フォント生成エンジン KAGE を自作する

博多市 (@hakatashi)

目次

第 I 章

剣戟少年と魔法技師

1　旅立ちの日

　何がおかしいって、去年でさえあれだけ渋っていた僕が今でもここにいるのが一番おかしい。

　あんまりおかしいので Skype で後輩に尋ねてみた。

　博多市「なんで部誌制作チームに僕が追加されてるんですかね」

後輩 S「え，部誌書くでしょ？」

後輩 H「あ、そうですか、期間が長すぎる、こんなに長くては 10 本かけてしまうぞと、そうですかそれはすみません」

後輩 H「じゃあ博多市さんだけ特別に期間短くしてあげますね (原文ママ)」

ちゃたい[1]「お、期待」

　若干つらいものがある。

2　魔法技師との出会い

　これを踏まえて考えると、やはり日本語フォントにおける最大の問題点は、その文字数の多さと、それに勝るとも劣らない文字の複雑さにあるといえるだろう。特に、ハネやウロコなど複雑な形を持つ明朝体は製作に時間がかかることで有名で、需要も高く美しい書体であるにもかかわらずフリーでのバリエーションは非常に限られており、高価な有料フォントを使用しないと美しいデザインが作れないというのが現状である。

3　機械都市アポリア

　フォント制作の手間を減らすにはどうすればいいのか。そこで登場したのがフォントの自動生成である。

[1] 灘校パソコン研究部の Skype 部屋に生息する人工無能

フォントの自動生成とは、その名の通り、これまで完全に人力で行っていたフォント製作を、コンピューターの手を借りて作業を補完することである。実はこれは決して真新しい技術ではなく、フォント製作にコンピューターが導入された当初から広く使われてきた手法である。

4　ジルコの願い

自動生成と言っても、コンピューターがどのように働くかという点で様々なものが考えられる。フォントがデザインである以上、コンピューターの役目はあくまで作業の補完にとどまるが、それでもフォント製作にはある程度のルーチンが存在し、それだけ自動化の這い入る隙間が存在する。

最も広く使われているのは、多ウェイト展開におけるウェイト間の補完作業である。ウェイトとはフォントの太さのことで、実際にこれらのフォントを使用する際にはウェイトを切り替えながら適切な太さを選んで使用するものである。ウェイトは一概には言えないものの基本的に細いものから太いものまでのバリエーションが多ければ多いほど好ましいとされ、中には 1 つのフォントに 10 種類以上のウェイトが存在するものもある。

しかしそれだけ多数のウェイトを、デザインを統一しながら手作業で全部デザインするのは非常に難しく、多数の労力を費やさなければならない。そこで自動化が登場した。[2]具体的には、現在開発でよく使われる方法では、ウェイト展開するうちの最も細い書体と最も太い書体を画数ごとに分けて用意し、その間のウェイトは機械的な計算によって自動的に生成するというものである。これならば、(ウェイトに応じてある程度の微調整は必須なものの、) 必要な労力を削減してフォント製作に集中することができる。

5　陰謀、そして裏切り

そしてもう一つ、自動化としてフォント製作にしばしば用いられるのが、グリフからフォントを生成する機能である。

グリフというのは文字の骨格のことであり、明朝体におけるウロコや、線の太さの違い、そのほか文字におけるあらゆる装飾を除いた文字構造のことである。すでにメジャーなフォントメーカーでは、ベンダごとに独自のフォント生成プログラムが開発されており、これにより複数の文字においても同じ形のウロコ、ハネ、線の太さを持つことが保証されている。

[2] というより、自動化が生まれたからこそウェイト展開が可能になったといえるだろう。

6　機械都市の秘密

今回製作するのは、そのような、グリフからフォントを生成するエンジンである。グリフというのは、基本的には明朝体でもゴシック体でも同じ文字なら大きく変わることがなく、異なる書体においてもこれを流用して使用することが期待できる。

さらに、漢字など多くの共通の形を持つ字形においては、(フォントではなく) グリフの形を変形し別の文字に持ち込むことにより、フォントを変形するだけではなしえなかった自然な形でのフォント生成が実現できる。

