TWL-SDK
Using the Pattern Recognition Library

Version 1.0.5

The content of this document is highly confidential and should be handled accordingly.

TWL-06-0039-001-B
Released: August 7, 2009

Table of Contents

Code

Figures

Revision History

Version	Revision Date	Description
1.0.5	2009/02/27	Deleted hiragana characters from the prototype database in section A.1 characterRecognition-1.
1.0.4	2008/10/16	Changed wording for inclusion in TWL-SDK.
1.0.3	2007/10/05	Updated information to match current conditions.
1.0.2	2005/02/18	Initial version.

1 Overview of Pattern Recognition

1.1 Introduction

TWL-SDK includes a pattern recognition library (PRC*) that provides rudimentary pattern recognition functionality. This document provides a basic explanation of how to use the pattern recognition library, characteristics of various recognition algorithms, and guidelines for tuning your application.

The pattern recognition library was designed to facilitate the use of the touch panel as an input device. If you only need handwritten character input functionality from the touch panel, consider using the Decuma Handwritten Character Recognition library, which is provided separately. You can use it for free if you agree to the terms in the End-User License Agreement.

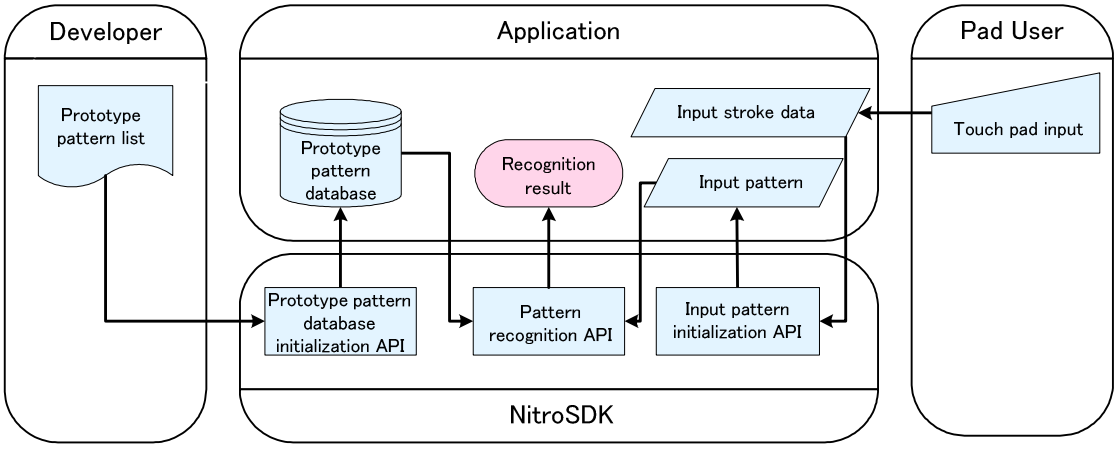
1.2 Library Functionality

The functionality provided by the pattern recognition library is fairly elementary. You must first prepare a list of pattern prototypes. Each entry in the prototype list contains a code number, stroke count, and the coordinates of vertices in the segments that make up each stroke.

The application program first creates a prototype database from the prototype (or prototype pattern) list. It then creates an array of input coordinates based on the touch panel input. When the application program begins the recognition process, it passes the input stroke data and prototype database to the library.

The library performs matching and returns the prototype database entry that has the closest match. Finally, the application program reads the code number of the entry and uses it as the recognition result.

Figure 1-1 Pattern Recognition Library



1.3 What the Library Can and Cannot Do

Examples of what the pattern recognition library can and cannot do are provided in the following sections.

1.3.1 Examples of Possible Uses

- **During a battle, the player writes a magic symbol on the touch panel screen, causing a spell to take effect in the turn (that follows immediately) after the symbol is written.**

  This feature is fairly easy to implement, because it is limited to a single stroke and it is clear that the recognition process should begin after the pen is lifted from the screen. The library returns both the recognition result and the degree of similarity. The application can be configured to permit the spell to take effect only if there is a close match.

- **When the player writes a symbol on a map displayed on the touch panel, a building appears in the location where the symbol was written.**

  If a bounding box is defined before the touch panel coordinate data is passed to the pattern recognition library, the recognition results can be displayed in the input location. The symbol must be devised in a shape that restricts the order in which it is written.

1.3.2 Examples of Possible Uses that Currently Require Workarounds

- **Recognizing patterns from multiple, continuous stroke input**

  Currently implemented recognition algorithms require the player to write each line in the correct order. In other words, the library must recognize precisely the stroke that initiates recognition and the stroke that completes it. If extraneous strokes are input at any point in the process, recognition becomes impossible. If strokes at the beginning or end are omitted, the pattern recognition library will return the entry that most closely matches the input. You can design your application to handle this result accordingly. However, if you design an application to avoid preprocessing recalculation, the restrictions will be applied to the recognition algorithm that can be used. (In particular, you must either fix the input size or use the Light algorithm, which does not require size normalization.)

- **Performing a calculation based on a formula written on the screen**

  If recognition of a series of drawn patterns is attempted simultaneously, the library may have trouble determining where each pattern begins and ends. This is essentially the problem described in the paragraph above. If you limit your application to formulas written horizontally, the library may be able to discern individual symbols by determining where their bounding boxes overlap, but this will require some creative coding.

  In trying to recognize a horizontal string of hiragana characters, the library may have trouble distinguishing “に” from “1こ”. However, this type of application could be implemented by using a combination of Dynamic Programming-based optimal splitting calculations and heuristics.

- **Reading commands from specific stroke input, similar to how mouse gestures can be used for PC input**

  You can create an interface that interprets a leftward stroke as a request to return to the previous screen and a rightward stroke as a request to advance to the next screen. A hook-shaped pattern of pen movement (upward and to the left) might indicate that the screen should be closed. You can use the pattern recognition library to implement this; however, if you need only recognition of up/down and left/right strokes, it may be simpler to code this on your own or use only the resampling functionality of the library to remove noise from pen strokes. (See `PRC_ResampleStrokes_*`.) The choice will depend on the complexity of your application.

