SqueakJS

A Modern and Practical Smalltalk that Runs in Any Browser

Bert Freudenberg
Communications Design Group
Potsdam, Germany
bert@cdglabs.org

Dan Ingalls
Communications Design Group
San Francisco, CA, USA
dan@cdglabs.org

Tim Felgentreff
Hasso Plattner Institute
University of Potsdam, Germany
tim.felgentreff@hpi.uni-potsdam.de

Tobias Pape
Hasso Plattner Institute
University of Potsdam, Germany
tobias.pape@hpi.uni-potsdam.de

Robert Hirschfeld
Hasso Plattner Institute
University of Potsdam, Germany
robert.hirschfeld@hpi.uni-potsdam.de

Abstract

We report our experience in implementing SqueakJS, a bit-compatible implementation of Squeak/Smalltalk written in pure JavaScript. SqueakJS runs entirely in the Web browser with a virtual file system that can be directed to a server or client-side storage. Our implementation is notable for simplicity and performance gained through adaptation to the host object memory and deployment leverage gained through the Lively Web development environment. We present several novel techniques as well as performance measurements for the resulting virtual machine. Much of this experience is potentially relevant to preserving other dynamic language systems and making them available in a browser-based environment.

Categories and Subject Descriptors D.3.4 [*Software*]: Programming Languages—Interpreters

General Terms Languages, Virtual Machines, Performance

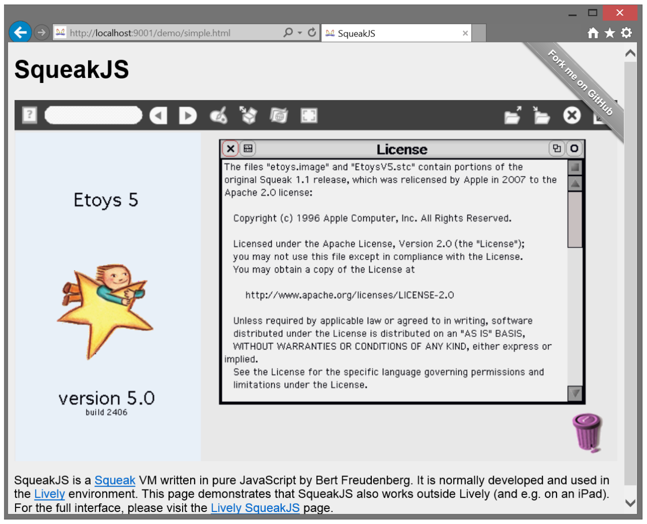
Keywords Smalltalk, Squeak, Web browsers, JavaScript

1. Motivation

The Squeak/Smalltalk development environment [8] and its derivatives are the basis for a number of interesting applications in research and education. Educational applications such as Etoys [9] and early versions of Scratch [12], however, suffer from restrictions against installing native software in scholastic environments. To cope with this limitation, web browsers have become the preferred target for those applications; Scratch has been rewritten in Flash, and for Etoys, a browser plugin for major web browser has been available for a several years. Yet, both approaches still need either software alien to the browser or a complete rewrite. With JavaScript as the de-facto standard for Web programming, a JavaScript-based, and thus browser-native, implementation of Squeak's virtual machine (VM) is desirable.

With SqueakJS, we aim to provide a fully backwards-compatible Squeak/Smalltalk system in the browser, able to directly run applications like Etoys or Scratch. Figure 1 shows SqueakJS running a Squeak Etoys image directly in the browser[1], on a plain HTML page.

Dynamic languages such as Python or Ruby have successfully been translated into JavaScript using the automatic translation toolchain Emscripten [20]. In contrast to such language implementations that host much of its core functionality inside of the VM, Squeak implements much of its kernel in Smalltalk [5] itself. Additionally, Squeak is a graphical programming system and, to run in the browser, requires at least minimal interaction with the browser's document object model (DOM).



SqueakJS

License

The files "etoys.image" and "EtoysV5.stc" contain portions of the original Squeak 1.1 release, which was relicensed by Apple in 2007 to the Apache 2.0 license:

Copyright (c) 1996 Apple Computer, Inc. All Rights Reserved.

Licensed under the Apache License, Version 2.0 (the "License"); you may not use this file except in compliance with the License. You may obtain a copy of the License at

http://www.apache.org/licenses/LICENSE-2.0

Unless required by applicable law or agreed to in writing, software distributed under the License is distributed on an "AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied. See the License for the specific language governing permissions and limitations under the License.

SqueakJS is a Squeak VM written in pure JavaScript by Bert Freudenberg. It is normally developed and used in the Lively environment. This page demonstrates that SqueakJS also works outside Lively (and e.g. on an iPad). For the full interface, please visit the Lively SqueakJS page.

Figure 1. SqueakJS running an Etoys image in a stand-alone Web page

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DLS '14, October 20–24, 2014, Portland, OR, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3211-8/14/10...$15.00.
http://dx.doi.org/10.1145/10.1145/2661088.2661100

[1] Squeak images are binary dumps of a running Squeak system's memory. SqueakJS supports reading and writing the same format as the standard Squeak VM.

