CORE
Metadata, citation and similar papers at core.ac.uk
Provided by Bournemouth University Research Online

PenDraw - A Language for Improving Take-Up of SVG

Keywords: 2D, graphics, programming, language, productivity

Dr Andrew Main
Managing Director
PenDraw Ltd
42 Fitzharris Avenue
Bournemouth
BH9 1BZ
United Kingdom
andrew.main@pendraw.co.uk

Biography

Dr Andrew Main is Managing Director of PenDraw Ltd, and Head of Computing at Bournemouth University (UK). He has worked in computer graphics since 1968. His exploration of graphics fundamentals came to fruition in the 1980s, when he ran his own software business. He built an interpreter for PenDraw, but found that the market was fixated on interactive graphics and ignored all else. The emergence of server-side operation as a key Internet architecture plus SVG as a universal graphics output and communication medium were the keys PenDraw needed. He restored and modernised the PenDraw project in 2002, releasing it as a free compiler and run-time system in 2005. The PenDraw web site is at www.pendraw.co.uk.
[pic]

Abstract

[pic]

The case is presented for preferring a dedicated Computer Graphics (CG) language to the traditional conventional language plus add-on. PenDraw is presented as an existing language that overcomes many problems of CG add-ons, providing compile-time checking and reduced need for run-time debugging. PenDraw produces well-formed SVG.

Evidence is given that PenDraw decreases development costs.

Evidence is presented that PenDraw has brought CG programming to a wider ability range of potential users than professional programmers.

It is argued that, given its qualities, PenDraw should be able to improve the take-up of CG programming.

Such take-up is expected to be slow at first, given the market-place focus on interactivity and 3D. But it is believed that PenDraw's expressive power and cost benefits should lead to growth in its use, and in use of SVG.

[pic]

Table of Contents

[pic]

1. 2D Computer Graphics Programming Problems

The paper presents PenDraw, a dedicated Computer Graphics (CG) language approach to generation of SVG, which is free from the problems of the traditional 'conventional language plus CG add on' approach and adds further benefits of its own. The paper provides evidence, from real cases of PenDraw commercial use, that PenDraw may be able to increase the take-up rate of graphics programming and therefore SVG based graphics. Server-side SVG 2-D graphics has not taken off as swiftly as it might. There are SVG add-ons to all popular web languages such as PHP, Perl, Python, and Java. Yet the low up-take problem persists despite the benefits of web

integration and plentiful language add-on supply. This paper argues that there are problems that hinder to the take up of graphics programming when it is based on the traditional 'language add-on' approach, and that PenDraw is an effective solution.

1.1 Graphics Programming & Low Productivity

The great majority of CG programming facilities require the programmer to work in matrix algebra. That inevitably deters some programmers, because they either do not have sufficient confidence in their mathematical ability, or they just do not have the ability. In addition, there is very limited compile-time checking either of types or of CG program structure, which makes testing and debugging inefficient and tedious - even tiresome.

In development projects, that may lead to cost-benefit based decisions to avoid CG and seek alternative solutions. System design choice may be driven by such things as the ratio of the price of presenting data in an SVG bar chart, to that of using an HTML table. The latter form an easy option, after all. It appears that the ratio is too high for many programmers, given the lack of diagrammatic graphics in use generally and on the Web in particular.

1.1.1 Matrix & Language Add-On

The transformation matrix is a CG 'given'. But instead of seeking to hide it beneath some level of abstraction, add-ons make the matrix a central progamming matter, as witness GPGL **[KULSRUD1]** in the 1960s, GKS **[ISO1]** in 1981, and OpenGL **[SGI1]** today in 2005.

Another 'given' is the language add-on, instead of a wholly graphics language (same references). The pro-add-on argument was (is) "Why invent a new language? Isn't it better for programmers to keep their favourite language and add graphics to it?". That argument has merit on the face of it. Yet if it were correct, one would have to ask why new programming languages are still developed and still flourish. Perl displaced C for much CGI work on the Web, for example, and PHP flourishes too. It appears that there is always room for new languages, especially if they have singular capabilities.

1.1.2 Problems of the Matrix

CG add-ons provide direct access to the transformation matrix for execution efficiency and tight programming control. But doing so ties the facility to a particular level of abstraction. **Example 1** shows typical matrix based code (drawing a shelf with two supporting brackets). The code is very matrix-centric. Its emphasis is less on graphics, more on matrix management.

```
|' Global Constants: shelf is sW wide and sT thick          |
|' Global Constants: brackets are W wide, H deep and T thick|
|                                                           |
|' Global Constants: shelf is positioned at (shelfX, shelfY)|
|                                                           |
|                                                           |
|Private Sub Bracket ( brackX, brackY )                     |
|glPushMatrix                                               |
|glLoadIdentity()                                           |
|glTranslatef( brackX, brackY, 0.0)                         |
|glBegin(GL_LINE_LOOP)     //draw bracket as a T            |
|glVertex2f(       0,   0 )                                 |
|glVertex2f(       W,   0 )                                 |
|glVertex2f(       W,  -T )                                 |
|glVertex2f( W/2+T/2,  -T )                                 |
|glVertex2f( W/2+T/2,-H-T )                                 |
```

```
|glVertex2f( W/2-T/2,-H-T )                                    |
|glVertex2f( W/2-T/2,  -T )                                    |
|glVertex2f(       0,  -T )                                    |
|glEnd                                                         |
|glPopMatrix                                                   |
|End Sub                                                       |
|                                                              |
|Private Sub Shelf ( )                                         |
|glBegin(GL_LINE_LOOP)     //draw shelf as rectangle           |
|glVertex2f(  0,  0 )                                          |
|glVertex2f( sW,  0 )                                          |
|glVertex2f( sW, sT )                                          |
|glVertex2f(  0, sT )                                          |
|glEnd                                                         |
|Bracket(    0, 0 )                                            |
|Bracket( sW-W, 0 )                                            |
|End Sub                                                       |
|                                                              |
|Private Sub Wall ( )                                          |
|glPushMatrix                                                  |
|glLoadIdentity()                                              |
|glTranslatef( shelfX, shelfY, 0.0)                            |
|Shelf( )                                                      |
|glPopMatrix                                                   |
|End Sub                                                       |
```

Example 1: Language Add-on Structure

1.1.3 Problems of the Add-On

A program is an expression of the way to do something; and language design can facilitate that expression. But as **Example 1** shows, add-on routine calls are not naturally expressive of what the programmer wants to do. Matrix push and pop calls get in the way, and reduce the clarity of the rest of the graphics. The CG add-on does not invite the programmer to think and work **naturally** in **graphical** terms of position, rotation and scale, but to think in **mathematical** terms of the state of the matrix.

