Reviving Smalltalk-78

The First Modern Smalltalk Lives Again

Dan Ingalls
CDG Labs,
San Francisco, CA, USA
dan@cdglabs.org

Bert Freudenberg
CDG Labs,
Potsdam, Germany
bert@cdglabs.org

Ted Kaehler Yoshiki Ohshima
Alan Kay
Viewpoints Research Institute,
Los Angeles, CA, USA
{alan.kay,ted,yoshiki}@vpri.org

Abstract

We report on a revival of Smalltalk-78 in JavaScript that runs in any web browser. Smalltalk-78 was a port of Smalltalk-76 to the NoteTaker, a portable computer based on the Intel 8086 processor. This same interpreter design and snapshot is the ancestor of Smalltalk-80 and Squeak systems of today. We describe our conversion to a completely different object memory with essentially no visible changes in the language or system, as well as our support of Smalltalk-78's linearized contexts that used the 8086 stack directly. We report how the Lively Web development system facilitated tooling for the project, as well as the integration of the final result with file access over the Internet. We report several performance results and describe how the resurrected system and IDE was actually used to build an entire slide composition and presentation system used to produce a present-day illustrated talk.

1. Background

Smalltalk-76 was the first modern Smalltalk, combining a compilable keyword message syntax with a compact and efficient byte code interpreter (Ingalls 1978). From its precursor (Smalltalk-74) it inherited an object-oriented virtual memory (OOZE) which enabled it to address over a megabyte of objects with sixteen-bit pointers, and a library of line-drawing, bitmap manipulation and text display routines.

At roughly the same time, Intel was developing its 8086 microprocessor to be introduced in 1978, and the Xerox Smalltalk team (see acknowledgements) determined to build a portable computer dubbed "NoteTaker" (Figure 1) around that chip, running Smalltalk as its operating system and application environment.

The challenge of porting Smalltalk-76 to a microprocessor with only 256k bytes of memory led to several more innovations that shaped the future of Smalltalk. The need for efficiency led to an experimental mapping of Smalltalk contexts directly onto the 8086 stack. The overwhelming task of rewriting all the graphics routines from ALTO assembly code to 8086 assembly code motivated a rewrite of all the Smalltalk graphics to use only the single BitBlt primitive operation. By this time the Xerox team had learned enough about how to build an effective IDE, so this port was also an excuse to pare down the system to a fairly lean self-supporting kernel of 100 classes and 2000 methods in an image of 200k bytes.

The NoteTaker Smalltalk, also referred to as Smalltalk-78 (Krasner 1983) is thus a manifestation of the original Smalltalk-76 language and interpreter architecture, together with the hallmark Smalltalk IDE and the earliest BitBlt-based graphic system. Because of its freedom from the complex OOZE virtual memory and the equally complex support for lines, text, and image manipulation in the original Smalltalk-76 system, we determined to revive Smalltalk-78 as a living artifact accessible in any browser.

Figure 1. NoteTaker hardware

While only a few NoteTakers were ever built, we are fortunate to have one Smalltalk-78 memory image preserved from that day. It was created by cloning (parts of) the running Smalltalk-76 system. Since this represents a fairly lean snapshot of the original Smalltalk-76 system, we thought it would be a particularly interesting target for reimplementation. A similar cloning process was used to produce an image to run on Xerox's high-performance Dorado machine, and that in turn was stepwise morphed into the publicly released Smalltalk-80 system.[1]

2. Notable features of Smalltalk-78

The constraints on NoteTaker deployment led to a particularly interesting point in the evolution of Smalltalk that can be summarized by the following relationship to its parent:

Language and bytecode architecture Unchanged from the original Smalltalk-76 design (Ingalls 1978).

Dynamic execution model Retains full context semantics, but with a linear stack matched to 8086.

Object memory Complex OOZE virtual memory replaced by simple object table model.

[1] Remnants of this duality can still be seen in the NoteTaker image: lower-level code in various places tests whether it is running on the NoteTaker or Dorado.

Graphics system Large library of assembly code for text, lines, etc. replaced by BitBlt alone.

Programming environment Includes full rich text editor, compiler, paned browser and debugger.

Source code Decompilation used in place of remote source code file.

System size VM reduced from approx 16k in Smalltalk-76 to 6k bytes of 8086 code. Entire IDE was 100 classes with 2000 methods, totalling 200k bytes

3. Revival process

To revive Smalltalk-78 and make it usable in a browser, we determined to leverage the similar SqueakJS project that runs Squeak Smalltalk using n JavaScript in a browser (Freudenberg 2013). We would reuse the innovative object model as is, modify the interpreter and BitBlt for Smalltalk-78 differences, and completely rewrite the context mechanism to follow Smalltalk-78's linear stack model. The SqueakJS browser port was developed in the Lively Web development environment[2], and we took advantage of this support in our revival of Smalltalk-78 as well. Figure 2 shows the initial Notetaker screen.

3.1 The initial snapshot

The Smalltalk-78 snapshot we used was created from a running Smalltalk-76 system. To fit within the NoteTaker hardware, many features were removed from that system.

The snapshot came in two files: an object table dump (30,976 Bytes) and an object memory snapshot (172,592 Bytes). These files were not a direct snapshot, but the result of a conversion program in a running Smalltalk-76 system. These snapshot files were added to the Lively interpreter object as literal arrays. The object table is a sequence of 4-byte entries. Each entry encodes the data address, along with some other bits including a reference count. Our implementation ignores the reference count since it uses a generational garbage collector.

The object data space is a sequence of 2-byte words. One header word encodes the class oop in the upper 10 bits, and the instance size in the lower 6 bits (class oops always had the lower 6 bits equal to zero in OOZE). If the size field is zero, then there is a word before the class with a 16-bit length. The size field is the object size in bytes, including the class (and size), so a String of length 1 has size = 3, and a Point would have size = 6.

The format of classes is (quoting from the system itself . . .)