- **Moving an army based on the rotation angle of a symbol written on a map**

  All currently implemented recognition algorithms are sensitive to pattern's orientation. The recognition algorithms will recognize a pattern written at a slight angle, but they cannot discern patterns written sideways or upside-down. One way to permit the rotation of a pattern is to rotate it in sixteen different directions and attempt a match for each pattern orientation. The rotated pattern that best matches the pattern in the database is selected. Note that this process will increase the recognition calculation time 16-fold; to explain, 16 match attempts are performed rather than a single match attempt, as would be the case in a simple 1-to-1 pattern matching. This process is best implemented on the application side. Conversely, you may also prepare 16 patterns by rotating a sample pattern in advance.

1.3.3 Applications Not Currently Possible

- **Asking the player to draw a character and recognizing which character it is**

  All currently implemented recognition algorithms use stroke information to find a match. (This method is called "online character recognition.") They can only recognize patterns written in the correct stroke order. To improve recognition of normal characters, you can store characters written with commonly made stroke order mistakes in the database of prototype patterns. However, the library cannot match line drawings that have no constraints on stroke order.

  It is possible to solve this problem by using a recognition algorithm based on bitmap images (this is known as "offline character recognition"), but the degree of matching accuracy will suffer. This algorithm is not included in the SDK.

- **Recognition of cursive writing**

  Currently implemented recognition algorithms rely on clear breaks between strokes. The recognition algorithms cannot recognize characters written without breaks or characters that have non-solid strokes. If there are few entries in the prototype database, for cursive characters you can store all possible character combinations as separate patterns in the prototype database; however, this approach may cause the number of entries to become unmanageable. Thus, an algorithm-based recognition approach is more practical.

2 Basics of Library Use

2.1 Data Structures

First, we will examine the data structures used with the pattern recognition library.

2.1.1 Basic Data Types

Code 2-1 Basic Data Types

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

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

typedef struct PRCBoundingBox
{
   s16              x1, y1; // Upper-left coordinate of bounding box
   s16              x2, y2; // Lower-right coordinate of bounding box
} PRCBoundingBox;
```

`PRCPoint` is a structure that expresses screen coordinates, and `PRCBoundingBox` is a structure that defines the bounding box. Note that the origin (0,0) is at the upper left and that the y-axis runs downward.

2.1.2 Prototype List Type

Code 2-2 Prototype List Type

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

   int                       normalizeSize;
} PRCPrototypeList;
```

This data type is used for the list of prototype patterns.

Prototype list comprises an array of `PRCPrototypeEntry` (explained in section 2.1.3 Prototype Database Entry Type) and its size, and an array of `PRCPoint` (used to store the vertex data in `PRCPrototypeEntry`) and its size.

The `normalizeSize` member defines the acceptable range of vertex coordinates. In the prototype list, all vertex coordinates must be within the bounding box defined by (0, 0) and (`normalizeSize` - 1, `normalizeSize` - 1). Before being used for actual recognition, this data is converted into a form that can be used by the prototype database.

2.1.3 Prototype Database Entry Type

Code 2-3 Prototype Database Entry Type

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

This data type is used for entries in the prototype database. Of its members, `code` and `data` can be freely used by the application as values that are linked to the entry. The `code` member is of type `u16` and can have a value of up to 65,535.

The `enabled` and `kind` members are referenced when the recognition function searches the prototype database for matches. Entries that have `enabled` set to `FALSE` are not considered for matching. The `kind` member uses a bit field to specify the type of pattern.

Example 1:

kind = 1 → Numeral

kind = 2 → Alphabetic character

kind = 4 → Half-size symbol

kind = 8 → Hiragana

Example 2:

kind = 1 → Level 1 spell

kind = 2 → Level 2 spell

kind = 4 → Level 3 spell

For example, if `kindMask` is set to 3 when the recognition function is called, matching will be limited to English letters and numerals, or Level 1 and 2 spells.

The `correction` value is used to calculate similarity between the input pattern and entry; it is of type `fx16`, and the 4,096 value corresponds to 1.0. If set to 0, there is no correction. A negative value results in a low-level of correction and a positive value in high-level correction. If the correction value is set to 4,096, the post-correction similarity will always be 1.0 (the maximum). The following formula is used (`score` is of type `fx32`).

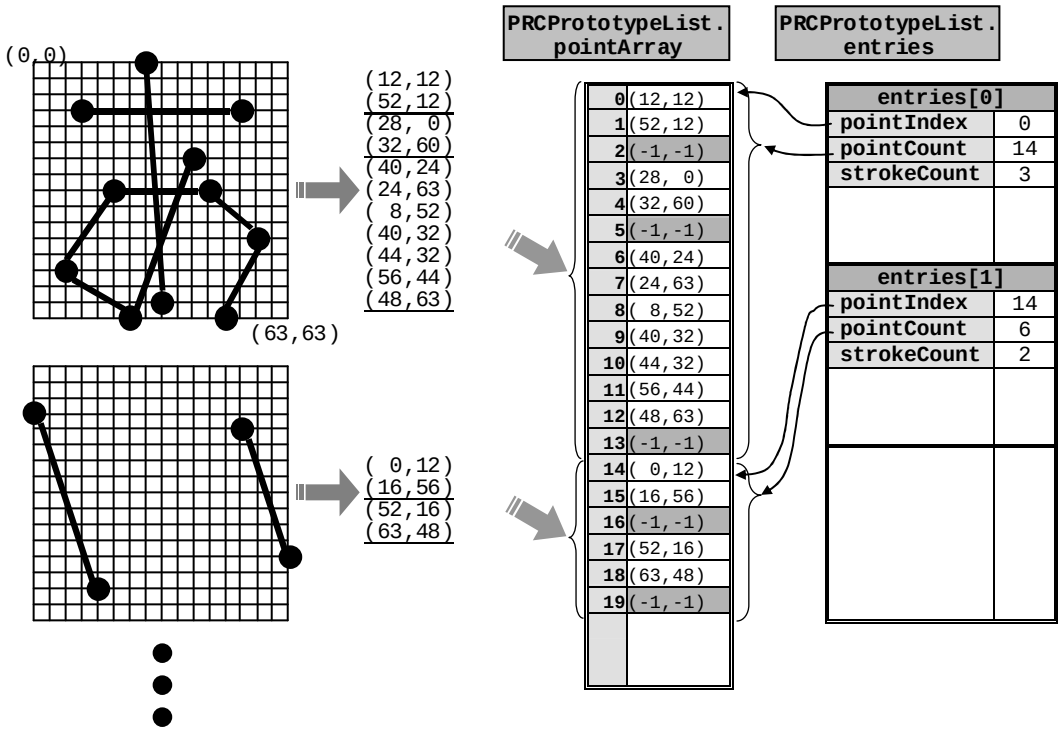
score = FX32_Mul(originalScore, FX32_ONE - correction) + correction