7　人造人間誕生

このような発想は、産業技術総合研究所の主導する CHISE プロジェクトにも大きく影響している。

CHISE プロジェクトとは、(特に日本語の) 文字情報処理における環境を改善するための総合的なプロジェクトであり、多くのサブプロジェクトが内部にセットアップされている。

その中でも特に開発が顕著なのが、大東文化大学の上地宏一講師の携わった一連のフォント自動生成技術であり、既に CHISE IDS や GlyphWiki、花園明朝などの多数の成果を上げている。これらのプロジェクトにおいて中核をなすのは、KAGE データと呼ばれる、文字のグリフを表現するデータ形式、およびそこからフォントを生成する KAGE エンジンである。これらは全てオープンソースで開発がされており、誰でも自由に利用することができる。こうして、再利用可能な形でグリフを表現することにより、困難な日本のフォント事情を打開しようというのがこのプロジェクトの狙いである。

8　機械に宿ったココロ

ところがそうは言っても、日本語には多数の文字が存在し、KAGE データのデータベースには大量の文字の収集と継続的な整備が欠かせない。この問題をフォークソノミー的な手法によって解決するのが GlyphWiki である。

GlyphWiki は KAGE データを内部に持つ、だれでも編集が可能な Wiki サイトである。このサイトは現在でも文字に関心をもつ多数の利用者によってメンテナンス、文字の収集が行われており[3]、すでに Unicode に含まれる漢字はすべて収集が完了している。ここに集積された KAGE データは完全に自由に利用することが可能であり、KAGE エンジンに手を加えればだれでも独自の Unicode フォントを生成することができる。

[3] 筆者も微力ながら力添えをさせてもらっている

9　人間と機械のための街、アポリア

このように、すでにフォントの自動生成のための土壌は形成されていると思っていいだろう。しかし、これらの一連の取り組みでもカバーできない部分は存在する。

ひとつは、明朝体特有のデザインによるものである。GlyphWiki などで集積された KAGE データは、汎用的な利用を目的としているものの、既存の KAGE エンジンとの親和性の問題から明朝体用にチューンアップされたものであり、そのまま他の書体に流用すると不具合が起こるものが多い。[4]そのせいもあってか KAGE データからフォントを制作するエンジンのバリエーションはなかなか制作されておらず、2014 年現在、CHISE プロジェクトによって正式に提供されている明朝体とゴシック体の生成エンジンの他には、丸ゴシック体を出力するものが唯一存在するのみとなっている。

[4] 例えばさんずいやにすいなどがこれにあたる。

第 II 章

実際に KAGE エンジンを組んでみる

能書きはそれくらいにして、とっととフォント生成エンジンを実装してみます。今回は実装を単純にするため[1]、単純な細ゴシック体を生成するエンジンを組みます。

公式で提供されてる KAGE エンジンは C++(deprecated) と、SpiderMonkey 互換の JavaScript で提供されていますが、今回は筆者の趣味により node.js で実装することにします。

1　使用データ

glyphwiki のダンプファイルは http://glyphwiki.org/dump.tar.gz で提供されています。今回は 2014 年 3 月 15 日現在のデータを使用しました。

2　KAGE データのパース

まずはダンプファイルをパースします。

kage.js:

```
var fs = require('fs');
var util = require('util');
var argv = require('optimist')
    .demand([1])
    .argv
;

// read argv[0] file and split them into lines
```

[1] すでに締め切りを大幅に過ぎているためとも言う

```
var lines = fs.readFileSync(argv._[0], {
    encoding: 'ascii'
}).split(/\n/);

// export all KAGE data to global
KAGE = [];

// parse all lines to KAGE structure
lines.forEach(function(line) {
    var kage = {};

    var splits = line.split('|');

    // trim formatting spaces
    var columns = [];
    splits.forEach(function(split) {
        columns.push(split.trim())
    });
    if (columns.length < 3) return;

    kage.name = columns[0];

    var strokes = columns[2].split('$');

    kage.stroke = [];
    strokes.forEach(function(stroke) {
        var params = stroke.split(':');
        kage.stroke.push(params);
    });

    KAGE.push(kage);
});

console.log(util.inspect(KAGE, {depth: null}));
```