- title "<String> for identification, printing"
- myinstvars "<String> partnames for compiling, printing"
- instsize "<Integer> for storage management"
- messagedict "<MessageDict> for communication, compiling"
- classvars "<Dictionary/nil> compiler checks here"
- superclass "<Class> for execution of inherited behavior"
- environment "<Vector of SymbolTables> for external refs"

The instsize is an integer (i.e. low bit = 1) with the following interpretation:

- 0x8000 – fields are oops, else not
- 0x4000 – fields are words, else bytes
- 0x2000 – instances are variable length
- 0x0FFE – instance size in words including class

[2] Seehttp://lively-web.org/

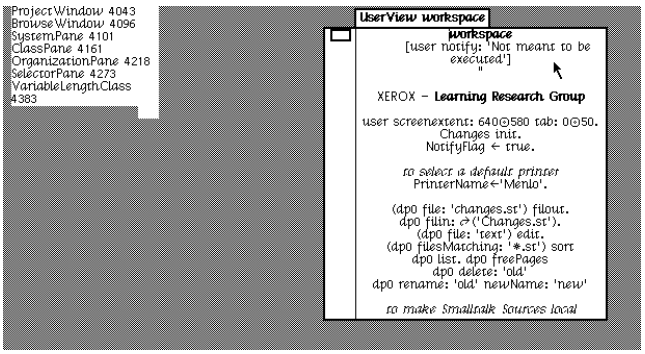

Figure 2. The initial NoteTaker screen.[3]

Thus Point has instsize = 0x8006 (class + 2 pointers) and Float has instsize = 0x4008 (class + 3 words). Floats on the NoteTaker use a non-standard format: 15 bits exponent in two's complement, 1 bit sign, and 32 bits mantissa. These floats are converted to standard IEEE 754 doubles (1 sign bit, 11 bits biased exponent, 53 bits mantissa) by bitshifts, resulting in native JavaScript numbers.

We discovered that some objects in the snapshot did not have the right class set. Specifically, we noticed that e.g. Array new: 5 failed with an inexplicable error. It turned out that the Array class was supposed to be an instance of VariableLengthClass which implements new:. But in the snapshot, it was an instance of Class. This problem did not manifest in the original NoteTaker VM because it intercepted new: as a primitive without actually looking it up. All other variable-length classes (String, UniqueString, Vector, Process, Natural, and CompiledMethod) had the same problem, so this probably came from a bug in the image cloning process.

3.2 The object model

Our VM mimics the Smalltalk-78 bytecode interpreter as faithfully as possible, but has an entirely different storage model from the original, borrowed from SqueakJS (Freudenberg 2013), which in turn was inspired by the Potato system, an implementation of the Squeak VM in Java (Ingalls 2008). Rather than emulating the original object layout scheme with an object table and a contiguous memory area for objects, it maps each Smalltalk object to a JavaScript object, and object references are simply fields of JavaScript objects. Tagged integers are mapped to JavaScript numbers, with typeof checks in places where the VM needs to distinguish objects from integers and floats.

The NoteTaker broke away from OOZE's object zones that placed uncomfortable limits on object size. Our VM goes further still by eliminating the complexities of reference counting. It allows more and larger objects (32K objects, virtually unlimited in size). It could easily be extended to allow even more objects (by representing oops with more than 16 bits in saved snapshots).

Instead of reference counting our VM uses a generational garbage collector as introduced by SqueakJS. Old objects are held in a linked list. New objects are not explicitly referenced, enabling them to be garbage-collected by the host GC of JavaScript. An object gets tenured when its oop is needed for the first time (typically for hashing). A full GC sweeping old and new space is rare, basically only for become operations and when snapshotting. This makes garbage disposal very efficient, since the vast majority of it is not handled explicitly but by the host language.

[3] The live system is available at `http://lively-web.org/users/bert/Smalltalk-78.html` (Freudenberg and Ingalls 2014). The initial snapshot can be loaded as "original image".

```
FIRST_TEMP: -1,// temps, followed by callee's stack
SAVED_BP: 0,   // rel link to caller's frame
CALLER_PC: 1,  // caller's supended PC
NUMARGS: 2,    // args were stacked right to left
METHOD: 3,     // method
MCLASS: 4,     // method class (needed for super)
RECEIVER: 5,   // top stack item in caller's frame
LAST_ARG: 6,   // stack item in caller's frame
```

Figure 3. Context frame layout (relative to BP)

```
MINSIZE: 0,     //
HWM: 1,         // not used
TOP: 2,         // top of stack rel to end
RESTARTCODE: 3, // code to run for restart
STACK: 4,       // bottom of stack ares
```

Figure 4. Process layout

3.3 Linear stacks

It was enough of a stretch to make a bytecode interpreter for Smalltalk perform acceptably on any machines available in 1978, but getting it to run on a 4MHz 8086 microcomputer required the Xerox team to explore every possible trick to improve speed. Because the 8086 offered special instructions for pushing and popping values on its stack, and also for switching from one frame of temporary variables to another, Smalltalk-76's general Context objects were recast into a series of overlapping context-like frames in the 8086's linear stack.

Figure 3 shows the layout of a Smalltalk-78 stack frame. The base pointer (BP) pointed to the base of the context frame, and this location held a link to the caller's base pointer, followed by the caller's supended PC. The remaining locations were similar to a normal Smalltalk context. Note that the stack grows downward toward lower addresses: arguments are first pushed, then the receiver, then the method info. This is followed by the caller's PC, and a link to the caller's BP, after which comes the state of the new context with temps first followed by the new context's stack cells. A conscious aspect of this design is that the layout of the top of the caller's frame is the same as the base of of the called frame, and these can simply be aliased so that no copying of receiver or arguments, nor allocation of a new context object is required to perform a normal Smalltalk message send.

The linear stacks themselves were Smalltalk objects of class Process, and these could be activated alernately to achieve a process switch. Each process object included a header that stored its current top of stack location, and it was possible to grow and shrink these variable-length objects as needed. Figure 4 shows the layout of a process object with many context frames in it, and Figure 6 shows an example.

The alert reader will recognize that it is not possible to emulate the operation of Smalltalk blocks with a linear stack alone. Smalltalk-78's RemoteCode objects (shown in Figure 5) provided the necessary added PC and remote return point needed to support out-of-line block execution, analogous to the RemoteContexts of Smalltalk-76. While use of remote code was no faster than in Smalltalk-76, almost all sends (well over 99 percent) in typical code ran using the overlapping stack frames.

3.4 BitBlt display

Smalltalk-78's BitBlt is relatively simple, supporting only black-and-white bitmaps, a 4x4 halftone pattern, four source rules (src, not src, halftone in src, halftone), and four combination rules (store, or, xor, and). Since each word stores 16 pixels, operations are relatively fast. For even more performance we use specialized inner loops, for example for filling, and for copying with the store rule.

