Achieving Interoperability of Pen Computing with Heterogeneous Devices and Digital Ink Formats

by

Xiaojie Wu

in Computer Science

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science

Faculty of Graduate Studies
The University of Western Ontario
London, Ontario
December 2004

THE UNIVERSITY OF WESTERN ONTARIO
FACULTY OF GRADUATE STUDIES

CERTIFICATE OF EXAMINATION

Supervisor

_____

Supervisory Committee

_____

_____

Examining Board

_____

_____

_____

The thesis by

Xiaojie Wu

entitled:

Achieving Interoperability of Pen Computing with Heterogeneous Devices and Digital Ink Formats

is accepted in partial fulfillment of the
requirements for the degree of
Master of Science

Date_____

_____
Chair of the Thesis Examination Board

Abstract

Pen-based computing has and continues to become accepted and increasingly used. Hardware and software vendors have typically stored and represented digital ink using proprietary or restrictive ink formats, and have provided software development toolkits to access or manipulate ink for user development on their devices. The variety of digital ink formats and device-dependent software toolkits has limited the ink exchange and application among heterogeneous devices. Our objective is to explore the interoperability of pen computing among heterogeneous devices and digital ink formats.

Our investigation has two aspects: ***digital ink formats*** and ***pen computing application programming interfaces*** (APIs). We consider three ink formats: UNIPEN, Jot and InkML, and two ink APIs: the IBM CrossPad API and the Microsoft Tablet PC API. Our objectives are twofold: (1) to accomplish conversions among UNIPEN, Jot and InkML; (2) and to develop a common abstract API for the CrossPad and the Tablet PC. In this thesis, the issues in conversion among three ink formats are discussed, and the conversion between UNIPEN and Jot is implemented. We also identify the incompatibilities between the CrossPad API and the Tablet PC API. The design of an abstract API is described, and a partial implementation is complete.

Keywords:
pen-based computing, digital ink, UNIPEN, Jot, InkML, CrossPad, Tablet PC

Acknowledgement

First and foremost, I would like to extend my sincerest gratitude to my supervisor, Dr. Stephen Watt, for his consistent guidance, encouragement and support during my graduate studies, and for his time and patience.

Many thanks to Mr. Kevin Durdle for his precious advice and help during the design of the abstraction API, and his sharing of the experience in Microsoft Tablet PC. Thanks to Mr. Igor Rodianov and Mr. Laurentiu Dragan for their valuable advice and configuring the development tools and operating systems that I have used for my thesis.

Thanks to Ms. Bethany Heinrichs for her help during my stay at the ORCCA lab. My thanks also go to Yuzheng Xie, Xiaofang Xie, Ben Huang and other ORCCA members and faculty for their friendship, encouragement and help.

Table of Contents

List of Tables

List of Figures

CHAPTER 1 INTRODUCTION

1.1 The "What" and "Why" of Pen Computing

Pen-based computing as a field broadly includes computers and applications in which a pen is the main input device. A special pen, often called a stylus, is often used to write on a digital tablet. The digitizer underneath captures the (x, y) coordinate data of the pen tip movement. With some handwriting recognition engines, the handwriting can be translated into text, commands, or otherwise just be left as digital ink.

Pen-based computing has held promise for decades, since the first pen-based computing device for handwriting was invented in late 1950s. It continues to draw a lot of research attention today. The interest in pen computing stems from a number of factors: First of all, for those small-size computing devices such as PDAs, Palm and some new exploratory applications for cell phones or pagers, a keyboard and mouse are obviously too big to accommodate. The pen is an alternative input mechanism. Second, for entering letters in ideographic languages like Chinese, or entering drawings, mathematical formulas, musical notation and other free form inputs, keyboard/mouse input appears cumbersome or infeasible. Third, there are some situations where pen-based input is superior to the usual style of input. For example, it is easier for a supervisor to amend his student's electronic articles on a pen-based computer. Finally, pen-computing is essential in some modern technologies such as the interactive whiteboard used in conference meeting or distance education.

1.2 Existing Technology in Pen Computing

The earliest technology in pen computing can be traced back to 1957, when T. L. Dimond presented a device call "Stylator" which could read handwritten characters. After that, more tablet and stylus devices were developed, but the success was limited because of their poor handwriting recognition capability and limited processing power. The Apple Newton was one of the unsuccessful examples. It was one of the earliest PDA products, sold from 1993 but discontinued in 1998. Besides its high price and large size (would not

fit in a pocket), the public criticism to its handwriting recognition caused its failure in the marketplace. It wasn't until 1996 when Palm Inc. launched its personal digital assistant (PDA) that mainstream acceptance from the public took hold.

The market was never so promising and competitive as today. We have seen the increased use of pen-based devices and computers. Companies such as Microsoft, IBM, Fujitsu, HP, Toshiba, ViewSonic and Wacom have developed and released various products. The most notable and widely used are the Palm Pilot, Pocket PC and Tablet PC. There are other products such as the Wacom Graphire2, Intuos2 and Cintiq, which are very professional pen tablets for image processing.

With the development of this new generation of hardware, digitizers can provide more powerful ink capture capabilities. In addition to recording (x, y) coordinates, these devices can sense the pen pressure on the digitizer, the angle of the pen, and so on. For example the Wacom Intuos2 has 1024 levels of pressure sensitivity and supports pen tilt. This gives a complete natural feel and very good control.

On-line handwriting recognition technology plays an important role in pen computing. The type of built-in handwriting recognition software shipped with devices has achieved a more satisfactory and acceptable feedback from public, but is still limited. Researchers are trying to improve the accuracy and speed of handwriting recognition, and to broaden the text recognition to other specific fields such as mathematical handwriting.

Numerous standards and specifications for representing digital ink have existed since the early 1990's. Most notably, ITU T-150, UNIPEN and Jot are targeted directly at the representation of digitized handwriting. Currently, a new specification, InkML, a XML data format for representing digital ink, is being developed under the World Wide Web Consortium (W3C).

1.3 Thesis Objectives

In view of the diversity of pen-computing devices developed by multiple vendors, the objective of this thesis is to achieve interoperability of digital ink among heterogeneous devices and platforms. This interoperability study is accomplished in two ways: to investigate conversions among various digital ink data formats so that ink can be shared in different applications; and to investigate unifying various ink APIs to achieve a device-independent API.

As mentioned above, there have existed numerous ink formats for some time. Since different standards have different design foci, ink has been defined in different ways. Some ink properties important in one format might be totally ignored in another. Some standards are public, while others are proprietary. Different hardware/software vendors store and represent digital ink using different data formats. This has severely limited ink sharing between applications with different data formats. UNIPEN, Jot and InkML are three well-known data formats. In this thesis we develop conversions among these formats to study the sharing of ink between applications.

It is often desirable to some users that they are provided some means to be able to port applications that have been developed on one device to a different type of device. However, in the current pen-computing world the application portability among different devices has not been well explored. Different device vendors provide their own software development toolkits (SDKs) for user's development. Some devices only work on a specific platform. For example, the Tablet PC and its SDK only work on the Microsoft .NET framework. This has limited ink applications' portability among different devices. We need a unifying API that is device-independent enough to make applications portable among devices. The current project of mathematical handwriting recognition in our lab also motivates a unified API for Palm, Pocket PC, CrossPad and Tablet PC so as be able to apply the mathematical handwriting recognizer to multiple devices. The second study of this thesis is part of this unifying project, exploring an abstract API unifying the CrossPad and Tablet PC.

In summary, the thesis objectives are twofold: first to develop converters among three digital ink data formats: UNIPEN, Jot and InkML, second to develop a device-independent abstraction API for the IBM CrossPad API and Microsoft Tablet PC API.

1.4 Organization of the Thesis

In this chapter, we have given an introduction to pen computing and the objectives of this thesis. The following is a brief overview of the remaining chapters.

Chapter 2 presents in some details of three notable standards for representing digital ink: UNIPEN, Jot and InkML. UNPEN and Jot have been widely used for years, while InkML is still being developed under the W3C Multimodal Activity Working Group. For each standard, its design goals and features are described.

Chapter 3 presents two popular digital ink Software Development Kits (SDK). These are the IBM CrossPad SDK and the Microsoft Tablet PC Platform SDK. The API of each SDK are described.

Chapter 4 and Chapter 5 concern digital ink data format conversions. In chapter 4, we discuss the issues related to format conversions. Chapter 5 describes the implementation of the conversions. Due to the incompleteness of the current InkML specification, only the conversion between UNIPEN and Jot is presented in this thesis.

Chapters 6 and 7 concern an abstraction API that covers both the CrossPad and the Tablet PC. Chapter 6 identifies the incompatibilities between the two APIs. Chapter 7 is the design of the new abstraction API that unifies the CrossPad API and the Tablet PC API.

Finally, chapter 8 presents our conclusions.

CHAPTER 2 REVIEW OF DIGITAL INK DATA FORMATS

This Chapter describes three digital ink data formats: UNIPEN, Jot and InkML. Please note that the specifications of the three formats presented here are based on the current publicly available information. The UNIPEN presented below is version 1.0, from 1994 [1]. Jot presented below is version 1.0, from 1993 [3]. Due to the demise of Slate Corporation, the founder of the Jot format, and the proprietary nature of this format, it is not guaranteed to be the most up-to-date version. Since InkML is still developing under the supervision of W3C, no complete W3C Recommendation for InkML currently exists. The InkML specification described here is based on currently available documents, including the third W3C Working Draft of InkML published on 28 September 2004 [4].

2.1 UNIPEN

This section is based on the UNIPEN 1.0 Format Definition [1] and the information from UNIPEN project website [10], giving an overview of the UNIPEN format. UNIPEN is a common data format to facilitate digital ink data exchange, primarily used by the technical and scientific community to store handwriting samples. It was designed in 1993, and over 40 institutions participated the work. The UNIPEN format incorporated the features of several institutions' internal ink data formats, including IBM, Apple, Microsoft, Slate (Jot), HP, AT&T, NICI, GO and CIC [10].

2.1.1 Motivation

UNIPEN was motivated by the need to store handwriting samples for on-line handwriting recognition research and development. In the early 1990's, pen computers and pen communication drew a lot of interest from the public, but handwriting recognition was still disappointing. Companies and universities working in this field collected their own handwriting databases for training and testing recognizers, but the data was not publicly available. To remedy this problem, and to encourage researchers to find better recognition techniques, the UNIPEN project was started, to make a large

corpus of on-line handwriting samples publicly available, and the UNIPEN format was then agreed upon.

2.1.2 Format Definition

UNIPEN is an extensible ASCII format. It is self-defined from 3 basic keywords: .COMMENT, .RESERVE and .KEYWORD. All keywords start with a dot. The UNIPEN definition can be divided into three parts: part A defines data types using the keyword .RESERVE; part B defines a number of new keywords using the keyword .KEYWORD; in part C, reserved strings are defined using the keyword .RESERVE. Below are pieces of a sample UNIPEN 1.0 format definition [1] :

.COMMENT	A – DATA TYPES
.RESERVE [N]	Integer or decimal number represented by digits separated by a dot; may start with a sign; no commas allowed.
.RESERVE [S]	String: any combination of keyboard ASCII symbols, except space, new-line, tabulations and words starting by a dot in the first column.

.COMMENT	B – KEYWORDS
.KEYWORD .KEYWORD [S] [R] [.] [F]	Define a new keyword: Keyword, argument types, documentation.
.KEYWORD .RESERVE [S] [F]	Define a new reserved string: reserved string, documentation.
.KEYWORD .PEN_DOWN [N] [.]	Pen down component: repeated sequences of coordinates as defined by .COOR, pen touching the pad surface
.KEYWORD .PEN_UP [N] [.]	Pen up component: same as .PEN_DOWN, but with the pen not touching the pad surface.

.COMMENT	C – RESERVED STRING GLOSSARY
.RESERVE T	Time in MILLISECONDS.
.RESERVE P	Pressure in units of P given by .UNITS_PER_GRAM.

2.1.3 The UNIPEN File

A data file in UNIPEN format consists of successions of instructions, each consisting of a keyword followed by arguments. The UNIPEN file is essentially a sequence of pen coordinates, annotated with various information about recording conditions, device information, writers, segmentation, data layout, labeling and so on.

The pen trajectories, the major part of the data file, are encoded as a sequence of components .PEN_DOWN and .PEN_UP, containing pen coordinates X, Y and other optional signals such as timestamp (T), pen pressure (P), rotational angle of the stylus (RHO), and so on. What signals are recorded depends on the arguments of .COORD specified. For example, if an ink stroke is recorded as a sequence of (X,Y) points indexed in time and pen pressure of each pen point, then the ink data in UNIPEN format defines: .COORD X Y P. Each line between the .PEN_DOWN and .PEN_UP pair (see following example data) represents a pen point, where the first two numbers record the X and Y coordinates of each point accordingly, and the third number records the pen pressure placed on the surface on that point. Recorded signals such as timestamp, pen pressure and angle of the pen provide the handwriting features and are important to the handwriting recognition research.

```
.PEN_DOWN
   5194     2821     5
   5195     2821     7
   5196     2822     11
   5197     2821     15
   5198     2820     19
   5198     2820     21
.PEN_UP
```

In a typical UNIPEN file, the keyword .VERSION specifies the version number of the format, .DATA_ID specifies the name of the database. The recording conditions are described by keyword .SETUP. The device information is described by the keyword .PAD. Segmentation and labeling are provided by the .SEGMENT instruction. Component numbers are used by .SEGMENT to delineate sentences, words, characters if that information is available. Data layout is specified by .X_DIM, .Y_DIM and .H_LINE, etc. Many more keywords and instructions may be used to record other data information. The format also provides a unified way to encode recognizer outputs to be used for benchmark purposes. A typical UNIPEN file from UNIPEN working group data collection [10] is given in Appendix 1.

2.1.4 Software Tools

Uptools3 is the latest version of software tools for viewing, editing and transforming UNIPEN files. It comprises a set of programs. Each program is described in Uptools3 introduction page [10] as following: *upview* is a X-Windows based program for visualizing UNIPEN files; *upread* is a program for transforming or extracting data from UNIPEN files; *uni2animgif* and *unipen2eps* transform data from UNIPEN files into animated gifs and encapsulated postscript respectively; *upworks* is a program using Tcl/Tk and X-Windows for browsing UNIPEN files and editing them. An example of the visualization of the UNIPEN file by upview program is appended in Appendix 1.

2.2 JOT

This section follows the presentation of the JOT specification [3] and gives an overview of the Jot format. Jot defines a common data format for the storage and interchange of electronic ink between software applications [3]. It was designed in 1992, by the efforts of Slate, Apple, General Magic, GO, Lotus and Microsoft. Unlike UNIPEN, whose design goal is to provide a standard format of digital ink samples for handwriting recognition research, the goal of Jot is to provide a simple and convenient format for digital ink exchange. It is intended to maintain complete likeness with the original ink as it was drawn.