ダンプデータから適当に一件抽出した dump.txt を用意してパースしてみます。

dump.txt:

```
u5fc3                                                    | u5fc3  |
    2:7:8:38:80:37:131:15:146$3:0:5:68:63:68:173:148:173$2
    :7:8:74:19:105:32:118:60$2:7:8:149:79:176:101:182:134
```

console:

```
hakatashi@UbuntuStudio:~/Projects/kage$ node kage.js dump.txt
[ { name: 'u5fc3',
   stroke:
    [ [ '2', '7', '8', '38', '80', '37', '131', '15', '146' ],
      [ '3', '0', '5', '68', '63', '68', '173', '148', '173' ],
      [ '2', '7', '8', '74', '19', '105', '32', '118', '60' ],
      [ '2', '7', '8', '149', '79', '176', '101', '182', '134' ] ] } ]
```

うん、よい感じ。

3 canvas に描画

node-canvas を使用してグリフを一画ずつ描画します。

node-canvas は HTML5 Canvas API の Node.js 実装であり、通常の Canvas API のほかに、PNG 出力などが使用できます。

今回は node-canvas への描画に paper.js を使用しています。

```
KAGE.forEach(function(kage) {
    // total glyph size is 200x200 since imaginary body is assumed to be
    // (12, 12) - (188, 188).
    var canvas = new paper.Canvas(200, 200);
    paper.setup(canvas);

    kage.stroke.forEach(function(stroke) {
        var path = new paper.Path({
            strokeColor: 'black',
            strokeWidth: 10
        });
        // The first parameter represents kind of stroke, and
        // second and third are shape of leading and trailing of
        // stroke. For now, we ignore shape of leading and trailing.
        switch(stroke[0]) {
            case 1: // 直線
                path.moveTo([stroke[3], stroke[4]]);
```

```
            path.lineTo([stroke[5], stroke[6]]);
            break;
        case 2: // 曲線
            path.moveTo([stroke[3], stroke[4]]);
            path.quadraticCurveTo(
                [stroke[5], stroke[6]],
                [stroke[7], stroke[8]]
            );
            break;
        case 3: // 折れ線
            path.moveTo([stroke[3], stroke[4]]);
            path.lineTo([stroke[5], stroke[6]]);
            path.lineTo([stroke[7], stroke[8]]);
            break;
        case 4: // 乙線
            path.moveTo([stroke[3], stroke[4]]);
            path.cubicCurveTo(
                [stroke[5], stroke[6]],
                [stroke[5], stroke[6]],
                [stroke[7], stroke[8]]
            );
            break;
        case 6: // 複曲線
            path.moveTo([stroke[3], stroke[4]]);
            path.cubicCurveTo(
                [stroke[5], stroke[6]],
                [stroke[7], stroke[8]],
                [stroke[9], stroke[10]]
            );
            break;
        case 7: // 縦払い
            path.moveTo([stroke[3], stroke[4]]);
            path.lineTo([stroke[5], stroke[6]]);
            path.quadraticCurveTo(
                [stroke[7], stroke[8]],
                [stroke[9], stroke[10]]
            );
```

```
                break;
            }
        });

        //Update paper view.
        paper.view.update();

        var stream = canvas.pngStream();
        var pngFileName = __dirname + '/' + kage.name + '.png';
        var output = fs.createWriteStream(pngFileName);

        stream.pipe(output);
    });
```

