図形認識ライブラリ解説

Ver 1.0.3

任天堂株式会社発行

目次

改訂履歴

版	改訂日	改 訂 内 容	担当者
1.0.3	2007-10-05	現況に合わせて情報を更新	清木
1.0.2	2005-02-18	表紙及び改訂履歴を作成	清木

1　図形認識ライブラリの概要

1.1　はじめに

NitroSDK には図形認識ライブラリ（PRC*）が含まれており、簡易的な図形認識機能を提供しています。このドキュメントでは、図形認識ライブラリの基本的な使用方法から、各認識アルゴリズムの特徴、チューニングの指針、などについて解説します。

図形認識ライブラリは、タッチパネルの入力機器としての利用を簡便にするために提供されるものです。タッチパネルからの手書き文字入力機能のみが必要な場合は、「Decuma 手書文字認識ライブラリ for NINTENDO DS」が提供されていますので、そちらの利用をご検討ください。利用許諾契約書の締結により、無償でご利用いただけます。

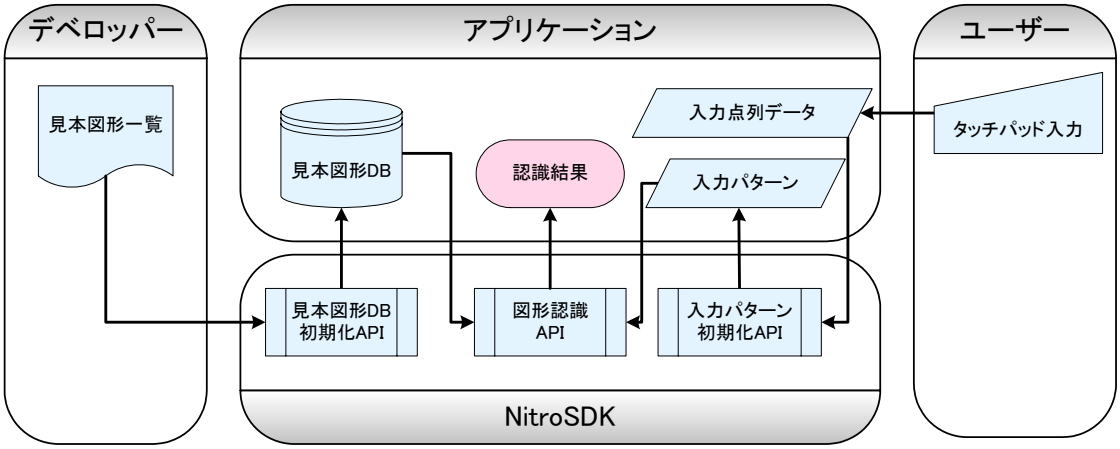
1.2　提供される機能

図形認識ライブラリが提供する機能は非常にシンプルなものです。

まず、あらかじめ見本となる図形の一覧を用意します。見本図形一覧の各エントリにはそのエントリのコード番号と画数、各画を構成する折れ線の各頂点の座標値などを収めます。

アプリケーションでは、まず、見本一覧から見本図形 DB を構築します。その後、タッチパネルからの入力座標の配列を作成し、認識を開始する時点でその入力点列の配列と見本図形 DB のペアを図形認識ライブラリに与えます。

図形認識ライブラリはマッチングを行い、類似度が高かった上位の見本 DB のエントリを結果として返します。最終的に、アプリケーションプログラムはそのエントリからコード番号を取り出し、認識結果として使用します。



1.3　できること・できないこと

図形認識ライブラリを利用することで実現可能・不可能な応用例は以下のとおりです。

1.3.1　実現可能な利用例

- **戦闘中にタッチパネル全面に一筆書きで呪印を書くと、書き終わった次のターンに魔法が発動する**
  一筆に限定することで、認識を開始するタイミングがペンの上がった瞬間と自明になり、もっとも実装しやすいシナリオです。認識結果とともに類似度が帰ってきますので、類似度が低ければ発動しない、などの設定も可能です。
- **タッチパネル面に表示した地図上に記号を書くと書いた位置に建造物が出現する**
  タッチパネルからの入力座標情報を図形認識ライブラリに渡す前に、そのバウンディングボックスを計算しておくことで、認識結果を入力位置に反映させることが可能です。ただし、記号は書き順が限定されるような形に工夫する必要があります。

1.3.2 現時点では工夫が必要な利用例

- **複数画の連続した入力の中から図形を抽出する**
  現在実装されている認識アルゴリズムはいずれも認識対象の画数が正しいことを必要とします。すなわち、認識開始の画と、認識終了の画を正確に知る必要があり、余計な画が前後や間に挟まっている場合は認識できません。前後の画を削りながら図形認識ライブラリに認識させ、類似度の最も高いものを採用することで、アプリ側で対応することは可能ですが、前処理の再計算が発生しないようにするには、使用できる認識アルゴリズムに制限がつきます。(具体的には、入力サイズを固定するか、サイズの正規化の必要がない "Light" を使用する必要があります)
- **画面全体に数式を書いてもらって、それを計算する**
  同時に複数の図形を入力させて一度に認識する場合、どの画で図形が切れるか問題になります。基本的には前項の問題と同様です。横書きの数式に限定すれば、バウンディングボックスの位置関係から判定できる可能性がありますが、工夫が必要です。
  横書きひらがなの文字列認識になると、「に」と「1 こ」の区別など問題は難しくなり、動的計画法による最適分割の計算とヒューリスティックを組み合わせて実装することになります。
- **PC のマウスジェスチャーのように、特徴的なストローク入力をコマンドとして認識する**
  左一直線にペンを走らせると前の画面に戻り、右一直線にペンを走らせると次の画面に行き、上→左とカギ型にペンを動かすと画面を閉じる、といったようなインターフェイスを実装したいケースです。これを図形認識ライブラリを用いて実現することも可能ですが、上下左右の動きだけとりたいのであれば自前で実装してしまったほうが軽く実装できるかもしれません。もしくは、ペン入力からノイズを取り除くために図形認識ライブラリのリサンプリングの部分だけ利用するという方法もあります(`PRC_ResampleStrokes_*` をご参照ください)。実現したい機能の複雑さに応じて選択してください。
- **マップ上に斜めや逆さに書かれた家紋を認識して、その向きにあわせて軍団を出現させる**
  現在実装されている認識アルゴリズムはいずれも図形の向きにセンシティブです。少し斜めになるくらいであれば認識可能ですが、横向きや逆さになると認識することはできません。一般に、回転を許容する文字認識の手法としては 16 方向程度でパターンを回転させながら全てのケースでマッチングをとり、もっとも類似度が高いものを採用する、という方法がとられますが、当然のことながら 16 倍の計算時間がかかります。入力パターンを回転させる処理は、アプリケーション側で実装してください。逆に、見本図形をあらかじめ回転させた 16 パターン分を用意するという方法もあります。