Contrast the add-on with its host language. The latter will almost certainly have implicit stack operations that the programmer leaves to the compiler. Not so the add-on: every stack raise and lower operation must be explicitly programmed. **Example 1** shows how conscious of the stack the programmer must be. Programmers today are used to compilers doing all that sort of thing for them, and in that sense programming CG can be deterringly primitive. The program expresses how to handle a matrix, when what the programmer wants is to express how to draw graphics.

In addition, the amount of matrix stack code is high for small effects. The programmer's simple thought "I'll put the shelf there" is expressed as four lines of stack-organising code, with only one line for invoking the shelf. The stack-organising code does not express the "put ... there" thought at all clearly.

At the practical level,**graphics stack push and pop form the graphics program structure**, but a structure in which the compiler can not detect any errors, even though one more/less push than pop is a disastrous error. The **risk** of push/pop imbalance **increases with program complexity**, of course, for pushes and pops may end up within if-then-else or other language structures. Debugging takes place at run-time. Testing and debugging become protracted. Development cost

increases.

1.2 Lessons from Basic

PenDraw language design drew on lessons derived from the Basic programming language. Kurtz & Kemeny invented Dartmouth Basic **[BASIC1]** to be accessible, in contrast to CG add-on design which has tended to focus on powerful features and processor efficiency. **Table 1** presents a comparison of the two. It effectively summarises the obstacles confronting those who try traditional CG programming.

No.	BASIC	CG ADD-ONS
1	Designed for the novice	Expert's toolkit - almost 'experts-only'
2	No special knowledge/skill required	Matrix mathematics ability required
3	Quick to learn	Expertise must be developed
4	Few unexpected effects	Surprises exist for the unwary, such as the rotate/translate translate/rotate effect
5	High level of abstraction - hid operating system and computer hardware realities	Low level of abstraction. Programmer must know the state of the transformation matrix, and must manage its stack.
6	Some compile-time checking. This is one area where Basic could have been better	No compile-time checking of graphics semantics at all

Table 1: Contrast between Basic and CG Add-ons

1.3 Dedicated CG Language - Design Goals

Given the foregoing analysis, a 'dedicated language' approach to CG offers real benefits. That was the reason for inventing PenDraw. It had to meet these 6 graphics language design goals.

1. To be easy to learn
2. To be graphically expressive - to provide primitives which match the construct they need to handle.
3. To handle the graphics stack implicitly
4. To hide the transformation matrix beneath an abstraction
5. To be strongly typed
6. To provide good compile-time error detection

2. PenDraw

The PenDraw language is based on a clear **metaphor**. It provides a high level of abstraction. It meets the 6 graphics language design goals outlined in the last chapter. It is side-effect free, as far as any procedural language can be. It enables the programmer to combine attributes, or to prevent attribute combination. Its parameter passing mechanism automatically adjusts geometric data when it is transferred from one frame of reference to another.

2.1 Origins

The PenDraw language was designed in 1981. An editor/interpreter was implemented for Intel 8088 processor PCs by 1983. Some copies were sold and were highly rated by customers (see case studies **Chapter 3** and **Chapter 4**), but the market was fixated on interaction and CG programming had little market appeal. A further problem was the lack of any standard output language or protocol. When development was suspended in 1989, the interpreter had 31 back-ends for different devices and the list grew all the time.

The Web's programmatic operation is an enabler for PenDraw, but until Flash, webCGM and SVG, the problem of the back-end was a major block. SVG was chosen for PenDraw, and the software was completely re-designed and re-coded, leading to its (re-) launch at the end of February 2005.

2.2 Procedure Hierarchy Architecture

PenDraw is based on what Foley and Van Dam called "procedure hierarchy"**[FOLEY1]**, unlike the traditional data structure hierarchy used in most CG facilities. As Foley and Van Dam observed, the two hierarchies suit different uses "... procedure hierarchy is syntactically convenient while data structure hierarchy allows greater run-time editing flexibility." Run-time editing is not needed on the Web, where applications run to completion, and where sessions are mimicked rather than real. Accordingly, procedure hierarchy was chosen for its benefits of better graphical expressiveness and programming flexibility.

2.2.1 Not Object-Oriented

Object orientation may *de rigueur* nowadays, but it brings no benefit to PenDraw. Graphics is naturally hierarchical, and procedures provide hierarchy very well indeed. Other programs in other languages can be used for complex data processing, and they can pass data to PenDraw for graphics. It is better to keep PenDraw simple and not to try to get it to do too much.

