  (bezier +0.2346510^+03 +0.2833812^+03)
  (angle +0.3544721^+03 +0.3410557^+03)))
```

エレメントの制御点と肉付けデータの対応を図5に示す. 肉付けデータは変数 minchowidth を変更して三つずつ表示している.

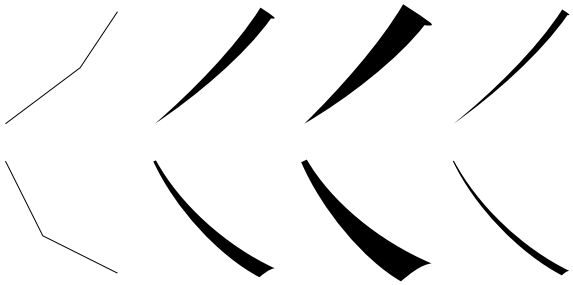

図 5: スケルトンと肉付け

基本エレメントを単なるデータでなく関数とすることは, デザイン上の自由度を増すことに役立っている.

3.2　飾りの肉付け

飾りの肉付けの記述も書体ファイルの中で行なう.

孤立の場合は制御点から曲線に引いた法線との交点 2 点とそこから曲線の接線方向に 2 点間の距離だけとった 2 点の計 4 点. 連結の場合は交点の 4 点を配列にして渡す. 関数が返すのは曲線であり, その始点と終点から, もとのストロークの輪郭線の始点, 終点を変更する.

このように単純化したことで飾りの定義は明朝体の場合で, 12 種類ですんだ.

```
;
(defkazari mincho (tate 0 yoko 1)
  (let ((p0 (vref cross 0))
        (p1 (vref cross 1))
        (p3 (vref cross 3))
  `((angle .,(plus2 p0
    (normlen2
     minchoheight
     (diff2 p1 p0))))
    (angle .,p3)))))
(defkazari mincho (tate 2 tate 3)
  (lets ((p0 (vref cross 0))
 (p1 (vref cross 1))
 (p2 (vref cross 2))
 (p3 (vref cross 3))
 (p4 (times2 0.5 (plus2 p0 p1)))
 (p5 (plus2 p1 (times2 1.0 (diff2 p3 p1))))
 (p6 (plus2 p0 (times2 0.6 (diff2 p2 p0)))))
  `((angle .,p6)
    (bezier .,(plus2 p6 (times2 0.7 (diff2 p0 p6))))
    (bezier .,(plus2 p4 (times2 0.7 (diff2 p0 p4))))
    (angle .,p4)
    (bezier .,(plus2 p4 (times2 0.8 (diff2 p1 p4))))
    (bezier .,(plus2 p5 (times2 0.8 (diff2 p1 p5))))
    (angle .,p5))))
```

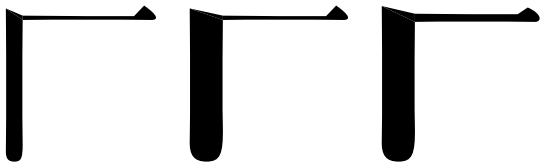

図 6: 飾りの例 (太さを変更)

4　組合せプログラム

複数のパーツを組み合わせて, 新たなパーツを作るという肝にあたる部分である. 現在実験を行なっているのは, 横方向の組合せと縦方向の組合せである. 第一水準の漢字 2965 文字のうち 2000 文字以上の字がこの 2 種類の組合せだけで作ることができる.