1.3.3 現時点では不可能な利用例

- **キャラクターの絵を書いてもらい、どのキャラクターかを認識する**
  現在実装されている認識アルゴリズムはいずれも運筆情報を利用するもの(オンライン文字認識)で、筆順が正しくなければ認識しません。通常の文字を認識するのであれば、よく起こる筆順違いを見本 DB に入れてしまうことで対応できますが、どこから書き始めるのかわからない線画をマッチングさせることは困難です。
  入力パターンをビットマップとして認識するオフライン文字認識のアルゴリズムを用いることで、認識精度は落ちるものの筆順に関係ない認識が可能になりますが、SDK には含まれません。
- **走り書きの認識**
  現在実装されている認識アルゴリズムでは、画の切れ目を重要視しており、続け字で画の切れ目が失われている場合や、画の途中でかすれて途切れてしまっている場合には認識できません。見本 DB のエントリが少ない場合は、続け字に関してはありうるパターンを全て見本 DB に入れることで対応可能ですが、エントリ数が一定以上になると現実的ではなくなり、認識アルゴリズムでの対応が必要になります。

2 基本的な使用方法

2.1 データ構造

まず、図形認識ライブラリで使用するデータ構造について解説します。

2.1.1 基本データ型

```
#include <nitro/prc/types.h>

typedef struct PRCPoint
{
    s16              x;
    s16              y;
} PRCPoint;

typedef struct PRCBoundingBox
{
    s16              x1, y1; // バウンディングボックスの左上の座標
    s16              x2, y2; // バウンディングボックスの右下の座標
} PRCBoundingBox;
```

それぞれ、スクリーン座標を表す構造体と、バウンディングボックスを表す構造体です。原点(0,0)が左上で、y 軸は下向きだということに注意してください。

2.1.2 見本図形一覧型

```
typedef struct PRCPrototypeList
{
    const PRCPrototypeEntry *entries;
    int                     entrySize;
    const PRCPoint          *pointArray;
    int                     pointArraySize;

    int                     normalizeSize;
} PRCPrototypeList;
```

見本図形の一覧を表現するデータ型です。

次項で説明する PRCPrototypeEntry 型の配列とサイズ、そして PRCPrototypeEntry で用いられる頂点データの本体である PRCPoint 型の配列とサイズ、のペアで見本の一覧は構成されます。

normalizeSize は各頂点座標がどのレンジで表現されているかを表す数値です。見本図形一覧内の全ての頂点座標は、左上(0, 0)右下(normalizeSize-1, normalizeSize-1)のバウンディングボックスに収まっていなければなりません。

実際に認識に用いる際には、このデータから見本図形 DB に一度変換してから使用します。

2.1.3 見本図形 DB エントリ型

```
typedef struct PRCPrototypeEntry
{
    BOOL             enabled;
    u32              kind;
    u16              code;
    fx16             correction;
    void*            data;
    int              pointIndex;
    u16              pointCount;
    u16              strokeCount;
} PRCPrototypeEntry;
```

見本図形 DB の各エントリを収めるためのデータ型です。このうち、code と data はこのエントリと結び付けられた値としてアプリケーションが自由に使うことができます。code は u16 型ですので、コード値は 65535 までの値をとることができます。

enabled と kind は、認識関数が見本 DB の中からマッチングを行う対象を選択する際に参照されます。enabled が FALSE であるエントリは決してマッチングが行われることはありません。kind はこのエントリの図形種をビットフィールドで指定するためのメンバです。

```
例 1）
  kind = 1 → 数字
  kind = 2 → アルファベット
  kind = 4 → 半角記号
  kind = 8 → ひらがな

例２）
  kind = 1 → レベル１魔法
  kind = 2 → レベル２魔法
  kind = 4 → レベル３魔法
```

こうしておくことで、例えば、認識関数の呼び出しの際に kindMask 値を 3 に設定することにより、認識対象を「英数字」や「レベル 1 と 2 の両方の魔法」というような形で選択できます。

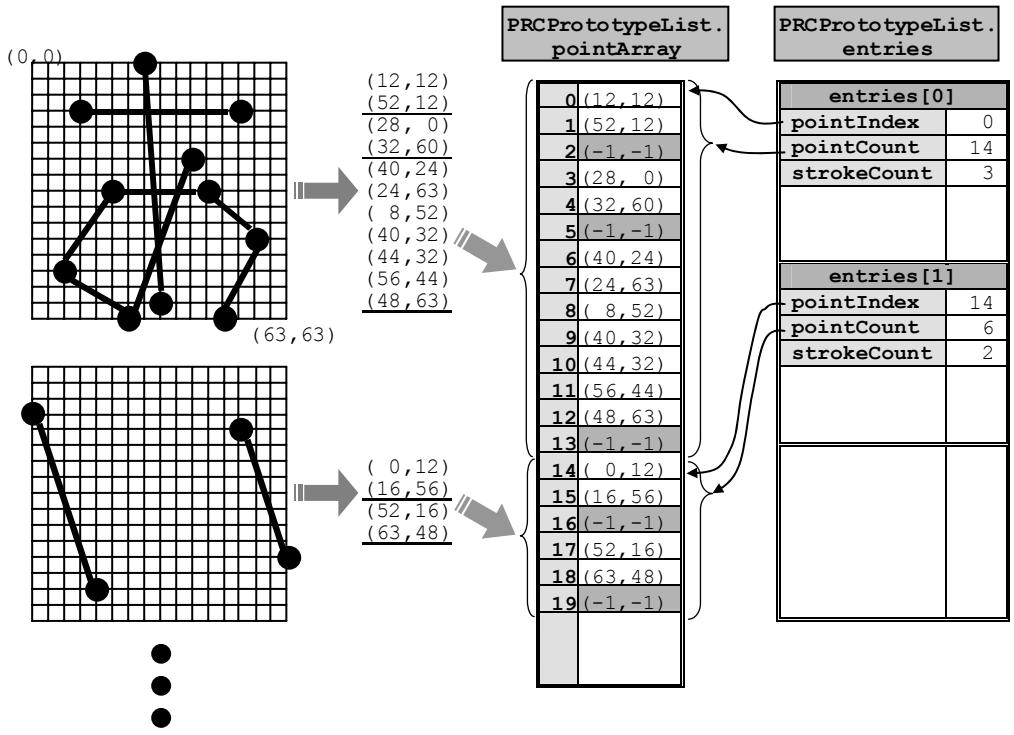
correction は入力パターンとこのエントリとの類似度を計算する際に用いられる補正値です。fx16 型ですので、4096 が 1.0 に相当します。0 で補正なしで、負で低めに補正、正で高めに補正です。補正値を 4096 に設定すると補正により類似度が常に 1.0(最高値)になります。計算式は以下のようになります。(score は fx32 型とする)

```
score = FX32_Mul(originalScore, FX32_ONE - correction) + correction
```