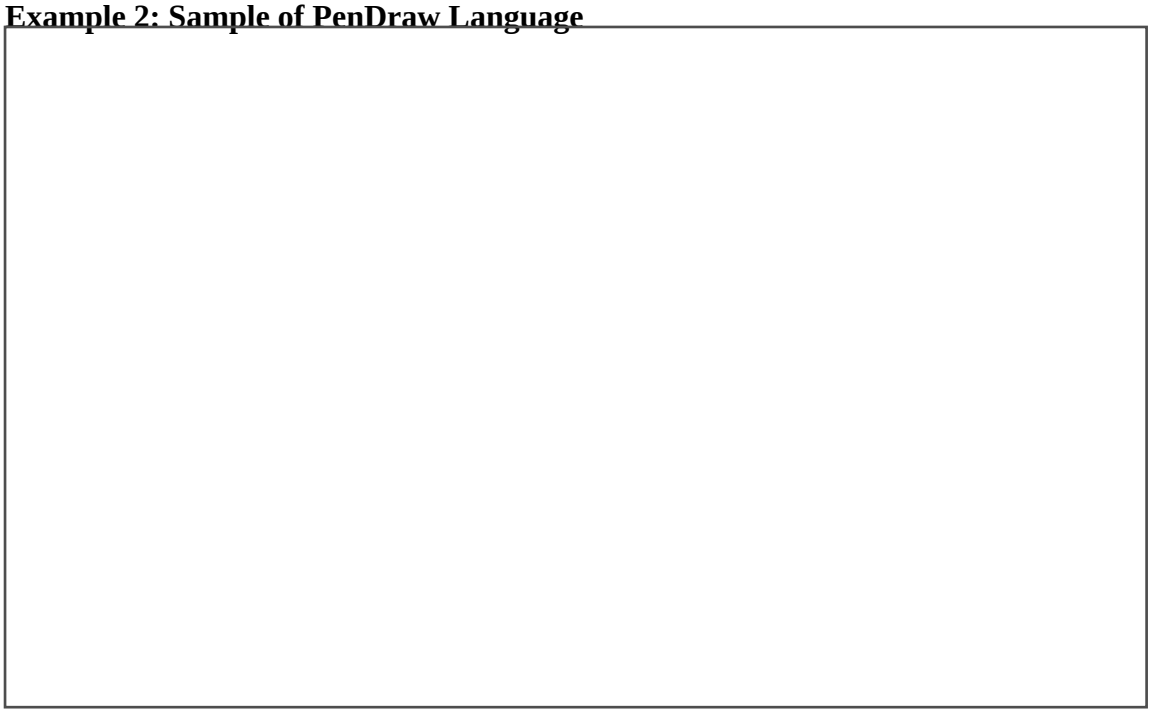
2.3 A Simple PenDraw Program

Here is the **same program** as in **Example 1**, but this one, **Example 2**, is written in PenDraw. Its output is shown in **Figure 1**. Its benefits are presented below the code.

```
|'Assume the following are declared as Global Const|
|                                                  |
|'shelf is Sw wide and sT thick                    |
|'brackets are W wide, H deep and T thick          |
|'shelf is positioned at (shelfX, shelfY)          |
|                                                  |
|Private Pic Bracket ( )                           |
|' underscore at line end denotes instruction      |
|' continues on next line                          |
|' In every Pic the Pen starts at (0,0)            |
|Pen To X W       , _                              |
|To Y -T      , _                                  |
|To X W/2+T/2 , _                                  |
|To Y -H-T    , _                                  |
|By X -T      , _                                  |
|To Y -T      , _                                  |
|To X 0       , _                                  |
|Close                ' closes loop                |
```

```
|End Pic                                                |
|                                                       |
|Private Pic Shelf ( )                                  |
|Pen To X sW, To Y sT, To X 0, Close                    |
|Draw { At (   0,0) } Bracket( )                        |
|Draw { At (sW-W,0) } Bracket( )                        |
|End Pic                                                |
|                                                       |
|Private Pic Wall ( )                                   |
|'1. shelf                                              |
|Draw { At ( shelfX, shelfY ) } Shelf( )                |
|'2. drawing border                                     |
|Pen {WidthR 0.5, Lines DimGray} _                      |
|To X 170, To Y 100, To X 0, Close                      |
|End Pic                                                |
```

Example 2: Sample of PenDraw Language



Showing the output for **Example 2** - note that fill defaults to none
Figure 1: Shelf with Two Brackets

Benefits in the above:-

1. Code is simple and direct.
2. Syntax is graphically expressive.
3. PenDraw organises the graphics stack automatically, at Pic beginning and Pic end.
4. There is no mention of matrices.
5. PenDraw is understandable: the 'At' attribute instructs PenDraw where to place a Pic, and is more intuitive than the more mathematically termed 'translate'.
6. PenDraw takes around half the number of instructions to do the same thing as the OpenGL

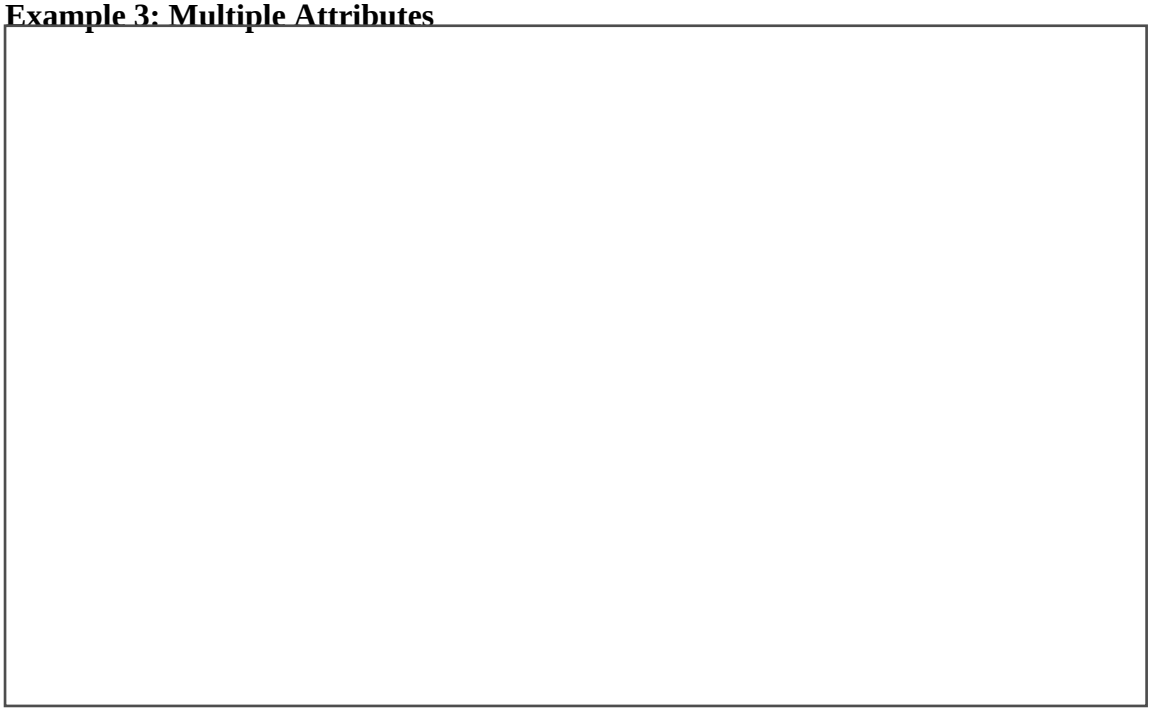
Example 1. (In general PenDraw source code is between 50% and 66% smaller than equivalent OpenGL-based source code.)