Recent attempts at re-implementing or re-engineering the Squeak VM have shown that good performance is difficult to achieve [2, 13, 17]. To us, Squeak appeared to be too slow to actually emulate in JavaScript, and too complex to subdivide into an interpreter layer and browser graphics layer.

Potato, an earlier Squeak VM experiment written in Java, allowed us to asses the performance of our approach, and served as basis for the SqueakJS design in many respects, like the mapping of Squeak objects to JavaScript objects or the separation of the VM into the interpreter and a primitive handler.

The fact that SqueakJS represents Squeak objects as plain JavaScript objects and integrates with the JavaScript garbage collection (GC) allows existing JavaScript code to interact with Squeak objects. This has proven useful during development as we could re-use existing JavaScript tools to inspect and manipulate Squeak objects as they appear in the VM. This means that SqueakJS is not only a "Squeak in the browser", but also that it provides practical support for using Smalltalk in a JavaScript environment.

In this paper, we report our experience with this implementation, highlighting the following items in particular:

- the mapping of Squeak objects in the VM to JavaScript objects with direct references,
- a hybrid garbage collection scheme to allow Squeak object enumeration without a dedicated object table, while delegating as much work as possible to the JavaScript GC,
- a Web-appropriate adaption of file management to load and store Squeak images and other file types supported within Squeak, and
- the interaction with JavaScript for clipboard access, graphics, debugging, and providing plugins, that allowed us to use a rapid-prototyping approach to VM development.

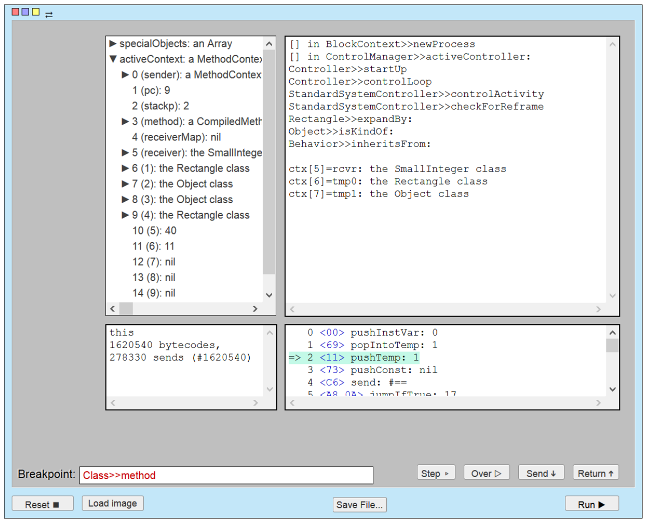
The rest of this paper is structured as follows: We describe our rationales for resolving various issues that emerged in our attempt to implement a Squeak VM in the browser. A description of the implementation details of SqueakJS, highlighting its innovative approaches, follows in Section 3. We evaluate our findings in Section 4 and put SqueakJS into perspective with other work in Section 5. Finally, Section 6 presents some lessons learned and suggestions for future work.

2. Approach

The core of SqueakJS is inspired by Potato, a Squeak VM written in Java. The sources of its precursor, Dan Ingalls's 2008 original "JSqueak", served as reference. Some fundamental design decisions, such as the mapping of Squeak objects to JavaScript objects or the separation of the VM into the interpreter and a primitive handler are exactly as in Potato. Some parts of the interpreter were transcribed literally, just removing Java's type annotations. Even some bugs were faithfully transcribed and only fixed later.

Apart from using another programming language, there are also differences in the SqueakJS implementation. For example, there is no object table, corresponding to the WeakReference array that is used in Potato for object enumeration.

These implementation differences, however, are less important than the context for which SqueakJS is intended. Java applications are less accommodating to direct manipulation development than JavaScript applications where live objects are inspected and manipulated with developer tools at run-time. Besides Browser-provided introspection tools, self-supporting development environments in the browser such as the Lively Kernel provide developers with malleable tools that can be easily adapted for specific use-cases. In Lively, most tools are *parts* [10]—parts are compositions of *morphs* [11] with associated, instance-specific behavior. In this context, the properties inherited from Potato are beneficial and allowed us to tailor Lively's tools towards inspecting and debugging the VM to provide immediate feedback during the development of SqueakJS.



```
▶ specialObjects: an Array
▼ activeContext: a MethodConte
   ▶ 0 (sender): a MethodContex
      1 (pc): 9
      2 (stackp): 2
   ▶ 3 (method): a CompiledMeth
      4 (receiverMap): nil
   ▶ 5 (receiver): the SmallIntege
   ▶ 6 (1): the Rectangle class
   ▶ 7 (2): the Object class
   ▶ 8 (3): the Object class
   ▶ 9 (4): the Rectangle class
      10 (5): 40
      11 (6): 11
      12 (7): nil
      13 (8): nil
      14 (9): nil
```

```
[] in BlockContext>>newProcess
[] in ControlManager>>activeController:
Controller>>startUp
Controller>>controlLoop
StandardSystemController>>controlActivity
StandardSystemController>>checkForReframe
Rectangle>>expandBy:
Object>>isKindOf:
Behavior>>inheritsFrom:

ctx[5]=rcvr: the SmallInteger class
ctx[6]=tmp0: the Rectangle class
ctx[7]=tmp1: the Object class
```

```
this
1620540 bytecodes,
278330 sends (#1620540)
```

```
   0 <00> pushInstVar: 0
   1 <69> popIntoTemp: 1
=> 2 <11> pushTemp: 1
   3 <73> pushConst: nil
   4 <C6> send: #==
   5 <A8 0A> jumpIfTrue: 17
```

