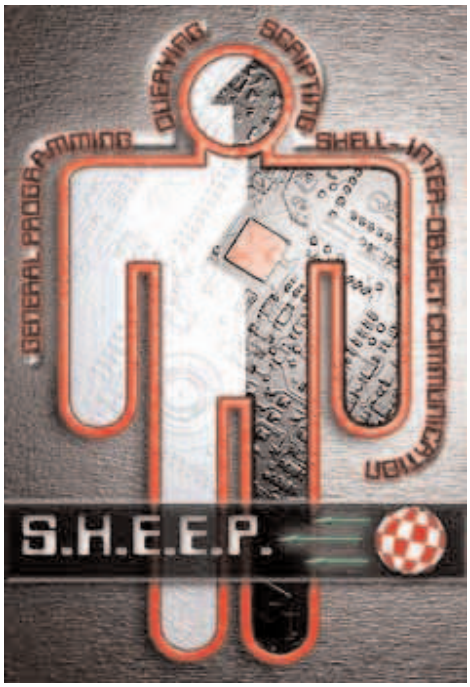
Making the Digital Environment Talk Sheep



Scripting is a great strength of the Classic Amiga, because it allows users to combine and automate operations from all different programs in the system, passing data automatically back and forth, using the best features of each to get results that no one program, however bloated, could match.

The new central scripting language of the Amiga Digital Environment (DE) is code-named SHEEP (This is an internal working name for this language. The final name will be different). It handles scripting, querying, Shell and inter-object communication, and general programming. It enables the user to interact with objects in the AmigaDE, and allows objects within the DE to control each other. It is the glue that holds the flexible Digital Environment together and makes everything universally scriptable, configurable and modifiable.

Having a language like SHEEP as an integral part of a system enhances its usefulness, as most applications will then be written with scriptability in mind. That integration allows users to harness the functionality available in the OS or applications and combine it with custom code to fill in what's missing. SHEEP will handle both GUI and command-line interaction and make it all as painless as possible.

Heading up the SHEEP project for Amiga is Wouter van Oortmerssen a programmer with a stellar reputation on the Classic Amiga who has jumped at the chance to help design the new one. Wouter is best known for his concise yet powerful Amiga-E language, yet this is just one of more than a dozen languages he's designed en route to his doctorate in Computer Science. He's now working full time for Amiga Inc., developing SHEEP.

Tim Sweeney, creator of the 3D shooter Unreal and programmer extraordinaire, summed up his thoughts of Wouter's talents this way: "Wouter is a brilliant programming language designer, and possibly the most prolific one on Earth--he wrote the E programming language back in the early Amiga days, and has since implemented a ton of imperative, functional, visual, reactive, linear, and other crazy languages. We email once in a while, and whenever I describe some new language idea I have, he's like: 'I implemented that 5 years ago, and here are the problems you run into...'"

Heritage

The Classic Amiga scripting language was ARexx, a standard component since release 2 a decade ago. ARexx was derived from Rexx, the original IBM mainframe version, with many Amiga-specific extensions. Versions of Rexx have since been ported to many other systems, like OS/2, Unix and Qdos. Rexx is a general-purpose high-level language, versatile yet easy to learn.

"If you want to get some job done on your computer, either it is very easy or it is extremely difficult," says Wouter van Oortmerssen. The decisive factor, he says, is whether you have an application suited to the job at hand. "And even when you have a GUI app that can do what you want to do, GUI's are traditionally very inflexible, especially for repetitive tasks. For those [times], there are very few alternatives to using a scripting language," he says.

The true significance of ARexx stemmed from the way it became a standard part of the Classic Amiga. Almost every system component, from applications to the desktop, and even emulators for other systems, had an ARexx port, so any operation that could be performed manually with the mouse or keyboard could be automated under ARexx control. In some cases this allowed operations or custom configurations that were only feasible with ARexx support.

There was no need for each program to have its own arcane scripting language built-in, as

Continued

has led to notorious compatibility and security problems in clumsier systems. While the idea was brilliant, well-executed and integrated, ARexx was not perfect—it dated back to the 1980s, and could be slow, prolix or clumsy at times. It did a lot, but could have done more.

Before ARexx became part of the system, old Amigas were shipped with an implementation of Microsoft BASIC. As on many home computers, users were encouraged to write their own programs in this dialect, extended with support for Amiga multimedia. While superior to their other offerings, Microsoft BASIC was slow, big and buggy. It was inflexible and incompatible with 32-bit systems, so third party packages like Amos, Blitz, Maxxon and HiSoft BASICs arrived in the 1990s to fill the gap between ARexx and professional programming languages like C, Oberon and hardcore assembly language.