After processing, a score below 0.0 becomes 0.0, and a score above 1.0 becomes 1.0. This number is the final measure of similarity.

The `pointIndex`, `pointCount`, and `strokeCount` members specify the actual pattern defined by this entry. The subscript `pointIndex` specifies the location of this pattern in the pattern list's `PRCPrototypeList.pointArray`.

An example of a prototype list data structure is shown in Figure 2-1, below.

Figure 2-1 Prototype List Data Structure



In this example, `PRCPrototypeList.normalizeSize` is 64. Information contained in the `pointCount` and `strokeCount` members of `PRCPrototypeEntry` is redundant, but you should include both members in your prototype database to speed up preprocessing.

2.1.4 Stroke Data Type

Code 2-4 Stroke Data Type

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

This structure is used mainly to manage the raw input coordinate data from the touch panel. `capacity` is the maximum number of points that can be stored and `size` is the current number.

Operations defined in the library are shown in Code 2-5.

Code 2-5 Library-Defined Operations

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

// Initializes the strokes structure
PRC_InitStrokes(&strokes, points, 1024);
// Adds a set of input coordinates (x, y) from the touch panel
PRC_AppendPoint(&strokes, x, y);
// Records the fact that the pen has been lifted from the screen
PRC_AppendPenUpMarker(&strokes);
// Checks if the structure has reached its capacity
PRC_IsFull(&strokes);
// Clears the structure
PRC_Clear(&strokes);
// Checks if the structure is empty
PRC_IsEmpty(&strokes);

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

// Makes a deep copy
PRC_CopyStrokes(&strokes, &anotherStrokes);

int i;
for ( i=0; i<strokes.size; i++ )
{
```

```
    if ( !PRC_IsPenUpMarker(&strokes.points[i]) )
    {
        // Ordinary processing
    }
    else
    {
        // The pen was lifted at this point
    }
}
```

2.1.5 Data Types Dependent On Recognition Algorithm

- **PRCPrototypeDB**

  PRCPrototypeList stores only the bare minimum of prototype data. To speed up recognition, you must preprocess the vertex data in the prototype list. Use PRC_InitPrototypeDB to preprocess PRCPrototypeList (the prototype list) and produce PRCPrototypeDB, which is the actual prototype database that holds the data passed to the recognition functions.

  Its internal structure depends on the recognition algorithm being used, but all currently implemented recognition algorithms use a common data structure. The following items are added to the initial data.

  - Indices to the starting point of each stroke
  - Length of each line segment
  - Length of each stroke
  - Total length of the pattern
  - Ratio of line segment to stroke length for each line segment
  - Ratio of stroke to pattern length for each stroke
  - Angle of each line segment
  - Bounding box for each stroke
  - Bounding box for the entire pattern

- **PRCInputPattern**

  The input coordinate data from the touch panel stored in PRCStrokes must also be preprocessed before it is passed to the recognition functions. Touch panel input is often sampled once per frame, which results in too many points for the recognition algorithm to use. You must resample the input pattern to extract the points that best define its features. To create the PRCInputPattern structure, the PRC_InitInputPattern function resamples the raw input stroke data and performs additional calculations similar to those done by PRC_InitPrototypeDB.

2.2 Examples Of Library Use

The following pseudocode excerpts are examples of library use.

Code 2-6 Examples of Library Use

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

#define RAW_POINT_MAX 1024 // How many raw input points to save
#define POINT_MAX 40 // Maximum number of points to accept after resampling
#define STROKE_MAX 4 // Maximum number of input strokes to accept
```

You cannot call `nitro.h` from the `PRC*` header file. To use the pattern recognition library, you must explicitly place `nitro/prc.h` in an include statement. Here, instead of placing `nitro/prc.h` in an include statement, we can select the default pattern recognition algorithm by specifying `nitro/prc/algo_*.h`. For more information, see section 3.2 Recognition Algorithms.

To use the pattern recognition library, a number of parameters must be defined as macro constants. The value specified by `RAW_POINT_MAX` is the maximum number of input points that can be accepted by the touch panel. Because the pattern recognition library processes the array of an entire series of points as a single target, the application must be able to store all of the input points. If the touch panel accepts 60 points every second and a single character requires at most 10 seconds to input, the application will need to store an array of 600 points.

During preprocessing, the raw input data handed off by the application is stripped down to its characteristic points. This is called resampling or characteristic point extraction. `POINT_MAX` and `STROKE_MAX` define the maximum number of points and strokes permitted after preprocessing. If `POINT_MAX` is set to a value that is too low, a long and complex set of input data for a single character can be truncated in the middle. The proper setting for this constant will depend on the complexity of the input pattern you require and on the number of points you want to preserve after preprocessing (`PRC_InitInputPattern*`).

Code 2-7 PRC_InitInputPattern

```
extern PRCPrototypeList PrototypeList;

// Allocates a work region for extracting the prototype database
PRCPrototypeDB protoDB;
void* dictWork;
dictWork =
    OS_Alloc(PRC_GetPrototypeDBBufferSize(&PrototypeList));
PRC_InitPrototypeDB(&protoDB, dictWork, &PrototypeList);
```

Think of `PrototypeList` as the prototype list data defined in a separate file.