2.2.1 Motivation and Goal

In the early 1990's there was no standard format for storing or representing electronic ink. This severely limited the capture, transmission, processing and presentation of digital ink by users and applications. Jot was therefore motivated by the need to share ink-based information. The goal of Jot was to provide application programs on the various platforms and operating systems a way to store and exchange ink data. As described in JOT specification [3], applications of Jot include: Sharing signatures and annotations between mobile, pen-based computers and a central database; sharing electronic mail

between handheld devices and desktop systems; taking and sharing notes throughout an organization.

2.2.2 Jot Format Overview

Jot is binary format and light-weight. It includes lossless compression with a "reserved encodings" (see 2.2.4) scheme to reduce the space for ink storage. It also has the ability to optionally reduce the amount of information retained for a particular piece of ink.

To maintain the ink fidelity, Jot supports a wide variety of ink properties, including multiple strokes of ink combined into single objects, bounds, scale, offset, color with opacity, pen tips, timing information, height of the pen over the digitizer, stylus tip force, buttons on the stylus and X and Y angle of the stylus. Applications can choose to recognize or ignore properties as required. In addition to the above the specified properties, new features can be added.

2.2.3 Format Definition

Jot is a record-based binary format. Ink information is stored in predefined structures. For example, the structure INK_POINT is defined to store data for one pen point, including the (x, y) coordinate and other attributes such as pen pressure and pen angle if available. The term "ink bundle" is used in the JOT specification to represent a piece of ink. Each ink bundle must begin with an INK_BUNDLE_RECORD structure and end with an INK_END_RECORD structure. Following is an example of ink bundle representation given in the JOT specification [3]:

```
INK_BUNDLE_RECORD       required    // for bundle number one
INK_SCALE_RECORD        optional    // sets the scale for rendering
INK_OFFSET_RECORD       optional    // sets the offset for rendering
INK_COLOR_RECORD        optional    // sets the color for rendering
INK_START_TIME_RECORD   optional    // sets the relative start time
INK_PENTIP_RECORD       optional    // sets the pen tip for rendering
INK_GROUP_RECORD        optional    // tags the following PENDATA
INK_PENDATA_RECORD      recommended // actual points
INK_GROUP_RECORD        optional    // tags the following PENDATA
```

```
INK_PENDATA_RECORD          recommended // actual points
INK_PENDATA_RECORD          recommended // more points in same group
INK_SCALE_RESET_RECORD      optional    // resets to default scaling/offset
INK_PENDATA_RECORD          recommended // actual points
INK_END_TIME_RECORD         optional    // relative time inking ended
INK_END_RECORD              required    // end of bundle number one
```

As we can see from the example, some records are required to record a stream of ink, while some records are recommended, and others are optional. The INK_BUNDLE_RECORD and the INK_END_RECORD are required. They indicate the beginning and end of the digital ink stream. In INK_BUNDLE_RECORD, all the features of ink stream are declared: whether the ink point value is compressed or not, whether pen angle data is present, whether ink force data is present, whether rotational data is present, and so on. The INK_PENDATA_RECORD is a key component in the format containing the actual pen data: x, y coordinate and other optional information such as force, angle, which varies in size depending on the flags set in the INK_BUNDLE_RECORD header. Other records listed above are optional, and they occupy space only when they are presented as required.

2.2.4 Encoding Schema

Ink data can be encoded in either *compacted* or *uncompacted* format in Jot. Both formats are delta-oriented formats. Each value is stored using a signed delta-value, which is added to the previous one. The first point in an INK_PENDATA_RECORD is relative to the defined default values for each component of the point. The difference between compacted and uncompacted format is that the delta value stored in the former is fixed length, while the delta value stored in the latter is variable length. Since the data is written most significant byte first in compacted format, the reading applications can determine how large the encoded delta is by reading the top 2 bits of the first byte (see following compacted format definition).

The “Reserved encodings” are applied in the compacted format. The reserved encodings are described as follows in JOT specification [3]: “Reserved encodings are

those encodings that, if real points, would fit into the next smaller delta size. The reserved encodings for 16 bit deltas are all 16 bit delta pairs where both X and Y are within the inclusive range MIN_S7 and MAX_S7. Similarly, the reserved encodings for 8 bit deltas are all 8 bit delta pairs where both X and Y are within the inclusive range MIN_S3 and MAX_S3."

Following is the compacted format definition described in JOT specification [3]:

32-bit absolute X/Y: Two 32 bit long words, first two bits are 00. Data is actually two S31s.

```
| 0 | 0 |        (30 low-order bits of X)              |
| X|       (sign bit of X plus 31 bits of Y)          |
```

16-bit short delta X/Y: Two 16 bit short words, first two bits are 0 1. Deltas are actually two S15s. Values that would fit into an 8-bit byte delta are reserved.

```
| 0 | 1 |     (14 low-order bits of delta-X)          |
| X|      (sign bit of X plus 15 bits of delta Y      |
```

8-bit byte delta X/Y: Two bytes, first two bits are 1 0. Deltas are actually two S7s. Values that would fit into a 4-bit nibble delta are reserved.

```
| 1 | 0 |      (6 low-order bits of delta-X)    |
| X|   (sign bit of X plus 7 bits of delta-Y) |
```

4-bit nibble delta X/Y: One byte, first two bits are 1 1. Deltas are actually two S3s.

```
| 1 | 1 | (S3 delta-X) | (S3 delta-Y) |
```

From the definition, we can see the data is encoded in the smallest power of 2 bytes that will fit. If the both delta X and delta Y are within the inclusive range MIN_S15 and MAX_S15 (–32768 ~ 32768), the data will be stored in two 16 bit short words with top two bits 0 and 1. If the both delta X and delta Y are within the inclusive range MIN_S7 and MAX_S7 (–128 ~ 128), the data will be stored in two bytes with top two bits 1 and 0. Similarly, if the both delta X and delta Y are within the inclusive range MIN_S3 and MAX_S3 (–8 ~ 8), then the data will be stored in 1 byte with top two bits 1 and 1.

For example, suppose we have an ink trace where the first two points are (1125, 8432) and (1148, 8475). Then delta X is 23 and delta Y is 43. The uncompacted Jot representation of the delta data is: 01000000 00010111 00000000 00101010; while the compacted Jot representation of the delta data is: 10010111 00101010. Clearly, the compacted encoding schema saves spaces, and is recommended to use.

2.3 InkML

InkML is an XML-based data format for representing, exchanging and storing digital ink. It is currently still being developed following the W3C process, and is expected to become an official W3C Recommendation. The work was first started in November 2000. IBM, Intel, Motorola, and the International UNIPEN Foundation have contributed to the proposal. This section follows the presentation of the third and latest InkML working draft [4] published on 28 September 2004, and give an overview of the InkXL format.

2.3.1 The purpose of InkML

Before InkML there already existed numerous standards for digital ink representation, storage and transmission. UNIPEN and Jot, presented above, are two of these. None of these standards, however, address all the concerns important for a digital ink standard. For example, UNIPEN is very focused on handwriting recognition requirements, with features to support labeling of ink data, but is not optimized for data storage or real time data transmission. Neither is it designed to handle ink manipulation applications involving colors, pen tip, image rotation, rescaling, etc. Jot is a proprietary format that avoids any abstract characterization of ink.

InkML is intended to unify various ink representations in a common modular format. It is to be a non-proprietary standard under the supervision of W3C. It is to provide the capability to capture, transmit, process and present ink across heterogeneous devices, and to be suitable for web-based applications. InkML can be used for various ink applications, some examples are: (1) real-time inking applications such as instant messaging, (2) off-line ink applications that capture and store ink for later processing, such as handwritten ink note archiving/retrieval, (3) interactive ink applications, such as using ink gestures to indicate actions.

2.3.2 InkML Elements

The InkML data format consists of two types of elements: primitive elements and application-specific elements.

2.3.2.1 Primitive Elements

The primitive elements form a set of rudimentary elements sufficient for all basic ink applications and have few semantics attached. All content of an InkML document is contained within a top-level <ink> element. The defined primitive elements include: trace and trace formatting elements, context elements and generic structure elements.

Trace and Trace Formatting Elements

A trace is the trajectory of the pen as the user writes digital ink. <*trace*> is the basic element used to record the actual trace data captured by the digitizer. It contains a sequence of points encoded according to the specification given by the <*traceFormat*> element. The simplest form of encoding specifies the X and Y coordinates of each sample point. For compactness, it may be desirable to specify absolute coordinates only for the first point in the trace and to use delta-x and delta-y values to encode subsequent points. Some devices record acceleration rather than absolute or relative position; some provide additional data that may be encoded in the trace, including Z coordinates or tip force. All these variations in the recorded information are supported through the <*traceFormat*> element.

<*traceFormat*> contains a <*regularChannels*> element listing those channels whose value must be recorded for each sample point (such as X, Y), and an <*intermittentChannels*> element listing those channels whose value may optionally be recorded for each sample point (such as F, pen tip force). Within a <*regularChannels*> or <*intermittentChannels*> element, channels are described using the element <*channel*> with *name*, *type*, *default* and *mapping* attributes. Following is an example of usage of <*traceFormat*>. The ink trace contains 10 points, it records (x,y) coordinates in regular channel and pen tip force in intermittent channel:

```
<traceFormat>
  <regularChannels>
     <channel name="X" type="decimal">
     <channel name="Y" type="decimal">
  </regularChannels>
  <intermittentChannels>
     <channel name="F" type="decimal">
  </intermittentChannels>
</traceFormat>

<trace id = "id001">
84 652:5’1’2:’2”2”-1:”2 4 1:4-1 21:0 13-9:-2-3-5:2-9 10:0 15 18:-
2-4-7:0;
</trace>
```

The trace is interpreted as following:

Trace	X	Y	F	vX	vY	vF	comments
84 652:5	84	652	5	?	?	?	
’1’2:’2	85	654	7	1	2	2	velocity values
”2”-1:”2	88	655	11	3	1	4	acceleration values
4 1:4	95	657	15	7	2	4	Implicit acceleration
-1 21:0	101	680	19	6	23	4	
13-9:-2	120	694	21	19	14	2	
-3-5:2	132	700	25	12	6	4	
-9 10:0	135	716	29	3	16	4	
15 18:-2	153	750	31	18	34	2	
-4-7:0	167	777	33	14	27	2	

Context Elements

A number of devices, data format and coordinate system details comprise the context in which ink is written and recorded. The <*captureDevice*>, <*brush*> and <*context*> elements address the contextual details. The <*captureDevice*> element describes the characteristics of devices, allowing specification of manufacture, model, sampling rate, sampling uniformity, latency and channel list. The <*brush*> element describes attributes of the brush used to capture the ink. The <*context*> element provides various attributes: *contextRef*, *canvas*, *canvasTransform*, *traceFormatRef*, *captureDeviceRef* and *brushRef*,

by which it both defines the shared context and serves as a convenient collection of contextual attributes.

Here is an example to define a device using the element <*captureDevice*>:

```
<captureDevice id="device1"
               manufacturer="IBM"
               model="Cross Pad"
               sampleRate="100"
               uniform="TRUE"
</captureDevice>
```

Here is an example using the element <*context*>. It defines a context using the predefined trace format "format1" and brush "brush1", and it shares the predefined canvas "canvas1":

```
<context id="context1"
         canvas="canvas1"
         traceFormatRef="format1"
         brushRef="brush1">
</context>
```

Generic Structure Elements

The most important elements of this category are <*mapping*>, <*bind*> and <*def*> elements. The <*mapping*> element is provided for various mappings in InkML. The mapping could be an identity mapping, a look-up table mapping or MathML mapping. A predefined mapping can be referenced and reused by other elements using a *mapRef* attribute. The following is a simple example of defining an identity mapping in a <*channel*> element:

```
<channel name="X" type="decimal" units="point" default="0">
   <mapping type="identity"/>
</channel>
```

The <*bind*> element is used to bind channels to entities. In the above identity-mapping example, if the source channel name is different from the channel being defined, then a <*bind*> element with a *source* attribute can be used to specify this as follows:

```
<channel name="X" type="decimal" units="point" default="0">
    <bind source="sourceDevX">
    <mapping  type="identity"/>
</channel>
```

The <*defs*> element provides a container for reusable content definitions that can be referenced by other elements via an *id* attribute. Three elements can be defined inside a <*defs*> block, they are <*context*>, <*brush*> and <*traceFormat*>. Here is a simple example to illustrate the usage of the <*defs*> element. Inside the context definition, it refers to the brush and trace format predefined in the <*defs*> block:

```
<defs>
   <brush id="greenPenRoundPoint"/>
   <brush id="yellowPenRectanglePoint"/>
   <traceFormat id="x-y"/>
   <traceFormat id="x-y-withAngle"/>
</defs>
<context id="context1"
         brushRef="greenPenRoundPoint"
         traceFormatRef="x-y-withAngle"
<context/>
```

2.3.2.2 Application-specific Elements

Application-specific elements provide a higher-level description of digital ink. They provide elements that support a specific category of applications, and can reference the primitive elements. For example, a document storage and retrieval application can use primitive elements to represent handwritten inks, while using application-specific tags such as <*page*> and <*keyword*> to organize the ink documents. Typically traces can be grouped into <*page*>s. Within a page, traces may be tagged as <*keyword*> or <*message*> etc. As another example, a handwriting recognition application can use primitive elements to record handwritten ink, and use a rich set of UNIPEN-like elements for annotation about recording condition, device information, writers, segmentation, data layout and so on. This can be used to offer the functionality previously available with the UNIPEN format to support the needs of online handwriting recognition developers.

CHAPTER 3 REVIEW OF DIGITAL INK SDKs

The CrossPad and Tablet PC are two popular pen-based devices. This chapter describes their associated software development toolkits. Through these users can access the handwritten ink captured by the device for further manipulation.

3.1 The IBM CrossPad SDK

The IBM Ink Manager SDK is a software development toolkit for processing the ink that originates on the devices including the IBM ThinkScribe™ digital notepad, or the A.T.Cross CrossPad™ digital notepad. Because this thesis is interested in the applications on the CrossPad, we will use the term "IBM CrossPad SDK" to refer to the IBM Ink Manager SDK in the later chapters, even though it applies to all these devices.

3.1.1 Introduction

The CrossPad portable digital notepads are pen-based input devices. They are used with ordinary paper and a digital pen. The digitizer under the writing surface can sense the pen, and capture and digitize the writing on the paper. The digitizer samples the pen location approximately 100 times per second, and records the pen location as a sequence of (x, y) coordinates. Handwriting is available in both physical ink on paper and digital ink simultaneously with the CrossPad.

The captured handwriting ink with the CrossPad can be uploaded to a computer via a serial port by the *InkTransfer* application. Then the ink can be displayed, managed or manipulated with a software application called the *Ink Manager* on PC.

The ink files produced by the *InkTransfer* upload application are called "ink device format files" (*.pad). The Ink Manager application uses a notebook format (*.nbk), which is a collection of uploaded files. SDK-based applications are able to read device format files and notebook files in addition to SDK created ink files (*.ink).