Breakpoint: Class>>method

Figure 2. A Lively tool to load Squeak images, inspect, and debug SqueakJS

2.1 Getting it Going

Squeak has been implemented a number of times by now. The process to get it up and running roughly consists of the steps described by Dan Ingalls [8]:

- implement a new image reader and writer,
- write a new interpreter and essential plugins such as BitBlt and FilePlugin, and
- compile the interpreter to make it practical.

Of these steps, only the first two apply to our context. We decided early on to transcribe our implementation from the Potato sources. The original Squeak VM is written in "Slang", a subset of Smalltalk with C semantics, which can be translated to C easily [6]. It does not use any of C's structured data types to describe objects. Instead, Slang operates on words in memory, much like machine code would. The Slang-to-C translator could be re-targeted to produce Java or JavaScript, but that would keep these low-level semantics, which does not map well to a high-level language. That is why Potato and SqueakJS are not based on the Slang implementation.

Potato's object representation departs significantly from that of Squeak. All Squeak objects appear as plain Java objects, with slots that store the information required by the VM such as instance variables, class, format code, and indexable data. The same scheme is carried forward in SqueakJS. In the context of Lively, this allows us to easily adapt existing JavaScript tools to work on Squeak objects. Once the image loader began working, we could compose various Lively tools to create a virtual machine inspector, as shown in Figure 2.

Because stack frames and threads (contexts and processes in Squeak parlance) are Squeak objects as well, we can use the inspector to watch the VM stack while it is executing and debug any problems directly. This was useful early on to verify that the image loader and scheduler work correctly, and that dispatch actually worked when the VM started executing. Additionally, the debugging support available in Squeak—to break at, step into, and to disassemble methods—is available to the inspector as well.

2.2 Drawing to the Screen

Besides the interpreter, some way of drawing to the screen is required. Squeak uses a BitBlt plugin to render into bitmaps, which is fairly well separated from the rest of the interpreter. Potato only supports 1 bit per pixel color depths and a minimal set of graphics features. To support full color and all graphics features used in modern Squeak and Etoys image, the much larger Slang code for the BitBlt and WarpBlt plugins had to be transcribed to JavaScript.

Both BitBlt and WarpBlt can be simulated in Squeak, so conceivably, the Smalltalk code could be executed directly to render graphics. The R/SqueakVM (based on the Spy [2]) is an RPython-based implementation of Squeak that uses this scheme [17]. However, even with RPython's tracing just-in-time (JIT) compiler [1], the performance of pure-Smalltalk BitBlt is still about an order of magnitude slower than a C-based implementation, indicating that a native (which in our case means JavaScript) implementation is still preferable to achieve good performance.

SqueakJS exposes a module interface similar to that of regular Squeak VMs, and BitBlt is implemented as an internal module, i. e., one that comes with the VM. A simple HTML5 canvas is used to mirror the contents of the Squeak display object, a bitmap object that contains an array of bytes. Whenever an update to the display is needed, the Squeak system calls a primitive, which we use to update the HTML5 canvas.

Another module required by modern Squeak images is the "Balloon" 2D renderer. It supports anti-aliased drawing of graphics primitives not available in BitBlt, e.g., Bézier curves (needed to render TrueType fonts), and color gradients (used on various user interface elements). Porting the Slang code for this module would have been a major undertaking, likely resulting in a very slow simulation. Instead, we use an off-screen HTML5 canvas to draw these shapes, read the pixels back, and composite them over the original Squeak bitmap.

2.3 Cleaning up Garbage

Many core functions in Squeak depend on the ability to enumerate objects of a specific class using the `firstInstance` and `nextInstance` primitive methods. In Squeak, this is easily implemented since all objects are contiguous in memory, so one can simply scan from the beginning and return the next available instance. This is not possible in a hosted implementation where the host does not provide enumeration, as is the case for Java and JavaScript. Potato used a weak-key object table to keep track of objects to enumerate them. Other implementations, like the R/SqueakVM, use the host garbage collector to trigger a full GC and yield all objects of a certain type. These are then temporarily kept in a list for enumeration. In JavaScript, neither weak references, nor access to the GC is generally available, so neither option was possible for SqueakJS. Instead, we designed a hybrid GC scheme that provides enumeration while not requiring weak pointer support, and still retaining the benefit of the native host GC.

