

# A Tour of the Squeak Object Engine

Tim Rowledge, tim@sumeru.stanford.edu

## *Introduction*

This chapter is intended to explain some basics of how a Virtual Machine (VM) works, why a VM is useful, what it does for the Squeak programmer and user, and how the Squeak VM might develop in the future.

## **What is a Virtual Machine and why do we need one?**

A Virtual Machine provides us with a pretense of being a machine other than the actual hardware in use. Using one allows systems that behave differently than the host hardware to run as if on hardware designed for them. The term *Object Engine* is less commonly used but is a useful concept that includes the lowest system areas of the language environment running on the VM. Since there is often some flux in the definition of which components are within the actual VM and which are part of the supported environment, Object Engine is useful as a more inclusive term.

The term *Virtual Machine* is used in several ways. When IBM refer to *VM/CMS* they are referring to a way of making a mainframe behave as if it is many machines, so that programs can assume they have total control even though they do not. Intel provide a somewhat similar facility in the x86 architecture(?), referred to as *Virtual Mode*. This sort of VM is a complete hardware simulation, often supported at the lowest level by the hardware.

Another sort of VM is the emulator - *SoftWindows* for the Mac, Acorn's *!PC*, Linux's *WINE* are good examples - where another machine and/or OS is simulated to allow a Mac user to run Windows programs, an Acorn RiscPC or a Linux machine to run Windows98 programs and so on. Emulators of games consoles, such as *Bleem*, are also popular, if a little legally contentious.

Many languages and some even applications are effectively built on a VM. **BASIC**, for example, is probably the most numerous VM. The BASIC interpreter does not just interpret the BASIC language but provides a varying amount of runtime support code depending on the precise system. PERL is another popular language that uses a similar form of VM.

We shall focus in this chapter on the form of VM used for Smalltalk. Many of the general principles apply equally well to other Smalltalk systems, and often to other dynamic languages such as Lisp, Dylan and even Java.

## ***The basic functionality of a Virtual Machine***

In a Smalltalk system all we can do is

- create objects,
- send them messages
- and let them die, at which point they are garbage collected.

All the rest is simply implementation details; however, those details are endlessly fascinating to VM hackers.

## **Creating Objects**

Unlike structures in C, Pascal, FORTRAN et al. , Objects are not simply chunks of memory to which we point, and so we need something more sophisticated than the C library **malloc()** function in order to create new ones.

Smalltalk creates objects by allocating a chunk of the object memory and then building a header that provides the VM with important information such as the class of the object, the size and some content format description. It also initialises the contents of the newly allocated object to a safe, predetermined, value. This is important to the VM since it guarantees that any pointers found inside objects are proper object pointers (oops) and not random memory addresses. Programers also benefit from being able to rely on the fresh object having nothing unexpected inside it. If a pointer object were not initialised the garbage collector would be prone to

attempting to collect random chunks of memory -- any experienced C programmer can tell you tales of the problems that random pointers cause.

Smalltalk allows for three kinds of object, which can contain object pointers (*oops*), machine words or bytes. Objects containing oops are initialised so that all the oops are nil and objects containing words or bytes are filled with zeros.

## Message sending

In order to do anything in Smalltalk, we have to send a message.

Sending a message is quite different to calling a function; it is a request to perform some work known by the message's name rather than an order to jump to a particular piece of code. This indirection is what provides much of the power of Object Oriented Programming, since it allows for encapsulation and polymorphism by allowing the recipient of the request to decide on the appropriate response. Adele Goldberg has neatly characterised this as "Ask, don't just touch".

To send a message to a receiver, the VM has to {diagrams?}:-

- I. find out the class of the receiver
- II. lookup the message in a list of messages understood by that class (the class's *MethodDictionary*)
  - A. if the message is not found, recursively repeat this in successive superclasses of the receiver
  - B. if no class in the superclass chain can understand the message, send a *doesNotUnderstand:* to the receiver so that the error can be handled in a manner appropriate to that object.
- III. extract the appropriate compiled method from the *MethodDictionary* of the class where the message was found
- IV. check for a primitive associated with the method
  - A. if there is a primitive, execute it.
  - B. If it completes successfully, return the result object directly to the sending Context.
  - C. Otherwise, continue as if there was no primitive called.
- V. establish a new execution context, setting up the pc, sp, sender, home etc
- VI. swap to that new context.
- VII. continue running in the new context.