ARexx was tightly integrated with the Amiga OS and applications and "allowed you to do the kind of programming that just isn't possible in any other way," says van Oortmerssen. "Besides, ARexx was also a nice design that fitted its purpose, simple and friendly so everyone could pick it up and do useful things with it quickly."

Enter SHEEP

SHEEP fills that niche, as well as the need for a standard, powerful scripting language. It's quite possible to write applications and utilities in SHEEP, and is in fact a very efficient approach for building prototypes or solving the sort of problems that might keep one programmer busy for anything from a few minutes to a few days. While compilers for Java, C, C++ and our ultra-efficient portable VP-code are more suitable for larger projects or routines likely to be run many times, SHEEP's interpreted, functional design delivers results sooner, if not always more quickly. It's ideal for interactive development and testing, an area where compiling languages cannot compete.

I expect SHEEP to go much further than ARexx," says van Oortmerssen. "ARexx depended on interfaces being implemented [into applications], so if the programmer didn't 'expose' certain functionality, ARexx had no way to get at it."

You can go from editing a SHEEP program to running it, and back into the editor of your choice, immediately. Future versions will allow your SHEEP code to be compiled after it has been tested, and to make secure, stand-alone programs that outperform Java and match the speed of compiled C++ in many cases.

SHEEP is ideal for installation scripts, communication between programs—much like ARexx, including access to ARexx in emulated Classic Amiga environments—and the sort of hacks that would be written in shell script, BASIC, E, Perl or Python on old systems. However, SHEEP is far more secure than ARexx, C or BASIC. There's no possibility of accidentally overwriting memory or clobbering some system resource unrelated to your program. SHEEP forswears the dangerous POKEs and pointers that fill the gaps in other languages, yet it can manipulate complex data structures simply and efficiently.

SHEEP can be used for database queries, as SQL (Structured Query Language) is on big systems. The new Amiga boasts immensely powerful ways of storing and organizing data; SHEEP provides a clear window on the Digital Environment.

Breeds of SHEEP

SHEEP can be interpreted, for ease of maintenance, or compiled into fast VP code. SHEEP is a synthesis of several Classic Amiga concepts and good ideas from elsewhere, into a powerful yet simple language that is easy to read and write, without the arbitrary and arcane punctuation of many other languages.

SHEEP works without the 'garbage collection' that unpredictably wastes time and memory in most other languages that allow dynamic data-structures. It does this by using new functional programming principles and automatic memory management, without forcing the programmer to use it any differently from conventional languages.

Continued

SHEEP is polymorphic so you can write programs oblivious of the exact types of data they will manipulate, and re-use code without making type-specific versions or complicated, slow and error-prone run-time tests. SHEEP lets you specify data types when it could make programs more efficient if they were preordained, but it doesn't force you to do so in more general cases. New types can be defined, with any combination of data inside. Types can be a superset of several types. Thus a tree with any number of branches or leaves can be defined in one line:

```
type tree = branch(left:tree, right:tree) | leaf(data)
```

This defines two types, 'branch' and 'leaf,' and a supertype, 'tree,' which can be used to refer to the value of either of those types. The vertical bar indicates that a tree (or sub-tree, implicitly) can be either a branch made up of further (sub) trees or a leaf, with a 'data' property.

""The new Amiga OS, is much more based on objects," says van Oortmerssen, "which will automatically expose their programmable interfaces to the world. The potential for ARexx-like scripting is therefore far greater." SHEEP provides a window to the AmigaDE and its immensely powerful ways of storing and organizing data.

Every value has a type, but it has the most general type 'any' unless you specify otherwise. If a general type is passed into a more specific context, as when a string of characters is passed to a routine expecting an integer, SHEEP coerces the string to a number, and raises a dynamic type error if it won't fit. If both types are strictly defined, the compiler detects mismatches before the program runs.

SHEEP reports errors in context. If in doubt, it issues a warning. SHEEP has built-in error-tracking and exception handling, so if something goes wrong you can identify the problem and fix it, or alter the program so that the special case will be spotted and resolved automatically, without cluttering up the source with conditional tests. This gives beginners support and reassurance, without cramping the style of experts.

Counting SHEEP

SHEEP uses the concept of 'vectors' to express structures normally stored in static arrays or dynamic lists. SHEEP vectors combine the efficiency of arrays with the dynamism of lists. Complete or partial vectors may be compared, extended or replaced.

Functions may take any number of parameters, applying defaults or patterns if necessary, and may return any number of values. You can tag parameters when you call a function, so you need only specify the parameters you care about, and they may be in whatever order you like. Thus you could specify desired details of a new screen, letting others default, by calling an 'openscreen' function like this:

```
openscreen width: 640
           height: 480
           title: "Goats and SHEEP"
           colourdepth: 32
           foreground: 1
           background: 0
```

"SHEEP has automatic memory management that requires almost no extra memory compared to what is in use, unlike most 'garbage collected' languages (like Java)" points out van Oortmerssen. "It can therefore run small scripts in as little as 10kb (code + data), which makes it suitable for Personal Data Assistants, embedded systems, etc." And small is good.