To display the bits on the screen we use an HTML canvas. We create a JavaScript ImageData object, which can be displayed on the canvas in a single drawing call. It needs 32-bit RGBA data, which we create pixel-by-pixel from the bits in the display bitmap. Doing this for the full screen (1024x768 by default) would still be quite expensive. Instead, we only do it for the rectangle affected by each BitBlt operation. Moreover, we record these "dirty" rectangles and merge them if possible, only actually flushing to the canvas when needed. This has the nice effect of reducing flicker, since not every individual drawing operation is seen by the user.

```
FRAMEOFFSET: 0, // offset of my frame in process
STARTINGPC: 1,  // PC to start or restart
PROCESS: 2,     // my process
STACKOFFSET: 3, // my saved stack pointer
```

Figure 5. RemoteCode layout

```
[1970]  savedBP: 8
[1971] callerPC: 139
[1972]  numArgs: 0
[1973]   method: a CompiledMethod: Window>>eachtime
[1974]   mclass: the Window class
[1975] receiver: a BitRectEditor
[1976] temp3/t4: true
[1977] temp2/t3: a BitRectEditor
[1978] temp1/t2: 1
[1979]  savedBP: 6
[1980] callerPC: 9
[1981]  numArgs: 1
[1982]   method: a CompiledMethod: UserView>>run:
[1983]   mclass: the UserView class
[1984] receiver: an UserView
[1985]  arg0/t1: false
[1986]  savedBP: 5
[1987] callerPC: 31
[1988]  numArgs: 0
[1989]   method: a CompiledMethod: UserView>>run
[1990]   mclass: the UserView class
[1991] receiver: an UserView
[1992]  savedBP: 5
[1993] callerPC: 103
[1994]  numArgs: 0
[1995]   method: a CompiledMethod: Process>>run
[1996]   mclass: the Process class
[1997] receiver: a Process
[1998]  savedBP: 0
[1999] callerPC: 0
[2000]  numArgs: 0
[2001]   method: a CompiledMethod: Process>>goBaby
[2002]   mclass: the Process class
[2003] receiver: a Process
```

Figure 6. Example stack frames, as shown in the VM debugger. The method Process≫goBaby was used to bootstrap the system, it is only executed when starting up the original image, but remains the top stack frame. The receiver slot always overlaps between frames, and if there were arguments, those too (e.g. at index 1984/1985: false and self were pushed by UserView≫run, then run: was sent)

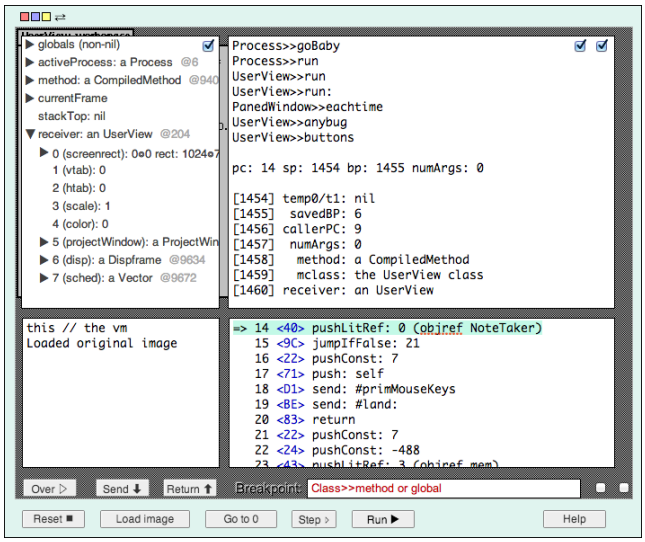

Figure 7. The debugger interface. It shows an object inspector in the upper left, an eval pane below it, the current stack in the upper right, the bytecodes of the current method in the lower right.

It was somewhat tricky to get the flushing right though: the original system did not use double-buffering[4]. We flush whenever an input primitive is called, because then we can assume that all intermediate drawing operations are finished. This works very well in general, but not for animations without user input checks. E.g. a "flash" operation reverses a portion of the screen twice. If we flush the screen after that, no change would be visible. So we had to modify the Smalltalk code by inserting a flush between the two reversals.

The canvas also shows the 16x16 pixel mouse cursor. We erase it from its previous position by drawing those pixels from the display bitmap just as after a BitBlt. Then it is shown at its new position.

3.5 Lively Debugging facilities

Having each Smalltalk object be a JavaScript object makes this VM convenient to debug using the Lively Web interface. Even before we had develped a nice VM viewer we could use Lively's inspectors and workspaces to interact with the VM. We soon added a bytecode disassembler and stack display to trace the execution, and a hierarchical inspector to explore object trees. These facilities make good use of the reflective nature of Smalltalk: While the VM normally does not care about instance variable names or the contents of selectors, we decoded them to make the debug display more meaningful (Figure 7).

Another helpful feature to get the system going were the decompiled sources. From independent work by Helge Horch, we had a complete set of decompiled source code, one class per file, and we attached to our debugger a little viewer that automatically jumped to the source code of a method when it was invoked. Since this used HTML text we had to map the unusual Smalltalk-78 characters to Unicode characters as best we could. Interestingly, not all the characters needed exist in Unicode[5]. We had to use some non-obvious mappings, like an open triangle for the open colon (Figure 8).

[4] Instead, it relied on specially selected "slow" phosphors in the cathode ray tube to reduce flicker.

[5] Perhaps there should be an effort to make them into official Unicode symbols? APL got its own section with all operators. We would need a white colon in particular, and perhaps an eyeball, quote, prompt and do-it chars

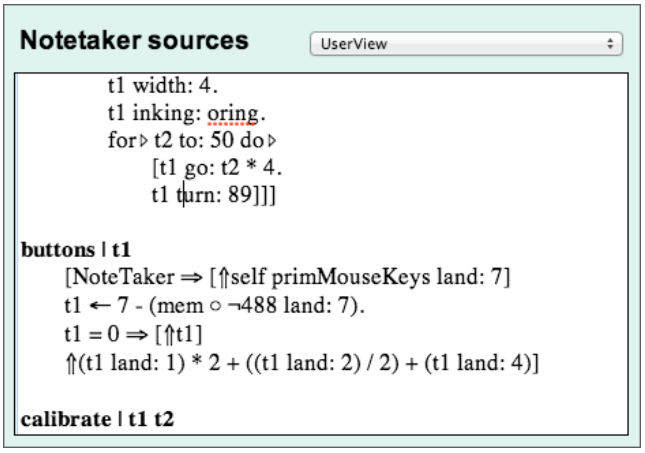

Figure 8. The sources view

4. Using a 36-year-old Smalltalk

4.1 Speed and Space

As the frequency of bugs dwindled, we were surprised how pleasant it was to use this old Smalltalk. This was partly a fortuitous result of the way in which we brought the system back to life. With the web browser came convenience and large clear bitmap graphics; with modern processors came more speed than the original native code; with our new object model most object size restrictions vanished, along with the need for any attention to reference counts.

Once things were running, there was still much work to be done, since Smalltalk-78 was never really finished. It was completed to the point of demonstration on the few NoteTakers that were actually built, but the machines were difficult to use with their small screens and marginal performance, and it was not easy to capture changes and feed them back into new releases. Originally, the Xerox group wrote a Smalltalk-78 image from a running Smalltalk-76 system and this was then moved to the NoteTaker and tried. After several iterations, one image worked well enough for demos. While a number of fixes were made and stored on NoteTaker floppy disks, those are long gone and they were never folded back into the snapshot we have.