2.3.1 Multiple Attributes

The braces { } in the Draw instruction contain the attribute set for the Pic to be drawn. They can contain multiple attributes. If we wanted to fill the shelf outline, and the bracket outlines, with blue colour we would write our code as in **Example 3** and the output would be as **Figure 2**

```
| Draw { At( shelfX, shelfY ), Fill LightSkyBlue } Shelf( )|
```

Example 3: Multiple Attributes



Showing the output for **Example 3** demonstrating the use of attribute Fill
Figure 2: Shelf with Two Brackets - Filled

2.3.2 Stack Hygiene

In PenDraw the stack is completely ’hygienic’: drawing Pic B from inside Pic A has no effect on the state of Pic A within the program (though it may affect the picture). The state of Pic A **after** drawing Pic B is identical to its state **before** drawing Pic B. In **Example 4** both Shelf Pics draw exactly the same thing, even though in one of them the Pen movement is interrupted by drawing two Bracket Pics.

```
|Private Pic Shelf ( )                |
|Pen To X sW, To Y sT, To X 0, Close  |
|Draw { At (   0,0) } Bracket( )      |
|Draw { At (sW-W,0) } Bracket( )      |
|End Pic                              |
|                                     |
|Private Pic Shelf ( )                |
|Pen To X sW, To Y sT                 |
|Draw { At (   0,0) } Bracket( )      |
```

```
|Draw { At (sW-W,0) } Bracket( )      |
|Pen To X 0, Close                    |
|End Pic                              |
```

Example 4: Stack Hygiene

2.4 PenDraw Programming

PenDraw is programmed to a consistent metaphor. Programs are made of Pics and Pen moves. Pics instances can have attributes. Pic definitions can have specialties, which control attribute application. There is a fixed set of nine types, four simple and five geometric. Pics can have parameters, and geometric parameters have sophisticated behaviour.

2.4.1 Programming Metaphor

A drawing, in PenDraw, is like a piece of paper.

Each Pic can be thought of as having its own draughtsperson, who is solely concerned with his/her own Pic, and who requests the **next** draughtsperson to draw any other Pics **and** defines the full set of attributes for that new draughtsperson.

Each draughtsperson draws their own Pic (with attributes given in the Draw instruction) on its own piece of stretchy (ie scalable), transparent film, which they hand to the **previous** draughtsperson to place on **their** piece of film - unless they are the **first** draughtsperson, in which case they put it on the paper.

The Draw instruction can define all attributes; any not defined are 'inherited' unchanged from the previous draughtsperson (this is **not** OO inheritance, it is run-time inheritance from one stack level to the next). Attribute values combine, where meaningful. For example, Scale combines: suppose Pic A is drawn Scale 0.5 relative to paper, and Pic A draws Pic B at Scale 0.4. Pic B's scale relative to paper would be 0.2 (0.5 times 0.4). Attribute values that do not combine, such as 'FontFamily', are either inherited unchanged or replaced by the new value.

Attribute inheritance/combination can be **prevented**, on a Pic by Pic basis, by permanent Pic properties, termed specialties (see **Section 2.4.4**).

The drawing is composed exactly as the sequence of instructions in the PenDraw program. So, for example, if two Pics overlap, the first one will appear to lie under the second. The only exception to that is the overall Pen movement within each Pic: all of a Pic's Pen movements are collected and then performed at the end of the Pic, after all/any Pics that it instances.

There is only one primitive: a Pic's native Pen movement. The Pen always starts at (0,0) in a Pic, and is always in the state (position, colour, etc) that the draughtsperson last left it in within a Pic.

Pen movement that has attributes (eg Pen {Lines Blue}...) is not native to the Pic, and is passed to another draughtsperson as if it were a Pic. Such Pen movements can only have non-geometric attributes (line-style, line colour, fill etc).

There are built-in Pics for *Text*, *Circle*, *Ellipse*, *Rectangle*, *Rounded Rectangle*, and *Polygon*.

2.4.2 Fixed Attribute Order

Attributes are always applied **in this order**.

1. Non-geometric attributes (line-style, line colour, fill etc).
2. At [default (0,0)]
3. Turn [default 0 degrees]
4. Mirror [default not mirrored]
5. Scale [default 1.0]

Fixed ordering creates consistent graphics stack operation, and frees the programmer to set out attributes within { and } in an expressive order without suffering any surprise effects. PenDraw is not only designed to be as side-effect free as possible, but to be a 'no surprises' language.

PenDraw translates before rotating because that is what novices expect and want. The much less frequently useful rotation before translation can be achieved by raising the stack twice, through use of an additional Pic. That approach conforms to two of Kurtz and Kemeny's eight principles for Basic**[KURTZ1]** which are that "It should be easy for the beginner." and "Advanced features had to be implemented in such a way that it was the 'expert' who paid the price, not the novice."

2.4.3 No Shearing

Scale operates on a single factor. X and Y can not be differentially scaled: PenDraw does not do shearing because it distorts distances, and prevents the specification of true distances between drawing elements. Shearing is a data transformation that should take place before graphical presentation, and should not be combined with it.

Mirroring is an explicit attribute, because only a single scale factor is used. The attributes are Xmirror and Ymirror (mirroring about one or other axis).

2.4.4 Specialties

Specialties are permanent properties of Pics, whereas attributes are instance-specific (one instance may be drawn with red fill, another with green fill).

Most CG systems offer control of text mirroring, so that, for example, the words HOT and COLD appear the right way round on taps, even if the taps are mirrored (as they might be if a sink-plus-taps assembly was drawn both right and left handed). But doing it for text is not enough. By convention the HOT tap is on the left, so we need control of the whole taps entity. PenDraw specialties extend attribute control to all graphical elements, not just text, and to all attributes, not just mirroring.