4.1　縦方向の組合せ

縦方向の組合せでは, 三つ以上のパーツの組合せもあるが, ここでは二つのパーツの組合せに関して説明

する．三つ以上の場合も同様の考え方でできる．
縦方向の組合せは，

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

という手順によって行なう．それぞれの手順について次から説明する．

4.1.1 中心あわせ

縦方向の組合せをする場合は中心合わせが必要だが，単純に重心を中心と定義するのでは，図 7の上のようになってしまう．縦方向に組み合わせることのできる部品は左右対称に近い形をしたものが多く，対称の中心を合わせる必要がある．

図 7: 中心合わせ

すべてのプリミティブについて部品の中心を定義するのは面倒なので，プリミティブのデータから対称の中心を求めることを試みた．まず，対称となりうるエレメント及びエレメントの対を人間が与えてやる (図 8)．

図 8: 対称とみなすエレメント

```
(setq xsymmetry
  '(
    ((yoko 0 1))
    ((tate 0 1))
    ((tatehidari 0 1))
    ((tatehane 0 1))
    ((hidari 0 2))
    ((ten 0 1))
    ((tate 0 1)(tate 0 1))
    ((ten 0 1)(hidari 0 2))
    ((hidari 0 2)(migi 0 2))
    ((tatehidari 0)(tatehane 0))
    ((tatehidari 0 1)(tate 0 1))
    ((hidari 0)(kokoro 0))
    ((tate 0 1)(tatehane 0 1))))
```

エレメントの型の後ろの数字は，何番めの制御点かをあらわしている．つまり，((ten 0 1)(hidari 0 2)) というのは，ten の 0 番めの制御点と hidari の 0 番めの制御点が対称で，ten の 1 番めの制御点と hidari の 2 番めの制御点が対称のとき，この ten と hidari が対称だとみなすということである．

プリミティブの中心を求めるには，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，単独で対称な縦棒，縦棒 + 左払い，縦棒 + はねがあるときはその対称の中心を使う．対称なエレメント，エレメント対がない場合は仮の中心をそのまま使う．

この方法でかなりのプリミティブについて正しく中心を求めることができるが，うまくいかない場合もある (図 9の上)．このような場合は，プリミティブの定義に，

```
(setq 電の下
 '(((190 225)
    (190 378)
 ......
    (73 272))
   ((kokoro (0 1 2 3))
 ......
    (yoko (8 4) (link 0)))
   (center . 190)))
```

のように center という属性を与えてやると，そちらの値を使って図 9の下のように修正することができる．この修正は，いまはファイルをエディタで直接操作することによって行なっているが，表示を見ながら対話的に行なうエディタも作るつもりである．

4.1.2 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 10を見ると幅の調節の必要性がよく分かるであろう．

プリミティブの幅は組合せによって変わる場合もあるが，それはあくまで例外であって，大概の場合プリミティブに固有の幅があるみなしてよい．

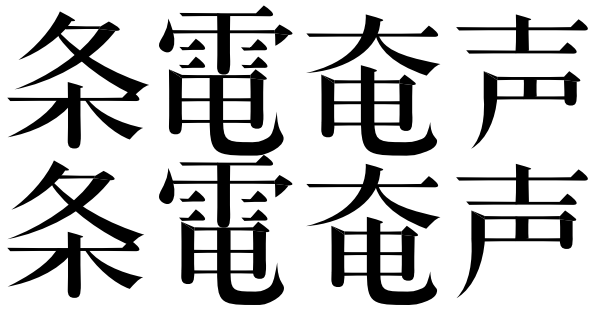

図 9: 中心の検出の補正

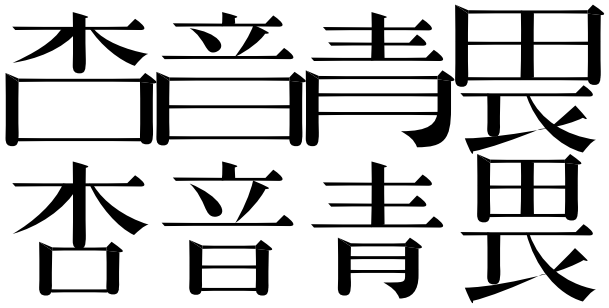

図 10: 幅の調節の有無

すべてのプリミティブについて固有の幅を与えればよいのだが, ある程度自動化することにした.

図 11のように縦に並べる場合, 口と日はほぼ同じ幅であり, 田はそれよりも幅広く並べるとバランスが良い. この例に限らず, 外形が同じ場合には中の縦線が多いほど幅が広くなる傾向がある. 縦線の数と全体の幅との関係をきめる際に, 同種の漢字を横に並べた時に縦線の間隔が等間隔になるようにした. こうすると, 「口」の幅は全体の 1/2 に, 「田」の幅は全体の 2/3 になる.