SqueakJS manages objects in an *old* and *new* space, akin to a semi-space GC. When an image is loaded, all objects are created in the old space. Because an image is just a snapshot of the object memory when it was saved, all objects are consecutive in the image. When we convert them into JavaScript objects, we create a linked list of all objects. This means, that as long as an object is in the SqueakJS old-space, it cannot be garbage collected by the JavaScript VM. New objects are created in a virtual *new* space. However, this space does not really exist for the SqueakJS VM, because it simply consists of Squeak objects that are not part of the old-space linked list. New objects that are dereferenced are simply collected by the JavaScript GC.

When full GC is triggered in SqueakJS (for example because the `nextInstance` primitive has been called on an object that does not have a next link) a two-phase collection is started. In the first pass, any new objects that are referenced from surviving objects are added to the end of the linked list, and thus become part of the *old* space. In a second pass, any objects that are already in the linked list, but were not referenced from surviving objects are removed from the list, and thus become eligible for ordinary JavaScript GC. Note also, that we append objects to the old list in the order of their creation, simply by ordering them by their object identifiers (IDs). In Squeak, these are the memory offsets of the object. To be able to save images that can again be opened with the standard Squeak VM, we generate object IDs that correspond to the offset the object would have in an image. This way, we can serialize our old object space and thus save binary compatible Squeak images from SqueakJS.

To implement Squeak's weak references, a similar scheme can be employed: any weak container is simply added to a special list of root objects that do not let their references survive. If, during a full GC, a Squeak object is found to be only referenced from one of those weak roots, that reference is removed, and the Squeak object is again garbage collected by the JavaScript GC.

2.4 Adapting Squeak for the Web

A general problem when running Squeak in a Web browser is the asynchronous nature of JavaScript. While most VMs are implemented as a main loop that simply executes until the VM is stopped, SqueakJS needs access to input events that are generated by the browser, and has to give the browser time to update the DOM. Our solution to this problem is to have SqueakJS regularly break out of its interpreter loop and schedule a timeout function to continue running after events have been processed.

Another common problem of running JavaScript applications is the question of file system access. While Squeak can work without a file system, a lot of functionality would be missing without the ability to store files. This presents a challenge for a system designed to run in a browser. One option would be to upload files to and retrieve them from a server. But this would require continuous connectivity, and we want SqueakJS to be usable offline.

Three options are available to store data in a browser. The first is browser cookies, on top these, simple key-value stores can be implemented. However, cookies are limited to 4 kB and are not a good fit to emulate a file system, considering that to save a modern Squeak image requires from ten to hundreds of Megabytes. The second option is the browser's local storage [7]. Like cookies, the amount of local storage per web page is limited, in most browsers to 5 MB. Again, this prevents us from saving most modern Squeak images. The third and most recent way to store files in a browser is IndexedDB [14].

IndexedDB is the only viable option that provides enough storage to be useful as a file system for SqueakJS. It can store arbitrary JavaScript objects and is intended to have both an asynchronous and synchronous application programming interface (API). However, all current implementations only provide the asynchronous API, whereas Squeak expects file system calls to work synchronously. Breaking out of the interpreter loop as described previously allows the browser to process asynchronous events. However, accessing the IndexedDB may take longer than one event loop. To be able to simulate a synchronous API, we additionally need to be able to freeze the VM—instead of scheduling the next interpretation step, the main loop simply stores a continuation to continue running when it is unfrozen. When a file primitive is called, the VM is frozen, and the callback that receives the IndexedDB unfreezes the VM. The same scheme can be used to synchronize any asynchronous JavaScript API that we want to expose. This is useful even for network operations, so that only SqueakJS, and not the entire browser, waits for a response.

3. Implementation

Our implementation closely follows the Potato VM in its basic approach, but deviates to accomodate JavaScript's execution environment. SqueakJS is publicly available on GitHub[2].

3.1 Object Representation

Implementation of a Smalltalk virtual machine is facilitated by the relatively clean separation between the byte code interpreter, the object memory and the graphics system. As noted above, Potato departed significantly from the Squeak object implementation without introducing any major perturbations in the rest of the virtual machine. All Squeak objects were mapped to Java objects with slots for

- Squeak hash (short)
- Squeak format code (short)
- class (pointer)
- pointer fields (array of pointers for instance variables and indexable fields)
- indexable binary data fields (array of bytes or ints)

As the only exception, SmallIntegers were represented by Java `Integer` objects. This is similar to the tagged pointers in other Squeak VMs which are used to represent immediate SmallInteger values. In contrast, Floats are represented by full objects, but their binary data slot was set to a Java `Double` so it could be accessed more quickly in arithmetic primitives.