これだけです。paper.js サイコー。

とりあえず何文字か噛ませてみます。

dump.txt:

```
u5fc3                                                    | u5fc3  |
    2:7:8:38:80:37:131:15:146$3:0:5:68:63:68:173:148:173$2
    :7:8:74:19:105:32:118:60$2:7:8:149:79:176:101:182:134
u591a                                                    | u591a  |
    2:0:7:91:16:68:56:24:80$1:2:2:80:34:140:34$2:22:7:140:34:97:103:17:126$2
    :7:8:62:53:81:61:90:80$2:0:7:125:80:99:123:46:146$1:2:2:111:99:173:99$2
    :22:7:173:99:131:179:25:188$2:7:8:94:117:114:127:124:147
u4e59                                                    | u4e59  |
    1:0:2:40:37:143:37$4:22:5:143:37:12:169:170:169:175:171
u89d2                                                    | u89d2  |
    2:0:7:69:14:55:55:14:86$1:2:2:63:30:129:30$2:22:7:129:30:120:47:101:65$7
    :12:7:51:65:51:127:51:165:16:188$1:2:2:51:65:159:65$1:22:4:159:65:159:182
    $1:32:32:105:65:105:135$1:2:2:51:100:159:100$1:2:2:51:135:159:135
```

出力は下のようになりました。

うん、よいかんじ。

4　引用グリフの実装

第一章で述べたとおり、KAGE には他のグリフを引用することにより字形の管理が容易になる工夫がしてあります。フォントを生成する際には、この参照されたグリフを指示通り変形して埋め込む処理を加えなければいけません。

以下の処理を Canvas 描画の前に挿入します。

```
function normalizeKAGE(kage) {
    var newKage = [];

    kage.forEach(function(params) {
        if (params[0] == 99) {
            // referenced glyph name lay in 8th parameter
```

```
            var refer = params[7];

            if (KAGE[refer] != undefined) {
                // invoke normalization recursively
                var refKage = normalizeKAGE(KAGE[refer]);

                var X = {start: params[3], end: params[5]};
                var Y = {start: params[4], end: params[6]};

                function normalizeToSection(section, point) {
                    return (
                        point * section.end +
                        (200 - point) * section.start
                    ) / 200;
                }

                refKage.forEach(function(stroke) {
                    for(var i = 3; i < stroke.length; i++) {
                        // parameter representing X lay in odd index number
                        if (i % 2 == 1) {
                            stroke[i] = normalizeToSection(X, stroke[i]);
                        } else {
                            stroke[i] = normalizeToSection(Y, stroke[i]);
                        }
                    }
                    // normalized stroke is stored here
                    newKage.push(stroke);
                });
            }
        } else {
            // stroke which doesn't need normalization goes straight here
            newKage.push(params);
        }
    });

    return newKage;
}
```