The IBM CrossPad SDK provides a rich set of programming operations for processing the ink that originates on the CrossPad. It allows users to write their own applications that use the CrossPad as a data input device. The Ink API is available in two variants, a C++ version and a Java version. IBM has achieved a very high level of consistency between these two toolkits. Since this thesis uses C++, we will review the C++ version of Ink API in the later sections.

3.1.2 Ink APIs

This section is based on the description of IBM C++ Ink Manager Pro SDK 1.0 Application Writer's Guide [5], and summarizes the C++ version of the CrossPad API's usages. The API is organized as eight packages. They are: api, forms, inkman, io, reco, system, util and export. The ***system*** package is the lowest level package. It provides system-level standard interfaces that all the other packages require. For example, Bytes and Enumeration are two classes in this package. The ***inkman*** package is at the highest level built on all the other packages. It provides the interface upon which the Ink Manager application is built. The ***util*** package provides miscellaneous utility classes. Besides packages mentioned above, the other five packages are most frequently used for developing user digital ink applications. The ***api*** package provides the basic ink manipulation functionality, such as creating new ink and modifying existing ink. The ***io*** package enables the input and output of digital ink between the internal ink format and the device file format. There is a handwriting recognition engine in SDK, and the ***reco*** package is the one to support the handwriting recognition functionality. The ***forms*** package provides the means for creating form specifications. The ***export*** package provides the functionality to export digital ink to image formats such as bitmaps etc. The remaining sections review and summarize the usages these packages.

3.1.2.1 API Classes

The ***api*** package provides APIs for performing the basic ink data functionality, such as creating, accessing and editing ink data. It includes five fundamental classes for

representing ink: *Point*, *Stroke*, *Scribble*, *Page* and *PageSet*, as well as other auxiliary classes such as *BoundingBox* and *Attribute*.

Point, *Stroke*, *Scribble*, *Page* and *PageSet* are the core classes in the SDK to define ink data structures. Their aggregation relationship is shown in Figure 1. The class *Point* represents ink points on the screen. Each Point is record in its (x, y) coordinate. The class *Stroke* represents ordered collections of Points starting from a "pen down" and ending with a "pen up". Every Stroke has a BoundingBox, a rectangle enclosing all the ink points of this stoke. Every Stoke also has an immutable timestamp recording the creation time of this stroke. The class *Scribble* represents ordered collections of Strokes. Like a Stroke, every Strokes has a BoundingBox and an immutable timestamp as well. The class *Page* represents the ink on a physical page of paper. A Page is composed of a collection of Scribbles, a PageSize, a creation date, a modification date and a unique ID. The creation date and ID of the Page are generated at creation time, so they are immutable. The class PageSet represents any collection of Pages.

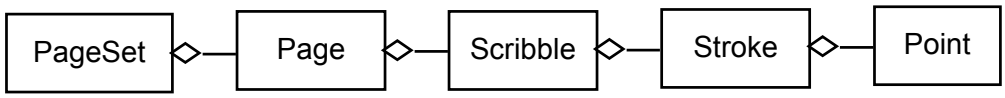

Figure 1. CrossPad Ink Class Aggregation

In addition to the five core classes presented above, there are three sets of classes are important and described in the following.

The first set is attribute classes. User may attach attributes to a Scribble, Page, or PageSet. The SDK defines a set of concrete attributes classes such as *BookMarkAttribute*, *HighlightAttribute*, *InkDisplayAttribute*, *TextAttribute* (derived by *KeyWordAttribute*). All Attributes eventually subclass from the abstract base class *Attribute*, and users are also able to create concrete classes derived from it for their needed attributes. The container *AttributeSet* class represents collections of Attributes on an ink object.

The second set of classes is associate to event Handling. CrossPad SDK employs Talker-Listener model to accomplish event handling. All relevant classes are eventually

derived from two abstract base classes: *Talker* and *Listener*. Classes that wish to have listeners attached should extend *Talker*. Within the SDK, the classes Scribble, Page and PageSet have a private data member derived from Talker. Similarly classes that wish to listen to some events should extend *Listener*. The derived classes *ScribbleListener*, *PageListener* and *PageSetListener* represent Listeners interested in being notified about modifications to Scribble, Page and PageSet, respectively. The container *ListenerSet* class represents collections of Listeners attached to an ink object.

The third set consists of the "walker" classes. There are three abstract base classes in the API used to develop applications in Walker pattern (also known as Visitor pattern): *ScribbleWalker*, *ScribbleSetWalker* and *PageSetWalker*. ScribbleWalker represents walkers that walk through every Point of a Scribble; ScribbleSetWalker represents walkers that walk through every Scribble of a ScribbleSet; similarly PageSetWalker represents walkers that walk through every Page of a PageSet.

3.1.2.2 I/O Classes

The ***io*** package provides interface for ink data read and write. It includes basic I/O, as well as Attribute I/O.

Basic I/O refers to classes that responsible for reading and writing ink file. Ink files are read using the class *Reader*. The Reader class provides functionality for reading upload device files (*.pad), notebooks files (*.nbk), and ink files generated by SDK-based applications (*.ink). To write ink, the SDK provides class *WriterV8*, which writes ink to files in the standard format used by SDK-based applications (*.ink).

Basic I/O does not support Attributes reading or writing. Instead, the SDK provides a pair of abstract base classes to enable applications reading and writing attributes attached to the ink: *AttributeDemarshaller* for reading, and *AttributeMarshaller* for writing. When an ink file is written, its attributes are converted to sequences of bytes using an instance

of an *AttributeMarshaller*; when an ink file is read, an instance of *AttributeDemarshaller* is used to convert sequences of bytes back into the Attribute instances they represent. [5]

Other classes in this package provide facilities of monitoring I/O progress, handling I/O errors, etc.

3.1.2.3 Reco Classes

The SDK includes a handwriting recognizer via the ***Reco*** package. To perform handwriting recognition, a user must create a *Context* object to specify the alphabet set and the *Wordlist* that the recognition engine will use to constrain the recognition result. This is then passed to the recognizer. The *InkRecognizer*, subclass of the interface *Recognizer*, is the concrete class used to perform the recognition and translate the ink Scribbles into characters.

3.1.2.4 Forms Classes

The ***forms*** package provides APIs for form operations. The abstract base class Field represents a general field in a form, and five derived classes: *BinaryField*, *BooleanField*, *CurrencyField*, *NumberField*, *StampField* and *StringField*, are used to define particular kinds of fields when creating a form. In addition, the class *FormDB* provides the ODBC (Open DataBase Connectivity) database interface and the class *FormIO* provides methods to perform reading or writing of a form.

3.1.2.5 Export Classes

The ***export*** package provides APIs for exporting ink to various image representations. The classes *Raster* provides the common raster formats interface; the classes *BMP*, *JPEG*, *PNG* and *TIFF* are derived from the class *Raster*, and provide methods that generate BMP, JPEG, PNG and TIFF image file representation of the ink. The ink also can be exported to PDF and PostScript format via classes *PDF* and *PostScript*.

3.1.2.6 Inkman Classes

The ***Inkman*** package is at the highest level. It is built on all the other packages and is an extension of lower-level packages to provide users more flexible means to develop ink applications. The Ink Manager application is built on this package. The *InkMan* package provides three groups of interface.

The first group consists of classes necessary to represent a notebook and provides an interface for manipulating ink on it. The classes include: *InkManagerNotebook, InkManagerSection*, O*rderedScribble* and the concrete derived classes, *ScribbleAttributeIndex* and its associated classes.

The second group consists of various Attribute and associate classes. The Scribble-Attribute classes are: AppointmentAttribute, DatedLabelAttribute, DatedTextAttribute, InkDisplayAttribute, KeywordAttribute, LabelAttribute, TextAttribute, TextNoteAttribute and ToDoAttribute. The Page-Attribute classes are: BookmarkAttribute, HighlightAttribute and KeywordListAttribute. Each Attribute describes an ink attribute and users can easily tell the attribute purpose from the class name. For example, InkDisplayAttribute class specifies the manner in which ink is rendered. When this attribute attached to a Scribble, it specifies the color and line-thickness of the scribble for rendering.

The third and smallest subset consists of four utility classes, for example the class Color, which represents colors using RGB values.

3.2 The Microsoft Tablet PC Platform SDK

3.2.1 Introduction

The Tablet PC is a format of tablet computers launched in 2002. Each is a general-purpose computer with an integrated interactive screen, accepting a pen stylus as an input device. A user can write on the screen with a pen stylus and thereby interact with the

computer. The digitizer inside the Tablet PC detects a hovering pen and takes samples of the pen's locations at least 100 times a second. The handwriting captured by the digitizer can be translated into text by some applications, or gestures by a built-in handwriting recognizer, or just treated "as ink", allowing the user to revise, edit, or re-purpose their handwritten input on the screen.

The Tablet PC uses the Microsoft Windows XP Tablet PC Edition operating system which is a version of Microsoft Windows XP bundled with special Tablet PC features that focus on the pen and digital ink. The Microsoft Tablet PC Platform SDK is a Software Development Kit to enable input, output and manipulation of handwriting data on Tablet PC as well as interchange of this data with other computers. It is based on .NET framework.

3.2.2 Tablet PC Platform SDK Overview

The remaining sections closely follow the presentation of the reference book "Building Tablet PC Applications" [6] and web source of MSDN Library (Windows XP Tablet PC Edition) [7], and give an overview of the Tablet PC platform SDK.

The Tablet PC platform can be divided into three distinct areas: (1) Ink collection (Pen API) for collecting ink from the digitizer; (2) Ink data management (Ink API) for managing the collected ink; and (3) Ink recognition (Reco API) for converting the ink into other types of data such as text [7].

The Tablet PC ink API is available in both the COM automation library and the .NET managed library. The automation library is the Tablet PC ink library with COM (Component Object Model) technology. COM has been used for many years in Windows applications, and enables software components to be linked and reused. The Tablet PC platform still supports this model. It also provides an automation library allowing developers more familiar with COM to write unmanaged codes with C/C++ and Visual Basic 6. The Tablet PC managed library is an ink library for the .NET framework. It

contains a set of managed objects that expose ink features. The term "managed" is .NET specific. It means the codes target the common language runtime, and the resources and services are managed at execution time. The majority of the objects in the Automation Library are identical to those in the Managed API, and an object in one library has a corresponding object in the other library. Figure 2 is a high-level view of the Tablet PC Platform SDK architecture presented in [6] (page 67). The COM automation APIs is at the lowest level, and directly uses Microsoft Win32 calls. The managed APIs are built on top of the COM automation APIs, essentially providing a managed wrapper for that functionality [6]. In addition, the Tablet PC Platform provides two controls: InkEdit and InkPicture, which allows easily add ink and handwriting recognition to Tablet PC applications.

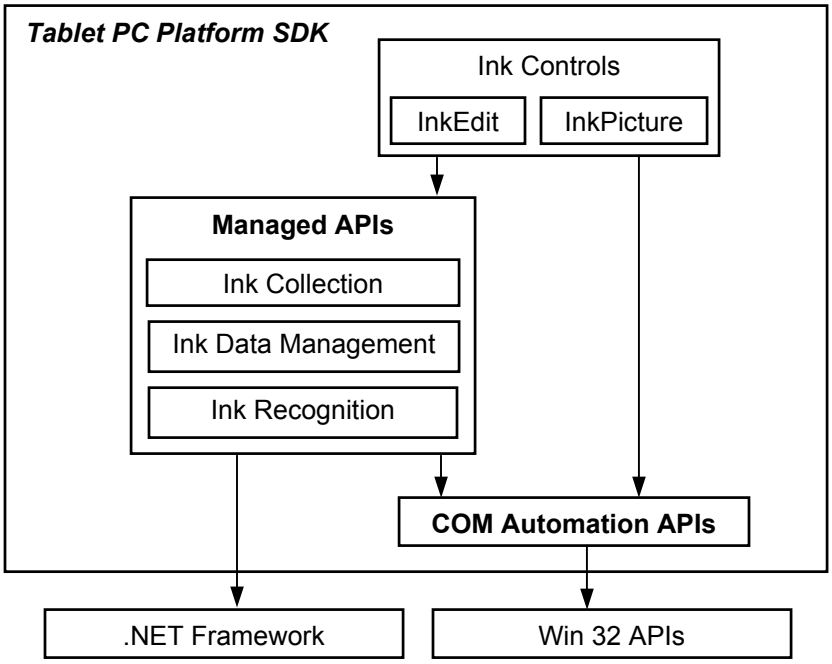

Figure 2. Tablet PC Platform SDK architecture

3.2.3 Ink Collection APIs

We have illustrated earlier that the Tablet PC ink API can be logically divided into three sets. In this section, we will have a brief review at the first set -- Ink Collection APIs. There are two key classes in this set that facilitate tablet input: the *InkCollector*

class and the *InkOverlay* class. They are both objects used to capture real-time ink from the tablet, and usually are used with windows form to capture ink. The difference between them is that *InkOverlay* supports interactive operations, such as selecting, moving, resizing and erasing, while *InkCollector* does not. *InkOverlay* has three different input modes indicated by property *EditingMode*: Ink, Select and Delete. If in ink mode, *InkOverlay* will act just like *InkCollector*. Once the digital pen touches the tablet, the ink is captured and drawn on the screen. In select mode, ink is selected when user taps or lassoes on it. In delete mode, ink is erased when it is touched by pen.

InkCollector and *InkOverlay* provide an extensive set of event notifications that can be used to trigger other operations. For better understanding, these events can be grouped into a number of categories. The first category of events includes *Stroke* and *Gesture* fired when ink is created. The second category can be referred as Pen Movement Events. Events are fired when some pen actions occur. For examples, *CursorButtonDown* and *CursorButtonUp* are two events fired when the button is either pressed or released. The third category is Mouse Trigger Events such as *DoubleClick*, *MouseMove*. The fourth category is Tablet Hardware Events including *TabletAdded* and *TabletRemoved*, occurs when a tablet device is either added or removed from the system. Two more categories, Rendering Events, and Ink Editing Events, are only provided in *InkOverlay* class.

The *InkCollector* and *InkOverlay* classes expose packet properties via the property DesiredPacketDescription. The packet property is used to describe an ink point such as X, Y coordinate and pressure. The property value is captured from digitizer, so it is device-dependant. For example, some devices support normal pressure, but some doesn't. By default, the *DesiredPacketDescription* property of *InkCollector* object is an array containing the X, Y. The user can add more packet properties by modifying the *DesiredPacketDescription* if the tablet hardware supports. Table 1 lists some of the PacketProperty members and their descriptions selectively extracted from MSDN web source [7].

PacketProperty Member Name	*Description*
X	Specifies the x-coordinate in the tablet coordinate space. The origin (0,0) of the tablet is the upper-left cornet.
Y	Specifies the y-coordinate in the tablet coordinate space. The origin (0,0) of the tablet is the upper-left cornet.
NormalPressure	Specifies downward pressure of the pen tip on the tablet surface. The greater the pressure on the pen tip, the more ink that is drawn.
TangentPressure	Specifies diagonal pressure of the pen tip on the tablet surface.
TwistOrientation	Specifies clockwise rotation of the cursor about its own axis
RollRotation	Requiring a three-dimensional digitizer, specifies the clockwise rotation of the pen about its own axis

Table 1. Some PacketProperty Members and their Descriptions

3.2.4 Ink Data Management APIs

This section describes the second part of the Tablet PC APIs – the Ink Data Management APIs. These provide the ability to interact with, manipulate, edit and save ink data.