Essentially the same scheme is used in SqueakJS. Since the layout of JavaScript objects does not have to be declared in advance, slots are added only when needed. The pointers (instance variables and indexable fields), words, and bytes slots are only present if the object actually needs them, that is, there is at least one element. We chose an array for storing instance variables rather than named JavaScript properties because the bytecodes reference them by index, not by name. SmallIntegers are directly represented by JavaScript numbers. In places where the VM needs to distinguish SmallIntegers from other objects it uses a `typeof` check.

For making common tests more efficient, SqueakJS also adds flags to special objects. For example, the `isNil` property is added to the `nil` object so that a test for being nil can be written without having to compare it to the VM's `nil` object. Floats are marked by an `isFloat` property, and the numeric value is stored directly in another property. The conversion to words happens on-the-fly, and only when the image tries to index individual words of the Float object.

In the C implementation, each object was uniquely identified through its position in memory, exposed to Smalltalk as the `oop` property. SqueakJS maintains the `oop` numeric property mainly for identifying objects when saving a snapshot. The `oop` also serves as a key when objects need to be added to a hash map temporarily (e.g., during "become" operations) because JavaScript does not provide an object-keyed hash map. New objects are assigned consecutive negative oops. A real oop is assigned when the object survives a full garbage collection, and during compaction oops are adjusted to account for released objects. This emulates the consecutive memory layout of the C Squeak VMs which is manifest in the snapshot format. The `oop` also helps to distinguish objects while debugging. The essential properties for a Squeak object in SqueakJS are thus:

- hash (`number`)
- format (`number`)
- class (`object`)
- oop (`number`)
- pointers (`Array`, optional)
- words (`Uint32Array`, optional)
- bytes (`Uint8Array`, optional)
- mark (`false`, present after surviving GC)
- nextObject (`object`, present only for objects in old space)

The implementation takes advantage of a few JavaScript idiosyncrasies like treating a non-existing property as `false`. For instance, a newly allocated object does not have the `mark` property, while an old object might have it set to `true` or `false` during GC. But the test can simply be written as "`if (obj.mark) ...`" because both cases test equally.

3.2 Object Enumeration and Garbage Collection

For object enumeration and finding all instances of a class, Squeak uses a primitive to answer the "next object" for a given object, or the next instance given a particular instance. The next object is easily found since all objects are contiguous in main memory, so simply adding the size of an object to the object's address gives the address of the next object. This is not possible for Potato or SqueakJS.

Since Java provides no facility for enumerating objects, Potato used an object table for the sole purpose of enumeration, as required by Squeak's enumeration facility as well as to support object mutation which involves enumeration implicitly. By making use of weak pointer objects in the object table, it was possible to leave the task of garbage collection entirely up to the Java runtime, a key to that port. In this regard, it is similar to Hobbes, a VM for the original Smalltalk-80 written for VisualWorks Smalltalk (see Section 5.) Unfortunately, current JavaScript engines provide no support for weak pointers. This means that the use of an object table for enumeration was out of the question, since it would have prevented all objects from being garbage-collected by the JavaScript runtime.

An important innovation in the SqueakJS object memory is the use of enumeration links in a subset of Squeak objects that corresponds to long-lived objects, or what is commonly referred to as "old space". When loading an object snapshot, objects are created as outlined above. Additionally, each object is assigned a `nextObject` slot pointing to the next object. The first and last object in this linked list is held in the VM, constituting the "old space". When a new object is allocated at runtime, it is not added to that list immediately. Instead, it is only held by references from other objects and released automatically when no longer referenced.

For old objects that are in the old space linked list, the "next object" primitive can simply return the object's `nextObject`. But for a new object, which does not have the `nextObject` slot yet, a full garbage collection is triggered. This adds the object (and all other surviving new objects) to the old-space list, setting the `nextObject`, so the enumeration can continue.

The full GC is a variation on mark-and-sweep: In the first phase, all objects reachable from the roots are marked recursively. If a new object is found in this phase, it is added to a temporary collection. In the second phase, the old-space linked list is traversed, unmarked objects are unlinked, and the mark of remaining objects is cleared. Finally, the new objects from the first phase are linked to the end of the old-space list. The discovery of new objects from the "virtual" new space during the mark phase is what sets this GC apart.

Interestingly, a full garbage collection does not occur regularly as in most other implementations, since our new space is essentially unlimited. No objects are tenured outside the full GC, so there is no need to regularly compact old space. In fact, that GC is invoked only from operations that are known to be expensive in the regular Squeak VM: `become:`, and `allInstances`/`allObjects`.

[2] `https://github.com/bertfreudenberg/SqueakJS/`

These operations are avoided in general because of their cost, and hence occur only in code sections that are not critical to performance. For example, during running the benchmarks in section 4.2 not a single full GC happened, even though about 15 million objects were allocated. In contrast, the startup code of an Etoys image does 12 `allInstances` calls resulting in as many full GCs, taking on average 300 ms, tenuring 980 of 150,000 allocated objects. But during normal operation, again, no full GCs occur.