In a typical system it often turns out that the same message is sent to instances of the same class again and again; consider how often we use arrays of **SmallInteger** or **Character** or **String**. To improve average performance, the VM can cache the lookup result. If the combination of the message and the receiver's class are found in the cache, we can avoid the full search of the *MethodDictionary* chain. Squeak currently uses a fairly simple and small global cache with two levels of reprobng if the initial probe misses; this reprobng reduces some of the yo-yo effect that can happen when the same message is sent many times to instances of two classes - imagine a collection with many alternating **SmallIntegers** and **LargeIntegers**. VisualWorks and some other commercial Smalltalks use inline caching, whereby the cached target and some checking information is included inline with the dynamically translated methods. Although more efficient, it is more complex and has strong interactions with the details of the cpu instruction and data caches.

## Primitives

Primitives are a way to improve performance of simple but highly repetitive code or to reach down into the VM for work that has to be done there - they are where a great deal of work gets done and typically a Smalltalk program might spend around 50% of execution time within primitives.

Primitives handle activities such as:-

Input/Output

## A Tour of the Squeak Object Engine

The input of keyboard and mouse events normally requires very platform specific code and is done in primitives such as **Interpreter>primitiveKbdNext** and **primitiveMouseButtons**. In Squeak, these in turn call functions in the platform specific version of **sq{platform}Windows.c** and associated files.

### Arithmetic

The basic arithmetic operations for SmallIntegers and Floats are implemented in primitives 1-17 (SmallIntegers) and 40-59 (Floats). There are also primitive number slots for LargeInteger primitives currently awaiting a Squeak implementation.

### Storage handling

In order to create an object we need a primitive (`primitiveNew` for objects with only named instance variables and `primitiveNew:` for objects with indexed variables) to allocate and initialise the memory for it. Reading from and writing to arrays is done by primitives, as is reading or writing to Streams.

### Process manipulation

Suspending, resuming and terminating Processes, as well as the signalling and waiting upon Semaphores are all done in primitives. Since changing structures such as the process lists and semaphore signal counts needs to be performed atomically, primitives are used to ensure no deadlocks occur.

In general it is best not to use primitives to do complex or heterogenous things; after all, that is something that Smalltalk is good for. A primitive to fill an array with a certain value might well make sense, but one to fetch, process and render an entire webpage would not! Primitives that do a great deal of bit-twiddling or arithmetic also make sense, since Smalltalk is not particularly efficient at that. See the Sound buffer filling primitives (`PluckedSound>mixSampleCount:into:startingAt:leftVol:rightVol:` is a good example) and Balloon graphics primitives such as the B3DEnginePlugin 'primitive support' protocol for examples.

## Garbage collecting

One of the most useful benefits of a good object memory system is that unwanted, or *garbage*, objects are collected, killed and the memory returned to the system for recycling. There are many schemes for performing this function, and quite a few PhD's have been awarded for work in this area of computer science. Dr Richard Jones keeps an impressive bibliography of garbage collection literature on his website at

<http://www.cs.ukc.ac.uk/people/staff/rej/gcbib/gcbib.html>

Essentially garbage collection (GC) is a matter of having some way to be able to tell when an object is no longer wanted and then to recycle it. Given some known root object(s) it is generally assumed that we can trace all possible accessible objects and then collect all those *not* accessible. If you read 'Smalltalk-80: The Language and its Implementation' [Goldberg Robson83] you will see that the exemplar system used a *reference counting* system combined with a *mark-sweep* system.

### Reference Counting

Reference counting relies on checking every store into object memory and incrementing a count field (often contained in the header) of the newly stored object and decrementing the count of the over-written object. Any object whose count reaches zero is clearly garbage and must be collected. Any objects that it pointed to must also have their count decremented, with an obvious potential for recursive descent down a long chain of objects.

One problem with a simple reference counting scheme is that cycles of objects can effectively mutually lock a large amount of dead memory since every object in the cycle has at least one referer. {diagrams}

Another problem is that *every* store into object memory will need the reference incrementing and decrementing to be done, even pushes and pops on the stack. Peter Deutsch [Deutsch Schiffman84] developed an improved way of dealing with reference counting the stack, known as **deferred reference counting**. This technique allows us to avoid reference counting pushes and pops by keeping a separate table of objects where the reference count might be zero but that might be on the stack -- if they are on the stack the actual count cannot be zero. At suitable intervals we scan the stack, correct the count values of the objects in the table and empty the table. Reference counting is still done for all other stores, and those objects that appear to reach a count of zero are added to the table.