Each specialty controls or prevents the application of specific attributes.

1. Default At - frame of reference returns to paper origin
2. Default Turn - x-axis becomes parallel to paper x-axis
3. Default Mirror - axes become right-handed
4. Default Scale - scale becomes that of paper
5. Default Overturn - if the thing would be drawn upside-down, it is rotated by 180 degrees: chiefly used with text.
6. Default Pen - line thickness, style etc become that of paper
7. Default Fill - fill becomes that of paper (none)

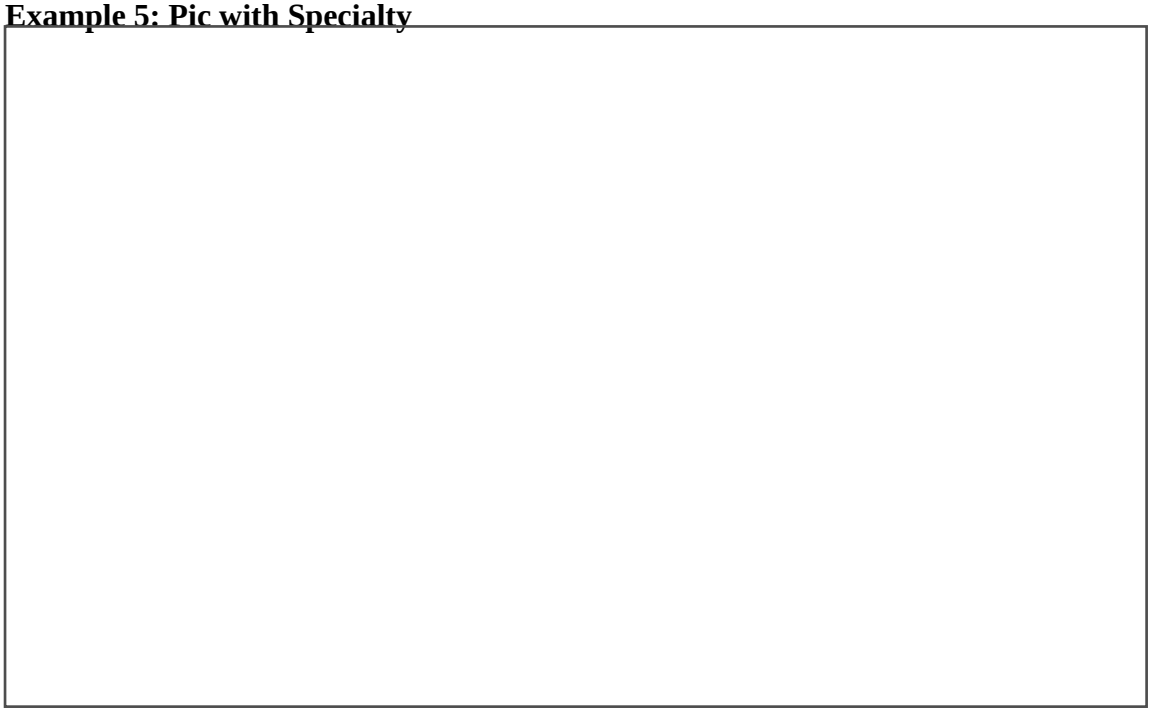
8. Default Font - font becomes that of paper

9. Default Mouse - item is not detectable

Example 5 demonstrates Default Fill. **Figure 3** shows the corresponding output.

```
|Private Pic Shelf ( )                                          |
|Default Fill          ’ --- Note this extra line               |
|Pen {Fill Orchid} To X sW, To Y sT, To X 0, Close              |
|Draw { At (   0,0) } Bracket( )                                |
|Draw { At (sW-W,0) } Bracket( )                                |
|End Pic                                                        |
|                                                               |
|Private Pic Wall ( )                                           |
|’1. shelf                                                      |
|Draw { At( shelfX, shelfY ), Fill LightSkyBlue } Shelf( )      |
|’2. drawing border                                             |
|Pen {WidthR 0.5, Lines DimGray} _                              |
|To X 170, To Y 100, To X 0, Close                              |
|End Pic                                                        |
```

Example 5: Pic with Specialty



Showing the output for **Example 5** where the specialty Default Fill has affected the application of attribute Fill

Figure 3: Shelf with Two Brackets - Default Fill

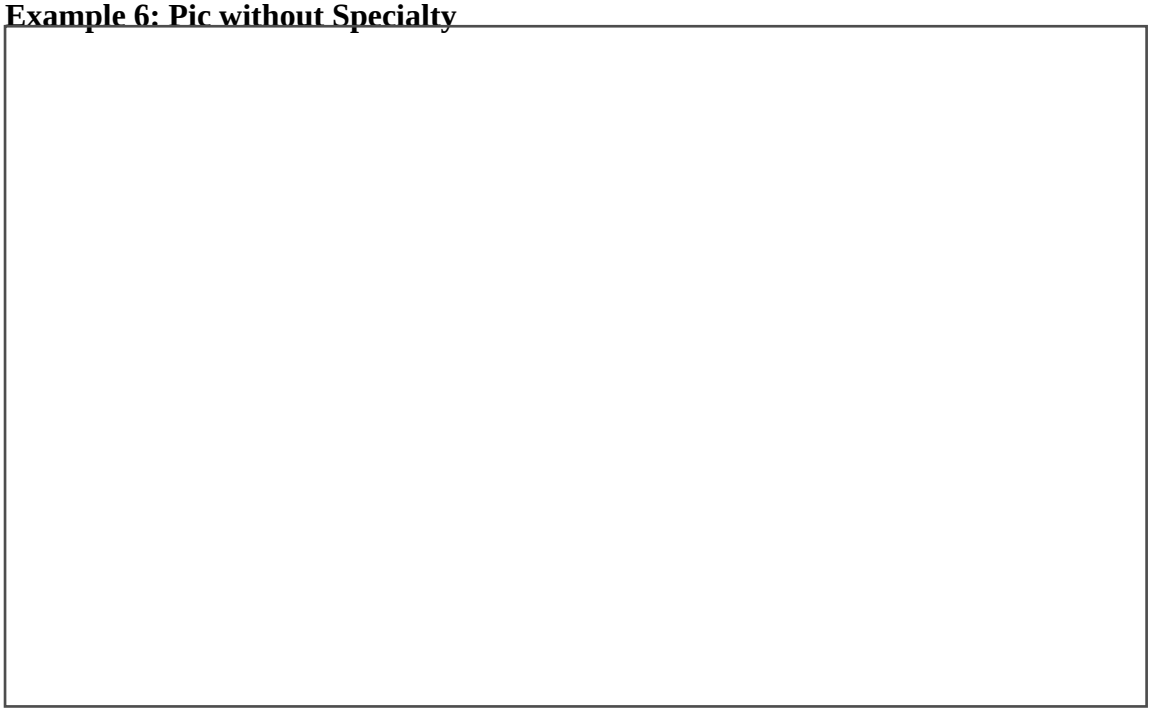
In **Example 5** above, Pic Shelf pen moves are filled Orchid, even though the Draw instruction in Wall says Fill Red. The ’draughtsperson’ uses the fill colour used on the paper (none). Then the {Fill Green} addition to the Pen move instructs that the Pen move should be filled in Green. The Brackets are not filled in colour. The Pic Shelf draughtsperson is using no fill, and that is what the Brackets draughtsperson inherits.