score はこの処理の後、0.0 ～ 1.0 からはみ出ないように上下で切られ、最終的な類似度となります。

pointIndex, pointCount, strokeCount はこのエントリが表す形を実際に指定しているメンバです。pointIndex はこのエントリが属する図形一覧の PRCPrototypeList.pointArray への添え字となっています。

見本図形一覧のデータ構造は下図のとおりです。



この例では、PRCPrototypeList.normalizeSize は 64 となります。

PRCPrototypeEntry の pointCount と strokeCount は重複した情報ですが、前処理の高速化のために両方の情報を見本 DB データには含めてください。

2.1.4 点列データ型

```
typedef struct PRCStrokes
{
    PRCPoint          *points;
    int               size;
    u32               capacity;
} PRCStrokes;
```

主に、タッチパネルからの生の入力座標情報を管理するための構造体です。capacity に最大格納可能数が、size に現在格納している点数が記録されています。

ライブラリで以下の操作が定義されています。

```
PRCStrokes strokes;
PRCPoint points[1024];

// strokes 構造体を初期化する
PRC_InitStrokes(&strokes, points, 1024);
// タッチパネルからの入力座標 (x, y) を追加
PRC_AppendPoint(&strokes, x, y);
// ペンがいったん上がったことを記録
PRC_AppendPenUpMarker(&strokes);
// 容量が一杯になったかをチェックする
PRC_IsFull(&strokes);
// クリアする
PRC_Clear(&strokes);
// 空か調べる
PRC_IsEmpty(&strokes);

PRCStrokes anotherStrokes;
PRCPoint anotherPoints[2048];
PRC_InitStrokes(&anotherStrokes, anotherPoints, 2048);

// ディープコピーする
PRC_CopyStrokes(&strokes, &anotherStrokes);

int i;
for ( i=0; i<strokes.size; i++ )
{
    if ( !PRC_IsPenUpMarker(&strokes.points[i]) )
    {
        // 通常処理
    }
    else
    {
        // ここでいったんペンが上がった
    }
}
```

2.1.5 認識アルゴリズム依存のデータ型

- **PRCPrototypeDB**

  PRCPrototypeList は最低限に近い見本図形の情報しかもっていません。しかし、認識処理を高速化するためには、見本図形一覧内の頂点データに対してあらかじめ前処理を行っておく必要があります。

  PRCPrototypeDB は見本図形一覧 PRCPrototypeList に PRC_InitPrototypeDB 関数によって前処理が行われた後のデータで、これを見本図形 DB と呼びます。実際の認識関数にはこちらを渡すことになります。

  内部構造は使用する認識アルゴリズムに依存しますが、現在実装されている認識アルゴリズムはいずれも共通のデータ構造を利用します。追加で計算されるものは以下の通り:各画の先頭点へのインデックス・各線分の長さ・各画の長さ・パターン全体の総長・各線分における画内での長さの割合・各ストロークの総長に対する長さの割合・各線分の角度・各画のバウンディングボックス・パターン全体のバウンディングボックス

- **PRCInputPattern**
  タッチパネルからの入力座標を収めた PRCStrokes のデータも認識関数に渡す前に前処理を行う必要があります。特に、タッチパネルからの入力はフレーム単位で取っている場合が多く、そのまま認識に用いるには点数が多すぎます。そこで、入力パターンの特徴を良く表している点のみを抽出するリサンプリング処理が必要になります。PRCInputPattern は PRC_InitInputPattern 関数によって、生の入力点列データに対し、リサンプリング処理および PRC_InitPrototypeDB と同一の追加計算を行った結果として作られる構造体です。

2.2 ライブラリ使用例

ライブラリの使用例を擬似コードの断片で以下に記します。

```
#include <nitro/prc.h>

#define RAW_POINT_MAX 1024 // 生の入力点をいくつまで保存するか
#define POINT_MAX 40 // リサンプリング後の入力点を最大いくつまで受け付けるか
#define STROKE_MAX 4 // 何画までの入力を受け付けるか
```

PRC* のヘッダファイルは nitro.h からは呼び出されません。図形認識ライブラリを使用する際には、必ず明示的に nitro/prc.h をインクルードする必要があります。ここで、nitro/prc.h をインクルードする代わりに、nitro/prc/algo_*.h を指定することで、デフォルトの図形認識アルゴリズムを選択することが可能になります。詳しくは後の各認識アルゴリズムの解説をご参照ください。

ここで、図形認識ライブラリを利用する際に最低限決めないといけないパラメータをマクロ定数として定義しています。

まず、RAW_POINT_MAX で指定している数値は、タッチパネルからの入力点をいくつまで受け取るかという最大数です。図形認識ライブラリは、認識対象の点の配列をまとめて受け取った上で処理しますので、いったんアプリケーション側で入力点をためておく必要があります。毎秒 60 点の入力を受け取る場合、1 文字の入力に最大 10 秒かかることを想定すると 600 点分の配列を用意する必要があります。

アプリケーション側から渡された生の入力情報は、前処理で特徴的な点だけに絞られます。これをリサンプリング処理、または特徴点抽出と呼びます。POINT_MAX と STROKE_MAX はその前処理後に許容される最大の点数と画数です。POINT_MAX を少なくしすぎると、長く複雑な入力をされた場合に、文字の後半が切れてしまうことになります。どのくらいの値が必要なのかは、入力させたいパターンの複雑さと、前処理(PRC_InitInputPattern*)でどの程度まで細かく点を残したいのかのパラメータ設定に依存します。

```
extern PRCPrototypeList PrototypeList;

// ワーク領域を確保して見本DBの展開
PRCPrototypeDB protoDB;
void* dictWork;
dictWork =
    OS_Alloc(PRC_GetPrototypeDBBufferSize(&PrototypeList));
PRC_InitPrototypeDB(&protoDB, dictWork, &PrototypeList);
```