3.2.4.1 Ink Data Structures

Ink, *Stroke* and *Strokes* are key classes for the representation of ink. The *Ink* class is the fundamental data structure used to represent ink captured by the Tablet PC. An *Ink* object is a container for *Stroke* objects, while a *Stroke* object is essentially an ordered collection of packets that is captured in a single pen-down, pen-move and pen-up sequence. A packet is the set of data that the tablet device sends at each sample point, such as (x, y) coordinates, pen pressure, pen angle, and so on, as specified by *PacketProperty* discussed in previous section. The *Ink* class typically exposes its *Stroke* objects through another collection class called *Strokes*. The *Stokes* is actually a collection of references to *Stroke* objects. Figure 3 shows the relationship between Ink, Stoke and Strokes classes.

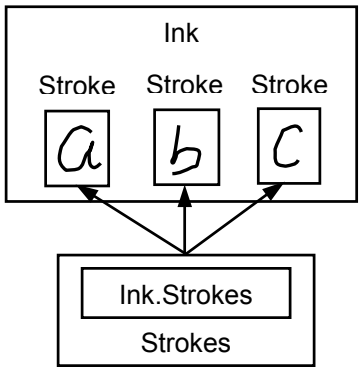

Figure 3. Relationship between Tablet PC Ink, Stroke and Strokes objects

3.2.4.2 Ink Creation and Deletion

An *Ink* object maybe created in two possible ways: either automatically created when an *InkCollector* or *InkOverlay* object is created and ink is collected, or created explicitly via constructors. Since a *Stroke* is always owned by an *Ink* object, the functionalities to copy, delete, remove or construct new *Stroke* objects are exposed in *Ink* class. Some often used methods are the following: The *CreateStroke/CreateStrokes* method constructs a new *Stroke/Strokes* object within an *Ink* object, The *AddStrokesAtRectangle* method is used to add existing strokes into an *Ink* object. The *Clone* method clones *Stroke* object. The *DeleteStroke* and *DeleteStrokes* methods destroy *Stroke* objects. *ExtractStrokes* is a method to copy or move Stroke objects from its owning Ink object into a new Ink object. Each *Stroke* object is assigned a unique ID at the creation time. This ID is immutable and remains unchanged for the Stroke's lifetime until the Stroke is destroyed.

3.2.4.3 Ink Rendering

The functionality for rendering ink strokes is encapsulated in class *Renderer*. Its primary purpose is to draw ink into a viewport and maintain a transformation on the ink space [6]. Figure 4 shows how a Reneder object draws ink strokes to a viewport, presented on [6] (page 209).

The *Renderer* class provide method *Draw* for drawing ink to either a Graphics object or a Windows GDI device context (HDC). The *Renderer* class also provides methods

InkSpaceToPixel and *PixelToInkSpace* to convert from ink space to pixels or vice versa, using either a Graphics object or an HDC to obtain the pixel dpi.

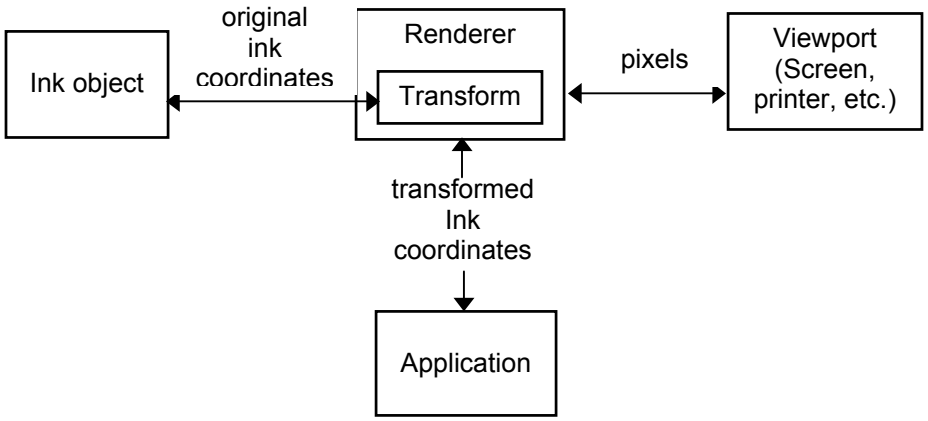

Figure 4. Two Functionalities of Renderer Class

Another ability *Renderer* provides is transforming data in the ink space. This is very useful to enable functionality such as zooming, rotating and resizing ink. There are two transformations maintained in the *Renderer* class: one is *view transformation*, and the other is *object transformation*. The difference between them is that view transformation will scale thickness of a stroke, while object transformation will not. The methods provided to work with the transformations are: *Move*, *Rotate*, *Scale*, *GetObjectTransform*, *GetViewTransform*, *SetObjectTransform* and *SetViewTransform*.

The Tablet PC provides the class *DrawingAttributes* to encapsulate the various properties and information that determine the ink's view when rendered to a device. The properties include: AntiAliased, Color, FitToCurve (ink rendered as a series of straight lines or Bezier curves), Height, IgnorePressure (whether the thickness of ink varies with pressure data or not), PenTip (ball or rectangle), RasterOperation, Transparency and Width.

3.2.4.4 Ink Interoperability

Ink data is always saved at the *Ink* object level, and this is accomplished with the *Ink* class' *Save* method. There are four persistence formats ink data can be saved shown in

Table 2, quoted from [6] (page 325). The method *Load* allows user to reconstitute ink from its persistence format into an *Ink* object.

PersistenceFormat Enumeration	***Description***
InkSerializedFormat	The Ink Serialized Format (ISF), typically used to save and load ink data.
Gif	The Graphics Interchange Format, typically used for viewing ink in Web browsers.
Base64InkSerializedFormat	The ISF, which is then Base64-encoded, typically used for storing ink in XML
Base64Gif	The Graphics Interchange Format, which is then Base64-encoded, typically used for viewing ink via .mht files.

Table 2. Ink Persistence Formats and Descriptions

The ink SDK also provides the method *ClipboardCopy* to cut or copy ink data from an *Ink* object to the clipboard in various formats according to the value of enumeration *InkClipboardFormats*. This provides choices such as Bitmap, CopyMask, Metafile, InkSerializedFormat, etc. Correspondingly, the method *ClipboardPaste* will read back the ink data from the clipboard and merge it into an *Ink* object.

3.2.5 Ink Recognition APIs

The third part of the Tablet PC APIs is the set of Ink Recognition APIs. Tablet PC ink recognition comprises two types of recognitions: gesture recognition and handwriting recognition.

Gesture recognition is the ability to translate ink strokes in predefined shapes to specific application commands, such as copy, paste, undo, and etc. For example, if the user scratches-out on the tablet with the digital pen, this gesture can be interpreted as an erasing command. The gesture recognition is accomplished by setting a proper ink collection mode on an *InkCollector* or *InkOverlay* object.

Handwriting recognition refers to the ability to translate handwritten ink into text for languages. The easiest way to perform recognition is to use the *Strokes* class' *ToString*

method. The strokes are sent to the default recognizer, and the highest probability result is returned as a string. The user also can perform more complex recognition by specifying an appropriate recognizer instead of using the default recognizer, or requesting more alternative recognition results instead of only having the highest probability result returned. The recognition also can be performed in asynchronous mode to make the process more interactive.

The classes *Recognizers*, *Recognizer*, *RecognizerContext*, *RecognitionResult* and *RecognitionAlternate* are the key classes provided to perform more advanced handwriting recognition. The *Recognizers* is a collection of different recognizers installed and used in the system. The *Recognizer* class represents the ability to translate the strokes into text. By calling method *CreateRecognizerContext*, the recognizer creates a *RecognizerContext* object, which is used to perform the actual handwriting recognition, retrieve the recognition result, as well as alternates. The result of the recognition is represented by the *RecognitionResult* class and the *RecognitionAlternate* class represents the possible alternate results.

Ink recognition is a computationally intensive operation. The Tablet PC therefore supplies two usage models to perform the recognition: synchronous and asynchronous. Synchronous recognition occurs when the thread requesting recognition results blocks until computation is complete. Asynchronous recognition is non-blocking: the thread requesting recognition result is allowed to continue, and is later notified that computation is complete [6]. So asynchronous mode allows more interactivities than synchronous mode. Another important concept is partial recognition, which refers to an incrementally recognition – the recognition begins as soon as any ink is given, and incrementally adjusts the computation as ink added or removed, or recognition properties change [6]. Partial recognition also improves the recognition's timely performance, since the strokes associated are kept up-to-date at all times, and computation is continuous. Otherwise the recognition is performed all at once, which may lead to pauses.

To perform advanced recognition with recognition APIs, the following steps are involved: The first is to obtain a recognizer. The Tablet PC platform allows multiple recognizers to be installed. The method *GetDefaultRecognizer* serves to get the default recognizer, and the method *Item* serves to get a specific recognizer by specifying an index in the collection. A recognizer's specific capabilities are obtained through the *Recognizer* class' *Capabilities* property, which returns a *RecognizerCapabilities* enumeration. For example, the value *DownAndRight* means the recognizer supports downward and rightward text flow as in the Chinese language.

After obtaining a *Recognizer*, a *RecognizerContext* needs to be created by invoking the *Recognizer*'s *CreateRecognizerContext* and setting the ink stroke to be recognized. This *RecognizerContext* object represents a recognition session, in which all the context information required for the recognition is associated. It contains the actual ink strokes to be recognized, parameters determining the recognition mode or improving the recognition accuracy, and any recognition results. An example of contextual information is the *Factoid*. A user may pass in the constant *Factoid.TELEPHONE* to *RecognizerContext*, which tells the recognizer that the ink it will perform recognition on is probably representing a phone number. This helps the recognizer can distinguish between letters and numbers such as 0 or O, and 1 or l.

The recognition results can be obtained either synchronously or asynchronously as discussed earlier. The *Recognize* method of the *RecognizerContext* class is used to obtain recognition results synchronously; while either the method *BackgroundRecognize* or the method *BackgroundRecognizeWithAlternates* is called to perform asynchronous recognition. If we also consider partial recognition, recognition can be in one of four modes: synchronous, synchronous with partial recognition, asynchronous, and asynchronous with partial recognition. Recognition results are encapsulated in *RecognitionResult* object, including result alternates, confidence and stroke association. The *TopAlternate* property of the *RecognitionResult* provides the top alternate for the result, returning an instance of the *RecognitionAlternate* class, while the method

GetAlternatesFromSelection is called to return a collection of other alternates by the way of the *RecognitionAlternates* class.

CHAPTER 4 ISSUES IN CONVERSION BETWEEN DATA FORMATS

In Chapter 2, we saw that UNIPEN, Jot and InkML are very different digital ink data standards with different design goals, focusing on different aspects of ink properties. This has limited what the ink applications can do in heterogeneous environments. We therefore want to develop conversions among the different ink formats to promote the portability of ink services. This chapter discusses some issues in the conversions among UNIPEN, Jot and InkML formats.

4.1 UNIPEN ↔ Jot

4.1.1 Correspondence of Data between Formats

This section illustrates the counterparts between UNIPEN and Jot, so we can convert from one format to the other without losing information.

The most basic component in recording ink is the set of ink points of the pen trajectory. Each point of the ink should at least store its (x, y) position, and can have optional data channels, such as pen pressure, pen angle, etc., depending on the capabilities of the digitizer. In the UNIPEN format, the keyword *.COORD* declares the data channels for the points, and actual data goes between *.PEN_DOWN* and *.PEN_UP* components. In Jot format, the data channel is specified by *INK_BUNDLE*_FLAGS, and all data is encapsulated in the *INK_POINT* structure. Table 3 compares the ink data channel recorded for each pen point in both formats. Most of the channels have correspondences between the formats, and the conversion is straightforward. The main exception is the T (time) channel, which is optional. In the UNIPEN format it is recorded for each point, i.e., each point has a timestamp. In Jot format, there is not a timestamp for every point. Instead the time is recorded in the structure *INK_STRART_TIME_RECORD* and *INK_END_TIME_RECORD* to indicate the timestamp of the first point and the last point for a sequence of "point values". This reduces the space occupied when the amount

of point data is large, and makes the Jot format more compact. Time information for the intermediate points may be reconstructed using knowledge of the sampling frequency.

UNIPEN	***Jot***	***Description***
X, Y	Position	X, Y position of the pen on the tablet
P	Force	Pen pressure on the tablet
Z	Height	Altitude of stylus above the tablet
BUTTON	Buttons	Barrel button states
RHO	RHO	Rotational angle of the stylus, measured in degrees from some nominal orientation of the stylus.
THETA, PHI	Angle	Tilt along the x-axis or y-axis
T		Time in milliseconds

Table 3. Data Channels for Pen Point in UNIPEN and Jot

Turning now to other parameters, both formats provide means to represent the x amd y resolutions of the data collection device (see Table 4). In UNIPEN, the keywords *.X_POINTS_PER_INCH*, *.Y_POINTS_PER_INCH*, *.X_POINTS_PER_MM*, *.Y_POINTS_PER_MM* are used. Correspondingly, in Jot, *INK_BUNDLE_RECORD* contains data members *penUnitsPerX* and *penUnitsPerY* to store this information. Some standard correspondences are as follows:

1000 points per inch digitizer == 39370 pen units per meter

500 points per inch digitizer == 19685 pen units per meter

200 points per inch digitizer == 7874 pen units per meter

254 points per inch (1/10 mm)== 10000 pen units per meter [3]

UNIPEN	**Jot**	**Description**
.X_POINTS_PER_INCH .X_POINTS_PER_MM	penUnitsPerX	X resolution of the data collection device
.Y_POINTS_PER_INCH .Y_POINTS_PER_MM	penUnitsPerY	Y resolution of the data collection device

Table 4. Other Corresponding Data Channels in UNIPEN and Jot

4.1.2 Information Lost in the Conversion

Unfortunately, not all data in one format has a counterpart in the other format. From the design point of view, UNIPEN is aimed to suit the needs of people testing handwriting recognition algorithms on large amounts of data, while Jot is application-oriented, to provide a terse and sufficient standard for applications running on small platforms like PDAs and pen-based notebook computers. UNIPEN therefore takes a lot of space for data annotations about ink file documentation, device information, recording condition, writers, segmentation, data layout, data quality, labeling and recognition results. But Jot does not have provisions for this information. As a result, this data will be lost when converting from UNIPEN to Jot. On the other hand, UNIPEN has no provisions for pen tip and ink color, but Jot has, since pen tip and ink color are more application-relevant properties. Therefore in conversion from Jot to UNIPEN, pen tip and ink color data will be lost.