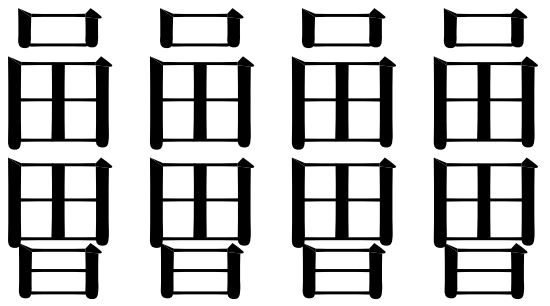

図 11: 口, 日, 田の大きさ

しかし, 縦棒がもっとも外側にはこない場合もあるし, 中に縦棒が出現しなくても, 他のエレメントがたくさんつまっていたら, 幅を広げなければいけない.

こういった場合を扱うために, ほかのエレメントも縦方向の成分をとって, その長さの和を高さで割ったものを縦棒の本数とするなどの操作を行なって, この方法の適用をはかっている.

ただ, 縦棒以外のエレメントが多く出現するパーツについてはうまくいかない例がある. また, 「昌」のようにパーツ固有の幅だけで扱うことのできない例もある (図 12の上).

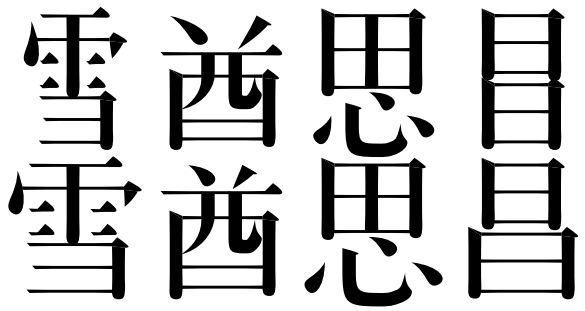

図 12: 幅の調節の補正

このような例について, 手で補正をするためにパーツに width という属性を与える. プリミティブ固有の幅を変更するには,

```
(setq あめかんむり
   '(((364 80)
      ...
     (344 32))
    ((hidari (0 1 2))
          ...
     (yoko (16 17) (link 5))
    (width . 190)))
```

のようにすればよいし, 組合せの際に幅を変更するには,

```
(setq 昌 '(tate2 日(xscale 1.4 日)))
```

のように, width の属性を変更したパーツを作る関数を通す (図 12の下).

4.1.3 高さの比率

上下の高さの比率を決定する際に根拠としたのは, 「複雑なパーツほど大きい」という経験則である.

複雑ということをどう定義するかというのは難しいが, とりあえずは面積当たりの線の長さとすることにした. 面積当たりの線の長さはもちろん高さの比率を変更すれば変わってくるので, 上下のパーツの比率をどうすれば黒さを同じにできるかは, 繰り返しによって求める.

その方法によって求めたのが図 13の上である. これを見ると, 縦棒と横棒だけからなるようなパーツの組合せについてはうまくいくが, ほかの斜めの線が過小評価されていることがわかる.

そこで, 斜めの線については線の長さでなく X 座標の差と Y 座標の差の和を用いるようにしたのが図 13の下である. この方式で, プリミティブについてエレメントの縦方向, 横方向の長さの総和をあらかじめ求めておけば, 2 次方程式を解くだけで高さの比率を計算できる.

これでもうまくいかないものは, プリミティブ, 組合せの単位で補正を行なう.

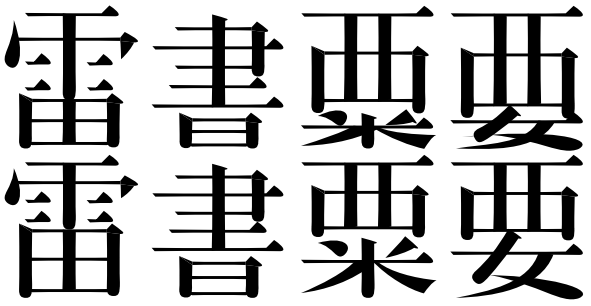

図 13: 高さの比率

4.1.4 食い込み

上下のパーツの高さの比率を決定した後, 下のパーツの上端と上のパーツの上端の距離を決定しなければならない.