PrototypeList は別ファイルで定義された見本図形一覧データだとお考えください。

PRC_InitPrototypeDB **関数を用いて、**PRCPrototypeList 型の見本図形一覧から PRCPrototypeDB **型の見本図形** DB **を構築**します。PRCPrototypeDB は内部で見本図形 DB の大きさに応じたメモリ量を必要としますので、PRC_GetPrototypeDBBufferSize によって得られたサイズのワーク領域を前もって確保しておき、それを初期化時に渡さなければなりません。

PRC_InitPrototypeDB は使用する見本図形のセットの総点数と総画数をカウントした後、あとの認識処理を高速化するための各画へのインデックス作成を行い、その後、各線分の長さや角度など、後の認識アルゴリズムが必要とする情報を事前計算して PRCPrototypeDB に収めます。

PRC_InitPrototypeDB **の兄弟関数**には、使用する見本図形種をビットフィールドで指定可能な**～**Ex もあります。**～**Ex を使用する際には対応する PRC_GetPrototypeDBBufferSizeEx **に** PRC_InitPrototypeDBEx **と同じ引数を与えて**ワーク領域のサイズ計算をしてください。

```
// 他の処理で使うワーク領域も確保しておく
void* inputWork;
inputWork =
   OS_Alloc(PRC_GetInputPatternBufferSize(POINT_MAX, STROKE_MAX));
void* recogWork;
recogWork = OS_Alloc(
   PRC_GetRecognitionBufferSize(POINT_MAX, STROKE_MAX, &protoDB)
);
```

ここで、後に認識処理で必要になるワーク領域も確保しておきます。複数の入力パターンを並行してプールすることを可能にするために、入力パターンの展開に必要なワーク領域と、比較処理に必要になるワーク領域を別々に確保しなければならなくなっています。また、認識処理をするごとにメモリを確保しなおさなくてすむように、あらかじめ必要になりうる最大値を指定して確保しておけるような仕様になっています。

```
// 入力点列データの初期化
PRCPoint points[RAW_POINT_MAX];
PRCStrokes strokes;
PRC_InitStrokes(&strokes, points, RAW_POINT_MAX);
```

タッチパネルからの生の入力データを保持する構造体を初期化します。

```
while ( 1 )
{
```

フレーム単位のループ処理に入ります。

```
   int x, y;
   if ( !PRC IsFull(&strokes) )
   {
      if ( タッチパネルから(x,y)に入力があった )
      {
         // (x,y) を入力点列に追加
         PRC_AppendPoint(&strokes, x, y);
      }
      else if ( 直前まで入力があった )
      {
         // ペンが上がったという印を挿入
         PRC AppendPenUpMarker(&strokes);
      }
   }
```

タッチパネルからの入力を PRCStrokes 構造体に追加していきます。タッチパネルからペンが離れたら、ちょうど 1 回だけ PRC_AppendPenUpMarker を呼び出して、ペンが上がったという印を追加しなくてはなりません。

```
   if ( 認識要求がきた )
   {
      // 現在の strokes の中身で認識開始
      // まずリサンプリング処理のパラメータを設定
      PRCInputPatternParam inputParam;
      inputParam.normalizeSize = protoDB.normalizeSize;
      inputParam.resampleMethod = PRC_RESAMPLE_METHOD_RECURSIVE;
      inputParam.resampleThreshold = 3;
```

生の点列データを認識処理用の PRCInputPattern 型のデータに変換するためのパラメータを指定します。normalizeSize に 0 以外の数値を指定すると、入力点列のバウンディングボックスを、指定されたサイズに合うように拡大縮小する正規化処理が実行されます。"Light" 以外の認識アルゴリズムは見本 DB とパターンの大きさが合っていることを前提に認識処理を行いますので、ここで見本図形 DB と大きさが等しくなるように正規化処理を行ってください。

resampleMethod と resampleThreshold では、生の入力点から特徴点だけ抽出する処理に用いるアルゴリズムとパラメータを指定します。詳しくは、後のリサンプリングのアルゴリズムの種類の解説を参照してください。

```
      // 生の入力点列にリサンプリングによるスリム化と
      // 長さなどの追加情報の事前計算を行い、inputPattern を作る
      PRCInputPattern inputPattern;
      PRC_InitInputPatternEx(&inputPattern, inputWork, &strokes,
          POINT_MAX, STROKE_MAX, &inputParam);
```

事前に確保しておいたワーク領域を使って、生の入力点列から PRCInputPattern 型の入力パターンデータを作成

します。

PRC_InitInputPattern は PRCInputPatternParam で与えられたパラメータに従って、サイズの正規化処理と、リサンプリング処理による特徴点抽出を同時に行い、その後、リサンプリングされた点列に対して、各線分の長さや角度などの事前計算を行い、PRCInputPattern 構造体に格納します。

```
        // inputPattern と protoDB の中の各エントリを比較し、認識を行う
        PRCPrototypeEntry* result;
        fx32 score;
        score = PRC_GetRecognizedEntry(&result, recogWork,
                                       &inputPattern, &protoDB);
```

認識の準備は全て整いました。あとは PRCPrototypeDB 型の見本図形 DB と PRCInputPattern 型の入力パターンデータを比較して、もっとも類似度の高い見本図形 DB のエントリを得るだけです。類似度は fx32 型で 0～1 の範囲、すなわち、int になおすと 0～4096 の範囲で返ってきます。

選択したアルゴリズムや見本 DB の分量などによっては数十ミリ秒以上かかる重い処理になりますので、メインの処理とは別のスレッドで動作させることを推奨します。実装例としてデモの prc/characterRecognition-1 をご参照ください。

また、PRC_GetRecognizedEntry の兄弟関数として、認識対象の図形種をビットフィールドで指定する～Ex 関数、類似度の上位 N 位まで返せる～Entries 関数があります。リファレンスマニュアルをご参照ください。

```
        // 結果を出力
        OS_Printf("code: %d¥n", PRC_GetEntryCode(result));
```

認識結果は PRCPrototypeList を構成している PRCPrototypeEntry へのポインタとして帰ってきます。PRC_GetEntryCode, PRC_GetEntryData でそれぞれのコード値とユーザデータを得ることができます。

```
    }
    V-Sync を待つ処理
}
```

3 各種設定エントリ

3.1 リサンプリングパラメータ

PRC_InitInputPattern で実行されるリサンプリング処理では、処理方法をいくつかのアルゴリズムから選択することができます。

3.1.1 PRC_RESAMPLE_METHOD_NONE

リサンプリング処理を行いません。同一座標の点が連続していた場合のみ、取り除かれます。すでにリサンプリング処理が行われている点列データに対して改めて処理を行う必要があるときに使用することを想定しています。