4.2 Finishing the job

As our reimplementation became usable (more so than the original), the entire team began working in it as though they had just downloaded a completely modern tool. It was gratifying to see the original design validated in such a way.

We fell almost instinctively into the process of "finishing" this software. This included such tasks as ...

- Making a convenient mechanism for saving and distributing changeSets
- Making an automatic update system for installing newly released changeSets
- Fixing bugs (there were several)
- Removing unused methods
- Taking advantage of the considerable increase in speed. For instance finding all senders of a message had been so slow that it was done by executing a code snippet in a workspace. With the greater speed, it became natural to present such retrievals as menu commands

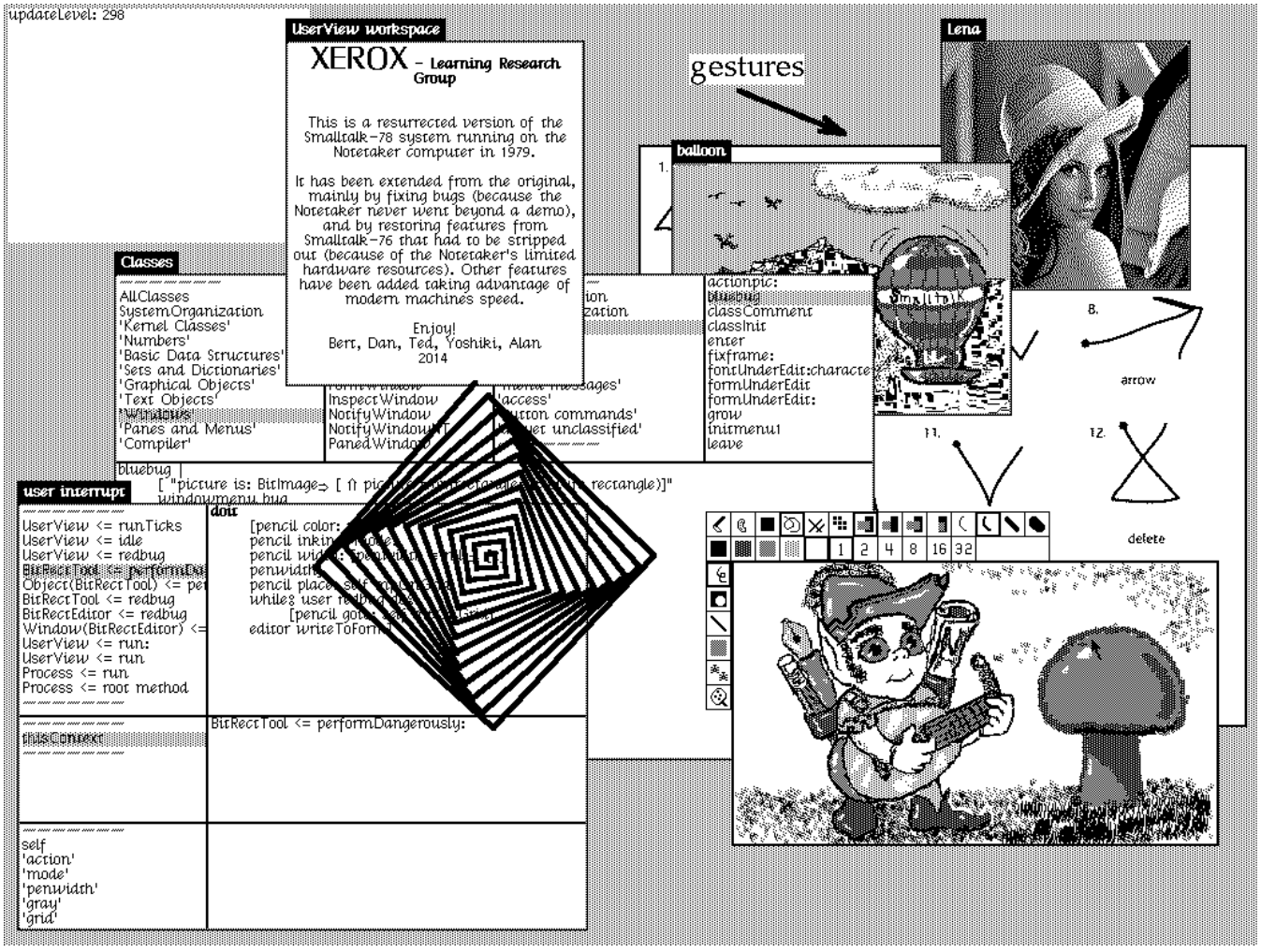

Figure 9. Various tools in the updated Smalltalk-78 system

- Completing the support of the excellent Smalltalk-76 debugger which had never been made to work completely with Smalltalk-78's linearized stack
- Recovering source code from an independently recovered Smalltalk-76 file

4.3 Recovering the source sode

Source code for a method in the NoteTaker had to be decompiled from the bytecodes of the method due to limited memory space. Decompiled code lacks meaningful names for temporary variables and it is also devoid of comments. Since the Smalltalk-78 snapshot was mechanically generated from a Smalltalk-76 most methods are identical to their Smalltalk-76 parents. With plenty of space avilable in the revived Smalltalk-78, we made an effort to restore full source code.

We were fortunate to find one file of source code for Smalltalk-76, although we had no way of knowing how well it matched our Smalltalk-78 snapshot. The file was simply a concatenation of all the methods with no indication of what class they came from, and only separated by arcane markers from a bygone text editor. We managed to isolate the methods and determine their classes in most cases, and then read them with an importer that would only accept methods if they generated the same bytecodes (actually if they produced the same decompilation) as the corresponding methods in our snapshot. (See Appendix B for the details.)

An immediate benefit from decoding the sources file was that we were able to import Kaehler's BitRectEditor, a tool similar to MacPaint, but developed in 1975 in an earlier Smalltalk.[6] The BitRectEditor has programmable tools, each composed of a texture ink, a BitBlt mode, and a nib. When a tool is selected, its components are shown in the top menu. A tool can be reconfigured by clicking and new tools can be created on the fly.

4.4 Life in the Cloud

With Smalltalk-78 running in the browser, work within the system became much more productive, but access to external files for reading and writing was actually more difficult than before. Here we were able to take advantage of hosting in the Lively Web to make access to changes files, snapshots, and image resources actually easier than before.

[6] In the earlier Smalltalks, the whole screen was too large to fit into a single Smalltalk object. Therefore images were stored as BitRects—objects that held striped data in 2k-byte chunks (which was optimal for OOZE). The image painting tool (BitRectEditor) would paint the BitRect's bits on the screen, do the editing using BitBlt on the screen only, and scrape the bits back into stripes in the BitRect when done.