3.3 Snapshotting

The Potato implementation, being a proof of concept only, did not support saving object images that could be stored and resumed elsewhere. The SqueakJS implementation provides this function in the following manner.

A Squeak object image consists of a file header for basic parameters followed by a literal memory dump of the object memory. Objects are laid out consecutively, with each object consisting of one to three header words followed by zero or more body words, as determined by bits in the header words. Object references (oops) are one word wide direct pointers to the address of the always-present "last" object header word immediately preceding the object body. That means the actual starting address of the object may be smaller than its oop, in the cases where it has a two- or three-word header.

Strictly speaking, oops matching the C VM memory layout are only needed while snapshotting. The SqueakJS runtime does not need them. They could be generated before taking a snapshot by walking the old-space list and accumulating object sizes. But for the reasons mentioned in section 3.1 they are kept up-to-date by the garbage collector, so after doing a full GC we can rely on all objects having accurate oops. The total number of bytes needed is maintained by the GC, too.

Snapshotting then means invoking a full GC which updates oops and sets the `nextObject` links. An `ArrayBuffer` of the needed size is allocated and wrapped in a `DataView` for typed access. The image header fields are written. The linked list of objects is traversed and each object writes a representation of itself to the memory buffer: one to three header words encoding the class oop, format, hash, and size, followed by the pointers (tagged for SmallIntegers and oops for all other objects), words, or bytes. The finished snapshot is stored as file in the browser's IndexedDB (see section 3.6). It can be resumed later in SqueakJS, or be exported to the user's disk and resumed in another Squeak VM.

3.4 Graphics System

As mentioned above the Squeak graphics system is relatively well separated from the rest of the virtual machine (VM) by the BitBlt and WarpBlt APIs. While the Potato proof of concept implementation did include BitBlt, it only supported the simplest 1 bit per pixel format and the simplest combination functions. While SqueakJS benefits from the similar syntax of C and JavaScript, the accurate and complete port of Squeak's BitBlt and WarpBlt functions was a significant part of the overall implementation task.

The implementation closely follows Squeak's, using the same approach of having various optimized variants of main loops for different arguments, or handling the start and end words of a line separately so the inner-most loop can be executed without conditionals. Most operations need to be done word-by-word, which is most efficient for 1 bit graphics because each word stores 32 pixels. The only place where more than one word can be processed at a time is when copying pixels without the need for shifting or color mapping. In that special case we use `Uint32Array`'s `subarray()` and `set()` functions to copy a full line at once. This optimization helps with scrolling, in particular.

The final stage of rendering the screen bitmap to an HTML5 canvas object uses the `putImageData()` drawing call. The image data is assembled in a `Uint32Array` by mapping each individual pixel of 1, 2, 4, 8, 16, or 32 bit to a single 32 bit ABGR pixel as required by the canvas API. This needs to be done even in the 32 bit case because the order of color components differs between Squeak and HTML. These conversions incur additional overhead compared to the standard Squeak VM. In Section 4, this additional overhead is visible in the greater slowdown incurred when actually drawing to the screen versus just blitting between bitmaps.

3.5 Main Loop

In most Squeak VMs, the VM itself has a main loop that executes byte codes until the system quits. Not so in SqueakJS: as described in section 2.4, the VM needs to break out of its interpreter loop periodically, to allow the web browser to update the screen or provide new input events.

The interpreter has two entry points: `interpretOne()` for executing a single byte code (which is used only when single-stepping in the VM debugger), and `interpret()` for executing multiple byte codes. The latter loops for a certain time, or until a primitive requests an immediate return of control to the browser. This happens e.g., after drawing to the screen: to minimize the delay between drawing and the change being visible to the user, the drawing primitive requests to exit the interpreter loop immediately.

Also, break points can be set for various conditions, e.g., when a certain method is executed or returns. When the condition is met, the interpreter loop exits to let the debugger show that state.

To support asynchronous functions in primitives that need to appear synchronous to the Squeak runtime, we implemented a feature to "freeze" the VM. A primitive can call the VM's `freeze()` function, which returns a callback function for unfreezing. All subsequent calls to `interpret()` are ignored until the unfreeze function was called by the primitive's asynchronous event handler. That means the VM's state remains unchanged until the unfreeze, so the primitive is allowed to defer pushing a return value onto the stack until the asynchronous operation is done. This is used for example to support asynchronous file access (see below).

Input Events Whenever SqueakJS breaks out of the interpreter loop, the JavaScript runtime has the chance to deliver new input events. The canvas object that draws the Squeak display registers to receive the available input events, such as `onMouseMove`, `onMouseDown`, `onKeyDown` and so on. In older Squeak images, two primitives regularly check the state of the input devices. To support that behavior, whenever SqueakJS receives an input event, we store it in a queue. When the running Squeak code calls the stateful input primitives, states are popped from the front of the queue, so all events are processed in order. Newer Squeak and Etoys images use an "event-driven" approach: they call a primitive that, instead of checking the *current* state of the input devices, releases an appropriate `Event` instance. If this primitive fails, the newer images can fall back to using the stateful primitives, so SqueakJS only implements these.