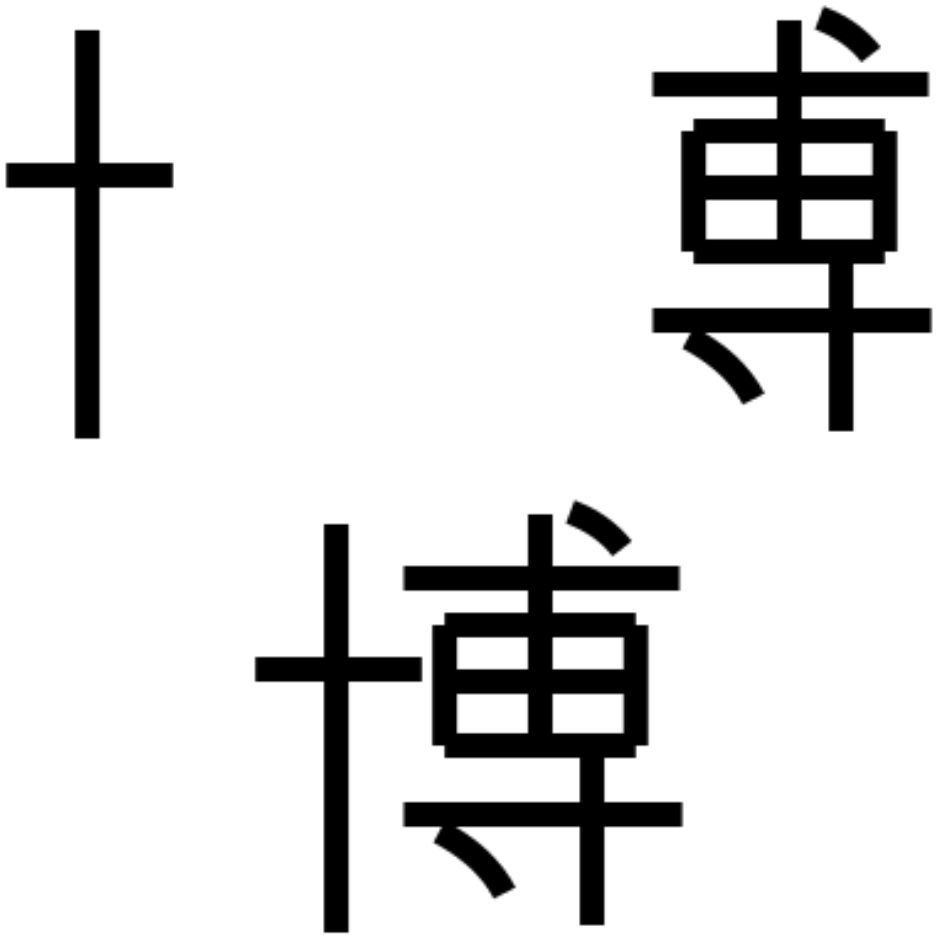
```
normalizeKAGE(kage);
```

引用グリフが含まれる文字を噛ませてみます。

dump.txt:

```
u5341-01                                                    | u5341  |
    1:0:0:14:79:86:79$1:0:0:49:20:49:186
u5c03-02-var-001                                            | u5c03  |
    1:0:0:86:42:187:42$1:12:13:101:61:101:110$1:2:2:101:61:171:61$1
    :22:23:171:61:171:110$1:0:32:136:16:136:110$1:2:2:101:84:171:84$1
    :2:2:101:110:171:110$2:7:8:147:15:159:20:166:30$1:0:0:86:138:188:138$1
    :0:4:155:115:155:183$2:7:8:99:145:116:155:123:170
u535a                                                       | u535a  |
    99:0:0:0:0:188:200:u5341-01$99:0:0:-22:0:200:200:u5c03-02-var-001
```

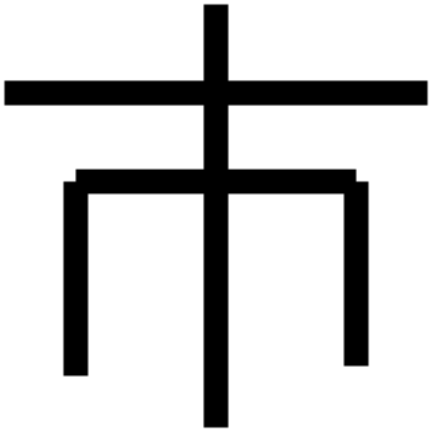
出力は以下のようになります。



すこぶる順調です。

5　隣接辺の結合

ここまでの文字は隣り合う画どうしを別々に描画していたので角の部分が分離して非常にブサイクでした。下の “市” の両肩のような部分はできればつながっている方がいいでしょう。つながる No.1。



以下の処理を Canvas 描画後に追記します。

```
// check if two path share their first or last segment
function needsJoin(pathA, pathB) {
    var firstA = pathA.firstSegment.point;
    var lastA = pathA.lastSegment.point;
    var firstB = pathB.firstSegment.point;
    var lastB = pathB.lastSegment.point;

    if (firstA.equals(firstB)
            || firstA.equals(lastB)
            || lastA.equals(firstB)
            || lastA.equals(lastB)) return true;
    else return false;
}

// join neighboring strokes
paper.project.activeLayer.children.forEach(function(child1, index1) {
```

```
        paper.project.activeLayer.children.forEach(function(child2, index2) {
            if (child1 !== child2 && needsJoin(child1, child2)) {
                child1.join(child2);
            }
        });
    });
```

