

## **Managing Procedural Knowledge**

**Sven Havemann     Dieter Fellner**

Institut of Computer Graphics, TU Braunschweig, Germany  
s.havemann | d.fellner@tu-bs.de

**Abstract:** Procedural knowledge is one of the most valuable assets of individuals as well as academic institutions and commercial companies. The ability to satisfy an order relies on the knowledge how similar tasks have been performed in the past. Thus the preservation of this knowledge is critical.

Procedural knowledge takes many different forms, which makes it very hard to reason about it. We propose a method to reduce it to its very essence. This method is very simple, and as such it is not new. But we argue that it is worthwhile to take a fresh look on an existing technology from a new point of view, because it may solve the problem of knowledge preservation that has become apparent in this form only recently. Although the technique is known for a long time, it appears that its potential for the management of procedural knowledge has not been realized so far. It is also a very elegant method since we can show that it serves both as a theoretical device to better understand the nature of processes, but it can also be directly operationalized to derive a new generation of user-friendly tools that support the preservation of procedural knowledge.

**Key Words:** procedural knowledge, generative modeling, generalized documents, 3D semantics

**Category:** H.3.7 (digital libraries), D.2.13 (reusable software), I.3.5 (geometric modeling)

### **1 Introduction on Procedural Knowledge**

One of the main reasons for the enormous success of the personal computer in the 1980's was a revolutionary killer application: the spreadsheet. Software systems such as VisiCalc [14, 5] gave average persons for the first time the possibility to perform complicated calculations easily and instantly. Every company, every store could keep track of all goods ordered and sold, make projections of the future, and compute the optimal price for their offers.

The remarkable thing about a spreadsheet is that it is a mixture of static and dynamic data. Once the relations between the different cells are set up, it permits to vary some of the data to see their influence on other data. Every cell may contain either a piece of data, usually a numeric value, or an expression that refers to other cells to compute that value. A spreadsheet calculator provides frequently used operations such as to sum rows and columns of the table, and it permits flexible copy and paste of cells and cell groups. In particular, the software supports the user by setting the references to other cells appropriately when a cell is copied, and it tries to maintain consistency when entire rows or columns are inserted or deleted.

A spreadsheet may provide a remarkable degree of functionality, some impressive examples can be found, e.g., on the website of the European Commission [6]. Spreadsheets gain their efficiency through a framework of restrictions, for example the fixed

table layout with cells labeled A1, A2, A3, etc. Within the framework, a great level of flexibility is possible. For more general tasks, the user must leave the restricted framework and resort to a more general method to tell the computer what to do: *Programming*.

### 1.1 Programming to Achieve Automization

Programming is a skill that is not easy to learn: There is not much difference between learning a programming language and learning a foreign language. Things can be expressed in many different ways, some are more elegant, some less. It may take months of practical exercise to get acquainted with the spirit of a new language, to find ways to express oneself clearly and unambiguously. And to some degree, it requires to adopt a new culture and a new way of thinking. Particularly beginners to the *art of computer programming* have great problems with the strict syntactic requirements of a formal language.

Users equipped with programming skills can create their own powerful tools, specifically tailored to their needs and habits. With programming, one can enhance the functionality of a spreadsheet, automatize administrative tasks with unix-style shell scripts, and query databases with SQL [4]. Larger interactive software packages have built-in scripting languages. Originally designed for recording macros, their functionality sometimes expands to the point where they allow to completely change the appearance of the application. Examples include VBScript from MS Office [7], EmacsLisp [16], and MEL for Maya [12]. Another approach is to embed a standard scripting language such as Perl, Tcl/Tk, or Python into an application. A third option is to use a *software development kit* (SDK) for direct low-level programming (C/C++), which can also be made accessible to a scripting language by using a *wrapper generator* such as SWIG [2].

Perhaps the strongest incentive for using scripting languages is *automization*. Interactive software packages offer deeply nested menus with dozens or even hundreds of different operations, many of which can be further configured with detailed dialogue boxes – interactively. A script however can reduce a task to its very essence: A single dialogue box permits to specify only the few really relevant options. All the other options of the operations involved are specified non-interactively by the script – which is more efficient and less error-prone.

### 1.2 Procedural Knowledge (PK)

Spreadsheets were a major step forward in enabling average users to take benefit from computers to facilitate their work. But for tasks beyond spreadsheets, to use the full power of the machine and the full flexibility of the available software, a user has to resort to programming – which is amenable only to a small minority of the user community. This has immediate consequences for knowledge management.

A technology for *all* users, i.e., also those with no programming skills, that

- permits to define a *process* by specifying a sequence of processing steps,
- allowing to integrate different applications into one process,
- but does not require any literal programming, and
- demands only a level of expertise comparable to creating a spreadsheet.