Use `PRC_InitPrototypeDB` to create `PRCPrototypeDB` (the prototype database) from `PRCPrototypeList` (the prototype list). You must allocate sufficient memory for `PRCPrototypeDB` based on the size of the prototype database. Allocate a region of memory based on the size obtained by `PRC_GetPrototypeDBBufferSize` and pass it during initialization.

`PRC_InitPrototypeDB` will count the total number of points and strokes in the prototype set, and then perform calculations that will speed up recognition processing. These calculations include creation of

an index of all strokes; the index determines the length and angle of each segment and other data required by the recognition algorithms, and the information is stored in PRCPrototypeDB.

PRC_InitPrototypeDB has a sibling function, PRC_InitPrototypeDBEx, that allows you to specify the prototypes to be used based on a bit field. When using PRC_InitPrototypeDBEx, be sure to calculate the work area size by providing PRC_GetPrototypeDBBufferSizeEx with the same arguments as used in PRC_InitPrototypeDBEx.

Code 2-8 Calculating Work Area Needed For Recognition Process

```
// Allocate a work area for other processing
void* inputWork;
inputWork =
   OS_Alloc(PRC_GetInputPatternBufferSize(POINT_MAX, STROKE_MAX));
void* recogWork;
recogWork = OS_Alloc(
   PRC_GetRecognitionBufferSize(POINT_MAX, STROKE_MAX, &protoDB)
);
```

Code 2-8 allocates the work area needed for the recognition process. To pool multiple input patterns in parallel, you need to allocate one work area for extracting input pattern and another for comparison processing. If at the outset you specify the largest values that you will need, you will not have to allocate new memory for each recognition process.

```
// Initialize the input stroke data
PRCPoint points[RAW_POINT_MAX];
PRCStrokes strokes;
PRC_InitStrokes(&strokes, points, RAW_POINT_MAX);
```

The code above initializes the structure that holds raw data input from the touch panel.

```
while ( 1 )
{
```

This loop is entered each frame.

Code 2-9 Adding Input from Touch Panel To PRCStrokes

```
int x, y;
if ( !PRC_IsFull(&strokes) )
{
      if ( there is (x,y) input from the touch panel )
       {
          // Append point (x,y) to the stroke
          PRC_AppendPoint(&strokes, x, y);
        }
        else if (  there was input in the previous frame )
        {
```

```
            // Insert a “pen up” marker
            PRC_AppendPenUpMarker(&strokes);
        }
    }
```

Code 2-9 adds input from the touch panel to `PRCStrokes` structure. If the pen is lifted from the touch panel, you must call `PRC_AppendPenUpMarker` once (but no more than once) to append a “pen up” marker.

Code 2-10 Setting Parameters For Converting Raw Stroke Data To PRCInputPattern data

```
    if ( there is a request for recognition )
    {
       // Start recognition using the current contents of strokes
       // First, set the resampling process parameters
       PRCInputPatternParam inputParam;
       inputParam.normalizeSize = protoDB.normalizeSize;
       inputParam.resampleMethod = PRC_RESAMPLE_METHOD_RECURSIVE;
       inputParam.resampleThreshold = 3;
```

Here, we set the parameters required for the conversion of the raw stroke data to the `PRCInputPattern` type data used for the recognition process. If `normalizeSize` is set to a non-zero value, the bounding box of the input stroke will be normalized (expanded or contracted) to match the specified size. All of the recognition algorithms except Light assume that the prototype database and the input pattern are of the same size. For input size to match the prototype database size, be sure to use normalization.

`resampleMethod` and `resampleThreshold` are used to set both the algorithm and parameters used to extract the characteristic points from the raw input data. For more information about resampling algorithms, see section 3.1 Parameters for Resampling.

Code 2-11 Processing Raw Input Points and Creating the PRCInputPattern Type Input Pattern Data

```
    // Use resampling on the raw input points to reduce the number of datapoints.
    // Perform preprocessing to determine additional information, such as length,
       and create inputPattern
    PRCInputPattern inputPattern;
    PRC_InitInputPatternEx(&inputPattern, inputWork, &strokes,
        POINT_MAX, STROKE_MAX, &inputParam);
```

Using the previously allocated work area, process the raw input points and create a `PRCInputPattern` type input pattern data.

Based on the parameters from `PRCInputPatternParam`, `PRC_InitInputPattern` performs normalization and resampling to extract the characteristic points. It then calculates segment lengths and angles from these points and stores this information in the `PRCInputPattern` structure.

Code 2-12 Performing Recognition

```
    // Perform recognition by comparing inputPattern with entries in protoDB
    PRCPrototypeEntry* result;
    fx32 score;
    score = PRC_GetRecognizedEntry(&result, recogWork,
                                   &inputPattern, &protoDB);
```

This method completes preparation for the recognition process. Next, we need to compare the input pattern data (`PRCInputPattern`) with the prototype database (`PRCPrototypeDB`) and find the database entry that most closely matches the input pattern data. The level of similarity is a type `fx32` that ranges from 0 to 1. (If converted to an `int`, it would range from 0 to 4,096.)

Depending on the selected algorithm and the size of prototype database, the processing could take more than several tens of milliseconds. We thus recommend using a separate thread for this processing. For an implementation example, see the `prc/characterRecognition-1` demo.

The sibling function `PRC_GetRecognizedEntryEx` allows you to use a bit field to specify the types of patterns for recognition. `PRC_GetRecognizedEntries` returns the N entries that are best matches. For details, see the reference manual.

```
     // Output the result
     OS_Printf("code: %d\n", PRC_GetEntryCode(result));
```

As a recognition result, the function returns a pointer to `PRCPrototypeEntry` in `PRCPrototypeList`. You can use `PRC_GetEntryCode` and `PRC_GetEntryData` to obtain the code and user data of the entry.

```
   }
   Processes that wait for V-Sync
}
```

3 Settings Entries

3.1 Parameters for Resampling

You can choose the algorithms to use for the resampling process conducted by `PRC_InitInputPattern`.

3.1.1 PRC_RESAMPLE_METHOD_NONE

No resampling is performed. This method removes only the points that duplicate the immediately preceding coordinates; it can be used when it is necessary to reprocess stroke data that has already been resampled.