paper.js の join はセグメントの重なりを自動的に判別して結合してくれるので非常に便利です。

以下のようになりました。

超いい感じ。

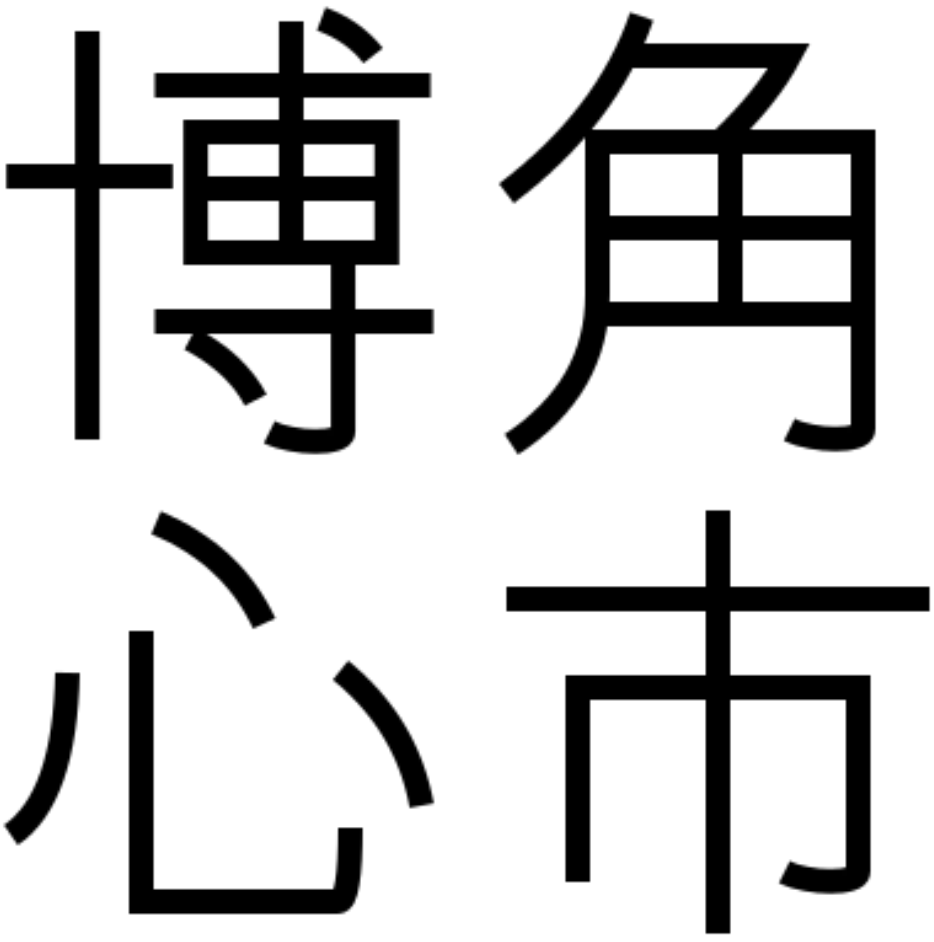
6　ハネの付加

ここまで、明朝体特有の頭形状、尾形状は完全に無視してきましたが、さすがにハネの部分は無視するわけにはいきません。適当にハネを付け足しておきます。

以下の 2 行を追加します。

```
// Add 'Hane' for Gothic design.
// The second parameter represents trailing stroke of stroke.
if (stroke[2] == 4) path.cubicCurveBy([0, 5], [-20, 5], [-30, 0]); //左ハネ
if (stroke[2] == 5) path.quadraticCurveBy([5, 0], [5, -30]); //上ハネ
```

以下のようになります。



若干ブサイクですが、まあ、いいんじゃないでしょうか。

7　svg で出力

これで字形生成はひととおり終了です。ここまでたったの 160 行。ここから生成した字形をフォントに詰め込む作業に移ります。

完成した字形は全て PNG 画像で出力していましたが、アウトラインフォントに流しこむにはベクター画像で出力しなければいけません。paper.js の svg 出力機能を使用して svg を出力します。