3.1.2 PRC_RESAMPLE_METHOD_DISTANCE

移動距離を基準にリサンプリングを行います。

各画に対し、始点と終点を採用したのち、始点から移動距離の累計が一定値を超えるごとに点を採用していきます。距離の計算には、通常のユークリッド距離ではなく、x 座標の差と y 座標の差の和であるシティブロック距離を用いますので、正確さは多少落ちるものの高速に処理されます。

resampleThreshold には、移動距離の累計がいくつ以上になったら次の点を採用するかを指定します。

他のものに比べて処理が一番早いですが、震えるペン先でゆっくりと入力された場合、すぐに移動距離が閾値を超えて点を取りすぎてしまうという問題があります。また、それ以外でもあまりいい特徴点抽出ができない傾向があります。

3.1.3 PRC_RESAMPLE_METHOD_ANGLE

曲がり方に応じてリサンプリングを行います。

各画に対し、まず始点と終点を採用します。その後、始点から出る線分の角度を記憶し、次々に線分を辿っていって角度の差が閾値を越える直前の点を、始点の次の点として採用します。その後は、前の前に採用した点と直前に採用した点とを結んだ線分の角度と、直前に採用した点から現在の点を結んだ線分の角度の差が閾値を越えた時点で現在の点を採用することを繰り返します。

角度計算には内部でテーブル引きを行う FX_Atan2Idx を使用しますので、速度はそれなりに出ます。FX_Atan2Idx の精度はあまり高くありませんが、この用途で使う分には十分に有効です。

resampleThreshold には、角度の閾値を指定します。この数値は一周を 0～65535 で表現した単位で表します。

なお、採用される点は前に採用された点からシティブロック距離が PRC_RESAMPLE_ANGLE_LENGTH_THRESHOLD 以上離れていなければなりません。これはあまり近すぎると有効な角度が取れないためです。PRC_RESAMPLE_ANGLE_LENGTH_THRESHOLD は現在は 6 で固定になっています。この距離計算は正規化前の座標値で評価されます。

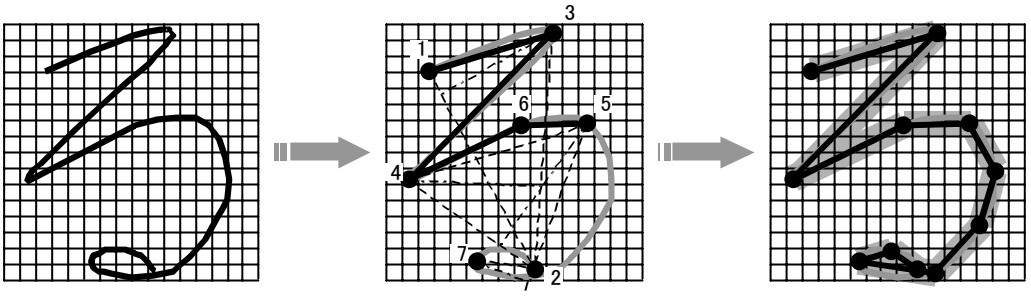
閾値を大きくしてリサンプリング後の点数をある程度少なくしたい場合でも、小さいループ部分まできちんと取ることができますので、いい特徴点抽出ができるケースが多くなります。反対に閾値を小さくしすぎると、ちょっとした入力のゆがみを拾ってしまいやすくなります。なお、計算時間は入力点数に対して線形です。

3.1.4 PRC_RESAMPLE_METHOD_RECURSIVE

再帰的に処理をし、もっとも特徴的な点から採用していきます。

まず、各画の始点と終点を採用し、それぞれ、点 A と点 B と置きます。点 A と点 B の間の全ての点について、点 A と点 B とを結ぶ直線との距離を求め、距離が最大になる点 C を求めます。その距離が resampleThreshold 以上であれば採用し、未満であれば点の選択を止めて点 A から点 B の辺を確定します。点 C を採用した場合は、点 A と点 C、点 C と点 B を 2 組の新しい点 A と点 B として、以上の処理を再帰的に繰り返します。

最終的には、リサンプリング後のパターンを左右に resampleThreshold ずつ膨らませた領域の中に、処理前の生の入力ストロークが完全に収まります。ただし、途中でリサンプリング点数の上限に達した場合は、その限りではありません。



生の入力データ　　リサンプリング途中　　処理結果の折れ線を中心として幅 resampleThreshold×2 の範囲に元のデータは収まっている

入力パターンとして想定される最小のループの大きさよりも小さな resampleThreshold を設定することで、ループの情報を失わず、かつ十分にコンパクトな点の集合までリサンプリング結果を落とすことができます。リサンプリング結果がコンパクトになると、実際に認識を行う際の計算時間が短くなります。

リサンプリング処理そのものの計算時間は、オーダーで考えると最悪で入力点数とリサンプリング後点数の積のオーダーになりますが、ひらがな程度の典型的な入力では、処理後に同程度のサンプリング点数となるようにパラメータを調整した PRC_RESAMPLE_METHOD_ANGLE より少々遅いくらいの時間で処理は終了します。

3.2 認識アルゴリズム

図形認識を行うアルゴリズムは、現時点で 4 種類実装されています。

認識アルゴリズムは、どのヘッダファイルを最初に include したかによって選択されます。

```
#include <nitro/prc/algo_light.h>     → 認識アルゴリズム "Light"
#include <nitro/prc/algo_standard.h>  → 認識アルゴリズム "Standard"
#include <nitro/prc/algo_fine.h>      → 認識アルゴリズム "Fine"
#include <nitro/prc/algo_superfine.h> → 認識アルゴリズム "Superfine"
```

#include <nitro/prc.h> とすると、上記の 4 つのヘッダファイルが全て読み込まれますが、最初に algo_standard.h が読み込まれるため、**"Standard" がデフォルトの認識アルゴリズムとして使用されます**。

ライブラリ関数の中で認識アルゴリズムに依存して変わるのは以下の型と関数です。

```
PRCPrototypeDB
PRCInputPattern
PRCPrototypeDBParam
PRCInputPatternParam
PRCRecognizeParam

PRC_Init
PRC_GetPrototypeDBBufferSize*
PRC_InitPrototypeDB*
PRC_GetInputPatternBufferSize
PRC_InitInputPattern*
PRC_GetInputPatternStrokes
PRC_GetRecognitionBufferSize*
PRC_GetRecognizedEntry*
```