4.1.3 Stroke Group

In UNIPEN, pen data is grouped in the unit of the stroke, i.e., ink data between a *.PEN_DOWN* and a *.PEN_UP* instruction is a stroke. In Jot, the smallest unit to record a sequence of ink points is *INK_PENDATA_RECORD*. It contains the actual pen data for one or more pen strokes. Multiple strokes are typically grouped into one record to increase the efficiency of the compression algorithm, though strokes may be stored individually, if desired. This causes the problem that there is no way to tell how many strokes are there in one *INK_PENDATA_RECORD*. Multiple strokes are not separated stroke by stroke as UNIPEN requires.

4.1.4 Ink Point Computation

UNIPEN is an ASCII format, and data in UNIPEN is recorded explicitly in absolute value. While Jot is a binary format, and data stored in Jot can be either in loss-less compression mode or non-compression mode. The detailed encoding schema of Jot was

described in section 2.2.4, and will not be repeated here. The computations of delta-calculation and compressing/uncompressing point data are needed in the conversion.

Jot records pen position relative to the lower-left (0, 0) corner of a logical page or window, and scale and offset properties cumulatively operate on the data. UNIPEN records ink positions in absolute values, therefore computation may required in the format conversion. In our conversion from UNIPEN to Jot, the scale is set to unity and offset is set to 0 by default.

4.1.5 Summary

In summary, the conversion between UNIPEN and Jot loses some information, especially form UNIPEN to Jot, where a large set of annotations is lost. Computation involved includes system unit transformation, scale and offset computation, compression and decompression.

4.2 JOT ↔ InkML

4.2.1 Overview

Jot is a format to maintain a complete likeness of the original ink as it was drawn, and is used for the storage and interchange of digital ink between applications running on small devices. It doesn't define any structures for applications in a specific area such as handwriting recognition. InkML however provides handwriting recognition specific elements and attributes. The conversion between Jot and InkML can be realized as a conversion between Jot and InkML primitive elements, since the set of primitive elements of InkML is sufficient for all the basic ink applications.

4.2.2 Ink Point Channels

As stated earlier ink point data in Jot is recorded through a sequence of *INK_POINT* structures, and whether an ink property, such as force or angle, is present depends on the specification given by the *INK_BUNDLE_FLAGS*. Similarly, InkML defines channels to

describe the ink data that may be encoded in a trace. Contiguous ink points are encoded within a <trace> element, and the <traceFormat> element defines the sequence of channel values that occurs within <trace> element. The comparison of ink data channels in both formats and their interpretation are listed in Table 5. The conversion of ink point data is realized by the conversion of *INK_POINT* structure in Jot and the <*trace*> element in InkML, and *INK_BUNDLE_FLAGS* structure in Jot and the <*traceFormat*> element in InkML.

Jot	InkML	Description
Position	X, Y	X, Y position of the pen on the tablet
Force	F	Pen pressure on the tablet
Height	Z	Height of pen above the tablet
Buttons	B1 … Bn	Barrel button / side button states
RHO	R	Rotation about the pen axis
Angle	Tx, Ty	Tilt along the x-axis or y-axis
	S	Tip switch state (touch/not touching the tablet)
	Az	Azimuth angle of the pen
	El	Elevation angle of the pen

Table 5. Comparison of Ink Point Data Channels in Jot and InkML

Form Table 5, we can see InkML provides more comprehensive channels for encoding ink data. Most ink properties have counterpart channels in both formats, except S, Az, EI. So the conversion from Jot to InkML can preserve all point data. On the other hand, in the conversion from InkML to Jot, the information about tip switch state, azimuth angle and elevation angle is lost if they are originally present.

There is a difference between Jot's *INK_BUNDLE_FLAGS* and InkML's <*traceFormat*>. InkML defines <*regularChannels*> whose value must be recorded for each sample point, and <*intermittentChannels*> whose value may optionally be recorded for each sample point. On the contrary, in Jot format, once a point element present is asserted in *INK_BUNDLE_FLAGS* of an ink bundle, the value of that element must be recorded for each ink point. This difference needs to be handled in the conversion between both formats.

4.2.3 Ink Mapping

Ink mapping often occurs in ink applications, especially in ink sharing among multiple devices. For example, an ink stream or file may contain traces that are captured on a tablet computer, a PDA device, and an opaque graphics tablet attached to a desktop. The size of these traces on each capture device and corresponding display might differ, yet it may be necessary to relate these traces to one another. They could represent scribbles on a shared electronic whiteboard, or the markings of two players in a distributed tic-tac-toe game. This may include two kinds of mapping: one is from original data captured by digitizing device to recorded trace values, the other is from the recorded trace to canvas coordinate system.

Recall that in InkML the correspondence between the trace data and the device channels is recorded using the *mapping* attribute of the <*channel*> element in the <*traceFormat*>. The transformation from trace coordinates to the shared canvas coordinate system is declared via the *mapping* attribute of the <*context*> element. In Jot, the *INK_SCALE_RECORD* and the *INK_OFFSET_RECORD* structures facilitate ink mapping and transformation. Ink scale and offset values are set by the storing application to be applied by the rendering application. For instance, if the storing application collected ink at scales of (2.0, 2.0), the storing application should insert an *INK_SACLE_RECORD* with a scale of (0.5, 0.5) for the rendering application to multiply all ink X and Y coordinates. This more likely corresponds to the mapping of the <*channel*> element in InkML, while the rendering in differing devices with Jot is left for the different rending applications. In addition, the scale and offset operations in Jot are cumulative, and the *INK_SCALE_RESET* record resets to the identity transformation matrix and zero offset. The conversion of ink mapping between InkML and Jot formats requires appropriate calculations. Form Jot to InkML, scale and offset can be converted to mapping attribute of the <*channel*>, but from InkML to Jot, the mapping attribute of the <*context*> may be lost.

In the conversion from InkML mapping attribute of <*channel*> in the <*traceFormat*> to Jot's scale and offset records, the information may not be reserved in some

applications. An example case is given below. In InkML, the mapping attribute has three forms. One is the value of "*" describing the identity mapping. Another is specifying an expression contains only channel names in the form of a "formula( )". The third is using a mapping value of the form "uri( )" refers to a resource such as a MathML document to describe more complex relations. Examples of these three formats are as follows:

```
<channel name="X" type="decimal" mapping="*"/>
<channel name="X" type="decimal" mapping="formula(3*X+5)"/>
<channel name="X"
         type="decimal"
         mapping="uri('http://www.uwo.ca/orcca')"/>
```

If the InkML file uses the third form, and the mapping rule is complicated (such as involving an integral), the information cannot be converted to Jot without explicitly computing all the points. Since the scale and offset in Jot is specified by a fixed value, only linear transformations will be considered. In the most general cases, the mapping will not be so complicated, and the computation to achieve the conversion between two formats is trivial.

4.2.4 Ink Encoding

As described earlier, in InkML, data is recorded in a <*trace*> element either with a regular channel or an intermittent channel. The specification defines that regular channels may be reported as explicit values, differences, or second differences, but intermittent channels are always encoded explicitly. Here is the example from 2.3.2.1:

```
<traceFormat>
  <regularChannels>
     <channel name="X" type="decimal"/>
     <channel name="Y" type="decimal"/>
  </regularChannels>
  <intermittentChannels>
     <channel name="F" type="decimal"/>
  </intermittentChannels>
</traceFormat>
```

```
<trace id = "id001">
84 652:5’1’2:’2 ”2”-1:”2 4 1:4-1 21:0 13-9:-2-3-5:2-9 10:0 15
18:-2-4-7:0;
</trace>
```

The trace is interpreted as following:

Trace	X	Y	F	vX	vY	vF	comments
84 652:5	84	652	5	?	?	?	
’1’2:’2	85	654	7	1	2	2	velocity values
”2”-1:”2	88	655	11	3	1	4	acceleration values
4 1:4	95	657	15	7	2	4	Implicit acceleration
-1 21:0	101	680	19	6	23	4	
13-9:-2	120	694	21	19	14	2	
-3-5:2	132	700	25	12	6	4	
-9 10:0	135	716	29	3	16	4	
15 18:-2	153	750	31	18	34	2	
-4-7:0	167	777	33	14	27	2	

In addition, the alphabetic characters may be used to encode small integer values. The letters “a” to “y” are interpreted as –1 through –25, “A” to “Y” are interpreted as 1 through 25, and “z” and “Z” are interpreted as zero.

The question of ink compression in InkML is still an ongoing debate. The readability of non-binary XML is an advantage, while the inefficiencies stemming from the document size is a disadvantage. It is believed by the working group that a binary encoding of InkML can be specified, however it hasn’t been incorporated in the current specification.

With Jot, ink data can be written in either uncompressed or compressed format. In uncompressed format, delta values are stored to represent a sequence of points, where the first point is always relative to the defined default values for each component of the point. This delta-oriented format corresponds to the single difference encoding of InkML. In

compressed format, point values stored are delta-oriented and compressed by reserved encodings as described in section 2.2.4.

In summary, in the conversion of ink data between two formats, string manipulations of InkML trace data, compressing and uncompressing of Jot data, and delta computations are involved.

4.2.5 Ink Group

Ink can be grouped in both Jot and InkML formats, and groups are nestable in both. Recall that, in Jot, *INK_GROUP_RECORD* groups *INK_PENDATA_RECORDs*, and each group is assigned a group number. In InkML, the elements <*traceGroup*> and <*traceRefGroup*> are used to group a number of traces. The conversion concerning the ink grouping between Jot and InkML is not straightforward.

When converting Jot to InkML, a single group of ink data can be encoded as a <*traceGroup*> or <*traceRefGroup*>, while nested ink groups must be encoded into <*traceRefGroup*>. Another issue is that users may change pens or colors when writing. This may cause the *INK_COLOR_RECORD* or *INK_PENTIP_RECORD* to be changed within an ink group in Jot because the interpretation of grouping is up to the application. For example, grouping could be used in drawing programs for the user to move or copy an entire group of ink with different pen tips or colors. This would cause a new <*traceGroup*> to be created within a group in the conversion due to the grouping criteria. On the other hand, InkML usually groups successive traces with common characteristics such as the same brush.

In the other direction, converting InkML to Jot, when a <*traceRefGroup*> is referenced in multiple groups, the trace may be copied multiple times in the resulting Jot format, since Jot does not provide any way for an ink record to refer to other records.

4.2.6 Ink TimeStamp

Jot and InkML both support ink timestamping, and time is measured in both milliseconds. Compared with Jot, InkML provides a relatively more flexible timestamping mechanism.

In Jot, *INK_START_TIME_RECORD* and *INK_END_TIME_RECORD* are used to record the start and end time of an ink bundle. There is no absolute time in Jot, all timestamps recorded are relative to the previous one, and the base timestamp is arbitrary.

In InkML, the element <*timestamp*> and attributes start, duration, timeOffset and timeRef are used for this purpose. The start time can be recorded either in an absolute value or a relative value. The absolute timestamp refers to the time in milliseconds since 1 January 1970 00:00:00 UTC. The attribute timeOffset along with the attribute timeRef is used to specify a relative timestamp. The timeRef refers to a <*timestamp*> element, and the timeOffset is the relative value to that timestamp. The attribute duration records the duration of a trace.

Thus, the ink time conversion from InkML to Jot is trivial. The start time in Jot is arbitrary and relative relationship is used, so the value of the end time in Jot format is actually the value of duration in InkML. However, this conversion loses information if an absolute start timestamp, or a reference timestamp is used in InkML format. On the other hand, it is not possible to convert timestamp in Jot to InkML accurately, because there is no way to set the timeRef attribute due to the fact that the base time in Jot is arbitrary.

4.2.7 Other Issues

INK_COLOR_RECORD and *INK_PENTIP_RECORD* are two records in Jot to describe ink attributes. In InkML, these attributes are captured in the <brush> element. The conversion between them is straightforward.

4.2.8 Summary

InkML is a more comprehensive ink data specification than Jot. For the records in Jot, there is always a corresponding element in InkML, so the conversion from Jot to InkML is almost lossless. However, the conversion from InkML to Jot is a lossy-conversion, because a large set of information in InkML has no appropriate way to be recorded in Jot. This includes context information for capture devices, canvas, and so on. In the above discussion, we have covered almost all records defined in the Jot format, but just some of elements definitions in InkML.

4.3 InkML ↔ UNIPEN

4.3.1 Overview

InkML provides a set of application-specific elements for handwriting recognition, which incorporate the features of the UNIPEN format. Like UNIPEN, it is intended to support the needs of online handwriting recognition developers requiring large corpora of handwriting samples stored in a common format. InkML provides the same rich annotation possibilities as the current UNIPEN format, and adds a number of improvements. Therefore, the conversion between UNIPEN and InkML can be realized as a conversion between UNIPEN and handwriting recognition-specific InkML. This uses InkML primitive elements to tag the ink data and application-specific tags to organize the document.

4.3.2 UNIPEN and UNIPEN-like InkML

All the keywords in the UNIPEN format have corresponding elements in InkML. For example, the keyword *.DATA_SOURCE* in UNIPEN corresponds to the element <*source*> in InkML, *.DATA_ID* and *.HIERARCHY* correspond to the <*dataBlock*> element's attributes *id* and *hierarchy*. An appropriate table matching all corresponding keywords and elements can be used by the converter when performing the conversion from UNIPEN to InkML, and vice versa.

On the other hand, the InkML handwriting recognition-specific schema not only offers the UNIPEN functionality, it also adds some improvements. Since InkML is XML based, one of the improvements comes from the use of XML complex types to group related annotations. For instance, in UNIPEN, the writer information is described with the keywords *.STYLE*, *.WRITER_ID*, *.COUNTRY*, *.HAND*, *.AGE*, *.SEX*, *.SKILL* line by line. In InkML, all this information is grouped together. The element <writerInfo>, with attribute *id*, contains elements <style>, <country>, <hand>, <age>, <sex> and <skill>, describing a given writer. As a result, in the conversion from UNIPEN to InkML, the UNIPEN components must be re-arranged and the related components must be grouped to conform to the InkML schema. For the other direction (converting from InkML to UNIPEN), the ungrouping of XML components and rearrangement to conform the UNIPEN format is necessary.

4.3.3 Summary

The conversion between UNIPEN and InkML is actually the conversion between UNIPEN and UNIPEN-like InkML. Regardless of the various improvements of InkML mentioned in the previous section, the conversion is lossless in both directions. Rearrangement of keywords in UNIPEN, elements in InkML, and computations of point values in both formats are involved. As to the ink encoding, UNIPEN records points using explicit values, while in InkML the values can be explicit values, differences, or second differences. The details of the InkML encoding have been discussed in the previous section, and will not be repeated here.

Being familiar with the ink data encoding in the above discussion of UNIPEN vs. Jot and Jot vs. InkML conversions, the computation of ink point data between UNIPEN and InkML is similar and straightforward.