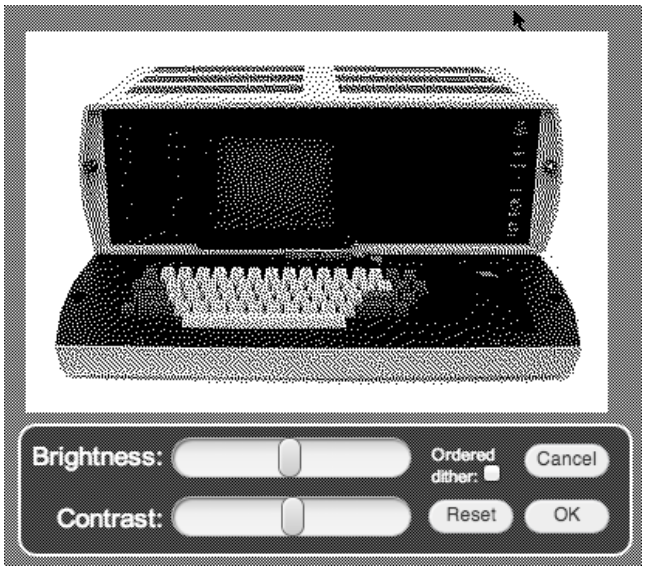

Figure 10. Bitmap importer with dithering UI

Files We reappropriated the existing "port" primitive (which had been used for file i/o) to implement a simple string-based file interface. It takes a file name string and a file contents string and stores them in a JavaScript dictionary. It is also written to the web browser's `localStorage`, which survives reloading of the browser page and thus provides persistence. A list of files is returned when passing an empty file name. These (and more) methods are provided by the user global, the current UserView instance:

```
1  user fileString: 'temp.txt' ← 'foo␣bar␣baz'.   "create file"
2
3  user fileString: 'temp.txt'        "retrieve file"
4          ⇒ 'foo␣bar␣baz'
5
6  user fileString: 'temp.txt' ← nil.      "delete file"
```

Files can be imported by drag-and-drop and are then available to Smalltalk.

Bitmaps A special case is importing bitmap files like JPEGs or PNGs. The NoteTaker code obviously can not load these files directly, as the formats had only been invented decades later. It did, however, define a binary serialization format (asInstance and fromInstance:) for black-and-white forms. This consists simply of the instance variables of the Form, a few Integers followed by a String for the bits. The Integers are stored as 16 bit big-endian numbers, the String has a one or two byte header encoding the length followed by the bytes. To support larger forms we extended this to a four-byte header. When we drop a bitmap into the browser, the system presents a Lively user interface for reducing the color range to black and white. It supports both error diffusion (Floyd/Steinberg algorithm) and ordered dithering. The user can adjust contrast and brightness, then a form file is generated and stored, which can be loaded from inside Smalltalk (Figure 10).

Networking The same VM primitive as for accessing local files is used to store files on the server and retrieve them. If the filename starts with "http:" then instead of storing it locally, we access it via a WebDAV server provided by Lively.[7] If the filename ends in a slash, a list of file names in that directory is returned. This enables seamless working in the cloud or locally.

[7] It is amusing to use the term "http" in a system that predates the invention of HTTP.

Table 1. Interpreter performance, measured on a laptop with a 2.2 GHz Intel Core i7 CPU

	Bytecodes/sec	Sends/sec
Chrome 35.0	1,600,000	70,000
Firefox 30.0	6,900,000	110,000
Safari 7.0.4	9,300,000	350,000

Update stream Using the network file access we set up an update mechanism to distribute changeSets. Each changeSet is a separate file. There is an index file named "updates.list" containing a list of all update file names. The image maintains its own number of loaded updates, so it knows how many new updates need to be loaded from the list. There are now already hundreds of changeSets in that stream. Because loading them from the start takes a long time, we also provide an updated image that is automatically downloaded when a user starts Smalltalk-78 for the first time.

4.5 Performance

We ported the tinyBenchmarks from Squeak to Smalltalk-78 for measuring raw bytecode and send speeds. Results are shown in Table 1. Performance depends considerably on the web browser's JavaScript VM. For this particular workload, Safari's Webkit JIT compiler outperforms Chrome's V8 engine by orders of magnitude, with Firefox in the middle.

In reality we throttle the interpreter when it is idle, as the interpreter speed is entirely sufficient. Idle detection works by measuring how often the image calls the input primitives. When it is rapidly reading the mouse and keyboard without the user providing input, we let the VM sleep for up to 200 ms, or until a user event arrives. Thus when the image is busy with some longer operation it will not check for user input, and thus will not be throttled. As soon as the user types something or moves the mouse, the VM resumes at full speed, and keeps going for at least 500 ms before throttling again.

Note that "full speed" here does not actually mean the interpreter runs continuously. Unlike a regular Squeak VM which has a main-loop that is only exited when the application quits, the Smalltalk-78 vm is callback-based, following the design of SqueakJS. That is, the bytecode interpreter loop runs for a limited time only (typically 20 msecs) and then returns control to the web browser. This is necessary so that the web browser can update the screen, which does not happen while JavaScript is executing. Similarly, events can only be processed while no other JavaScript function is active. Once the interpreter loop finishes, a timer event is scheduled to restart the interpreter loop. When throttling, the timeout is 200 ms. When not throttling, it is 0 ms, meaning to call back into the interpreter as soon as possible.

To get the lowest possible delay between user actions and display response, we not only run the VM at full speed while user events arrive. We also break out of the interpret loop early, as soon as something was drawn to the screen (see section 3.4). Otherwise the interpreter would continue using its current time slot and only then return control to the browser. This would delay updating the screen noticeably, and make the system feel sluggish.

4.6 A real test drive

As an experiment to validate not only the underlying design and development system, but also the ability of the system to support an as yet unanticipated application, we built a fairly capable PowerPoint-like presentation system (all in 1-bit graphics, of course). Two example screenshots appear as Figure 11 and Figure 12. The first of these is performing the role of a slide sorter, with a clever BitBlt scheme