3.1.2 PRC_RESAMPLE_METHOD_DISTANCE

This method resamples based on the traveled distance. It captures the starting and ending points of each stroke and captures a point each time a stroke travels more than a predefined cumulative distance from the starting point. Measured distance is not Euclidean; it is the change in the *x* position plus the change in the *y* position, or the “city block” or “Manhattan” distance. This calculation of distance is less precise than that of Euclidean distance, but it is faster to process.

The `resampleThreshold` specifies the cumulative distance that the stroke has to travel before capturing the next point. This is the fastest method to process, but strokes drawn slowly with a shaky pen may cause the threshold to be reached quickly, resulting in too many points being captured. Also, this method tends not to capture the best characteristic points.

3.1.3 PRC_RESAMPLE_METHOD_ANGLE

This method performs sampling based on the curvature of each stroke.

First, the starting and ending points of each stroke are captured. Next, the angle of the segment connected to the starting point is stored. The connecting segments are followed in succession until the angle difference reaches the threshold angle. The point immediately before the point where threshold is exceeded is captured as the second point in the series. The angle of the segment that connects the two immediately preceding points is compared to the angle of the segment that connects the preceding point with current point. If the difference is greater than the threshold angle, the current point is captured. This process is repeated.

`FX_Atan2Idx`, which uses an internal table lookup, speeds up the process by calculating the angle. `FX_Atan2Idx` is not highly accurate, but it is sufficient for this purpose.

The `resampleThreshold` specifies the threshold angle. The values range from 0 to 65,535, with 1.0 representing 1/65,535th of a full circle.

Because valid angles cannot be measured at very short distances, every captured point must have a city-block distance from the previous point, and the distance must be greater than that specified by

PRC_RESAMPLE_ANGLE_LENGTH_THRESHOLD. Currently, PRC_RESAMPLE_ANGLE_LENGTH_THRESHOLD is fixed at 6. The distance calculations use the raw, non–normalized coordinates.

Even if you set the threshold high to reduce the number of resampled points, this method can still accurately capture points in small loops and extract good characteristic points. Conversely, if the threshold is set too low, slight stroke bends will be picked up. The calculation time is linear to the number of input points.

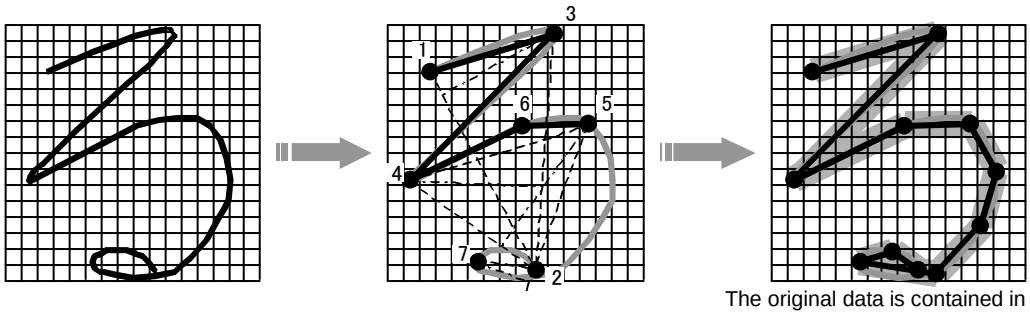
3.1.4 PRC_RESAMPLE_METHOD_RECURSIVE

This method processes input recursively and captures the most characteristic points.

First, the starting and ending points are captured and defined as points A and B. All points between A and B are tested, and the point farthest on a straight line drawn between A and B is defined as point C. If the distance is greater than resampleThreshold, C is captured. Otherwise, the line from A to B is retained. If C is captured, the process is reiterated for A and C and C and B.

Ultimately, this process will completely capture all the original raw input stroke data in the region bounded by the resampleThreshold distance on both sides of the resulting line segments. However, if the number of resampling points reaches the upper limit during the process, this method will not capture all points.

Figure 3-1 PRC_RESAMPLE_METHOD_RECURSIVE



The original data is contained in the region bounded by the resamplingThreshold on both sides of the resulting line segments.

If you set resampleThreshold to a value smaller than the smallest loop you expect in the input pattern, you should be able to generate a relatively compact set of resampling data without missing any loops. If your resampling results are compact, the recognition process will be faster.

The calculation time for the resampling process itself is, in the worst case, proportional to the result of multiplication between *[the number of input points]* and *[the number of resulting resampling points]*. With typical input data, such as hiragana characters, this method will take slightly longer than PRC_RESAMPLE_METHOD_ANGLE. This is based on the assumption that the parameters have been set to have both methods generate the same number of resampled points.

3.2 Recognition Algorithms

At this time, four pattern recognition algorithms have been implemented. The first header file placed in an include statement is selected for the recognition algorithm.

```
#include <nitro/prc/algo_light.h>        →      Light recognition algorithm
#include <nitro/prc/algo_standard.h>     →      Standard recognition algorithm
#include <nitro/prc/algo_fine.h>         →      Fine recognition algorithm
#include <nitro/prc/algo_superfine.h>    →      Superfine recognition algorithm
```

If you describe `#include <nitro/prc.h>`, all four of the above header files will be loaded. Because `algo_standard.h` is loaded first, Standard is the default recognition algorithm.

The following library functions and types vary depending on the recognition algorithm.

`PRCPrototypeDB`

`PRCInputPattern`

`PRCPrototypeDBParam`

`PRCInputPatternParam`

`PRCRecognizeParam`

`PRC_Init`

`PRC_GetPrototypeDBBufferSize*`

`PRC_InitPrototypeDB*`

`PRC_GetInputPatternBufferSize`

`PRC_InitInputPattern*`

`PRC_GetInputPatternStrokes`

`PRC_GetRecognitionBufferSize*`

`PRC_GetRecognizedEntry*`

Each recognition algorithm uses an identifier shown above, with the algorithm's name appended to it as a suffix. The above identifiers will be treated as aliases of those in the first loaded header file. To use the recognition algorithms placed in an include statement after the first header, you must explicitly use types and function names with the suffix `_<algorithm name>`: for example, `PRCRecognizeParam_Light` or `PRC_InitPrototypeDBEx_Fine`.