CHAPTER 5 IMPLEMENTATION OF DATA FORMAT CONVERSIONS

5.1 Notes to the Conversions

The issues in converting among UNIPEN, Jot and InkML have been discussed in the previous chapter. This chapter is about the implementation of these conversions. Because the InkML specification is still evolving, we will have to leave the conversions of InkML vs. UNIPEN and InkML vs. Jot for future work. Here we describe the implementation of the conversion between UNIPEN and Jot. As mentioned earlier, the conversion is based on the UNIPEN 1.0 definition and the JOT 1.0 specifications.

5.2 UNIPEN ↔ JOT Conversion

5.2.1 Principle Design

The specification of JOT 1.0 is written in the C language, thus we have chosen to do the conversion between Jot and UNIPEN in C. The data having counterparts in both formats shown in Table 3 and Table 4 can be converted from one format to another without losing information. Two programs accomplish the conversion: one converts UNIPEN to Jot, another performs the conversion in the opposite direction. The conversion scheme (see Figure 5) is straightforward.

To convert UNIPEN to Jot, (1) we read a data stream from the UNIPEN file, (2) parse and extract the data that can be converted, (3) convert to Jot format by the mapping rules, and then (4) write the stream to the output file. The mapping rules for conversion are hard-coded in the programs. The rules are based on the correspondence representation between the two formats shown in Table 3 and Table 4. Similarly, to convert Jot to UNIPEN, we read the data stream from the Jot file, and convert the Jot structures to the UNIPEN format. The length of the data stream to read is encapsulated in the header of each structure.

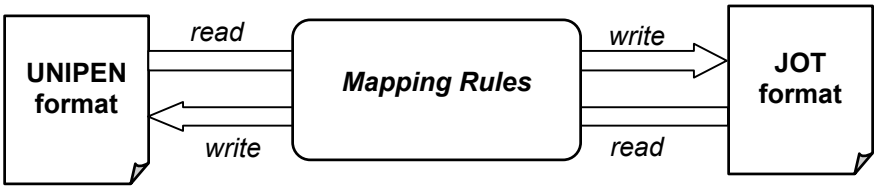

Figure 5. UNIPEN vs. Jot Conversion

As discussed in previous chapter, ink data in UNIPEN is recorded using absolute values, while in Jot uses delta values. The ink point data in Jot can be written in either compacted or uncompacted format, as specified by compactionType in INK_BULDLE_RECORD. Therefore computation is involved in the conversions. This includes the delta-value computation, and compression/de-compression. The program converting UNIPEN to Jot uses a command line argument to determine whether ink to be compacted or not.

In Jot format, the first ink point in a pen data record is always written using an absolute value, while the proceeding points are stored in signed delta values, each added to the previous value. If the data is compacted, the encoding algorithm uses "reserved encodings" (we have described earlier in 2.4.4). Let us take the point position compression from UNIPEN to Jot as an example. For clarity, we show the compact format definition for (x, y) position again (quoted from [3]), and then give the pseudo-code for compressing (X, Y) position data in conversion from UNIPEN to Jot:

32-bit absolute X/Y: Two 32 bit long words, first two bits are 00. Data is actually two S31s.

```
| 0 | 0 |        (30 low-order bits of X)              |
| X|       (sign bit of X plus 31 bits of Y)          |
```

16-bit short delta X/Y: Two 16 bit short words, first two bits are 0 1. Deltas are actually two S15s. Values that would fit into an 8-bit byte delta are reserved.

```
| 0 | 1 |     (14 low-order bits of delta-X)          |
| X|      (sign bit of X plus 15 bits of delta Y      |
```

8-bit byte delta X/Y: Two bytes, first two bits are 1 0. Deltas are actually two S7s. Values that would fit into a 4-bit nibble delta are reserved.

```
| 1 | 0 |      (6 low-order bits of delta-X)    |
| X|   (sign bit of X plus 7 bits of delta-Y) |
```

4-bit nibble delta X/Y: One byte, first two bits are 1 1. Deltas are actually two S3s.

```
| 1 | 1 | (S3 delta-X) | (S3 delta-Y) |
```

The pseudo-code for compressing (X, Y) position is as follows:

```
if (the first point)
   write as 32-bit absolute X/Y
else
   compute deltaX, deltaY
   if((MIN_S7<deltaX<MAX_S7)&&(MIN_S7<deltaY< MAX_S7))
      write as 8-bit byte delta X/Y
   else if((MIN_S3<deltaX<MAX_S3)&&(MIN_S3<deltaY<MAX_S3))
      write as 4-bit nibble delta X/Y
   else
      write as 16-bit short delta X/Y
```

5.2.2 Results

The UNIPEN-Jot converter was tested by a set of data, publicly available at the UNIPEN official website (http://hwr.nici.kun.nl/unipen/) [10]. All ink point data was converted from one format to another successfully. The set of data used to test UNIPEN-to-Jot converter is composed of UNIPEN files collected by Apple, Go, IBM, NICI and MIT. The files record ink with (x, y) position and pressure data.

Unfortunately, we could find no standard Jot format data set. We therefore tested the Jot-to-UNIPEN converter using the Jot data previously converted from the original UNIPEN data mentioned above. The results of converting UNIPEN format to Jot are summarized below.

Original UNIPEN File (byte)		***Converted Jot uncompacted (byte)***	***Converted Jot Compacted (byte)***
apple001.dat	87601	34887	11547
go001.dat	17672	13431	6183
ibm001.dat	34817	38249	12635
nici001.dat	621944	228893	71669
mit001.dat	3636	805	403

Table 6. Comparison of Size after Conversion from UNIPEN to Jot

The results conform to the features of both formats well. In general, when UNIPEN format is converted to Jot, the file size decreases because all the annotation information is lost. Compared with uncompacted Jot, the compacted Jot format further decreases the file size by 1/2 to 2/3, which supports the claim that Jot is relatively light-weight.

CHAPTER 6 INCOMPATIBILITIES BETWEEN THE TBLET PC AND CROSSPAD APIs

The IBM CrossPad API and the Microsoft Tablet PC API are both application programming interfaces aimed at processing digital ink captured by pen-enabled devices. They have many similar concepts in representing and manipulating ink, which makes the abstraction of a common API based on them possible. However they are two software development toolkits developed by two separate vendors, and serve two rather different devices - the CrossPad and the Tablet PC - so they have many incompatibilities. This chapter examines these issues. To preserve the logical progression, we repeat a few details, at times, that have been previously mentioned.

6.1 Managed and Unmanaged Code

Recall that the IBM CrossPad API provides two highly consistent versions: a C++ version and Java version. It is obvious that C++ is the only choice to implement an abstract API for the CrossPad and Tablet PC, given that the Tablet PC API is not available in Java. On the other hand, the Tablet PC API is available both in a Managed Library and an Automation Library for the .NET framework. The automation library is implemented for the Microsoft COM interface. Due to the inconvenience of COM type libraries, we chose to use the managed library interface for this thesis.

Any incompatibility then stems from the managed library of the Tablet PC for the .NET framework. As we know, the Common Language Runtime is the foundation of the .NET framework. Code that targets this runtime is known as “managed” code, while code that does not target the runtime is know as “unmanaged” code. Any development on the CrossPad is written in native C++, which is unmanaged. In contrast, on the Tablet PC, we need to use C++ with managed extensions to access managed objects on the .NET framework. The need to mix unmanaged and managed code brings some difficulties in our abstraction API implementation.

Managed Extensions for C++ are extensions to the Visual C++ compiler and language to allow them to create managed C++ code and enable access to the functionality of the .NET Framework. Managed C++ is an extension to C++, the runtime defines a particular object model but unfortunately does not support all features of the C++ language. For example, multiple inheritance of classes is not supported, const modifiers on member functions are not supported either. Many syntax incompatibilities also occur. For instance, a managed array is itself a __gc object, inheriting from System::Array. In contrast, an ordinary C++ array is not self-describing, so we have to specify the length of the array if we want to pass an array as parameter in a method. With __gc array, this is not the case. Another problem is that the mixture of managed and unmanaged codes is restrictive. For example, a managed class cannot be derived from an unmanaged class; an unmanaged class cannot contain a pointer pointing to a managed object, and so on.

6.2 A Document Model of Ink

Collections of ink can conform to various different semantically structured models. For example, if one is developing a "PowerJot" application in which the user writes words and sentences, these are the semantic elements. On the other hand, maybe the application is "Super-Doodle", in which case the digital ink is most likely a series of small drawings.

The Tablet PC API does not support a particular document model. It provides only a flat view of digital ink, an *Ink* object is simply a container for *Stroke* objects, and a *Strokes* collection references *Stroke* objects. The *Stroke* objects are essentially a collection of packets, and that's it.

In contrast, the CrossPad API supports a document model with semantic meanings for ink. *Strokes* are collected into a *Page*, representing the ink on a physical page of paper with a page size. Pages of ink are collected into a *PageSet*, representing any collection of *Pages*, say a notebook. A *Scribble* attached with an *AppointmentAttribute* represents an

appointment. A *Scribble* attached with a *KeywordAttribute* represents a keyword. A *Page* attached with a *BookmarkAttribute* represents bookmarks on a page.

The different document models of the Tablet Ink and CrossPad Ink bring concerns in defining the ink objects for an abstraction API. Will the abstract API follow the CrossPad model, or otherwise simply leave the ink as a plain view of ink like the Tablet ink?

6.3 Memory Management of Ink

Recall that *Point*, *Stroke*, *Scribble*, *Page* and *PageSet* are five ink data classes to represent ink in the CrossPad. Each can exist by itself, and their relationship is aggregation: A Stroke is an array of Point, Scribble is an array of Stroke, Page is an array of Scribble, and PageSet is an array of Page. The memory management system allows a given Scribble to be a member of multiple Pages, a given Page to be a member of multiple PageSets, and to be added to or deleted from a given Page or PageSet without affecting its status on other Pages or PageSets. As to the Point, Stroke and Scribble, an existing Point may not be altered, an existing Stroke may not be altered by any change to its constituent Points, and an existing Scribble may not be altered by any change to its constituent Strokes.

In the Tablet PC, the ink data classes are *Ink*, *Stroke* and *Strokes*. The Ink class is the outermost entry point into the Ink Data API. An Ink object owns a collection of Stroke objects, and a Stroke cannot exist without an Ink object as its owner. Although a Stroke may be transferred between different Ink objects, it can be contained by exactly one Ink object. That's why there is no explicit Stroke constructor in the Stroke class. A new Stroke is constructed through the CreateStroke() method in the Ink class. In here, the Stokes collection is actually just a collection of references to Stroke objects.

6.4 Ink Input

Another key area of difference is the way the two APIs use input. The Tablet PC uses real-time inking, but the CrossPad uses a fetch model. The Tablet PC API has packaged

real-time inking functionality into the *InkCollector* and *InkOverlay* classes. They use a Windows Forms-based window as an ink canvas to capture ink on the tablet. For example, the following two lines of code implement ink collection using any installed tablet device on the Tablet PC:

```
InkCollector * inkCollector = new InkCollector(Handle);
inkCollector -> Enabled = true;
```

Here we create a new InkCollector, and we use a windows form for the host window by passing the form's handle property in the InkCollector. We then activate the inking functionality by setting the Enabled property to true. At this point the user is free to write on the form interactively, and the handwritten ink is collected.

On the other hand, in the CrossPad API, there is no collection class and it is not necessary to take care of ink input because the CrossPad ink collection mode does this for us. The ink is recorded by the CrossPad offline, when the user writes on the tablet and simultaneously on the physical paper. The ink is later uploaded to a computer by the Ink Transfer application.

The Tablet PC API is composed of three subsets: the Tablet Input API, the Ink Data Management API and the Ink Recognition API. On the other hand, the CrossPad API only provides functionalities in Ink Data Management and Ink Recognition. For our abstraction API, we invent a friendly "Ink Player" to simulate the ink collection scenarios on the CrossPad.

6.5 Ink Properties available from the Hardware

The Tablet PC supports much richer ink properties than the CrossPad. The CrossPad digitizer captures the pen movement (x, y) coordinates of ink, and records the timestamp of each Stroke, and each Page. In addition to the (x,y) coordinates of the cursor and timestamp information, the Tablet PC digitizer hardware may provide other data such as pen pressure, tilt angle and rotation angle depending on the device. The various properties available from the digitizer are known as packet properties. These properties are represented through PacketProperty class in the Tablet PC API. The API uses

globally unique identifiers (GUIDs) to identify packet properties. Table 7, extracted from [7], shows a partial list of the packet properties supported in the Tablet PC platform and their descriptions. The proposed abstraction API must have a way to represent these properties on CrossPad even though they are not real.

Field	***Description***
X	The *x*-coordinate in the tablet coordinate space.
Y	The *y*-coordinate in the tablet coordinate space.
Z	The *z*-coordinate of the pen tip from the tablet surface.
PacketStatus	Private Wisptis data
TimerTick	The time that the packet was generated.
SerialNumber	Identifies the packet.
NormalPressure	Downward pressure of the pen tip on the tablet surface.
TangentPressure	Diagonal pressure of the pen tip on the tablet surface.
ButtonPressure	Pressure on a pressure sensitive button.
XtiltOrientation	The angle between the *y,z*-plane and the pen and *y*-axis plane.
YTiltOrientation	The angle between the *x,z*-plane and the pen and *x*-axis plane.
AzimuthOrientation	Clockwise rotation of the cursor about the *z*-axis.
AltitudeOrientation	The angle between the axis of the pen and the tablet surface.
TwistOrientation	Clockwise rotation of the cursor about its own axis.
PitchRotation	Whether the tip is above or below a horizontal line that is perpendicular to the writing surface.
RollRotation	The clockwise rotation of the pen about its own axis.
YawRotation	Whether the tip is moving left or right around the center of its horizontal axis (pen is horizontal).

Table 7. Tablet PC PacketProperty Fields and their Descriptions

6.6 Ink Rendering

The C++ version of the CrossPad SDK does not provide any graphics features. In the Tablet PC SDK, the class *Renderer* is designed to provide the ink rendering functionality. The *Renderer* class is to used to draw ink into a viewport and maintain a transformation on the ink space. It supports drawing ink to either a *Graphics* object or a Windows GDI device context (HDC) with the *Draw* method. It also provides two methods

InkSpaceToPixel and *PixelToInkSpace* to convert from ink space to pixels or vice versa, using either a Graphics object or an HDC to obtain the pixel dpi. Another ability *Renderer* provides is maintaining the transformation that is very useful to facilitate functionality such as zooming, resizing and scrolling ink.

6.7 Ink Display/Drawing Attributes

In both APIs there are classes to support various properties that define the ink's visual characteristics. In the CrossPad API, the class *InkDisplayAttribute* represents the manner in which ink is displayed, likewise in the Tablet PC API, the class *DrawingAttributes* encapsulates the formatting information that defines the style ink is rendered with.

The CrossPad's *InkDisplayAttribute* is attached to a Scribble. It defines two attributes of the ink display: color and line-thickness. The Tablet PC's *DrawingAttributes* can be associated to a Stroke or a Cursor, and specifies more settings to make ink rendered more realisticly or in more styles than the CrossPad. Table 8, quoted from [6] (page 225), lists the members of *DrawingAttributes* and their descriptions.