## A Tour of the Squeak Object Engine

To avoid completely running out of memory due to cycles, most systems that use reference counting also have a *mark-sweep* collector that will start at the known important root objects and trace every accessible one, leaving a mark in the object header to check later. Once the marking phase is completed, every object marked can be swept up (or down) the memory space, effectively removing all the untouched ones, and the mark removed. There are important details to consider with respect to object cycles, which objects are actually roots, how to move the objects memory and update all the referers to it and so on. One useful side effect of the mark-sweep collector is that it compacts all the objects and leaves all the free space in a contiguous lump. In most cases this can be used to make subsequent memory allocation simpler. {diagrams}

### **Generation Scavenging**

The most commonly used garbage collection scheme in current commercial Smalltalk systems is based on the generation scavenger idea originally developed by David Ungar [Ungar 87]. He noticed that most new objects live very short lives, and most older objects live effectively forever. By allocating new objects in a small *eden* area of memory and frequently copying only those still pointed to into another small *survivor* area, we immediately free the eden area for new allocations. Subsequently we copy surviving objects from this area into another survivor area and eventually into an *old* space where they are considered *tenured*. Various refinements in terms of numbers of survivor generations and strategies for moving objects around have been developed. {diagrams}

An important part of a generation scavenger is a table of those old objects into which new objects have been stored. Each time an object is stored into another, a quick check is done to see if the storee is new and the storand is old; this is simpler than the check needed by a reference counting system. If needed, the storand object is added to the table, sometimes referred to as the remembered set. This table is used as one of the roots when the search for live objects is performed; each of the objects is scanned and new objects pointed to are scavenged. Any object that no longer points to new objects can be removed from the table. Once all the new objects pointed to by the table and other roots have been scavenged, these 'survivors' are also scanned and any new objects found are also scavenged. The process continues until no further objects are found.

Note that scavenging a generation is similar to a mark and sweep of the memory in which the generation lives, though the objects can get moved to a different space rather than simply compacted within the same space.

### **Why is an Object Engine better than C and a library?**

One might reasonably ask why this rather complex seeming system is better than a 'normal' programming environment and a C library.

The most obvious answer is that the message sending support allows, helps, and strongly encourages an object oriented form of programming. Good object oriented programs form extremely useful libraries that can increase programming productivity greatly. Although it is possible to use almost any language to work in an object oriented style, doing so without proper support makes it much harder than just 'going with the flow' and slipping into procedural programming.

### **Memory handling**

Much of the code in a complex C program is taken up with storage management; memory allocating, freeing, initialising, checking and error handling for when it goes wrong! A good VM will handle all this automatically and reliably (making some assumptions about the quality of the VM) and transparently. Although many C programs do not bother to check the bounds of arrays, exceeding those bounds is a major cause of problems. Consider how many viruses are spread through or otherwise rely upon buffer overflow and other bounds problems. A Smalltalk VM checks the bounds of any and all accesses to its objects, thus avoiding this problem completely. Furthermore, since an object is absolutely a member of its class (there is no concept of 'casting' in Smalltalk) you cannot fool the system into allowing you to write to improper memory locations. This is called 'Referential Safety' and it is a very useful property of Smalltalk systems. Sadly, java does allow casting and its concomitant problems.

## ***Meta-programming capability.***

One of the great virtues of a virtual machine is that you can precisely define the lowest level behaviour that the higher level code can see. This allows us to provide a reflective system with meta-programming capabilities and thus to write programs that can reason about the structure of the system and its programs. One good example is the Smalltalk debugger. Since the structure of and interface to the execution records (the **Context** instances) is defined within Smalltalk, we can manipulate them with a Smalltalk program. We can also store them in memory or in files for later use, which allows for remote debugging and analysis.

## **Threads & control structures programmer accessible**

When it is possible to cleanly manipulate the Contexts, we can add new control structures. See the article "Building Control Structures in the Smalltalk-80 System" by Peter Deutsch in the Byte August 1981 issue which illustrates this with examples including case statements, generator loops and coroutines. With only a tiny amount of support in the VM, it is possible to add threads to the system; although they are known as `Processes` within the class hierarchy.

## **Portability of the system**

In much the same way that the VM allows meta-capability, it can assist in providing high levels of portability. The interface to the machine specific parts of the VM is uniform, the data structures are uniform and thus it is possible to make a system that has total binary portability. No recompiling, conversion filters or other distractions are needed. Squeak, like several commercial Smalltalks, allows you to save an image file on a PC, copy it to a Mac, or a BeOS, or almost any Unix, or Acorn machine and simply run it. The bits are the same, the behaviour is the same, barring of course some machine capability peculiarities.