However, of the types and functions in the current implementation, the following library functions are common to all recognition algorithms.

`PRCPrototypeDB`

`PRCInputPattern`

`PRCPrototypeDBParam`

`PRCInputPatternParam`

`PRC_Init`

`PRC_GetPrototypeDBBufferSize*`

`PRC_InitPrototypeDB*`

`PRC_GetInputPatternBufferSize`

`PRC_InitInputPattern*`

`PRC_GetInputPatternStrokes`

These functions and types use the `_Common` suffix, which is referenced by all algorithms: for example, `PRCPrototypeDB_Common`.

Apart from the algorithms related to `PRC_GetRecognizedEntry*` (which performs actual recognition), all other algorithms currently use the same libraries. The *prc/characterRecognition-2* demo exploits this feature to access a shared prototype database and shared input pattern data by using all recognition algorithms simultaneously. For an example of using multiple recognition algorithms simultaneously, refer to the demo.

An overview of each algorithm is presented in the following sections. In this discussion, we often use vague terms because the accuracy and calculation time for each algorithm depend considerably on a number of factors. Statements about processing speed are for reference purposes only. Select your recognition algorithm and set your parameters only after thorough testing with the data used in your application.

3.2.1 Light Algorithm

The Light algorithm is the most lightweight recognition algorithm. It is ideal for situations where patterns in the prototype database are distinct (making recognition errors unlikely) or when you want to recognize patterns that consist of only a single stroke.

This algorithm compares only angles. Strokes of the input pattern and prototype are expanded or contracted, so that the total length of each is 1. The integral of the difference in angles is then taken, and the degree of similarity is computed and returned. The values are adjusted such that a similarity of 0.0 is returned if all angles differ by 180°, and a similarity of 1.0 is returned if all angles match perfectly.

Figure 3-2 Angle Difference

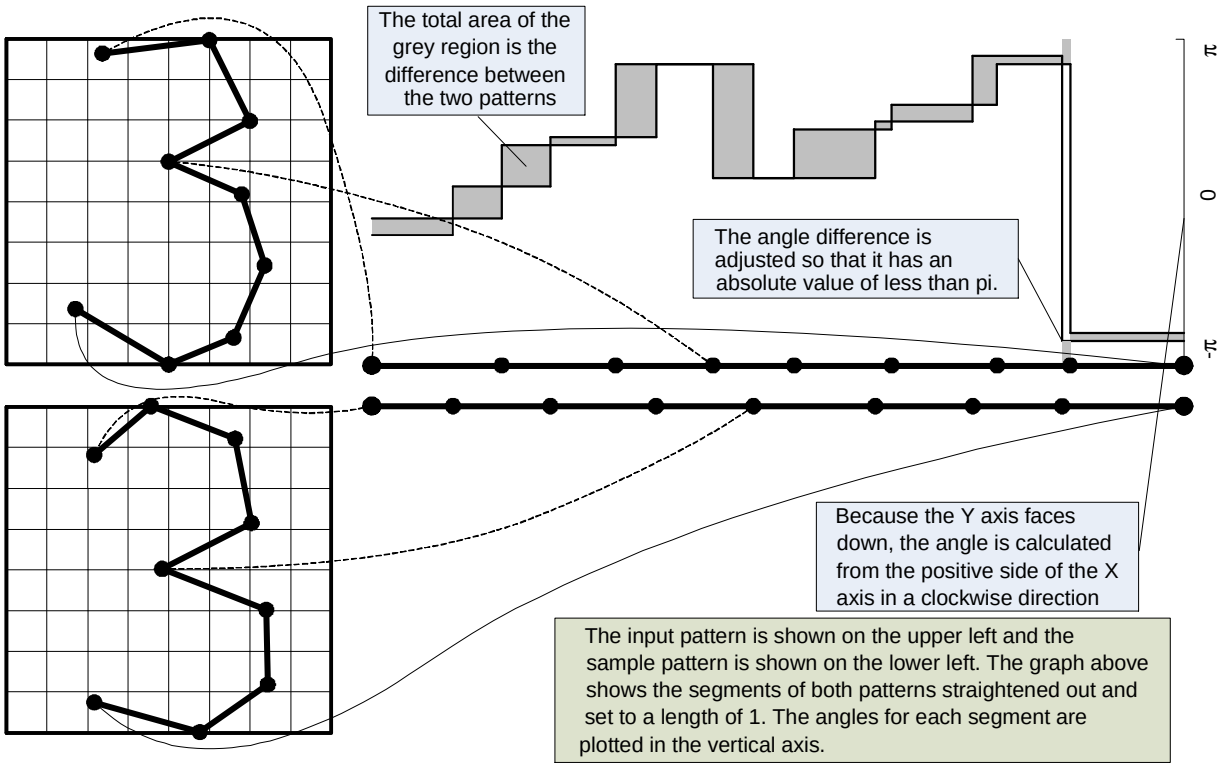


Figure 3-2 shows the angle difference in a graphic form.

When comparing patterns with multiple strokes, the Light algorithm performs the same calculations on each stroke and then computes the weighted average of all similarity scores, weighting each stroke in the prototype based on its length relative to the entire pattern. However, this algorithm does not examine the relative positions of each stroke; it thus has the inherent drawback of not being able to distinguish "T" from "+". This algorithm was designed mainly to recognize single-stroke patterns at the fastest speed possible.

The calculation time will be proportional to the result of multiplication between *[the number of points in the input pattern]* and *[the number of entries in the prototype database]*.

3.2.2 Standard Algorithm

This algorithm was designed as a standard recognition algorithm. It is ideal for situations that require the player to correctly enter patterns like magic symbol.

This algorithm compares both angles and positions. Like the Light algorithm, it adjusts the length of the input pattern and the prototype so that they are both 1, and takes the integral value of the angle differences multiplied by the position differences. Instead of taking the difference between exact points, distances are measured with the “city block” method and approximated to the closest sampling point coordinates. Like the Light algorithm, the Standard algorithm adjusts the similarity values such that 0.0 indicates lack of similarity and 1.0 indicates perfect match, and returns the result as a score.