まず, プリミティブを制御点を結んだ線分の集合として, 交叉点がない範囲でもっとも近付いた状態を求める (図 14の上). 横線と横線および縦線と横線の間の間隔は異なるものとしなければならないので, 交叉する点における線分同士の角度を考慮して間隔を決定する (図 14の下).

図 14: 食い込み

自動的に決定したものに不満がある場合は, 手で直すことができる.

4.1.5 プリミティブの変形

縦方向の組合せの場合は, プリミティブの変形を必要とする場合がある. これを解決する場合の安直な方法の一つとして, 変形したプリミティブを独立のプリミティブとして用意するものがある.

しかし, これはプリミティブの数をなるべくすくなくするという方針に反するので, もっとも一般的な上のパーツの下端が左払いと右払いからなる場合のプリミティブの変形は関数でおこなうことにした.

変形を必要とする「発」のようなパーツは,

```
(setq 発 '(tate12 はつがしら (kashira 発の下)))
```

のように記述する. kashira はパーツを引数として, 上端を縮めたパーツを作り出す関数である. これによる変形を図 15にあげる.

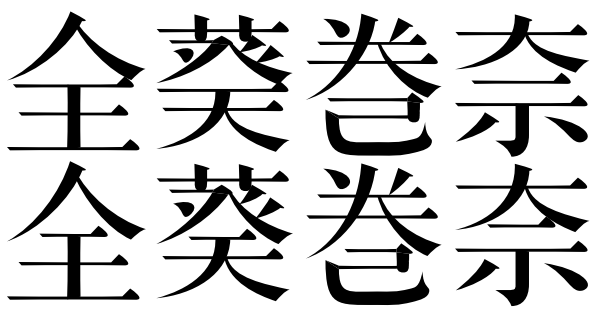

図 15: 変形の例

4.2 横方向の組合せ

漢字の組合せでもっとも多く現れるのが横方向の組合せである.

本システムでは, 横方向の組合せは次のような手順によって行なう.

1. 両方のパーツの上下を合わせる
2. パーツの幅の比率を決定する
3. 食い込みを決定する

デザイナによって作られるフォントの中には, 図 16([4] より) のように, 横方向の組合せによってパーツの形が変化するものもあるが, すべてのフォントで変形するわけではないので扱わない. このようなフォントを作るには変形したプリミティブを用意する必要がある.

図 16: 組合せによる変形

4.2.1 上下あわせ

パーツを横方向に組み合わせる時, 単純にパーツの中の制御点の Y 座標の最大, 最小をあわせると, 図 17の上のように最大最小に横線などのエレメントが来た時に上下にはみだしたような印象を与える.

そこで, Y 座標が最大最小になるエレメントを横棒に近似して, その幅によって上下に補正するようにした (図 17の下).

しかし, すべてのパーツをこの方法で上下あわせすると単純なパーツが, 大きくなりすぎてしまう (図 18の上). これに対する補正を自動的に行なうアルゴ

図 17: 上下あわせ

リズムが決定できなかったため, 次のように手で与えることにした.

```
(setq つちへん
  '(((21 305)
     ...
     (153 127))
    ((migiue (0 1 2) (link 3))
     (tate (4 3))
     (yoko (5 6)))
    (updown 10.0 . 80.0)))
```

補正の結果, 図 18の下のようになる.