You can define several instances of a function, depending on the type or value of parameters you want to deal with. SHEEP automatically chooses the right one to match a particular function call. Gratuitous brackets are not needed, either in the call or the function definition.

Pattern matching is a key feature of SHEEP, simplifying programs. SHEEP can search out patterns within lists, splitting them according to the position of the pattern. It can just as readily merge patterns and lists. Pattern matching also allows particular values to be extracted from user-defined data types.

Continued

There are several neat ways to test large amount of data in one step. 'Find' performs a conditional search through a vector, returning an index of the first match. 'Filter' is similar but returns a vector denoting all the matches. 'Fold' combines all the matches, accumulating a total.

Spacing

Sheep is a no-limits language, with no restriction other than available memory on the length of strings, vectors, identifiers, programs or individual lines.

SHEEP notes the indentation of the first line in a list. If a new line is indented the same amount as the previous one, it's a new statement. If it's indented more, it's taken as a continuation, allowing long statements to be presented neatly. If it's indented less, the next token implies the end of the list.

Functionality

One striking feature of SHEEP is the way that you can teach it rules in a simple format, and expect it to find and apply them in context. You can add functions, or definitions, without having to wrap them in obscure syntax. To take a very simple example, imagine you need to report the number of files a program has copied in a human-friendly way. Rather than resort to a sequence of tests, or sub-literate laziness like "n file(s) copied," you can write:

```
print (nfiles n) + " copied"
define nfiles 0 -> "no files"
define nfiles 1 -> "one file"
define nfiles n -> n + " files"
```

Strings can be built up as simply as they could be in Java. So if you want special formatting - like padding, or like this example—you insert extra functions. References to "file(s)" become a thing of the past, hurrah! The minimal Sheep program is just one line long. Here's the classic "Hello World" program, fully expressed in SHEEP:

While SHEEP's focus is on simplicity and small tasks—it is not meant to replace C++/Java for most professional programmers—it is designed to scale up from scripting to more complex code very effectively. This is achieved with features transparent to the beginner programmer but available (and useful) for more advanced programming. "And the language design," points out van Oortmerssen, "in general allows for faster code, all without hurting its friendliness." As such, SHEEP is aimed squarely at its target: Helping users maximize their productivity in the Digital Environment. It is a key component of the new Amiga, and its unique singularity and tremendous strengths will bring programmers flocking to the New Amiga.

```
print "Hello, World!\n"
```

The only complication here is the "\n" sequence, which tells the system to make a new line after printing "Hello, World!" This is the same syntax as C, VP code and other programming languages. Tabs can similarly be written \t, embedded double quotes are written \", and \\ stands for one literal backslash.

Sorting

Here's a slightly more complicated example—an implementation of the QuickSort algorithm, which requires a couple dozen lines of code in many languages. SHEEP can do it in just four (the lines starting with two dashes are comments) with no weird symbols apart from brackets, plus signs and arrows, all of which have relatively obvious meanings.

```
-- polymorphic quicksort
define qsort [] -> []
define qsort [pivot] + rest ->
 (qsort filter x in rest where before x pivot) + [pivot] +
 (qsort filter x in rest where not before x pivot)
```

Continued

You'll notice there's no need to declare the types of data items like x, rest or pivot. This is not sloppiness—the data type simply doesn't matter to SHEEP. The same code will happily and efficiently sort text, whole numbers, or fractions, whereas old-fashioned languages would need a separate routine for each.

In this case,

```
print qsort [5,2,6,3,8]
```

returns

```
[2,3,5,6,8]
```

You don't have to warn the program about the length of the vector or check it inside the program. One size fits all, as the program is recursive—it works by calling itself repeatedly, splitting the list into smaller parts until they're all in order, but you don't have to worry about programming or checking the stack—SHEEP looks after that for you.

The definition of what order the sorted list should be delivered in is similarly flexible. The term 'before' in the listing could have any name, and any definition associated with it which can determine the order of two items. If we want to sort a vector of whole numbers, known as int(egers), we define 'before' like this:

```
define before x:int y:int = x<y
```

This says that when 'x' and 'y' are integers, 'before' is true if 'x' is less than (<) 'y.' Equivalent rules may be defined for other types of data, and you need not be limited by them—if you want punctuation at the end of a sorted list of words (rather than in ANSI or Unicode order), you may define an arbitrary function to put things the way you want them.