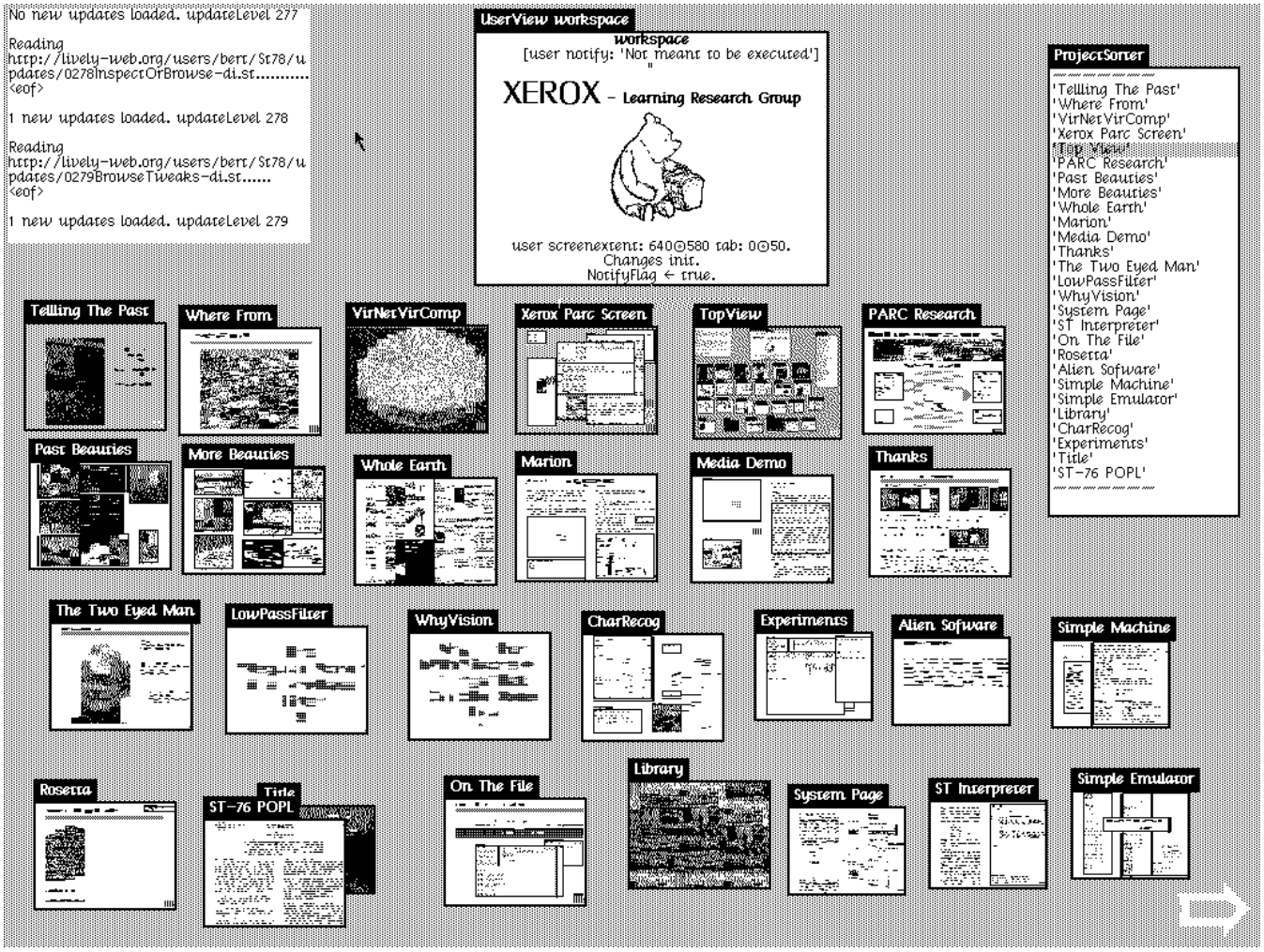

Figure 11. Presentation slides

to produce shrunken thumbnail images on the fly. The second shows one of the slides during a full-screen presentation.

While resurrected software systems are often fragile, we were impressed by the robustness of Smalltalk-78. Our reimplementation had a completely different object memory and a completely different Context discipline, but remained extremely stable throughout our work on "finishing" the system and building this presentation system.

As an example, besides the normal "builds" required in a presentation system, we wanted to support multiple concurrent animations running on the screen while editing and other operations were being done. This required a rework of the window scheduler to provide a queue of ticking objects, and for all existing idle loops to yield to the scheduler in this regard. All of these changes were made in a couple of hours in the running system with very little problem. The picture of the ball in Figure 12 is constantly bouncing (and appearing squashed when it hits the ground) even when other text is being edited.

As mentioned earlier, some interesting features were stripped out in the Smalltalk-78 image from the original Smalltalk-76 image. Most notably, a pen stroke gesture recognizer was missing. To demonstrate the richness of experiments, we have reimplemented a stroke gesture recognizer. The implementation is based on a modern algorithm called $1 recognizer (Wobbrock et al. 2007) rather than the one that was used in the Smalltalk-76 system. The $1 algorithm heavily relies on floating point number computation, but by writing a few primitives to support the algorithm, it works responsively.

5. Things we learned

Small is beautiful. Systems like Smalltalk that are self-describing are highly leveraged. This made it possible to implement Smalltalk-78 in only 6k of 8086 assembly code on the NoteTaker. Similarly we were able to get the system running in a browser with only roughly 3,000 lines of JavaScript. This number grew to 4,000 as we added various comforts such as the support for web-based file access, but the kernel remained small. This same leverage made it a fun project, as we were able to see "bits on the screen" after only about 4 man-weeks of work.

Speed is nice. The improved performance of our implementation over the original made this an exciting project as well. Many facilities that had been barely usable on the NoteTaker and its parent Smalltalk-76 system were delightfully responsive, and the system therefore surprisingly productive.

Clean object API. As with most Smalltalk systems, Smalltalk-78 had a clean interface to storage, and very little work ws needed to completely change from a reference-counted object table model