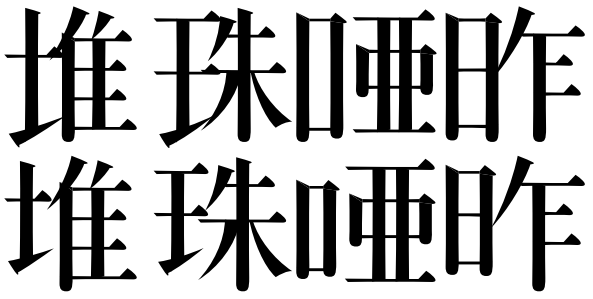

図 18: 上下あわせの補正

4.2.2 幅の比率と食い込み

幅の比率と食い込みは, 縦方向の組合せと同じアルゴリズムを用いた. しかし, 図 19のように補正を必要とするものが多いことから考えて, 別のアルゴリズムを考案する必要があるかもしれない.

図 19: 幅の比率と食い込み (補正前と補正後)

5 今後の課題

5.1 縦横方向以外の組合せ

縦方向横方向の組合せだけで, 漢字の大部分が扱えるといってもフォントとして使うには最低でも第一水準の漢字すべてが扱えなければ使いものにならない.

それ以外の組合せは, 一つのパーツの中の空白部分にもう一つのパーツを埋め込むという方法で統一的に扱うことができるが, 空白部分が単純な長方形とは限らないので, そこへの埋め込みは縦や横の食い込みのような単純な方法では扱えないであろう.

また, 埋め込まれるパーツが複雑な場合にはもとのパーツを空白部分が広がるように変形させる必要があるかもしれない. この変形もどのように統一的に扱うか問題が多そうである.

5.2 書体の変更

実用的なフォントにするには, 最低限ゴシック体は用意しなければいけない. 明朝体のデータに対して肉付けアルゴリズムだけを変更してゴシック体の字を作ることができるような漢字もあるが, エレメントの座標を変更する必要がある漢字や, エレメントの種類を変更する必要がある漢字もある (図 20).

この変更をある程度自動的に行なうことはできないか, また, 手で変更する必要があるのはどの程度の数かを調べるのが今後の課題である.

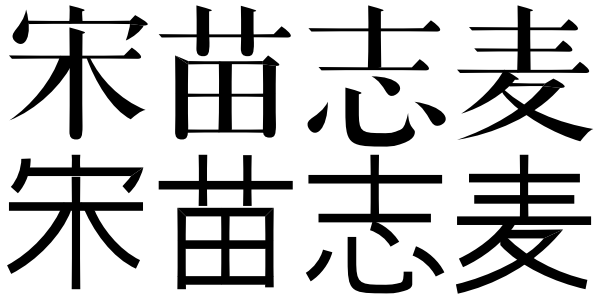

図 20: 明朝体のスケルトンにゴシック体の肉付けをした例

5.3 ひらがな, カタカナ

ひらがなやカタカナは数が少ないので, アウトラインフォントを作るにしてもデザインの手間はそれほどでもないが, 漢字と同じ精神で統一的に扱うことができるならそのほうがよい.

これについては, 研究室の卒論生がいろいろ実験を行なっている最中である.

5.4　ツールの開発

プリミティブのデザイン, 肉付けアルゴリズムの決定, パーツ組合せの補正などの書体製作を素人でも簡単に行なうことができるよう, 対話的ツールを開発中である. そのため, Xlib をリンクする方法でなく UtiLisp から X-Window のサーバと直接通信する形のライブラリも作っている所である. これを使ったツールはシンポジウム当日のデモ会場で展示する予定である.

6　おわりに

漢字スケルトンに明朝体の肉付けをするのは困難とされてきたが, Lisp を使い自由度の大きい肉付けアルゴリズムを使うことによって, 少量の定義によって使いものになる肉付けを得ることができた.

漢字をパーツに分解して定義し, 組合せプログラムによって合成する試みは縦方向と横方向についてある程度達成された. 他の組合せについても試みて, すべての漢字を表現できるよう, さらなる努力が必要である.

参考文献

[1] Plass, M. : Curve-Fitting with Piecewise Parametric Cubics, Computer Graphics, vol. 17, No. 3, pp. 229-239(1983).

[2] 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).

[3] 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. 32, No. 2, pp. 209-218(1990).

[4] 明朝系 3 書体, 視覚デザイン研究所 (1981).