In **Example 6** below, however, Pic Shelf’s pen moves are filled Green, because the ’local’ fill

overrides the earlier fill, whereas the Brackets are filled in Red, because the Pic Shelf draughtsperson is using Fill Red and that is what the Brackets draughtsperson inherits. **Figure 4** shows the corresponding output.

```
|Private Pic Shelf ( )                                          |
|Pen {Fill Orchid} To X sW, To Y sT, To X 0, Close              |
|Draw { At (   0,0) } Bracket( )                                |
|Draw { At (sW-W,0) } Bracket( )                                |
|End Pic                                                        |
|                                                               |
|Private Pic Wall ( )                                           |
|'1. shelf                                                      |
|Draw { At( shelfX, shelfY ), Fill LightSkyBlue } Shelf( )      |
|'2. drawing border                                             |
|Pen {WidthR 0.5, Lines DimGray} _                              |
|To X 170, To Y 100, To X 0, Close                              |
|End Pic                                                        |
```

Example 6: Pic without Specialty



Showing the output for **Example 6** with multiple Fill attributes
Figure 4: Shelf with Two Brackets - Multiple Fills

2.5 Types

PenDraw is strongly typed, with 9 types, 4 conventional, and 5 geometric types.

Boolean, Double, Integer and String are conventional.

cCircle, cEllipse, cLine, cPoint, and cVector are geometric. The prefix c is there because they are used primarily for ***c***onstructing geometric shapes. There is a rich set of operators and functions for constructing geometry as illustrated by **Example 7**

```
|' Calculate the intersection of the two cLines L1 and L2|
```

```
|                                                          |
|                                                          |
|P1 = L1 / L2                                              |
|                                                          |
|' Calculate the distance from cPoint P1 to cLine L1       |
|                                                          |
|D  = P1 <-> L1                                            |
|                                                          |
|' Use Sub CcTangents to calculate the tangent points      |
|' on two circles, when they are joined by two straight    |
|' lines that are tangents to both circles.                |
|' C1 and C2 are of type cCircle                           |
|' TangentPoints is an array of type cPoint                |
|' iCode is an Integer that tells how many cPoints are     |
|' returned - there can be 0, 4 or 8 tangent points        |
|' TangentPoints and iCode are OUT parameters              |
|                                                          |
|CcTangents(C1, C2, TangentPoints, iCode)                  |
```

Example 7: Sample Operators/Functions

2.6 Parameter Passing

PenDraw has one more feature that we believe is unique. It makes PenDraw very different from a language add-on. The component values of geometric variables are adjusted **numerically** when they are passed in to or out of a Pic, in order to keep the variable the same **geometrically**. **Example 8** shows what happens. Pic A defines a cPoint P1 to be (20,20) and then passes it to Pic B, which is at (5,10) with respect to Pic A axes. Now if P1 coordinates were still (20,20) inside B, P1 would have moved: it would not be where A defined it to be, which would be wrong.

```
|Private Pic B (OUT P1 As cPoint)                          |
|'P1 will have the value (15,10) at entry to Pic B         |
|                                                          |
|'Now change P1                                            |
|P1 = (6,11)                                               |
|End Pic                                                   |
|                                                          |
|Private Pic A ( )                                         |
|Dim P1 As cPoint                                          |
|P1 = (20,20)                                              |
|Draw { At (5,10) } B(P1)                                  |
|'Now P1 has the value (1,1)                               |
|End Pic                                                   |
```

At entry to and exit from Pic B, the coordinate values of P1 are adjusted, **so that P1 stays in the same place**. That saves the programmer from complex and error-prone calculations when passing data between Pics that have different frames of reference. Such adjustments are automatic, and happen no matter how complex the differences between frames of reference.

Parameters are passed by value, except for OUT parameters, which are passed by reference.