**Figure 1:** The missing link between spreadsheets and programming.

On a larger scale, task automatization leads to *workflow management*. Following a bottom-up approach, the specification of a processing sequence within one tool can be generalized to the specification of which tool to use, and how to use it, in the different steps of processing an order or realizing a project. In any company or institution, orders and projects require several people to work together in an organized fashion. The knowledge about the organization of the processes, e.g., to carry out an order, is in the minds of the employees: Which sub-tasks to assign to which individuals, when to set deadlines, what results to expect from sub-processes, etc. We call knowledge of this type *procedural knowledge* (PK), as opposed to declarative or relational knowledge. The latter two may be more common subjects of knowledge management, but they unfortunately fail to represent PK adequately (unless PROLOG [13] is used, maybe).

Many sectors, e.g., the print or automotive industries, have radically adopted digital technology, and the traditional workflow has become a digital one. Consequently, one solution to streamline typical processes is to hire a software company that sets up a full-scale workflow management system. But this often creates additional problems (communication etc.), and is not feasible for smaller companies, or for smaller automatization tasks.

The link is missing between (limited) spreadsheets on the one hand and (large) workflow management systems on the other: A new technology that permits all employees to automatize their work as well as to preserve their procedural knowledge (Fig. 1).

### 1.2.1 The Code Generation Problem

The employees' procedural knowledge must be somehow mapped to an unambiguous, computer-readable representation. This raises the question of how to encode procedural knowledge. The encoding is crucial for the efficiency of PK generation, maintenance, and re-use (see Fig. 2), which gives a quality measure for comparing different encodings.

One representation that was used already in the 70's is the data flow network: Large diagrams with symbols for 'if', 'input', and 'output' have proven to be too low level and hard to maintain. Today a more abstract notation is used in the form of UML dia-

- PK generation: mapping human knowledge to computer-readable form
- PK maintenance: browsing in, updating, or changing processes
- PK re-use: using existing PK in similar new situations

**Figure 2:** Criteria to assess the efficiency of PK representations (encodings).

grams, which represent (static) relations between object-oriented classes. They have deficiencies, however, in representing algorithms/processes.

Probably the most common encoding of PK is as the *source code* in some formal programming language. To facilitate PK generation, users should not be required to write programs with an ascii text editor. Coding should be avoided. Instead, source code should be *generated* in background. Unfortunately, with most conventional programming languages (such as C, Java, VisualBasic), automatic code generation is far from trivial: The source code, regarded as a character string, contains many interdependencies (declaration/definition), must obey to a complicated syntax, respect a strict type system etc.

This affects negatively not only the ease of PK generation, but also its maintainability. The administration and management of procedural knowledge represented as C++ or Java code, stored in large source code repositories, requires extremely sophisticated tools [11, 15] and gifted source code administrators.

As a consequence, a formal language for representing PK should be much, much simpler in order to be more manageable – for easier code generation, easier maintenance, and, hopefully, better re-usability. So what we did was to try the simplest type of programming language we could possibly find, and examine its suitability for representing procedural knowledge. The result was surprising.

## 2 A Simplistic Encoding of Procedural Knowledge

The following considerations have three different levels: An abstract description of the elements of procedural knowledge, a practical level (task automatization), and a formal, syntactical level. The purpose is to develop a very natural encoding of procedural knowledge.

Our assumption is that procedural knowledge can be understood in a strict sense as a collection of process descriptions. Thus we define a *process* to be the most important entity of PK. Furthermore a process is a sequence of simpler entities, which we call *operations*. In the simplest case, an operation may be constant, i.e., just a static piece of data - a number, a string, etc. Such data are to be kept for later use. This leads quite naturally to a stack-based approach: An arbitrary amount of data is just put 'on stack', until it is used by some operation. One example is to select data items interactively, and to apply a (non-constant) operation such as 'make text bold', 'move files', or 'sum rows' to them.

A more complex scenario is the application of a whole sequence of operations to one (or more) data items. In this case, every operation takes some pieces of data, processes them, and produces new, transformed data. A typical practical example is batch processing of, e.g., an image, such as: to crop a part, enhance its contrast, apply a bump map, and add it back to the original image. This linear sequence of operations can also be regarded as one new *combined operation*, essentially a macro. In an interactive application, it could be added to a drop-down menu as a *user-defined tool* that automatizes a specific task and extends the set of built-in tools. And it can, of course, also be used to create new combined operations out of old combined operations – which is nothing but programming.

An operation can often be further configured with a dialogue box. This is nothing but a graphical form with a number of key-value pairs, essentially a *dictionary*. A practical example is a print dialogue with its typical entries: 'pages: 1-12', 'paper size: A4', 'orientation: landscape', 'color mode: greyscale', etc.