各認識アルゴリズムごとに、上記の識別子名の後ろに "_<アルゴリズム名>" という suffix がついたものが定義されており、最初に読み込んだヘッダファイルにおいて、それらが上記の標準名に alias されます。2 番目以降に include した認識アルゴリズムを使用したい場合は、明示的に "_<アルゴリズム名>" を後ろに付けた型名・関数名を使用してください。（例:PRCRecognizeParam_Light, PRC_InitPrototypeDBEx_Fine）

しかし、これらのうち、以下のものは現在の実装においては全ての認識アルゴリズムにおいて共通となっています。

```
PRCPrototypeDB
PRCInputPattern
PRCPrototypeDBParam
PRCInputPatternParam

PRC_Init
PRC_GetPrototypeDBBufferSize*
PRC_InitPrototypeDB*
PRC_GetInputPatternBufferSize
PRC_InitInputPattern*
PRC_GetInputPatternStrokes
```

これらに関しては、各アルゴリズムが共通に参照する、"_Common" という suffix がついた型・関数が用意されています。(例:PRCPrototypeDB_Common)

現状では、認識を実際に行う PRC_GetRecognizedEntry* 関係のライブラリ以外は共通です。それを利用し、デモの prc/characterRecognition-2 では、共通の見本 DB と入力パターンデータを使用して全ての認識アルゴリズムを同時に使用しています。複数認識アルゴリズムの同時利用のサンプルとしてご参照ください。

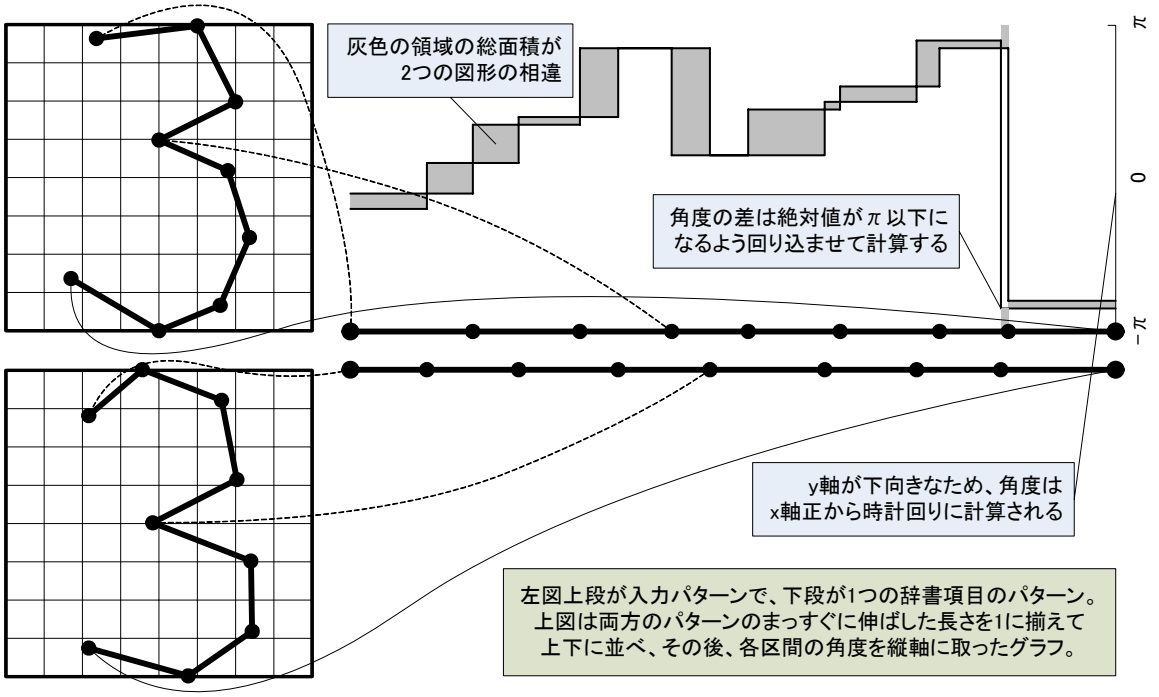
以下に、各アルゴリズムの概要を述べます。

以下の記述において、あいまいな表現がしばしば出てきますが、これは条件により計算時間や認識精度などが大きく変わるためです。速度に関する記述などはあくまでも参考程度にお考えください。必ず、アプリケーションで用いるデータで実測して確認した上で、認識アルゴリズムや各種パラメータを選択してください。

3.2.1 認識アルゴリズム："Light"

"Light" は最も軽量な認識アルゴリズムです。見本 DB 内の各図形が互いにあまり似ていなく（誤認識を起こしづらく）、かつ一筆書きの図形を認識したいのであれば、最適なアルゴリズムとなります。

"Light" は角度のみを比較します。入力パターンと見本パターンの画の双方の長さが全体が 1 になるように拡縮させたのちに、角度の差の積分を取り、全て 180 度逆だったときが 0.0、全て一致していたときが 1.0 になるように調整した値を類似度として返します。



上図が角度の差をグラフ化したものです。

"Light" は複数画のパターン同士の場合は、画ごとに同様の計算を行った後に、見本パターンにおける各画の長さの比で重み付けをして類似度の平均をとることで対応しますが、各画の相対位置は一切見ないため、"T"と"+"の区別がつかないなどの根本的な問題を抱えています。基本的に、一筆書きの図形をできるだけ高速に認識するためのアルゴリズムとして設計されています。

計算時間は入力パターンの点数と見本 DB エントリ数の積に比例します。

3.2.2 認識アルゴリズム："Standard"

"Standard" は標準的に使用できる認識アルゴリズムとして設計されています。呪印を正しく入力せよ、などユーザに対し手本どおりに入力することを強要できるケースにおいて、最適に用いることができます。

"Standard" は角度と位置の両方を比較します。"Light" と同様に入力パターンと見本パターンの長さを 1 にそろえ、角度の差と位置の差の積の積分を取ります。このとき、位置の差はシティブロック距離で、なおかつ正確な位置の差ではなく、近いサンプリング点の座標で近似されます。その後、"Light" と同様に 0.0 がもっとも離れている場合の類似度、1.0 が同一だった場合の類似度となるように調整を行い、スコアとして返します。