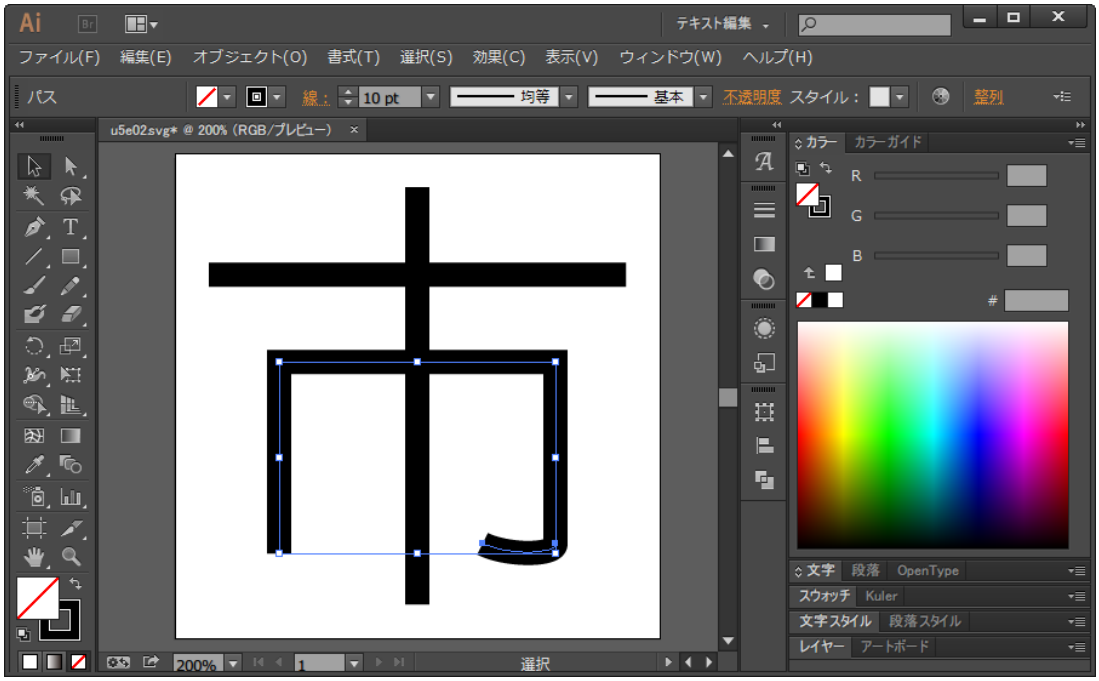
画像出力部を以下で置き換えます。

```
// output as svg format
var svg = paper.project.exportSVG({asString: true});
var output = __dirname + '/' + name + '.svg';
fs.writeFile(output, svg);
```

下のようなファイルが出てきます。

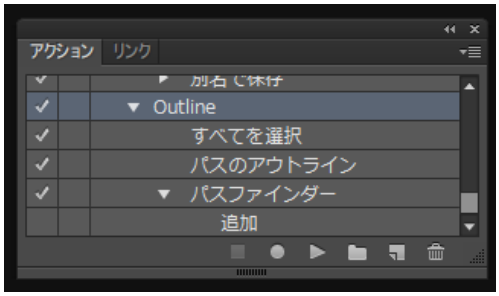
u5e02.svg:

```
<svg x="0" y="0" width="200" height="200" version="1.1"
    xmlns="http://www.w3.org/2000/svg"
    xmlns:xlink="http://www.w3.org/1999/xlink">
  <g fill="none" stroke="#000000" stroke-width="10"
      stroke-linecap="butt" stroke-linejoin="miter"
      stroke-miterlimit="10" stroke-dasharray=""
      stroke-dashoffset="0" font-family="sans-serif"
      font-weight="normal" font-size="12"
      text-anchor="start" mix-blend-mode="normal">
    <line x1="100" y1="14" x2="100" y2="50"></line>
    <line x1="14" y1="50" x2="186" y2="50"></line>
    <path d="M43,165L43,86L157,86L157,161c0,5 -20,5 -30,0"></path>
    <line x1="100" y1="53" x2="100" y2="186"></line>
  </g>
</svg>
```



8　アウトライン化

できればすべての作業を node.js 上で行いたかったのですが、残念なことに paper.js には現在ストロークをアウトライン化する機能がありません。(Roadmap には記載があるんですが) そこで、今回は Illustrator のバッチ機能を用いることにします。