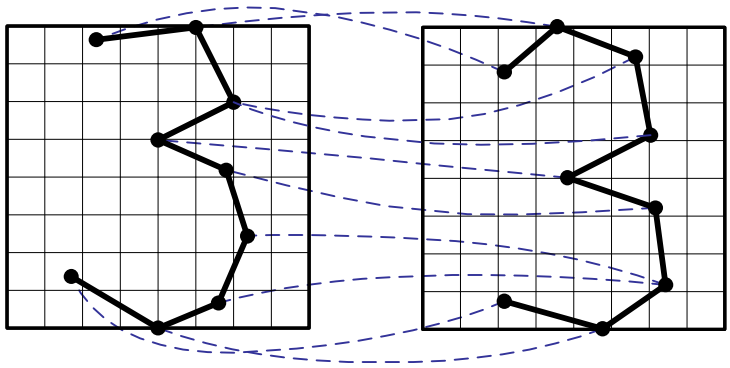
Because the Standard algorithm considers positioning, it can easily recognize patterns with multiple strokes. When determining the similarity score, after performing the above calculations on each stroke this algorithm computes the weighted average of similarity scores, giving each stroke in both the database entry and input pattern a weight proportional to its length relative to the entire pattern.

The calculation time for the Standard algorithm is two or three times longer than that required for the Light algorithm. However, even if you set up a recognition thread that runs when the main thread is idle, the result should come back in an acceptable period of time.

3.2.3 Fine Algorithm

This algorithm was designed to handle even distorted characters. It is useful when the application needs to salvage such character input data as the distorted input from a user. In addition to comparing both angles and positions, this algorithm performs elastic matching. Rather than matching input and prototypes by changing their size, it expands and contracts individual strokes, and looks for the matches that result in a high evaluation score.

Figure 3-3 Elastic Matching



An example of elastic matching is shown in Figure 3-3. Vertices in the input pattern (left side) are compared with those in the prototype database entry (right side). You can see that sometimes the vertices are mapped to a single vertex more than once. The Fine algorithm searches for combinations that produce the highest score, while permitting more than one point to be mapped to a single point. This allows the algorithm to easily handle distortions like those shown in the figure and to generate a high score for a “3” drawn such that the upper and lower sections are not of the same size as the prototype. Elastic matching is good at correctly interpreting distorted input.

To compute the score, the following formula is used on each matching vertex.

(normalized size × 2 – city block distance) × (π – difference between angles of the segments entering the vertex)

The average of the vertices is then taken, and the result is distributed over the range of 0.0 to 1.0. The vertices are matched to find the vertex that generates the highest score.

Elastic matching is performed using an algorithm based on Dynamic Programming (DP) matching. It does not implement a beam search. Accordingly, the calculation time is proportional to the result of

multiplication between *[the number of points in the input pattern]* and *[the number of points in the prototype]*. In a typical application, the Fine algorithm usually takes several times longer than the Standard algorithm.

3.2.4 Superfine Algorithm

The Superfine algorithm is the recognition algorithm that requires the longest processing time among the currently implemented algorithms. However, it is not always more accurate than the Fine algorithm. Use the Superfine algorithm when you find that the Fine algorithm is not accurate enough.

Like the Fine algorithm, Superfine algorithm uses elastic matching. The Fine algorithm takes the evaluation values used by elastic matching and returns the values as the score; Superfine algorithm uses elastic matching to obtain information about the points that should map. Elastic matching determines the most likely vertex matches; vertices without a certain match are matched with hypothetical points on the other pattern based on lengths of the segments before and after the vertex in question. The Superfine algorithm then computes a final score in the same manner as the Fine algorithm.

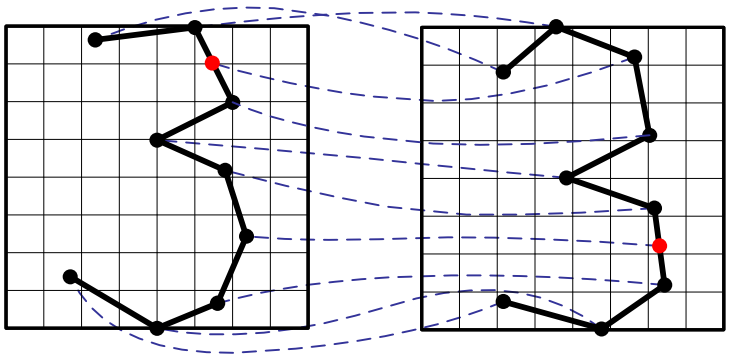
Unlike the Fine algorithm, Superfine uses the following formula for each point to compute the final score.

(normalized size × 2 – city block distance) × (cos of the difference between the angles of segments entering the vertex)

The Superfine algorithm then computes a weighted average of all the points based on the lengths of segments connected to each point relative to the entire pattern.

When finding vertex pairs using DP matching, the Superfine algorithm does not treat segment lengths the same way as the Fine algorithm. The angle score is computed using a cosine function.

Figure 3-4 Computing Angle Score Using a Cosine Function



In the graph above, the hypothetical points that result from interpolation are in red. In addition to performing the operations used by the Fine algorithm, the Superfine algorithm must perform frequent division to generate the interpolated points. The calculation time required by the Superfine algorithm to generate the interpolated points is often several times longer than that required by the Fine algorithm.

4 Tips and Tricks

4.1 Parameter Settings

The easiest way to learn about parameter adjustments is to change parameters in the *prc/characterRecognition-2* demo and watch the effect on memory use, calculation time, and accuracy. Because performance will depend considerably on the nature of the prototype database, you need to make your parameter adjustments using the data that is as close as possible to the prototype database that you will use in actual application. For instructions on using the *characterRecognition-2* demo, see section A.2 characterRecognition-2.