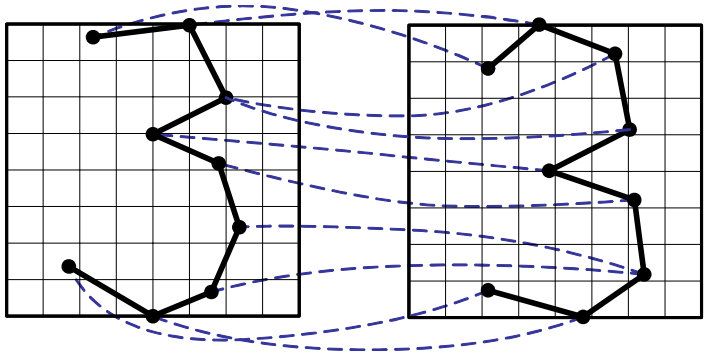
位置も考慮しているため、複数画の図形も問題なく認識することができます。類似度を求める際には、画ごとに上記の計算を行った後に、見本 DB エントリのパターンと入力パターンの両方について、パターン全体に対してその画の長さが占める割合を調べ、より大きいほうの値で重み付けを行い、全ての画のスコアの平均を取ります。

計算時間は "Light" の 2～3 倍程度とそれなりにお手ごろで、メインスレッドの空き時間に認識スレッドを走らせたとしても、それほどストレスにならない時間で結果が返ってくることが期待できます。

3.2.3　認識アルゴリズム："Fine"

"Fine" は歪んだ文字も認識可能なアルゴリズムとして設計されています。文字入力など、ユーザの歪んだ入力をアプリケーション側でできるだけ救済しないといけないケースにおいての使用が想定されています。

"Fine" は角度と位置の両方を比較するのと同時に、伸縮マッチング（elastic matching）を行います。入力パターンと見本パターンを当比率での拡縮で合わせるのではなく、画に沿って部分的に引き伸ばしたり縮めたりして、もっとも評価値が良くなるようにマッチングをとります。



上図は伸縮マッチングの例です。左の入力パターンと右の見本 DB エントリのパターンとで、頂点同士のマッチングを取ります。お互いに複数の点が一つの点にマッチングされているのが分かるかと思います。複数点が同一の点にマッピングされることも許容しながら、もっともスコアが高くなる組み合わせを探すことにより、例えば上図では歪みをうまく吸収し、見本とは上下の大きさのバランスが異なった「3」にも高いスコアをつけることができています。このように、伸縮マッチングは入力の歪みに強い認識手法となります。

スコアの値は、対応付けられた各頂点に関して、

（正規化サイズ×2 － シティブロック距離）×（π － その頂点に入る線分の角度の差）

を計算し、平均したものを 0.0～1.0 に分布させた値です。この値を最大にするように頂点の対応付けが行われます。

伸縮マッチングは動的計画法（Dynamic Programming）を用いたアルゴリズム（DP マッチング）で行います。ビームサーチは実装されていません。そのため、計算時間は入力パターンの点数と見本パターンの点数の積に比例し、典型的な利用法では "Standard" に比べて数倍からそれ以上遅くなることが多くなります。

3.2.4　認識アルゴリズム："Superfine"

"Superfine" は現在実装されている中でもっとも計算時間が長い認識アルゴリズムです。しかし、"Fine" に比べて常に認識精度がよいというわけでもありません。"Fine" では認識精度が出ない場合に使用してみてください。

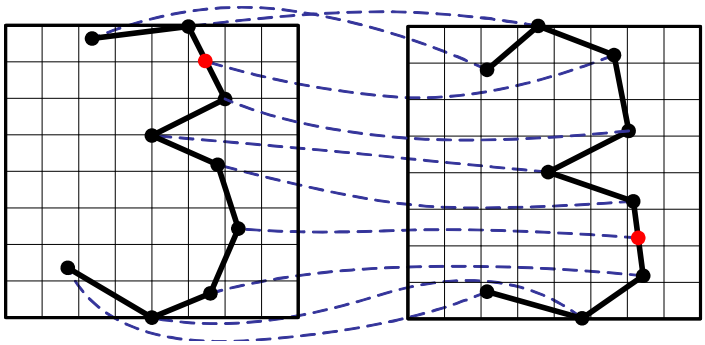
"Superfine" はまず "Fine" と同様の伸縮マッチングを行います。"Fine" では伸縮マッチングで用いた評価値をそのままスコアとして返しますが、"Superfine" では伸縮マッチングはどの点とどの点が対応するのかという情報を抽出するために用いられます。伸縮マッチングによってもっとも確からしい頂点対を決定し、対応する点が確かでない頂点は前後の辺の長さに比して相手に仮想的な点があるものと補完した上で、"Fine" と同様なスコア計算を行います。

この際、最終的なスコアの値は、"Fine" とは異なり、各点対に対して

（正規化サイズ×2 － シティブロック距離）×（cos その頂点に入る線分の角度の差）

をさらにその点に入る辺の長さ（画全体に対する比率）による加重平均したものになります。

なお、頂点対を DP マッチングで決定する際は、辺の長さに関しては "Fine" と同様に考慮されませんが、角度のスコアは cos を使用したものになります。



上図の赤い点が補完結果の仮想的な点です。

"Fine" と同様の処理を行った上に、補完点の計算のために割り算が頻繁に発生するため、多くの場合 "Fine" の数倍程度の計算時間がかかります。

4 活用のためのヒント・テクニック

4.1 パラメータの設定

まず、デモの prc/characterRecognition-2 でいろいろとパラメータを変化させてみて、消費メモリ量と計算時間と精度の関係を体感するのがパラメータ調整の早道でしょう。その際、与える見本図形 DB の性質によって、振る舞いが変わりますので、できる限り実際のアプリケーションで用いる見本図形 DB に近いデータを載せて調整することが必要です。characterRecognition-2 デモの使用法はこのドキュメントの最後の Appendix に記載されています。

認識されすぎる図形などがあった場合は、各図形見本エントリの correction 値により細かい調整が可能ではありますが、微妙な調整を無駄に繰り返すような事態に陥りがちです。同じコード値に対して複数の見本エントリを登録することが可能ですので、認識しづらい図形について、認識するようになるまで新しい見本データを追加していくほうがバランスは取りやすいでしょう。

4.2 FAQ

□ **Q. `PRC_InitPrototypeDB*` と `PRC_GetRecognizedEntry*` と、両方に `kindMask` が指定できますが、どちらで図形種を選択すればよいのでしょうか。**

A.

どの程度比較対象の図形種を頻繁に切り替えるかによります。PRC_InitPrototypeDB の段階で制限すると、見本 DB の展開に必要なメモリが減りますが、その代わりに柔軟にマッチング対象の図形種を切り替えることはできなくなります。

□ **Q. 複数の画で構成される図形をできるだけ軽く認識させたいのですが、"Light" は使えないのでしょうか。認識精度はそこまで要求しないのですが、さすがに "p" と "b" をまったく区別できないのは困ります。**

A.