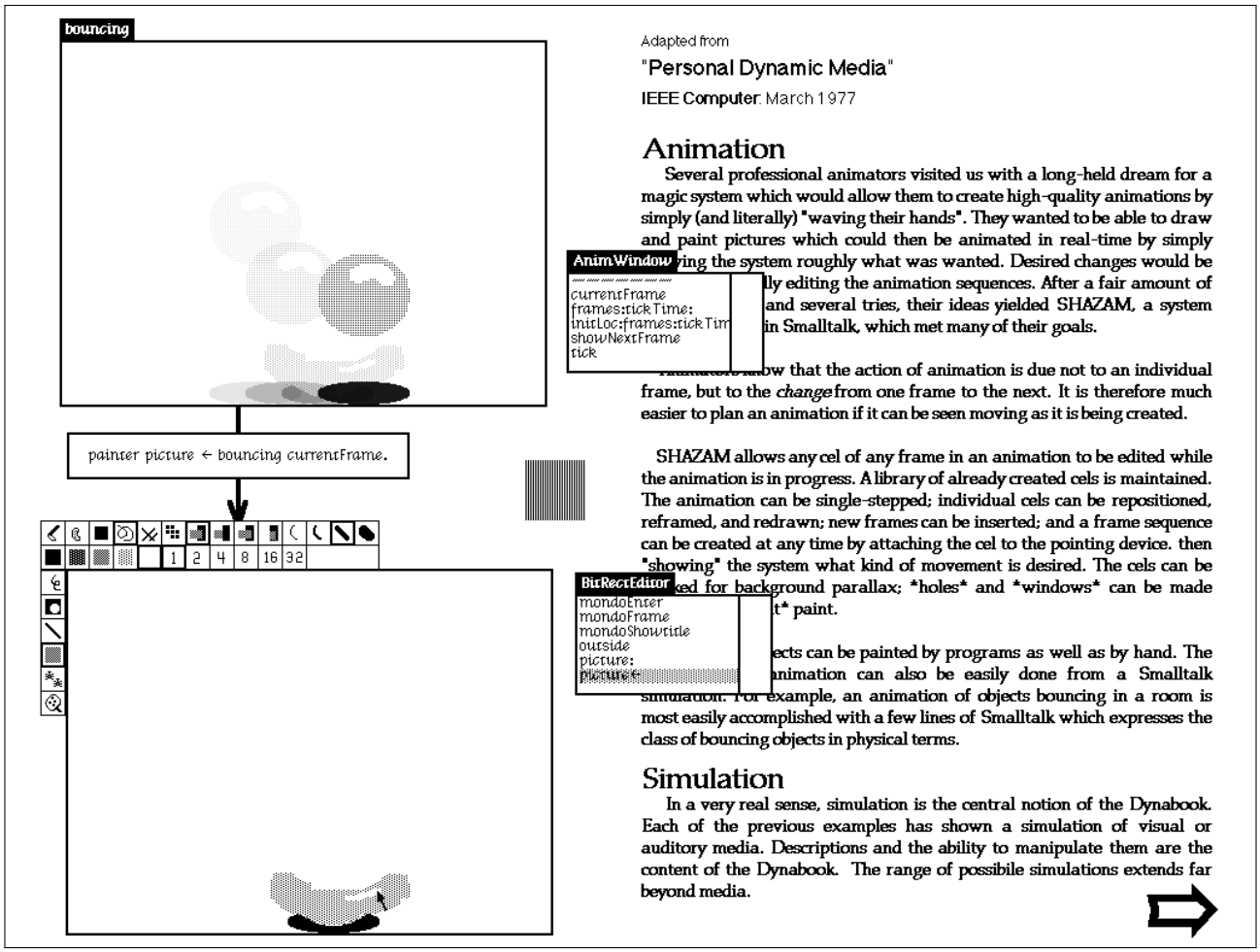

Figure 12. Editing a slide: An animation frame is repainted while the animation is playing. This feature is not "built-in", but was added on-the-fly by connecting two objects with one line of code. (This figure shows a composite of four successive screenshots)

to a direct pointer garbage-collected model. We were, of course, fortunate to inherit a relatively complete JavaScript Smalltalk object model from SqueakJS (Freudenberg 2013).

Browser and Cloud as a universal platform. Finally we learned through this and the earlier SqueakJS project how to adapt the earlier file-based Smalltalk systems to take advantage of the conveniences of browsers and web-based storage facilities. Much of this work was facilitated by our use of the Lively Web development environment, although our completed Smalltalk-78 artifact can operate entirely on its own.

6. Related work

In 2004 Helge Horch got our same Smalltalk-78 snapshot from Dan Ingalls, along with the original 8086 code listings. From this he wrote a relatively complete resurrection in Java, that is yet to be published.

Our Smalltalk-78 VM is based on Bert Freudenberg's "SqueakJS" VM (Freudenberg 2013). It shares the overall design, and parts of the implementation. For example, our BitBlt is a simplification of SqueakJS's BitBlt. SqueakJS in turn was inspired by Dan Ingalls's "Potato", a Squeak VM written in Java (Ingalls 2008).

Another Smalltalk that now runs in a web browser is Smalltalk-72 via Dan Ingalls's Alto emulator (Ingalls 2013). A major difference to our approach is that this emulates Alto machine code which then executes the interpreter, rather than building a new interpreter running the Smalltalk bytecodes.

There are various attempts to implement different languages for the web-browser. Among those, Amber[8] is notable for being a Smalltalk dialect implemented in JavaScript. Most of such languages are implemented as a translator from the language to JavaScript. The key difference in our approach is that we implement a virtual machine that is compatible with the actual old one; this allows us to revive the exact system.

Acknowledgments

The original Smalltalk-78 implementation was done in 1978 mainly by Dan Ingalls and Ted Kaehler (with BIOS by Bruce Horn) at Xerox PARC, and never written up except in the introduction of "Smalltalk-80: Bits of History, Words of Advice" (Krasner 1983, p. 17). A decade later, "The Early History of Smalltalk" (Kay 1993) gave an overview of early Smalltalks, including the NoteTaker version.

The NoteTaker hardware was designed and built by Doug Fairbairn, with help from the hardware support group at PARC. Engineers at Xerox El Segundo debugged the boards. Some NoteTaker

[8] `http://amber-lang.net/`

firmware, schematics, and memos are at `http://bitsavers.org/pdf/xerox/notetaker`

The Smalltalk font in the screenshots is *Cream* by Bob Flegal, based on a design by Alan Kay.

We are grateful to Al Kossow and Dave McDougall for preserving a number of Alto diskpacks, of which the NoteTaker Smalltalk export disk was one. Al and Larry Stewart moved the files to modern media, and they are now well preserved.

References

B. Freudenberg. SqueakJS, 2013. `https://github.com/bertfreudenberg/SqueakJS`.

B. Freudenberg and D. Ingalls. Lively Smalltalk-78, 2014. `http://lively-web.org/users/bert/Smalltalk-78.html`.

D. Ingalls. The Smalltalk-76 programming system design and implementation. In *Proceedings of the 5th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '78, pages 9–16, New York, NY, USA, 1978. ACM. . URL `http://doi.acm.org/10.1145/512760.512762`.

D. Ingalls. JSqueak/Potato, 2008. Original site: `http://web.archive.org/web/20080729033427/http://research.sun.com/projects/JSqueak/`, current home: `https://www.hpi.uni-potsdam.de/hirschfeld/projects/potato/index.html`.

D. Ingalls. Lively Alto emulator running Smalltalk-72, 2013. `http://lively-web.org/users/Dan/ALTO-Smalltalk-72.html`.

A. C. Kay. The early history of Smalltalk. In *The Second ACM SIGPLAN Conference on History of Programming Languages*, HOPL-II, pages 69–95, New York, NY, USA, 1993. ACM. ISBN 0-89791-570-4. . URL `http://doi.acm.org/10.1145/154766.155364`.

G. Krasner. *Smalltalk-80 : bits of history, words of advice*. Addison-Wesley Pub. Co, Reading, Mass, 1983. ISBN 0201116693.