Property name	Type	Description
AntiAliased	Bool	Turns antialiasing on (true) and off (false).
Color	Color	The color used to draw the ink.
FitToCurve	Bool	Whether ink is rendered as a series of straight lines (false) or Bezier curves (true).
Height	Float	The height of the ink specified in ink coordinates when using the rectangle pen tip.
IgnorePressure	Bool	Whether to avoid varying the thickness of ink with pressure data (true) or not (false).
PenTip	PenTip	The style of tip used to draw ink: Ball or Rectangle.
RasterOperation	RasterOperation	The raster operation used when drawing ink. The most common value is RasterOperation.CopyPen, though highlighter ink use RasterOperation.MaskPen
Transparency	Byte	The transparency amount of the ink, where 0=opaque, and 255=invisible
Width	Float	The thickness of the ink when using the ball pen tip, or the width of the ink specified in ink coordinates when using the rectangle pen tip.

Table 8. Tablet PC DrawingAttributes members and their Descriptions

6.8 Point Value

In the CrossPad, the (x, y) coordinates of the Point are floating point values. They measure in virtual units, which happen to correspond to centimeters, with a resolution of 0.01cm. The origin (0, 0) is at the upper-left corner of the screen. In contrast, the (x, y) value of each ink packet within tablet coordinate space is measured in HIMETRIC units, which are integer values. Each HIMETRIC unit is 0.01 millimeter. The origin (0,0) of the tablet is also the upper-left corner. The float-valued points lead to CPU-intensive computation. Unifying the measurement unit is necessary in the abstraction API.

6.9 Event Handling

The CrossPad and the Tablet PC API employ different event models. The former is based on Java Delegation Event Model, while the later is based on C# Event Model.

Java Delegation Event Model – CrossPad Event Handling

The Java Delegation Event Model is based on four concepts of *Event Source*, *Event Listeners*, *Event Listener Interface* and *Event Message*. An event source generates an event and sends it to all the registered event listeners. The event source object notifies an event listener object by invoking a method on it and passing it an event message. All event listeners for a particular type of event must implement a corresponding event listener interface.

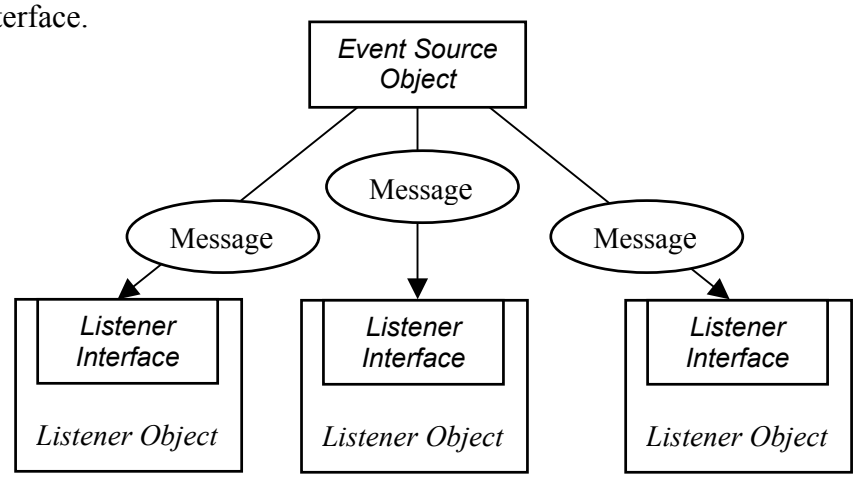

Figure 6. Java Delegation Event Model

In the CrossPad API, the classes *Scribble*, *Page* and *PageSet* are concrete event source classes derived from abstract base class *Talker*. They a provide registration method *addListener*, a de-registration method *removeListener* to add or remove corresponding listeners. It implements notification methods as well. For instance, in the *Page* class, the method *notifyScribbleAttributeChanged* notifies all attached *PageListeners* that the Attributes of a Scribble on the Page changed. The abstract base class *Listener* provides a common base class for all Listeners. The three pre-defined abstract Listeners *ScribbleListener*, *PageListener* and *PageSetListener* provide an event listener interface, and define a set of "update" methods. A concrete event listener object must implement the interface. Listeners are stored in a ListenerSet maintained by the corresponding event source classes. For example, users will subclass *PageListener* and implement various update methods such as *updateAttributeChanged, updateScribbleAdded, updateScribbleDeleted*. The appropriate methods of the class *Page* are defined to call the appropriate update-methods of all attached *PageListeners*, so that, the particular task will be invoked when an Attribute of its Page is changed, when a Scribble on its Page is added or deleted.

The C# Event Model – Tablet PC Event Handling

The C# event model is similar to Java Delegation-Event model. It still has the event source, event consumer (event listener) and the event object (event message). However, unlike the Java Delegation-Event model, the C# event model uses a special type of "delegate".

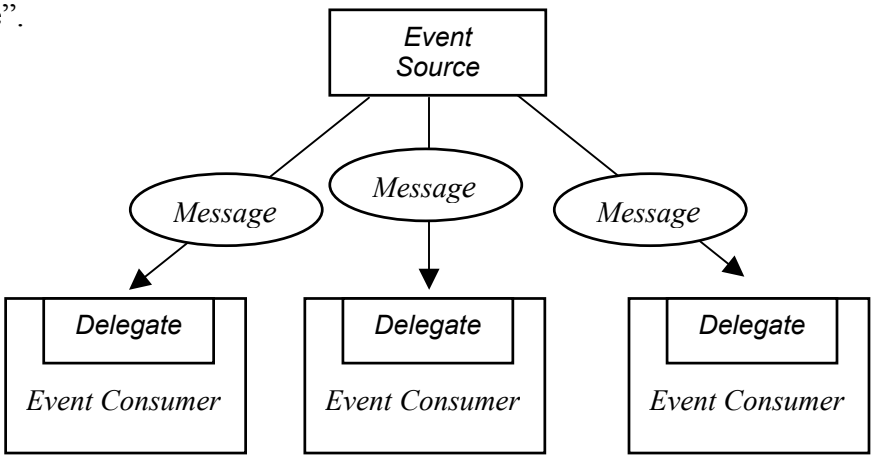

Figure 7. C# Event Model

The “Delegate” is a new concept in the .NET framework. It provides the first class support events as class members. Delegates can be thought of as a special type – something like a class. It is a class type derived from *System.Delegate* in the .NET Framework. Its main job is to encapsulate one or more methods. When you invoke a delegate instance, the methods it encapsulates are also invoked. Therefore, a delegate allows one to pass methods of one class to objects of other classes that can call those methods. Delegates are similar to C++ function pointers. However, unlike function pointers, delegates are object-oriented.

As mentioned just now, the C# event model is based on the concepts of event source, event consumer and event object. The event source is the object that potentially causes an event to happen. It provides a way for interested event consumers to register, and keeps a list of registered event consumers so that when the event occurs, the registered consumers in the list are notified. The event consumer is the object interested in listening to a particular event. An event consumer contains a special method called the event handler. This method takes the event object as parameter. When an event occurs in the event source, a new event object is created. This event object is then passed over to the event consumer’s event handler method as parameter.

The following is an example of how the Tablet PC handles events. *CursorDown* is one of events in InkCollector class. The event is fired when the cursor tip has touched the surface of the digitizer. The API is:

```
public delegate void InkCollectorCursorDownEventHandler(
                     object sender,
                     InkCollectorCursorDownEventArgs e);
public event InkCollectorCursorDownEventHandler CursorDown;
```

In this case, the *InkCollector* object is an event source. It maintains a list of registered event consumers. The delegate *InkCollectorCursorDownEventHandler* is an event consumer registered in *InkCollector*, listening to the cursor down event. To add this event consumer to the event source InkCollector, one uses the += operator:

```
ic.CursorDown +=                  // ic is an InkCollector object
 new InkCollectorCursorDownEventHandler(inkCollector_CursorDown);
```

inkCollector_CursorDown is a user-defined function encapsulated in the delegate which is called when the event is fired. It has the same signature as the delegate, and takes an sender object and an event object as parameters:

```
void inkCollector_CursorDown (object sender,
                              InkCollectorCursorDownEventArgs e){
   // user-defined to deal with the event
}
```

InkCollectorCursorDownEventArgs is an event object, which contains the event data. It is passed from event source over to the event consumer's event handler method as a parameter.

In summary, the difference between two event models stems from the concept of *delegate*. To implement an event is a two-step process with both APIs. With the CrossPad API we must: (1) create a concrete listener inheriting from listener interface, and implement the behavior of its update methods; (2) attach listener to the event source object by invoking the method addListener. With the Tablet PC API we must: (1) attach a defined event to the event source object using the += operator; (2) define the function, encapsulated in the delegate to be called when the event fired (for example *inkCollector_CursorDown* in the above analysis).

6.10 Ink Persistence and Interoperability

Ink persistence and interoperability are important features for an application that uses ink. The Tablet PC accomplishes ink persistence and interoperability by allowing users to save/load ink data with full fidelity through the Ink class' *Save* and *Load* methods, and move to and from other Microsoft windows-based application using the clipboard though Ink class' *ClipboardCopy* and *ClipboardPaste* method.

The *Save* method produces a byte array in one of the four formats that we have discussed in section 3.2.4.4 (See Table 3): Base64Gif, Base64InkSerializedFormat, Gif and InkSerializedFormat. With the byte array, ink can be further written to files, exported to .gif image, or stored in RTF, HTML or XML-based formats. Reconstitution of ink is

done with the *Load* method, which takes the byte array previously returned by the Save method. The *ClipboardCopy* method can cut or copy ink data from an *Ink* object to the clipboard in many different formats. The *ClipboardPaste* method will read the supported data formats from the clipboard and merge it into an *Ink* object.

The CrossPad also provides facilities allowing the user to save and load ink data, but the capability of ink interoperability with other applications is not available. As we have seen in section 3.1.1, the device format (*.pad), notebook format (*.nbk) and ink format (*.ink) are three relevant file formats for the CrossPad. Both *.pad and *.nbk files are produced by InkTransfer upload application. With CrossPad API, the ink read and write is accomplished by *Reader* and *Writer* classes. The *Reader* class can read all three formats mentioned above, while the ink files written by *Writer* class is in *.ink format. In addition, the CrossPad provides ink-data export classes allowing the user to export ink to images in BMP, JPEG, PDF, PNG, PostScript and TIFF formats.

6.11 Handwriting Recognition

Tablet PC ink recognition comprises gesture recognition and handwriting recognition. Two usage models are supplied to perform the recognition: synchronous mode and asynchronous mode. Synchronous recognition occurs when the thread requesting recognition results blocks until computation is complete. The method *Recognize* performs recognition synchronously. For asynchronous recognition, the thread requesting a recognition result is allowed to continue, and is later notified that computation is complete. The methods *BackgroundRecognize* and *BackgroundRecognizeWithAlternates* perform recognition asynchronously. Another important concept of the Tablet ink recognition is partial recognition. This refers to an incremental recognition – the recognition begins as soon as any ink is given, and incrementally adjusts the computation as ink added or removed, or recognition properties are changed. Partial recognition improves the recognition time performance, since the strokes associated are kept up-to-date at all times, and computation proceeds.

The CrossPad recognition API is relatively simpler. It does not distinguish the synchronous, asynchronous, or partial concepts. The *Recognition* class provides one *recognize* method to translate scribbles to characters all at once. Since recognition is off-line, the question of synchronicity is not relevant.

6.12 Some Advanced Functionalities of the Tablet PC not on the CrossPad

This section identifies some functionalities supplied by the Tablet PC, but not available in the CrossPad, beyond what we have seen in previous sections.

Bezier Curve Fitting

Curve fitting is the process of taking some points and figuring out a smooth curve that passes near all the points [6]. The Bezier curve was developed in 1970's for CAD/CAM. The algorithm is able to detect inflection points, or cusps. In the Tablet PC API, the Stroke class' *BezierPoints* property provides the control points of the Bezier curve. The method *GetFlattenedBezierPoints* computes the actual (x, y) points that approximate the Bezier curve. Unfortunately, the CrossPad doesn't provide any functionality to calculate Bezier Curve, Bezier Curve fitting is one of the most significant improvements to digital ink that Tablet PC provides. To implement the curve fitting with CrossPad API is non-trivial.

Cusp

A cusp in ink data is defined as a point at which the direction of the ink changes in a discontinuous fashion [6]. Cusps are useful for logically dividing a stroke into segments, and aid in performing gesture/handwriting recognition or stroke segment erasing. The Tablet PC can compute two kinds of cusps: polyline cusps and Bezier cusps. The Stroke class' *BezierCusps* and *PolylineCusps* properties return an array of integer point indexes at which a cusp was determined. However, the CrossPad doesn't provide any way to compute cusps. Cusp implementation is also non-trivial.

Intersections

Computing the intersections of ink strokes can be useful for performing recognition. The Tablet PC provides three kinds of intersections: self-intersection (a stroke crosses itself); stroke intersection (a stroke crosses another stroke); and rectangle intersection (a stroke crosses the bounds of a rectangle). The Stroke class' *SelfIntersections* property, and methods *FindIntersection* and *GetRectangleIntersections* compute these three intersections respectively. The returned intersection points are floating-point indexed. A floating-point index is a value that defines an arbitrary position along the length of an ink stroke. For example, index 2.2 means that the point is 20 percent of the way along the line segment between point at index 2 and 3. However, the CrossPad does not provide the functionality to compute intersections.

6.13 Some Advanced Functionalities of the CrossPad not on the Tablet PC

This section identifies some functions supplied by the CrossPad, but not available on the Tablet PC, beyond what we have seen in previous sections.

Form Processing

As described in section 3.1.2.4, the CrossPad provides APIs for forms processing. It defines many different kinds of fields of form, provides the ODBC (Open DataBase Connectivity) database interface for forms, as well as methods to perform stream I/O of the specifications of the fields of forms. This enables applications to collect form field data and permits the data to be automatically entered in a database. The Tablet PC API does not specifically support form processing.

Walker Pattern

As we have seen in section 3.1.2.1, the CrossPad API provides three abstract base classes *ScribbleWalker*, *ScribbleSetWalker* and *PageSetWalker*, each of which is a walker interface. It allows flexible definition of new operations by users without modifying the interface of the ink data classes. In contrast, the walker pattern is not provided by the Tablet PC API.

CHAPTER 7 IMPLEMENTATION OF A COMMON ABSTRCT API FOR THE TABLET PC AND CROSSPAD

We have completed a partial implementation of an abstract API for the Tablet PC and CrossPad. In this chapter, we describe the principle design and issues in the implementation.

7.1 Primary Design

The task of creating a common abstract API for the Tablet PC and CrossPad is essentially to wrap Tablet PC objects and CrossPad objects on their own platform, extract the common part, extend the uncommon part, and provide them with the same API. The architecture of the abstract API is shown in Figure 8.