Example 8: Geometric Parameters In & Out

Here, **Example 9** (output as **Figure 5**) shows **Example 1** reworked with OUT parameters, to

allow a line to be drawn between the two Bracket mid points, illustrating how geometric values can be passed around.

```
|’Assume the following are declared as Global Const      |
|’shelf is Sw wide and sT thick                           |
|’brackets are W wide, H deep and T thick                 |
|’shelf is positioned at (shelfX, shelfY)                 |
|                                                         |
|Private Pic Bracket ( Out P As cPoint )                  |
|’ underscore at line end denotes instruction             |
|’ continues on next line                                 |
|’ In every Pic the Pen starts at (0,0)                   |
|Pen To X W       , _                                     |
|To Y -T      , _                                         |
|To X W/2+T/2 , _                                         |
|To Y -H-T    , _                                         |
|By X -T      , _                                         |
|To Y -T      , _                                         |
|To X 0       , _                                         |
|Close                ’ closes loop                       |
|P = ( W/2, -H/2-T/2 )                                    |
|End Pic                                                  |
|                                                         |
|Private Pic Shelf ( Out P1 As cPoint, Out P2 As cPoint )|
|                                                         |
|Pen To X sW, To Y sT, To X 0, Close                      |
|Draw { At (   0,0) } Bracket( P1 )                       |
|Draw { At (sW-W,0) } Bracket( P2 )                       |
|End Pic                                                  |
|                                                         |
|Private Pic Wall ( )                                     |
|Dim P1 As cPoint                                         |
|Dim P2 As cPoint                                         |
|Draw { At ( shelfX, shelfY ) } Shelf( P1, P2 )           |
|Pen Up, To P1, To P2 ’(Pen drops after: To P1)           |
|End Pic                                                  |
```

Example 9: Sample of PenDraw Language

Showing the output for **Example 9** showing how OUT parameters pass the position of the Bracket mid-points to Pic Wall, where they are used to draw between them.

Figure 5: Shelf with Two Brackets - using OUT parameters

2.7 Text

There are built in Pics for Text. PenDraw's unified approach to graphics attributes includes text. In addition to the usual text characteristics of font-family style variant and weight, PenDraw treats each text string as a graphic element. The programmer has full control, and specifies the following

1. Its position
2. Its angle of rotation
3. Whether it is mirrored or not
4. Its scale

Text has the following characteristics

1. Its size is specified in drawing units, not points
2. Text is rectangular, and is inside a metaphorical box. The box can be positioned by any of its 9 'Principal Points': 4 corners, 4 side mid-points, 1 centre
3. Text is subject to Specialty (attribute control)

2.7.1 Box Specialty

Any Pic can have the Box specialty: it is then metaphorically given a rectangular box. Text always has it: the text is entirely within the box. For other Pics the dimensions have to be specified. The Pic/Text can be positioned by any of 9 corners of mid-points, called Principal Points (PP).

When a Pic/Text has to be de-mirrored (for Default Mirror specialty) the box stays in the same place and the Pic/Text is un-mirrored within the box.

When a Pic/Text has to be de-overturned (for Default Overturn specialty), the box stays in the same place and the Pic/Text is rotated 180 degrees within the box.

3. Productivity Case Study - The Oil Well

3.1 Background

PenDraw was used in completing a software contract *in record time*, in the 1980s. The software company Adrian Tesson Associates (ATA), with whom the author worked at the time, won the contract from a major multi-national oil exploration and production company. For reasons of commercial confidence we shall call the oil company AB Ltd, or ABL.

ABL's IT department intended to develop an Oil Well inventory system, complete with automatic drawing generator, to run on a DEC VAX computer. ABL investigated CAD systems for this, and concluded that they needed the benefits of automation, not the slowness of interaction.

ABL had estimated that the project would take between *1 and 1.5 years*, using up to *2 programmers*, in *FORTRAN, using ABL's 2D graphics Subroutine Library add-on*. ATA speculatively built a prototype in three days and demonstrated it to ABL. ABL awarded ATA a contract to supply the system, on PC hardware, using a relational database, and PenDraw.

When the software was delivered, ABL contracted ATA to convert the PenDraw interpreter to run on DEC VAX computers, and to re-work the database on DEC VAX, thus making the Well inventory system available worldwide on their corporate VAX network.

3.2 Oil Wells

Oil wells are not just simple tubes that run from underground reservoir to surface. They have an outer tube (casing) and an inner tube that has engineering devices at intervals. The inner tube can be withdrawn and reconfigured with different devices to match prevailing well performance. There are many functions for these devices, but here is one example.

Oil flows up the inner tube because of pressure in the underground oil reservoir, but when a well nears the end of its life, its pressure decreases and oil flows more slowly. To increase flow, the well is re-configured. Near the bottom, one device injects detergent. Higher up, a second device turns the oil into foam by injecting gas. The column of foam in the tube flows more quickly, weighing less than pure oil, and the well produces more oil per hour.

Real wells are more complex and they have many more devices. Accurate configuration records are vital to well performance analysis and re-design. Hitherto, well engineers recorded well inventories manually on a schematic drawing which was annotated with configuration data. It was slow and open to error.

The cost of errors is potentially high. Incorrect records could result in incorrect design. If it was found that records were wrong, when the tubing was pulled out of the well, the well would have to re-designed and delays would result.

A non-functioning oil well represents significant lost revenue. Automating drawing from data was a very attractive move.

3.3 Well System Design

Data involved three entities, (oil) Field, Well, and (well) Components, with simple one-to-many relationships. One Field has many Wells. One Well has many Components. The data suited database technology, and an RDBMS was deployed for data storage and retrieval.

The power, effectiveness and automation of PenDraw made it ideal for the work.

For PenDraw and the RDBMS to work together, the RDBMS simply had to produce a report with Field, Well, and Component data for an individual well. The PenDraw program read the file and executed the drawing automatically. A schematic drawing element was developed for each type of well component.

System design can be summarised as

1. RDMBS for data entry, storage and retrieval
2. RDBMS for generating a report to contain data for PenDraw
3. PenDraw for picking up the data, and producing graphics
    1. User choice
        1. draw a specific well
        2. draw all wells in a specific field
        3. draw all wells in all fields
    2. Drawing(s) on either paper or monitor

3.4 Well System Development

Adrian Tesson did the RDBMS work. The author did the PenDraw programming. A prototype was developed and demonstrated to ABL after 3 days, who issued a contract to do the work. The remainder of the work was largely producing the full range of schematic drawing elements. It was completed in about 6 additional working days. After testing, the system was delivered.

ABL saw how quickly PenDraw enabled the graphical side of the system to be developed. They decided that they would like the schematics to be more realistic and drew up a more sophisticated set of schematics to replace the earlier ones. This work was completed in a further two weeks. ABL were again surprised, and did the same thing again, producing yet more sophisticated schematics. Development took a further two weeks. Thus in six weeks the graphics library for the system was produced three times.