3.6 Files

As mentioned in section 2.4, a goal for SqueakJS was to provide for offline storage. Persistent offline storage is a relatively new feature for web browsers. The most common way is the browser's `localStorage`, which is synchronous, and very easy to use, but only supports storing strings, and is usually limited in size to 5 MB. Fortunately, all major browsers now support the IndexedDB client-side storage: Firefox since version 4, Chrome since version 11, and Internet Explorer since version 10. Safari support for IndexedDB is forthcoming with version 8 (and we implemented a fallback for older versions). IndexedDB stores arbitrary JavaScript objects, and has more generous size quotas, but current browsers only provide an asynchronous API.

	Squeak VM (Slang)	R/SqueakVM (Python)	SqueakJS (JavaScript)
Interpreter	5000	700	1000
Image & Object Memory	4000	1600	800
Primitives	2000	1200	1800
BitBlt	3000	900	1000

Table 1. Approximate lines of code for various VM concerns in different Squeak VMs

To optimize access times, the FilePlugin is implemented with a combination of `localStorage` and IndexedDB. Directory entry information (name, size, directory flag, creation date, modification date) is serialized as a JSON string per directory and saved in `localStorage`. It is accessed very often so the ability to access it synchronously is helpful. Also, it takes a relatively small amount of space. Due to idiosyncrasies of Squeak, often files are opened and closed without actually accessing the file contents.

The file content is put into the IndexedDB. When the file content is needed, it is loaded from the database into memory. Since every database access is an asynchronous operation, the VM is frozen until this read operation is finished. Write operations are performed in memory only. It is stored to the database only when the file handle is closed or explicitly flushed. The VM does not need to wait until the write finishes (similar for deleting a file). A single copy of the file contents in memory is shared for all open file handles of the same file. The file handles are reference-counted, and the memory is released when the last handle is closed.

A potential drawback of this scheme is that the two storages could get out of sync. This has not been a problem so far. If it becomes a problem, a file system check could restore consistency, or directory information could be stored in IndexedDB as well.

To interact with the host file system, SqueakJS handles drag and drop of files from the disk into the Lively world—Squeak image files are loaded and run, graphics are read in-image and loaded as Bitmaps. Files can also be stored into the host file system, by generating an object URI in the HTML page, which can be saved to the file system by the user.

3.7 Embedding

While SqueakJS is developed inside Lively, we have taken care not to depend on Lively. The VM is implemented in a single JavaScript file. For demonstration purposes we have written a simple harness that runs Squeak without Lively. It consists of a 10 line HTML page, 30 lines of JavaScript to emulate the Lively module and class system, and 200 lines to download an image, set up input and output, and run the VM. Because the overhead of that harness is considerably lower than the full Lively environment, it works nicely even on mobile devices such as an iPad, and supports browsers that cannot run the full Lively system, such as Internet Explorer and Firefox.

3.8 Extensions

SqueakJS can be extended with modules, just like most regular Squeak VMs. There is an internal and an external modules interface. Internal modules are part of the VM itself, they simply expose primitives by name instead of by index. An example for an internal module is the FilePlugin.

External modules are implemented outside the core VM. The interface is purely object-based. Which module is implemented in which file is unknown to the VM. A module object is simply registered by name:

```
Squeak.registerExternalModule('ModuleName',
                              moduleObject);
```

The module object needs to have a field named `exports` which references all provided primitive functions by name. If a function is exported as `initializeModule` then it will be executed when loading the module, passing the VM's primitive handler as argument. This is similar to passing the `interpreterProxy` in C-based VMs, it allows the module's primitive functions to interact with the VM, foremost to read arguments from the stack and push a result back. An example external module is `SimplePlugin.js` as shown in listing 1.

```
var SimplePlugin = function() {
    var proxy;
    function initializeModule(interpreterProxy) {
        proxy = interpreterProxy;
    };

    function primitiveNavigatorInfo(argCount) {
        if (argCount !== 1) return false; // fail
        var which = proxy.stackInteger(0);
        if (!proxy.success) return false; // fail
        var result;
        switch (which) {
            case 1: result = navigator.userAgent; break;
            case 2: result = navigator.language; break;
        }
        if (!result) return false; // fail
        var resultObj = proxy.makeStString(result);
        proxy.vm.popNandPush(1 + argCount, resultObj);
        return true; // success
    };

    return {
        exports: {
            initializeModule:
                initializeModule,
            primitiveNavigatorInfo:
                primitiveNavigatorInfo,
        }
    }
};

// register plugin in global Squeak object
Squeak.registerExternalModule('SimplePlugin',
                              SimplePlugin());
```

Listing 1. A simple Squeak Plugin

4. Evaluation

We have evaluated SqueakJS on two accounts: the amount of code it took us to produce a working version, and the performance compared to the regular Squeak VM. Because we are interested in running SqueakJS in any modern browser, we provide performance results for different major browsers.