"Light" で複数画の図形をそれなりに処理させる方法もあります。それは、PenUpMarker を使わない、という手法です。ペンが上がったときに、通常は PRC_AppendPenUpMarker で画が途切れたことを記録しますが、その代わりにただ新規の点を追加しないという処理に変更することにより、図形認識ライブラリは複数画の入力をひと繋がりの画として処理します。同様に、見本 DB もあらかじめ画の切れ目を入れずに作成しておくことにより、"Light" アルゴリズムでも複数画の位置関係を反映した認識が可能になります。

また、このテクニックは、続け字や、画の途中の切れへの対応としても有効です。ただし、当然、意図せぬマッチングが起こる可能性が上がりますので、パターンの選定には十分に注意をしてください。

□ **Q. リサンプリング処理の結果をゲーム内のほかの処理にも利用したいのですが、できるでしょうか。**

A.

PRCInputPattern に対して PRC_GetInputPatternStrokes を使用してください。なお、PRCInputPattern の内部データへのポインタを直接張りますので、第 1 引数は PRC_InitStrokes で初期化しておく必要はありません。書き換えたい場合など、必要に応じて PRC_CopyStrokes でコピーしてから使用してください。

リサンプリングのみを行う場合は、PRC_ResampleStrokes* が使用できます。インデックスの配列で帰ってきますので、そこから PRCStrokes への変換はアプリケーション側で実行してください。

* Appendix

A.1 デモ

NitroSDK の $NITROSDK_ROOT/build/demos/prc/ 以下には、図形認識ライブラリのデモが収められています。以下はその概要です。

A.1.1 characterRecognition-1

図形認識ライブラリを使用する場合は、1 フレームで計算時間が収まらない場合がしばしば起こり得ることと、入力パターンの複雑さによって計算時間が大きく異なることが問題となります。そのため、実際に使用する場合には、メインスレッドとは別に図形認識用のスレッドを立てて、メインスレッドが処理を終えてから V-Blank 割り込みが発生するまでの空き時間に処理を行う形の実装が望ましいでしょう。characterRecognition-1 はそうした別スレッドを立てる場合の参考実装となっています。

A ボタンで認識を行い、B ボタンで画面をクリアします。

デモでは、テスト用に数字・英小文字・ひらがな、そして少量の記号を登録した、総エントリ数 161 の見本図形 DB が入っています。英数字には 1 つの文字に対して異なる字形のパターンが複数入っていますので、認識可能文字種類は 117 となっています。なお、これはデモ用見本 DB です。実際に使用される際には、要求される処理速度と精度に応じたサンプリング点数と、標準的な字形を用い、アプリケーションに合わせた見本図形 DB を作り直す必要があります。

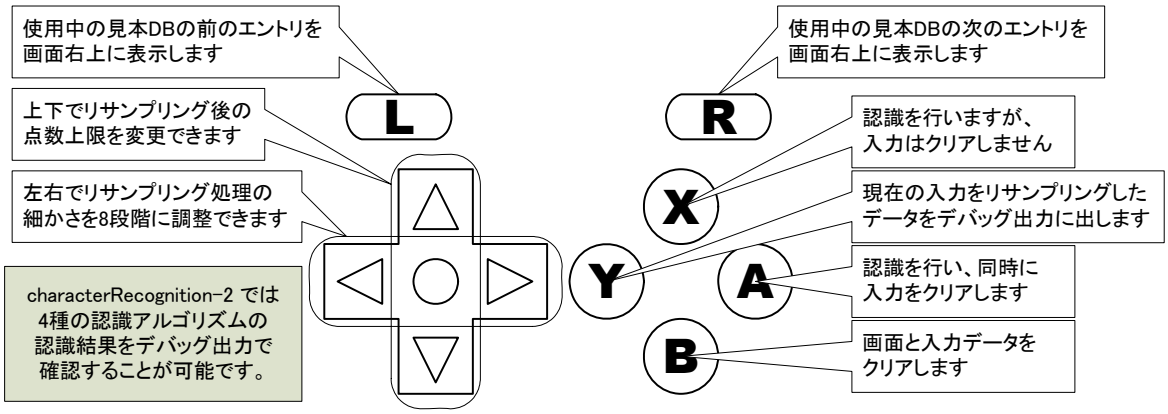
A.1.2 characterRecognition-2

複数の図形認識アルゴリズムを比較することを目的とした認識デモです。使用している見本 DB について、サンプリング後に許容する最大点数 maxPointCount を変更するとワーク領域のサイズがどのくらいになるのか、リサンプリングのパラメータを調整すると認識にかかる時間と結果はどのように変化するのか、などを実機で実行しながら確認することができます。

リサンプリングのパラメータとしては、3 種類のリサンプリングアルゴリズムの結果が似たようなサンプリング点数になるように調整した threshold の組を荒いものから細かいものまで 8 セットが用意され、実行時に切り替えることが可能です。

characterRecognition-1 と同様のデモ用見本図形 DB が搭載されていますが、実際にアプリケーションで用いる見本図形 DB にデータを載せ代えて、各種パラメータの調整に使用することを想定しています。

実行すると、タッチパネルに自由に図形を書き、A ボタンを押すと、画面上部に4つの図形が表示されます。左から 3 つがリサンプリング処理の結果で、順に PRC_RESAMPLE_METHOD_DISTANCE, ANGLE, RECURSIVE です。一番右側が認識結果の図形の見本データです。デバッグ出力には各アルゴリズムでの詳しい認識結果が表示されます。



また、このデモは簡易的なパターン作成ツールとしても使用可能です。

左右でリサンプリングのパラメータを調整したのちに、ペンで図形を書き Y ボタンを押すと、デバッグ出力にリサンプリングアルゴリズム 3 種それぞれを適用した場合のリサンプリング結果のパターンデータをテキスト形式で出力します。このテキストデータを行単位でコピー＆ペーストし、必要なパターンを集めたテキストファイルを作成した上で、

```
$ perl $NITROSDK_ROOT/tools/bin/pdic2c.pl <出力正規化サイズ> <見本DBテキストデータ>
```

を実行すると、標準出力に、そのまま図形認識ライブラリに読み込ませることが可能な見本図形一覧の C ソースコードが出力されます。図形認識ライブラリの動作確認などにご利用ください。

pdic2c.pl の入力フォーマットなどの詳細は、リファレンスマニュアルに記載されています。