J. O. Wobbrock, A. D. Wilson, and Y. Li. Gestures without libraries, toolkits or training: A $1 recognizer for user interface prototypes. In *Proceedings of the ACM Symposium on User Interface Software and Technology (UIST '07)*, pages 159–168. ACM, 2007.

A. Smalltalk-78 Syntax

The syntax of Smalltalk-78 is unchanged from Smalltalk-76, which introduced the regularity of unary, binary, and keyword messages that is still used in modern Smalltalks. Readers may refer to the Smalltalk-76 reference for a full description. There are no blocks as in Smalltalk-80, no booleans, and no meta classes.

The system uses the ASCII-1963 character set (which had ↑ and ← arrows in place of the ^ and _ characters in later ASCII versions). From the control character range, only Tab (9) and Carriage Return (13) are used as in ASCII, while some others are used for special characters not found in ASCII. These include return and implication arrows (⇑ ⇒), comparison operators (≤ ≥ ≠ ≡), indexing and point creation operators (∘ ⊙), a curved arrow for literal quoting (used like # in Smalltalk-80), a unary minus (for prefixing negative numbers, as in 3 − -4), a possessive operator (evaluates an expression in the context of another object, e.g. obj 's 'instvar'), and a few more. See Figure 13 for a complete list. This code snippet also demonstrates a few idiosyncrasies:

Conditional execution A block of code can be conditionally evaluated using the ⇒ operator. It maps directly to the "jump if not false" bytecode (where false is just an instance of Object known to the VM compared by identity). If the condition is false, the following block of code is skipped. Any object other than false causes the conditional jump not to be taken, so the code block is executed, followed by an unconditional jump to the end of the surrounding code block. This allows putting an "else" case right after the conditional block. If execution should resume after the else case, another set of brackets needs to be put around it, as demonstrated in Figure 13.

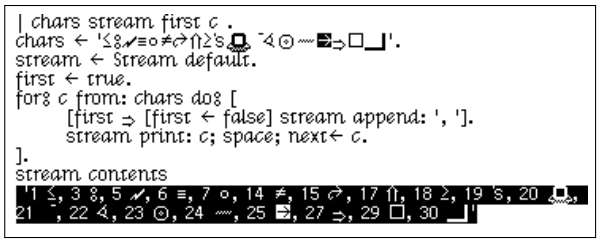

Figure 13. Non-ASCII characters used in Smalltalk-78

Deferred evaluation Brackets are merely a syntactic device to group statements, they do not create a block object as in Smalltalk-80. How then does e.g. "for:from:do:" work? The magic is *not* in the brackets, but in the two variants of the colon in a keyword message. The regular colon ":" causes the keyword argument to be evaluated immediately. For an open colon ":" however, the compiler generates a "remote copy" of the current context. This RemoteCode can be evaluated later, and even repeatedly if desired.

There are two variants for evaluation, with and without an argument. The form with an argument (value←) can be used to assign a value to a variable, the form without an argument (eval) just evaluates the code. These work together, as there is no way to pass arguments to a block other than "remotely" assigning to temporary variables. This is perhaps best explained with an example. A possible implementation of "for:from:do:" could look like this:

```
1 for: var from: vec do: code | stream item
2 [
3     stream ← vec asStream.
4     while: [item ← stream next] do: [
5         var value ← item.
6         code eval.
7     ]
8 ]
```

Note that "for:from:do:" is not actually a method, but a compiler macro (as evidenced by the missing receiver). It could, however, be a regular method as just described.

Assignment selector The last part of a keyword selector can be an assignment arrow. E.g. "stream next← 5" writes an item to the stream, whereas "stream next: 5" reads 5 items from the stream. To the method this is just like any other argument. But on the sending side, the parser treats the expression after ← as if it was an assignment. That means no parentheses are needed around that expression. This also works for binary operators extended by an assignment arrow, making it a ternary operator. E.g. "a ∘ 1 ← b ∘ 1" uses the ∘ and ∘ ← operators. It is equivalent to Smalltalk-80's "a at: 1 put: (b at: 1)" but reads a lot nicer and needs no parentheses.

No metaclasses The expression "Stream default" looks like it invokes a "class-side method" if it was Smalltalk-80. But Smalltalk-78 did not have a metaclass hierarchy in parallel to the regular class hierarchy. All classes are an instance of Class so no class-specific methods are possible. Instead, Class provides a couple of methods that dispatch to a new instance. E.g. the implementation of Class≫default is "⇑ self new default". In the Stream case, this invokes Stream≫default which initializes the stream for writing on a new String.

B. Source code recovery

The Smalltalk-76 sources file we found was "Smalltalk.Sources.5.5k" from November 22, 1980. Methods in the file have no class name associated with them. Instead, every method inside a running ST-76 has an offset pointing the beginning of its text in the sources file. Unlike a Smalltalk-80 sources file, a method from the Smalltalk-76 file is not ready to be 'filed in' to a running Smalltalk. The methods in Smalltalk-76 sources are grouped by class, but we did not always know which class it was. It was also hard to tell where one class ended and another began.

The first thing we did was to build a table of all methods in the 5.5k sources file in a modern Squeak. The table had an entry containing the selector, the method source text, and a space for the class. We could write out any table entry as an expression that could install that method in Smalltalk-78. Freudenberg arranged that dropping a file on a web browser running the interpreter placed the file in a list that Smalltalk-78 could see. From a menu inside the running Smalltalk-78, we could "file in" that file.

The selective importer let us bring in source code without changing the bytecodes, but we still worried about two methods in different classes having wildly differing comments. Methods named comment were a prime example, but also +, −, /, =, copy, to:, and printOn:. There were 89 of these ambiguous selectors in Smalltalk-78. We wrote out every (class, selector) pair and included a crude hash for its bytecodes. This allowed us to detect same-name methods where both versions would pass the decompile test.

For the first round, we wrote out all of the methods in Smalltalk-76 that would be non-ambiguous in Smalltalk-78. We still did not know which of several implementations of a selector in Smalltalk-76 was the right one. We simply wrote out all of the implementations! For a given selector, we sent every source version to every class that had it. We depended on the importer to accept only the correct version for each class. This worked. We collected the accepted methods in changeSets and wrote them to files. We sent 1644k of source, of which 241k was accepted.

Back in Squeak, we parsed the Smalltalk-78 fileouts to discover what class had claimed each version of a method. We wrote those class names into the table of Smalltalk-76 methods.

How could we get the right class attached to source code for the ambiguous selectors? The code for each class was contiguous in the Smalltalk-76 sources file. We assumed that for every ambiguous selector, at least one other method in that class was unambiguous. If we started at an ambiguous method, and looked in both directions in the Smalltalk-76 table, we would come across a class name in each direction. If the two classes were different, we simply sent two copies of the method to the merge test. We sent 20k of sources to Smalltalk-78 for the ambiguous selectors, and 16k was accepted.