## 2.1 A Stack-Based Language

The abstract and practical levels have been discussed. The formal level is an immediate consequence: the representation of a process using a stack based language. Although well known, it is described in the following. Note that the description is very short - and it is complete! This clearly indicates the simplicity, and thus, the suitability as PK encoding, of this type of languages. And although the following is taught in every introductory course on programming languages, it provides new insights with respect to knowledge management, as will be shown in the next section.

First of all it is natural that the encoding of a process is a string of characters, and the individual operations are words separated by whitespaces. Every such word is called a token. It can either be a piece of data, or it can actually perform an operation – i.e., a token can be either *literal* or *executable*. To execute a process then only means to execute each token in the sequence, from begin to end. To execute a literal token just means to put on the *operand* stack. Executable tokens fall into either of two categories: Elementary (built-in) operators, and combined (user-defined) operators. A built-in operator, when executed, takes a number of arguments from the stack, processes them, and pushes the results back. Before explaining the execution of combined operators, the two higher-level data structures need to be introduced, arrays and dictionaries.

To create an array, the literal bracket [ is put on the stack as a marker. The closing ] is an operator. It pops all stack elements until it finds a marker, creates an array from them, and pushes a single token as a reference to the array. The dict operator creates a new empty dictionary and pushes a reference as well. A stack of dictionaries is used for name lookup: searching top to bottom through the *dictionary stack* the first dictionary that contains the name as a key defines the value. A new dictionary is pushed on the dictionary stack with begin, and it is popped again with end. This yields a very flexible mechanism for name lookup, since the scope can also be changed within a function:

```
dict begin /x 4.4 def dict begin /x 5.5 def x end x end → 5.5 4.4
```

The code to the left has the effect on the stack shown to the right, where the stack top is 4.4. Names come in two flavours, literal and executable: `/x` is a literal name that is just put on the stack; the `def` operator expects a name and an object, and makes them a key-value pair in the current (topmost) dictionary of the dictionary stack. To retrieve the value, the preceding slash is left away: `x` is an executable name, which means that it is looked up, and the object found is executed – in the example it is a literal number, which is put on the stack.

Arrays also exist in two flavours. Executable arrays are created much like the literal arrays before, except that curly brackets `{ }` are used, and that the first `{` starts the *deferred mode*. In deferred mode, all tokens are treated as literals, i.e., just pushed on the stack. Deferred mode stops when the matching `}` for the first `{` is found in the token stream (pairs of `{ }` can be nested). Then an array is created as before, but this time it is flagged as executable. An executable array is executed just like a process, by executing each token in the sequence, from begin to end.

If the currently executed array contains an executable name that refers to another executable array, this array is called as a sub-routine: the currently executed array is pushed on the *execution stack*, and the new array becomes the currently executed array. When it is finished, the previous array is popped from the execution stack, and its execution continues. This makes it very easy to create macros for often needed sub-sequence: Simply enclose it in `{ .. }` and give it a name. The `dup` operator duplicates the stack top, and `mul` replaces the two elements on top of the stack by their product. A number is then squared by multiplying it with its duplicate, as in the following three lines of code, which are all equivalent:

```
1.5 dup mul 2.5 dup mul → 2.25 6.25
/sqr { dup mul } def 1.5 sqr 2.5 sqr → 2.25 6.25
/sqr { dup mul } def [ 1.5 2.5 ] { sqr } forall → 2.25 6.25
```

Note how the stack-based approach permits to separate data from operations by re-ordering the token sequence: the same operations can be applied to different data, and the same data can be used with different functions. Effectively there is no difference between a *process*, a *combined operator*, a *function*, and an *executable array*; all these are synonyms. So a stack-based language is also a functional language, with functions being 'first class citizens' just like arrays are. Functions are just 'action sequences', and they can be created and changed just like any other array. In particular, it is possible to create new functions by concatenating arrays, and to alter existing functions by changing items in the array (self-modifying code). This recombability is the basis for 'programming without coding': The result of a program can be another program.

## 2.2 The Generative Modeling Language (GML)

We have implemented one version of a stack-based scripting language, the *Generative Modeling Language* (GML) [9, 1]. It was originally designed to serve as a low-level file

format for procedurally defined 3-dimensional objects, such as gears, staircases, and Gothic architecture [10]. In order to exploit the procedural properties of these models, we have chosen our file format to be a low-level programming language, rather than just a collection of geometric primitives (triangles, spheres, NURBS etc.).

The stack-based approach has proven very powerful, and it appears that its usefulness goes far beyond its traditional job of representing 2D graphics with PostScript [3] or 3D objects with the GML. Our experiences indicate that the simplicity of attaching an operator-based interface to existing source-code make this type of language an ideal 'glue' to tie also different applications together. And the simplicity of code generation makes it the ideal 'invisible language': Recording a macro is as simple as creating an array from the individual GUI actions.