こんな感じでバッチします。せっかくなのでパスの合成もやっておきました。

FontForge の ExpandStroke() を使ってもいけるような気がしますが、僕の実行環境ではエラーを吐かれました。

9　フォント生成

生成した svg をフォントファイルに流し込みます。今回は老舗のフォント編集ソフト FontForge を使用します。

FontForge は基本的に GUI ツールですがスクリプティング機能も備えており、GUI で実

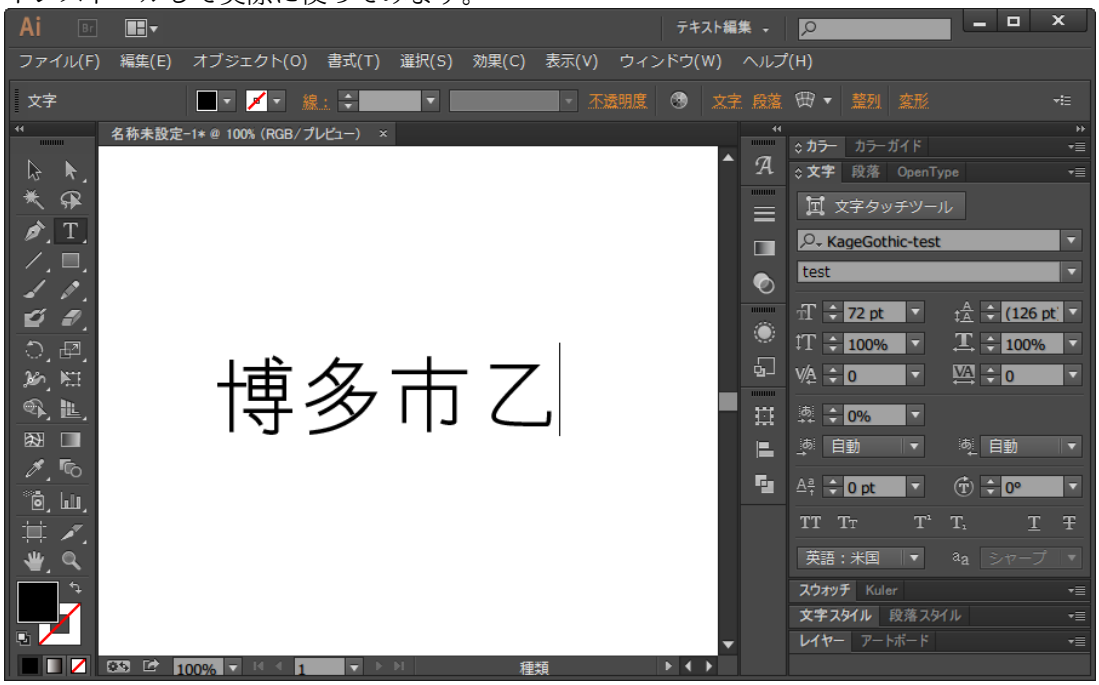
行できる操作は全てスクリプトからアクセスできます。

というわけで、以下のスクリプトを実行します。

```
New()
Reencode("unicode")
Import("u*.svg")
SetFontNames("KageGothic-test", "KageGothic-test", "KageGothic-test")
Generate("KageGothic-test.otf")
Quit()
```

独自言語なのが非常にきも^H^H 面白いですが、これだけでちゃんとした OTF フォントが出てきます。

インストールして実際に使ってみます。



ほいきた!

第 III 章

あとがき

えー、フォント制作というと完全にデザイナーの仕事の範疇というイメージがありますが、このようなプログラマブルな方法も長年研究されています。現在では GlyphWiki のような字形データを利用することにより、今回示したように非常に簡単にフォントを生成することが出来ます。今回製作したのはフォントの中でも特に単純なゴシック体でしたが、パラメータを変えたり、更に細かい処理を施すことによって、明朝体などの複雑な字形も作ることができるようになります。この記事によって、そのような機械的な字形処理に興味を持っていただければ幸いです。

なお、今回製作したフォント生成エンジンは https://github.com/hakatashi/KageEngine-test で公開しています。PDF 上では読みにくいかと思いますので、ぜひそちらもご参照ください。

以上、OB の博多市からでした。