## **How the Squeak VM differs from the Blue Book design**

### ***Object format***

The Blue Book definition of the ObjectMemory used an object table and a header word for each object to encode the size and some flags. Each object table entry contained the oop for the object's class and a pointer to the memory address for the body of the object. Any access to the instance variables or array elements of the object required indirecting through the object table to find the body. {Diagram}

Squeak uses a more direct method, whereby the oops are actually the memory addresses of a header word at the front of the object body. This saves a memory indirection when accessing the object, but requires a more complicated garbage collection technique because any object that points to a moved object will need the oop updating. With an object table, the oop stays constant through the life of the object. {Diagram}

Since we still need to have the class oop, size and some flags available for each object, Squeak generally uses a larger header. The canonical header is three words:- flags/hash/size encoded in one word, class oop, size. However as a high proportion of objects are instances of a small set of classes, and since few objects are large, it was decided to use three header formats as follows:-

One word - all objects have this header.

3 bits reserved for gc (mark, old, dirty)

12 bits object hash (for HashSets)

5 bits compact class index, non-zero if the class is in a group of classes known as 'Compact Classes'

4 bits object format

6 bits object size in 32-bit words

2 bits header type (0: 3-word, 1: 2-word, 2: forbidden, 3: 1-word)

Two word - objects that are instances of classes not in the compact classes list. This second word sits in front of the header shown above.

30 bits oop of the class

2 bits header type as above

Three word - objects that are too big for the size to be encoded in the one-word header, i.e. more than 255 bytes. This third word sits in front of the two shown above  
{diagrams}

The set of compact classes that can be used in the compact headers is a Smalltalk object that can be updated dynamically; you can even decide that no classes should qualify. See the methods **Behavior>becomeCompact** and **becomeUncompact** as well as **ObjectMemory>fetchClassOf:** and **instantiateClass:indexableSize:** for illustration. Whether the space savings are worth the complexity of the triple header design is still an open question in the Squeak systems community. With the flexibility to designate classes as compact or not on the fly, definitive experiments can someday answer the question.

## Garbage Collection

Squeak uses a hybrid of generation scavenging and mark-sweep collection as its garbage collector. Remember that generation scavenging takes advantage of the observation that many newly created objects last a very short time before being abandoned, and that those which live longer tend to last a very long time. Thus if we can collect up all the newly created objects that are still alive and move them out of the way, all the remaining memory is populated only by dead objects - and dead objects are no objects at all. Squeak makes use of the previously mentioned similarity to the mark-sweep algorithm to implement a simple two generation scavenger. An important part of a generation scavenger is a table of those old objects into which new objects have been stored. This table is used as one of the roots when the search for live objects is performed; each of the objects is scanned and new objects pointed to are scavenged. Any object that no longer points to new objects can be removed from the table. Once all the new objects pointed to by the table and other roots have been scavenged, these 'survivors' are also scanned and any new objects found are also scavenged. The process continues until no further objects are found.

You can find the code for this scavenger in `ObjectMemory` in the 'garbage collection' protocol.  
{Diagrams}

Since older, long lasting objects get collected in lower memory, we can decide to ignore memory below a certain point for most of the time. However, sometimes we do need to clean and compact all of memory and Squeak has a secondary scheme very similar to the Blue Book description of the mark-sweep collector. This secondary collection has to sweep all of object memory and may take several seconds on machine with large memories or lower performance. As an example, whilst typing this document on an Apple PowerBook 400MHz, the average scavenge pause was about 3 milliseconds occurring roughly once per second, and a full garbage collect took 400 milliseconds. The mark-sweep collector methods can be found in the `ObjectMemory` 'gc -- mark and sweep' and 'gc -- compaction' protocols.

-- finalizer for weak pointers

Explain the idea of weak pointers?

## CompiledMethod format

Was a strange hybrid format; not weird format anymore with NCM, intended for Image 3.

## Primitives

- additional ones (sockets, serial etc)

Squeak has added many primitives to the list given in the Blue Book. They include code for serial ports, sounds synthesis and sampling, sockets, MIDI interfaces, file and directory handling, 3D graphics and more. See the later section for an idea of the full collection.

-- not there (asOop, events, )

Some primitives have been dropped. A few no longer make sense, such as `#asOop` which used to take a number, treat it as an oop and return the object with that oop. This is useful in some circumstances when there is an object table but is quite meaningless without one.

## atCache to speed up at: & at:put:

The messages at: and at:put: are very commonly used - they are both already installed as special bytecode sends. Squeak uses a special cache to further try to speed them up.

{method references?}

## BitBLT and Graphics extensions (little endian etc, warpblt, alpha, colour)

Blue Book BitBLT worked on monochrome bitmaps to provide all the graphics for Smalltalk-80. Squeak has an extended BitBLT that can operate upon multiple bit-per-pixel bitmaps to provide colour. It can also handle alpha blending in suitable bpp bitmaps to give anti-aliasing.