The second factor that makes a stack-based language very useful is its relative closeness to pure data formats. Note that a sequence of atomic values already *is* a program of a stack-based language. With, e.g., a stream editor, it is easily possible to intersperse the data with names of operations. Sometimes it is surprisingly simple to achieve complex results with this approach.

As a (final) example, there is a remarkably simple transformation from XML to our GML. The first of the following three lines is XML syntax (HTML), the last is GML syntax, but note the line in between:

```
<b> this is bold <i> and italic text </i> just bold again </b>
b >this is bold< i >and italic text< ni >just bold again< nb
b "this is bold" i "and italic text" ni "just bold again" nb
```

The line in the middle can be considered a GML program if the tokenizer is made to accept > and < as symbols that open and close a character string. Leaving away the first '<' and the terminating '>' character just reverses the notion of normal text: Instead of structured formatting symbols inserted into normal text, the document is now made of character strings that are arguments to operators. Yet syntactically, there is not much difference. And note that this is just what an XML parser does, since each chunk of consecutive characters is placed into its own text node in the DOM tree.

For the example above to actually work it is just necessary to provide some operators such as `b` and `nb` to switch the 'bold' property on and off as a side effect to consuming the text. The next example shows that when using GML, it is not necessary to resort to a second technology (e.g., DTDs) just to be able to define styles concisely. The first and second lines show the XML and GML versions just like above.

```
<font size="+1" color="red"> large red text </font>
pushfont +1 fontsize "red" fontcolor >large red text < fontpop
```

The effect of the following two lines is identical to the second line above. But this time a style is defined, so that the font manipulation does not have to be always repeated:

```
/myfont1 { pushfont +1 fontsize "red" fontcolor } def
myfont1 >larger red text < fontpop
```

When the style `myfont1` is declared in the preamble, it can simply be used throughout the whole document – which makes it much easier to change its definition.

### 3 Summary and Conclusions

We have discussed the fundamental problem of maintaining procedural knowledge. As principal constituent of PK we have identified the *process*, which consists of a sequence of individual processing steps. It may also involve loops and conditional decisions, which naturally leads to a stack-based programming language approach.

It must be pointed out that depending on the type of procedures to be described, programming languages of any form may not be the appropriate form of description. Humans who describe things or procedures normally refer to implicit prior knowledge, and the resulting descriptions are not precise enough to be executed by a computer. The computer, on the other hand, is often perceived as unintelligent because literally every action it is to perform needs to be described precisely, which is tedious and error-prone.

We believe that good tools should support the user in realizing his or her ideas, which means more precisely: To turn his or her implicit knowledge into explicit knowledge. The functionality of an application is determined by the expressiveness of the file format it uses. A more flexible and dynamic general file format standard, one that can represent also procedural aspects contained with the data, is therefore a reasonable point of departure for a new generation of user-friendly tools that will hopefully surpass even VisiCalc in flexibility, usefulness, and ease of use.

We gratefully acknowledge the support from the German Research Foundation (DFG) under the Strategic Research Initiative *Distributed Processing and Delivery of Generalized Digital Documents (V<sup>3</sup>D<sup>2</sup>)* [8]. – We would also like to thank one of our anonymous reviewers.

### References

1. Gml scripting language website. <http://www.generative-modeling.org>.
2. Swig wrapper generator website. <http://www.swig.org>.
3. Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley, 1999.
4. ISO ANSI. Database language sql iso/iec 9075:1992, 1991.
5. Dan Bricklin. Visicalc: Information from its creators. In *Bricklin.com*. WWW, 2004.
6. European Commission. Cordis – community research & development information service. <http://www.cordis.lu>.
7. Microsoft Corporation. Visualbasic script. <http://www.microsoft.com>.
8. Dieter W. Fellner. Strategic Initiative V<sup>3</sup>D<sup>2</sup> – Distributed Processing and Delivery of Digital Documents. German Research Foundation (DFG), <http://graphics.tu-bs.de/dfgspp/V3D2>, 1998-2003.
9. Sven Havemann. *Generative Mesh Modeling*. PhD thesis, Technical University Braunschweig, Germany, 2005.
10. Sven Havemann and Dieter Fellner. Generative parametric design of gothic window tracery. In F. Giannini and A. Pasko, editors, *Proc. Shape Modeling and Application (SMI'04)*, pages 350–354, Genova, June 2004. IEEE.
11. IBM. Rational rose. <http://www-306.ibm.com/software/rational>.
12. Alias Inc. Maya embedded language (mel). <http://www.alias.com>.
13. ISO. Prolog standard iso/iec 13211-1, 1995.
14. D. J. Power. A brief history of spreadsheets. In *DSSResources.COM*. WWW, august 2004.
15. Wind River Software. The sniff+ software repository. <http://www.windriver.com>.
16. Richard M. Stallman. *GNU Emacs Manual*. FSF, 2002. <http://www.gnu.org>.