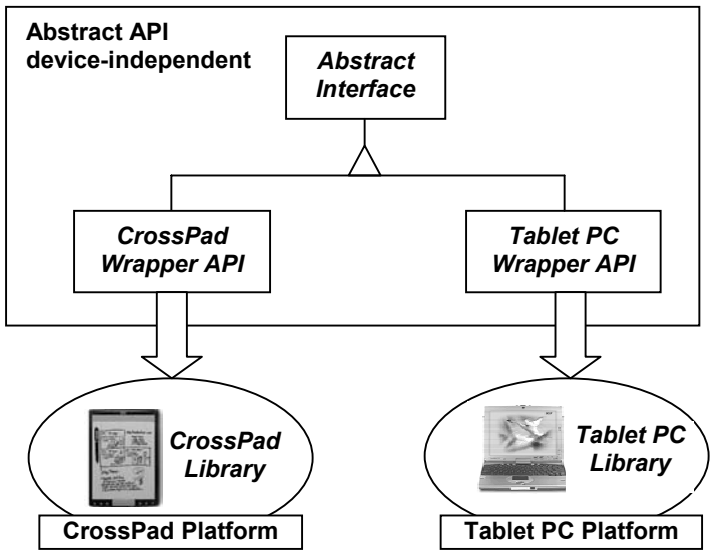

Figure 8. Architecture of Abstraction API upon CrossPad and Tablet PC

The abstract interface consists of abstract base classes defining pure virtual functions that are implemented in their derived classes: the CrossPad wrapper classes and Tablet PC wrapper classes. The wrappers are classes that contain a pointer or a reference to a real object, and must implement all functions provided by the abstract interface. For example, both the CrossPad and Tablet PC have a *Stroke* class. The abstraction API for stroke defines the interface *IgStroke* class. The derived classes are

CrossPad::*gStroke* and TabletPC::*gStroke* (see Figure 9). The private member of the CrossPad::*gStroke* class is a pointer to a CrossPad::*Stroke* object, while the private member of the class TabletPC::*gStroke* is a pointer to a TabletPC::*Stroke* object. In this way, the abstract API can provide the common functionalities available in both devices, as well as extend as many functionalities as possible that are available in one device but not another.

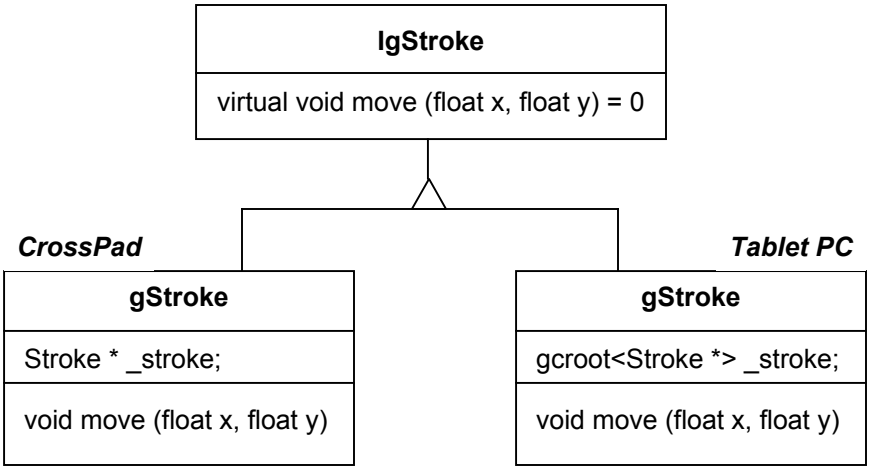

Figure 9. Example of abstract interface and derived classes

7.2 Abstraction Ink Classes

The abstraction API uses the CrossPad document model to represent ink. The key classes are *gPoint*, *gStroke*, *gStrokes*, *gInkPage* and *gInkPages* to represent a point, a stroke, a collection of strokes, a page of ink and pages of ink respectively. The class name starts with small letter "g" meaning generic. These ink classes are wrapper classes actually wrapping the corresponding object of the CrossPad or Tablet PC (see Table 9). *ElectricInk* is the namespace to access ink library on CrossPad, while *Microsoft::Ink* is the namespace to access the ink library on Tablet PC.

Abstraction API	Wrapped Object	
	CrossPad	Tablet PC
gPoint	*ElectricInk::Point*	*System::Drawing::PointF*
gStroke	*ElectricInk::Stroke*	*Microsoft::Ink::Stroke*
gStrokes	*ElectricInk::Scribble*	*Microsoft::Ink::Strokes*
gInkPage	*ElectricInk::Page*	*Microsoft::Ink::Ink*
gInkPages	*ElectricInk::PageSet*	*Microsoft::Ink::Ink*

Table 9. Ink Classes in Abstract API

7.3 Managed and Unmanaged C++

As discussed in section 6.1, the abstraction API is implemented in C++. The abstract interface, and derived classes on the CrossPad are implemented in native C++. Since we are using the managed library of the Tablet PC, the implementation goes differently on the Tablet PC side. The fact that a managed class cannot inherit from an unmanaged class determines that the classes derived from the abstract interface on Tablet PC must be unmanaged. We want the unmanaged wrapper class to contain a pointer to the managed Tablet PC object. Unfortunately, it is illegal for an unmanaged class to declare a member to have a managed pointer type. In order to point to a managed object from the C++ heap, the header file vcclr.h provides the type-safe template gcroot. Use of this template allows the programmer to embed a virtual __gc pointer in an unmanaged class and treat it as if it were the underlying type. Therefore, all the wrapper classes on Tablet PC will have a virtual __gc pointer to the corresponding managed object as its data member. For example, the private data member declaration of *gStroke* class on Tablet PC will be gcroot<Stroke *> _stroke.

CHAPTER 8 CONCLUSIONS

The objective of our research has been to achieve ink interoperability and application compatibility among heterogeneous devices and ink formats. This thesis has studied both digital ink formats and pen computing APIs. We give here some conclusions on these two aspects of our work and discuss some possibilities for future investigation.

8.1 Digital Ink Format Conversion

Three notable digital ink data formats: UNIPEN, Jot and InkML have been studied and compared. In general, InkML is going to be the most comprehensive specification for capturing, transmitting, processing and presenting digital ink. It incorporates the features of UNIPEN and Jot, but is an improvement over them. Jot is a proprietary format, which severely limited the opportunities for using ink as a communications medium. InkML is an open XML-based format that allows exchange of digital ink across heterogeneous devices developed by multiple vendors and web-based applications. Jot is a binary format, while InkML supports binary mode as an optional layer. This appeals to application developers who object to a binary encoding of ink. Jot does not support any abstract characterization of ink, in contrast to InkML with application-specific schemas. InkML is an improvement over UNIPEN because it replaces UNIPEN's flat attribute organization and record-like structure by supporting a more sophisticated labeling scheme and by leveraging other standards.

We wanted to realize the conversions among UNIPEN, Jot and InkML. The conversions could be valuable for sharing ink between applications, especially the conversions between UNIPEN and InkML and between Jot and InkML. The conversion between UNIPEN and Jot will lose a large set of information because of their very different design goals and format definitions. We have implemented the conversion between UNIPEN and Jot. As to the other conversions, because the InkML is still not a finalized standard, we have to leave the conversions between it and UNIPEN and Jot for future work. In this thesis, we have pointed out the issues of conversions between InkML and UNIPEN/Jot, which should be useful for this future work.

8.2 API Interoperability

Our goal was to develop an abstraction API for the IBM CrossPad and Microsoft Tablet PC to achieve application compatibility between these two devices. We have studied these two APIs and identified the incompatibilities between them. We have also made a partial implementation of the abstraction API. Generally speaking, the Tablet PC API provides more comprehensive and powerful functionalities. On the CrossPad, to provide Tablet PC functionalities sometimes is not trivial. For example, the implementation of Bezier curve on the CrossPad is difficult. After having identified the incompatibilities between the two APIs, the principle design of the abstract API has been completed. The basic abstract ink classes have been designed and partially implemented. Future work will focus on the identified incompatibilities between the two APIs.

There remains considerable work to do in the area of portability of ink data and ink handwriting programs. This thesis has made some progress in this area by studying format conversions and API abstraction.

References

1. Isabelle Guyon. *UNIPEN 1.0 Format Definition.* AT&T Bell Laboratories, 1994.

2. Isabelle Guyon, Lambert Schomaker, Réjean Plamondon, Mark Liberman, and Stan Janet. *UNIPEN project of on-line data exchange and recognizer benchmarks.*

3. Slate Corporation. *JOT – A Specification for an Ink Storage and Interchange Format.* 1993.

4. *Ink Markup Language, W3C working Draft 28 September 2004.* W3C®, 2004. http://www.w3.org/TR/2004/WD-InkML-20040928/ .

5. IBM. *IBM C++ Ink Manager Pro SDK 1.0, ApplicationWriter's Guide.* 2001.

6. Rob Jarrett and Philip Su. *Building Tablet PC Applications.* ISBN: 0-7356-1723-6, Microsoft Press, 2002.

7. MSDN Library. *Windows XP Tablet PC Edition*. http://www.msdn.microsoft.com/library/default.asp

8. Heng Ngee Mok. *From Java to C#: A Developer's Guide.* ISBN: 0-321-13622-5, Addison-Wesley, 2003.

9. Microsoft. *Managed Extensions for C++ Specification.* http://www.gotdotnet.com/team/cplusplus/articles/mcspec.doc

10. UNIPEN project web page. http://hwr.nici.kun.nl/unipen/

Appendix1 A Typical UNIPEN File and Upview Visualization

```
.VERSION 1.0
.COMMENT File s001n19
.COMMENT See "A Comparison of Approaches to On-line
          Handwritten Character Recognition"
         (Rob Kassel, MIT PhD Thesis, 1995)
         for complete specifications and
         benchmarks on this data.

.DATA_SOURCE MIT_LCS_SLS
.DATA_ID MIT_Natural_Handprint_95
.DATA_CONTACT
 Name: Rob Kassel
 Email: rob@goldilocks.lcs.mit.edu
 Phone: 1-617-253-3049
 Fax: 1-617-258-8642
 Address: Spoken Language Systems Group
       Laboratory for Computer Science
       Massachusetts Institute of Technology
       Room NE43-601
       Cambridge, MA  02139, USA

.SETUP
 Site: MIT
 Writer motivation: Paid
 Writer physical position: Seated at desk
 Instructions given to the writer: Minimal; printing required;
  capitalize initial letter; no correcting; writing to be
  examined later.
 Prompting: Aural prompts, both string and spelled
 Recognizer feedback: no
 Form layout: large writing area, no guides, minimal left/right bias

.DATA_INFO
 Alphabet: English alphanumerics plus symbols to indicate
       connections, ligatures, embellishments, and pen skips
.PAD
 Machine name: Wacom 648A
 Brand: Wacom
 Type: 648A
 Serial Nr.: 160039
 Sensor: Electromagnetic, wireless pen
 Pen: Untethered, tip switch only
```

Driver: Mircosoft Windows for Pen Computing V1.0
Sampling mode: Using Microsoft Visual Basic 2.0 controls
Sampling rate: 193 Hz
Resolution: 0.001 inches/unit
Accuracy: 0.01 inches
Display: Backlit LCD screen, 640x480
Inking: 1 pixel wide black on white
.X_DIM 4975
.Y_DIM 3058
.X_POINTS_PER_INCH 100
.Y_POINTS_PER_INCH 100

.ALPHABET "A" "B" "C" "D" "E" "F" "G"
"H" "I" "J" "K" "L" "M" "N"
"O" "P" "Q" "R" "S" "T" "U"
"V" "W" "X" "Y" "Z"
"a" "b" "c" "d" "e" "f" "g"
"h" "i" "j" "k" "l" "m" "n"
"o" "p" "q" "r" "s" "t" "u"
"v" "w" "x" "y" "z"
"0" "1" "2" "3" "4"
"5" "6" "7" "8" "9"
"!" "&" "*" "+"
.WRITER_ID 1
.STYLE PRINTED
.HAND R
.AGE 29
.SEX M
.WRITER_INFO
Group: Training
Weight: 190
Student: Yes
Where educated: OH
Home language: English
Name: Jim

.COORD X Y T
.HIERARCHY WORD LETTER
.SEGMENT WORD 0-5 ? "10342"
.SEGMENT LETTER 0 ? "1"
.SEGMENT LETTER 1 ? "0"
.SEGMENT LETTER 2 ? "3"
.SEGMENT LETTER 3-4 ? "4"
.SEGMENT LETTER 5 ? "2"

.COMMENT Prompt string: "10342"

.COMMENT Recognizer string: "10342"
.COMMENT Transcriber Comment: ""
.PEN_DOWN
1319 718 0
1298 739 5
1298 698 10
1308 635 16
1319 583 21
1329 520 26
1371 448 31
1392 395 36
1392 343 42
1402 302 47
1433 260 52
.PEN_UP

.PEN_DOWN
1663 677 275
1652 635 280
1631 541 285
1610 458 291
1631 375 296
1704 354 301
1829 364 306
1944 427 311
2017 500 317
2017 541 322
1944 593 327
1798 645 332
1673 677 338
1663 677 343
1683 666 348
.PEN_UP

.PEN_DOWN
2142 645 659
2121 666 664
2121 687 669
2183 729 675
2319 750 680
2423 729 685
2454 698 690
2454 656 695
2381 614 701
2340 583 706
2340 583 711
2340 583 716
2392 562 722
2444 520 727
2475 468 732
2485 427 737
2465 375 742
2413 323 748
2350 302 753
2288 270 758
.PEN_UP

.PEN_DOWN
2683 698 1209
2673 698 1214
2673 687 1219
2673 645 1225
2683 573 1230
2715 520 1235
2767 479 1240
2860 448 1245
2944 448 1251
3017 448 1256
3079 458 1261
.PEN_UP

.PEN_DOWN
3017 698 1428
2996 604 1433
2985 500 1438
2965 385 1444
2965 302 1449
2996 229 1454
3027 218 1459
.PEN_UP

.PEN_DOWN
3183 708 1648
3183 718 1653
3194 729 1658
3246 750 1664
3371 750 1669
3444 718 1674
3496 677 1679
3496 635 1684

3444 562 1690
3402 500 1695
3371 458 1700
3360 416 1705
3392 364 1710
3433 343 1716
3496 323 1721

3558 323 1726
3610 323 1731
3642 323 1737
.PEN_UP

.COMMENT End of File

UniPen Viewer upview 4.02

VITA

Name: Xiaojie Wu

Post-secondary Education and Degrees:

Shanghai JiaoTong University
Shanghai, China
1991 ~ 1995 B.Eng

University of Western Ontario
London, Ontario, Canada
2000 ~ 2001 B.Sc

University of Western Ontario
London, Ontario, Canada
2002 ~ 2004 M.Sc

Honors and Awards:

Dean Honor List, 2000 ~ 2001

Faculty Association Scholarship, 2001

Ontario Graduate Scholarship, 2002 ~ 2003

Related Work Experience:

Teaching Assistant, 2002 ~ 2003
Computer Science Department
University of Western Ontario
London, Ontario, Canada

Software Developer, 2003 ~ present
Liberate Technologies
London, Ontario, Canada