{examples?}

There is a WarpBLT that can perform strange transforms on pictures by mapping coordinate systems between groups of rectangles and interpolating the pixels as needed.

{Diagram, examples}

Perhaps most excitingly, there is a new geometry based rendering system known as Balloon, which can be used for 2D or 3D graphics.

## Self described VM kernel

The VM is largely generated from Squeak code that can actually run as a simulation VM. See the Interpreter and ObjectMemory classes for most of it.

You can try out the C code generator by evaluating:-

```
Interpreter translate: 'interp.c' doInlining: true
```

to build the C file that is most of the VM code. It will take some time, perhaps a couple of minutes or more depending on the speed of your machine.

## Generated parts of the VM

- the core of the interpreter is generated from the classes Interpreter and ObjectMemory, by the CCodeGenerator
- some sound sample mixing primitives are generated from code in FMSSound and other AbstractSound subclasses
- the Balloon 2 and 3 D graphics primitives are generated from a variety of classes in the Balloon3D hierarchy. See B3DEngine class>translateB3D
- many VM plugins are generated from various subclasses of InterpreterPlugin using the messages explained below.

## Non-generated parts of the VM, both platform specific and portable

See Ian Piumarta's chapter on porting Squeak for more information on what each file requires.

- directory handling primitives, in file sq{plat}Directory.c

All the code that reads directory entries. This is usually platform specific, unlike the file open/read/write/close code which is done via the ANSI C library calls that appear to be adequately portable.

- sockets and resolver primitives in sq{plat}Sockets.c

Code to handle sockets and resolvers, which tends to be very platform specific.

- serial port primitives, in sq{plat}Serial.c

- MIDI interface primitives, in sq{plat}MIDI.c

- sound system interface in sq{plat}Sound.c ,

The portable parts of the sound system are generated, but there is a non-portable section where the filled sound buffers need to be sent to the OS.

- window & display, input events, clipboard handling etc in sq{plat}Windows.c

Code to gather the OS input events for keypresses, mouse movements etc and convert them to Squeak

compatible forms. Code to display the updated parts of the Squeak Display Window on the actual machine

## A Tour of the Squeak Object Engine

display, potentially converting bitdepth.

- tablet & joysticks primitives, in `sq{plat}Joystick.c`

Read an attached joystick or graphics tablet. Obviously very machine specific

- external primitive interface in file `sq{plat}ExternalPrims.c`

Call primitives implemented in VM Plugins - see the chapter on how to implement them for more details.

## How to build a Squeak VM

Look at the class methods of `Interpreter`, in the 'translation' protocol:-

`Interpreter translateDoInlining`,

`Interpreter translate: [filename] doInlining: [boolean]`

will both produce a C file of the main interpreter component, with inlining of the small functions or not depending on the boolean.

`InterpreterSupportCode writeSupportFiles` will write out the other generic required files and

`InterpreterSupportCode writeMacSourceFiles` will create all the files needed for a Mac, including the makefile.

For non-Mac platforms you will need to go to [www.squeak.org](http://www.squeak.org) and find the archives for the VM sources for your platform. You will need to take care to check that the files are of the right vintage, since changes in the `Interpreter` class will affect several of the other files indirectly. In particular, 'sq.h' is very sensitive to such changes.

The various VM plugins can be generated by

`[PluginClass] translate`,

`[PluginClass] translate: [filename] doInlining: [boolean]`, or

`[PluginClass] translateDoInlining: [boolean]`

depending on whether the default name is acceptable to you and whether you want the code inlined or not.

The `Balloon3d` plugin is built from a number of classes and requires us to evaluate:-

`B3DEngine translateB3D`

to build the single file.

>>Things for the future

>>

>>external call interfaces.

Not like VM plugins. Problem with marshalling of args. Suggest VM plugin with call to dll as option for easier usage. Callbacks. Andreas?

>>

>>modular VM.

Splitting out to plugins. Options for configuration. Size reduction, performance improvement, simplicity.

>>

>>going faster

Remember better ST code is often best improvement!

>>

>>jitter

Ian? Describe it as opposed to explaining the development?

>>

>>native compilation

>>

>>threaded code BitBLT.

Maybe Andreas?

## **Bibliography**

Goldberg Robson 83

Adele Goldberg & David Robson, "Smalltalk-80: the Language and its Implementation", Addison-Wesley  
May 1983

Deutsch Schiffman 84

L. Peter Deutsch & Allan M. Schiffman, "Efficient Implementation of the Smalltalk-80 system", Proc. POPL  
'84

Ungar 87

David Ungar, "The Design and Evaluation of a High Performance Smalltalk System"