If certain patterns are recognized too frequently, you can prevent this by adjusting their correction values in the database. However, you can easily end up making a large number of unnecessary minor adjustments. You can use the same code value for several database entries. If you have patterns that are not being recognized, it might be easier to add new prototypes to the database until you start getting matches for those patterns.

4.2 FAQ

Q. `kindMask` can be specified with both `PRC_InitPrototypeDB*` and `PRC_GetRecognizedEntry*`. Which should be used to select a certain type of pattern?

A. This depends on how often you want to change your selection criteria. Specifying with `PRC_InitPrototypeDB` will reduce the memory required for extracting the prototype database, but you will not be able to easily change the set of patterns you want to target.

Q. I want a lightweight algorithm that will recognize patterns with multiple strokes. Can the Light algorithm be used for this purpose? I don't need a high level of recognition accuracy, but the inability to distinguish "p" from "b" is going to be a problem.

A. Light algorithm can be used to recognize patterns that have multiple strokes. This is accomplished by not using `PenUpMarker`. Normally, when the pen is lifted, a `PRC_AppendPenUpMarker` is used to show that the stroke was completed, but if you omit this operation, the pattern recognition library will treat a series of strokes as a single connected stroke. By populating your prototype database with patterns that have a single unbroken stroke, the Light algorithm will be able to perform recognition that reflects the positional relationships of multiple strokes.

This technique is also useful for handling joined characters and lines that fade out partway. Nonetheless, the possibility of unintentional matches will naturally increase. To avoid this, select your patterns accordingly.

Q. Is it possible to use the resampling results for processing outside of the game?

A. For `PRCInputPattern`, use `PRC_GetInputPatternStrokes`. This creates a pointer that points directly to the data contained in `PRCInputPattern`, so there is no need to initialize the first

parameter with `PRC_InitStrokes`. If you want to change the contents, you can copy the structure with `PRC_CopyStrokes` before using the contents.

If you only want to perform resampling, you can use `PRC_ResampleStrokes*`. The results of this function will be returned as an index array. Use the application to convert the results to the `PRCStrokes` type.

Appendix A Demos

The pattern recognition library demos are stored in the `$TWLSDK_ROOT/build/demos/prc` directory of the TWL SDK.

A.1 characterRecognition-1

Several problems can occur when the pattern recognition library is used. Calculation time can sometimes exceed a single frame and can depend considerably on the complexity of the input pattern. Therefore, set up a pattern recognition thread that is separate from the main thread. Ideally, the application should perform pattern recognition during the idle period after main thread processing is finished and before the V-Blank interrupt is generated. The `characterRecognition-1` demo is an example of an application that uses a separate thread.

Perform recognition with the A Button and clear the screen with the B Button.

In the demo, the prototype database has 97 entries that can be used for testing. This prototype database contains Arabic numerals, lowercase alphabets, and some symbols. Because there are multiple patterns for each numeral, the total number of characters that can be recognized is 50. This prototype database is used only for demonstration purposes. For your application, you should build a new prototype database using sampling points and standard patterns that meet your requirements for speed and accuracy.

A.2 characterRecognition-2

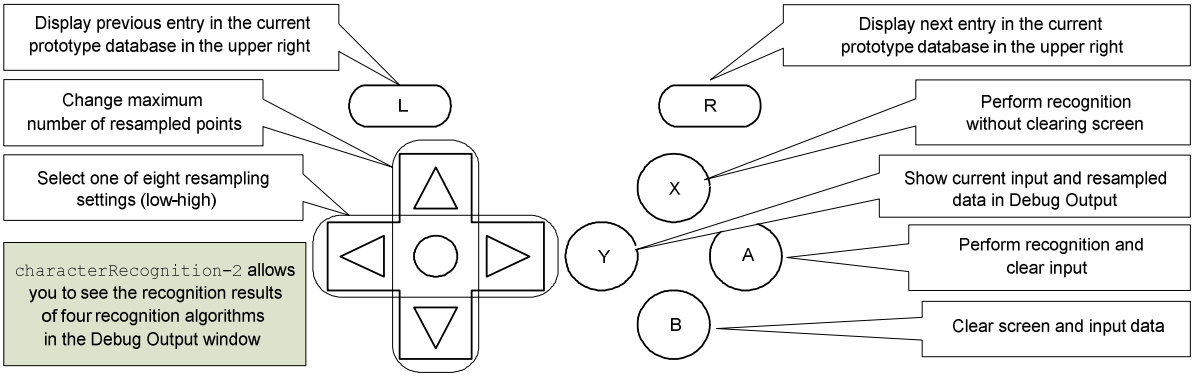
This demo application is designed to compare various pattern recognition algorithms. It allows you to use a prototype database on the production unit to see the effect of changing `maxPointCount` (the largest number of sampling points accepted) on the size of work area; it also helps you understand how the adjustment of resampling parameters can affect recognition time and results.

There are eight threshold combinations (from low to high) that have been tuned to generate a similar number of sampling points using the three sampling algorithms. These settings can be changed during runtime.

This demo employs the prototype database used in the `characterRecognition-1` demo. By repopulating the database with actual application data, you can use this demo application to help determine optimal parameter settings.

Start the application and draw a pattern on the touch panel. When you press the A Button, four patterns will appear on the screen. The three patterns on the left, `PRC_RESAMPLE_METHOD_DISTANCE`, `ANGLE`, and `RECURSIVE`, are the sampling results. The rightmost pattern is the recognition result prototype data. The Debug Output window displays detailed recognition results for each algorithm.

Figure A-1 characterRecognition-2 Demo



This demo can also be used as a basic pattern creation tool.

Set the sampling parameters using the +Control Pad (Left/Right) and draw a pattern with the pen. Press the Y Button; the resampling result pattern data for each of the three resampling algorithms will appear as text in the Debug Output window. One line at a time, you can cut and paste the text data for various patterns into a text file; to obtain a C source code listing for the prototype list that can be read by the pattern recognition library, run the following demo sample:

```
$ perl $TWLSDK_ROOT/tools/bin/pdic2c.pl <normalized size for output>
<prototype database  text data>
```

To check operation of the pattern recognition library, use the source code. For more information on the input format used by `pdic2c.pl`, see the reference manual.