The 'filter' operation scans through a list, assigning each value in turn to 'x' and asking 'before x pivot?' If this is true, the value 'x' precedes the value 'pivot', so 'x' is moved to the result. Once all the 'rest' has been scanned, the list is more nearly in order, as early values have been removed. 'Quicksort' calls itself for progressively shorter lists, until every value is in its proper place.

SHEEP delivers some of the same excellent properties as ARexx: "It's simple and friendly right from the start," says van Oortmerssen. "ARexx, however, had the problem that people easily outgrew it: As soon as they mastered the language, they noticed that slightly larger scripts would run dreadfully slow and easily become messy. SHEEP is designed to last you a lot longer," he says.

Splitting and Joining SHEEP

The expression '[pivot] + rest' shows how lists of values (or 'vectors') may be split and joined in SHEEP. The expression takes a vector and assigns the first element to 'pivot' and the remainder to 'rest.' The '+' operation can both join and split patterns. It allows us to create vectors like this:

```
a = [1,2,3]
a = a+[4,5,6]
```

This leaves 'a' holding the vector [1,2,3,4,5,6]. The '+' operation constructs vectors, while pattern matching inspects and deconstructs them. Imagine a function 'chop,' which is passed a vector. The following instruction takes the vector apart:

```
define chop [1,2]+v
```

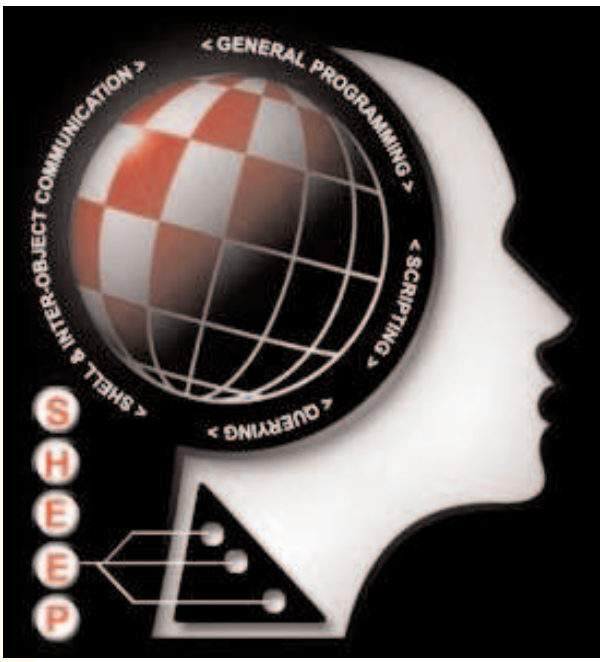
The pattern matching operation splits the parameter in one vector of length two, which must contain the integers 1 and 2, and puts the remainder of the vector into 'v,' so 'v' becomes [3,4,5,6] if 'chop' is passed the value assigned to 'a.'

Continued

Graphics

The last example shows how SHEEP can express a classic graphics algorithm. It assumes that a device with appropriate width and height is ready to accept output from the 'plot' statements.

This example shows that the functional sophistication of SHEEP is not an obstacle to conventional programming. Apart from the lack of syntactical clutter, the Mandelbrot listing could be in any block-structured language. It can be run and edited as readily as interpreted BASIC, yet has the speed and structure of a modern compiled dialect:




```
-- SHEEP Example Mandelbrot fractal generator
width = 640
height = 480
maxdepth = 128

define count x:real y:real do
 xc = x
 yc = y
 i = 0
 while i<maxdepth and x*x+y*y<4.0 -- |z| < 2
  t = x
  x = x*x-y*y+xc    -- z = z*z + c
  y = (t+t)*y+yc
  i = i+1
 end
 return i
end
w = 3.5 -- change these to zoom or move about
h = 2.8
top = -1.6
left = -2.0
for x = 0 until width do
  for y = 0 until height do
   plot w x y (count x/width*w+left y/height*h+top)
 end
end
```

Conclusion

When there are already thousands of computer languages in existence, you need special reasons to propose a new one, let alone develop something as sophisticated as SHEEP, which is simple on the surface yet built on cutting-edge Computer Science within.

SHEEP borrows concepts from classics like ARexx, Prolog, BASIC and Perl, and newer languages like Python and Java, and benefits from the practical experience of 15 years of Classic Amiga product development and integration. SHEEP adds significant new ideas, eliminates dangerous or over-complicated gimmicks, and has the cohesion that comes from one brilliant mind, rather than a committee of lobbyists.

SHEEP demonstrates that if a job's really worth doing, it's worth approaching afresh. SHEEP not only **is** something new and exciting—it allows and facilitates great things that would not otherwise be practical. It is a key component of the new Amiga, and nothing else can match it.

Contents