3.5 Case Study Analysis

ABL estimated approximately 125 programmer-weeks to create the system. It was created with PenDraw in 12 person weeks, of which 6 were for CG programming. Part of the higher development speed was due to the use of an RDBMS instead of FORTRAN and file storage. The rest was due to the use of PenDraw.

At the time, it was estimated that of ABL IT’s 125 person-weeks, 60 were allowed for CG programming. PenDraw’s equivalent was 6.

Although the schematics were developed three times in PenDraw, it is likely that the time estimate allowed for similar refinements to be made to the FORTRAN schematics. Hence it appears that for development PenDraw was around 10 times as fast as FORTRAN + add-on.

3.6 Well Case Study Conclusion

PenDraw achieved productivity benefits of around a factor of 10 when compared to a development based on a conventional programming language coupled with CG add-on..

4. Ability Case Study - A Drawing Office

PenDraw was introduced to a steel fabrication company drawing office, again in the 1980s. Programming was carried out by draughtsmen/engineers (users).

4.1 Success with Non-Programmers

PenDraw was not presented to them as a programming language, but as a new way of generating drawings. Users were able to produce their first drawing program after around 3 days, with some training from the author. After two weeks they were working fully independently.

They produced a range of drawings, from single beam-to-column connections, through column baseplates, to entire portal frame rafters. One draughtsman's program could make an A0 drawing showing an entire, fully dimensioned, portal frame rafter, complete with roofing purlins, bracing cleats and haunches, and engineering notes. He developed the program entirely by himself.

PenDraw was genuinely able to bring CG programming to a wider ability range of potential users than professional programmers.

Those users would have refused to write graphics programs in a CG add-on programming environment.

4.2 Computational Efficiency Issues

PenDraw prioritises programmer efficiency over computational efficiency. Back in the 1980s, on large complex drawings, the trade-off won user approval for ease of use, but not always for run-time duration. The draughtsman would enter the data for the above rafter program and then run it. It took 15 minutes, on an Intel 8088 PC, to produce the drawing to monitor or pen-plotter. That was a deterring delay.

However, on a modern 3GHz Pentium IV PC the same program would take about 0.2 seconds.

In 3D CG work, computational efficiency is (rightly) assumed to merit very high priority. That assumption has spilled over into 2D graphics, but increasing processor power has made it less important. Provided that programs are not too computationally inefficient, it seems right to increase the emphasis on programmer efficiency.

Nowadays PenDraw can produce significant drawings in under a second. As yet little effort has gone into PenDraw's efficiency and there is much scope for improving it. It is suggested here that the emphasis on programmer efficiency is appropriate.

5. Status

5.1 PenDraw

The interpreter (compiler and run-time combined) pendraw.exe is free. The compiler will be separated from the run-time at some future point. Both will remain free. The .exe runs from the command-line. It runs stand-alone or on Web server. It produces SVG. At time of writing, pendraw.exe runs under Windows. It should be available under Linux later in 2005.

The interpreter is not open source, because we wish to retain control of developments. The language has some very unusual features, and experience has been that expert graphics programmers tend initially to suggest changes that would make it conform with what they are used to, which would yield negative results. Suggestions are very welcome, however, for the intention is to be responsive to the PenDraw community. It is not believed that PenDraw is beyond improvement! When the PenDraw community is

more mature, it may be appropriate to move the interpreter to open source.

5.2 PenDrawDev

There is an Integrated Development Environment, pendrawdev.exe, known as PenDrawDev. It runs under Windows operating systems only. It is low-cost. It combines a built-in syntax highlighting full-screen editor with interpreter and debugger. The debugger provides step/trace/run, plus data stack display, plus graphic display of geometric type variables. When a PenDraw program is stepped-through, it auto-completes the SVG so that the picture-so-far can be viewed in a browser. There is comprehensive syntax prompting/generation available in the editor on mouse right-click.

5.3 Intellectual Property

Programs written in the PenDraw language are like those written in any language as regards intellectual property. That will not change.

6. Conclusions

PenDraw is a dedicated CG language. It offers a real alternative to traditional computer graphics (CG) programming facilities of conventional language + CG add-on (library/package/etc), which present novices with quite a high cost of entry, and thus have tended to make CG programming an experts-only domain.

PenDraw's natural metaphor basis accelerates both programmer learning and program development.

PenDraw has several potential advantages. As an existing mature language and implementation, it has shown that the advantages are achievable and real, and that they translate into increased novice usability and greater productivity.

PenDraw produces SVG, and because PenDraw enforces rigorous program structure, its output SVG is well-formed and its consistent SVG structure produces good compression.

Given those qualities, PenDraw may be able to improve the take-up of CG programming and SVG. Take-up of PenDraw itself may be slow at first, however. Few are looking for a PenDraw-like development. CG market-place expectations are static, in that new developments are expected to focus a) on interactivity and 3D, not on 'batch processing' and 2D, and b) on add-ons and point-and-click editors, not on new languages. Furthermore, 2D CG expert programmers are content with what they already have, while non-experts tend to avoid CG programming because of its perceived problems.

Over time, however, as PenDraw's effectiveness, simplicity, expressive power, and programming cost benefits become known, its use should grow, thus contributing to growth of CG programming and SVG.

References

[BASIC1]

*BASIC: A Manual for BASIC, the elementary algebraic language designed for use with the Dartmouth Time Sharing System*Dartmouth College (1964).

[FOLEY1]

Fundamentals of Interactive Computer Graphics Foley, J. D., and Van Dam, A., (1982). Addison-Wesley.

[ISO1]

Graphical Kernel System (GKS), Version 6.6 I.S.O., (1981). International Standards

Organization, May 1981.

[KULSRUD1]

A General Purpose Graphics Language Kulsrud, H.E., (1968). Communications of the ACM, 11(4), April 1968, pp. 247-254.

[KURTZ1]

True BASIC is the Ideal First Step Kurtz, T. E., (1999). Available at (http://www.truebasic.com/downloads/D2006.pdf)

[SGI1]

The OpenGL Graphics System: A Specification (Version 2.0 - October 22, 2004) Segal, M., and Akeley, K., Silicon Graphics International, 1992-2004.