4.1 Lines of Code

SqueakJS aims to be small enough to be understandable and modular enough to be easily extended. Table 1 compares the lines of code in the regular Squeak VM, R/SqueakVM, and SqueakJS directly used for certain VM concerns. Note that both the Squeak VM and the R/SqueakVM require a translation toolchain to generate C code, which has not been counted towards the required lines of code. For

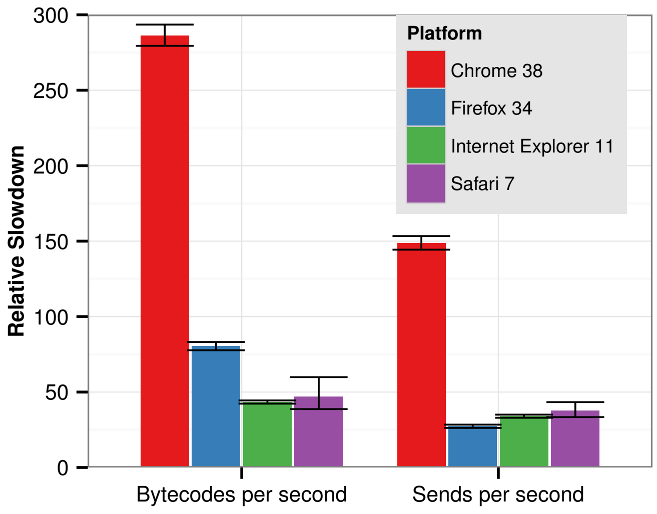


Figure 3. Overall speed: Bytecodes per second, sends per second. Each bar shows the arithmetic mean of 5 runs, normalized to the Squeak 3.9 interpreter VM.

Squeak, we have only counted the source code required for the so-called *Interpreter* VM, that does not include code for a JIT.

4.2 Performance

To measure the performance of SqueakJS in various scenarios, we used five benchmarks across Chrome 38, Firefox 34, Internet Explorer 11, Squeak VM 3.9, and Safari 7. We used an older version of the Squeak VM, because newer versions cannot load the Squeak mini image we used for testing. Our benchmarks are as follows:

blt measures the efficiency of the BitBlt implementation. It renders alternating patterns of pixels to a bitmap, without drawing it to the screen.

draw measures how fast the VM can push updates the the screen. A Mandala is drawn repeatedly and flushed to the display.

fib measures the impact of large stack depths and many sends by calculating Fibonacci numbers recursively.

fill measures the performance when allocating a large object and filling it with constant data.

prims measures raw bytecode performance by finding primes in a loop using a prime sieve.

richards measures allocation and send performance by simulating an operating system kernel.

Timing results of these benchmarks are presented in Figure 4. To make our measurements comparable, we report the arithmetic mean of five benchmark runs along with bootstrapped [4] confidence intervals showing the 95 % confidence level. The raw numbers for these browsers as well as those of Squeak 3.9 and Potato/JSqueak are included in the appendix (Table B and Table C).

A method, `tinyBenchmarks`, is included in Squeak to measure raw speed of sends and bytecode processing. It is used by various VMs to give an indication of the number of bytecodes and sends per second. Inverse speedup (i.e., slowdown) in bytecodes per second and sends per second is given in Figure 3.

These results show that SqueakJS is, depending on the browser, consistently between one and two orders of magnitude slower than the Squeak Interpreter in C. Note also that these numbers do not change significantly if we disable breaking out of the VM to process browser events or run them on the headless V8 VM.

We have not spent time optimizing the JavaScript code because using SqueakJS already feels responsive. We already regard these performance results as acceptable, but also see much room for optimization. For example, we are planning to add a just-in-time compilation to JavaScript.

5. Related Work

Squeak virtual machines The root for all Squeak implementations is the original Squeak VM [8], that aims to provide an open Smalltalk-80 system [5]. Initially, the original VM did only provide an interpreter-based execution, hence it is sometimes called *interpreter* VM; it serves as the reference implementation for other Squeak VMs. A *Squeak Web Plugin* based on this VM has been available for several years; however, the plugin runs completely isolated from the browser, thus, not providing any interaction from within the browser, let alone JavaScript.

Another well-known Squeak VM is the CogVM [15], providing a fully functional JIT compiler for Squeak. Cog's primary concern is performance. Both are written in *Slang* [6], a Smalltalk subset that directly translates to C.

The RoarVM [16] implements Squeak with a focus on exploiting manycore hardware. It is written directly in C++.

Potato[3] and its ancestor JSqueak are implementations of Squeak in Java. R/SqueakVM, based on Spy [2], is an implementation of Squeak in the PyPy framework [17]. It is implemented in a subset of Python, RPython, to test its viability as VM platform.

NaClSqueak[4] is a deviation of the Squeak *interpreter* VM, modified to run natively on the NativeClient framework of the Chrome browser family. Emscripten provides a similar option to NaCl, compiling C code directly into JavaScript. While to the best of our knowledge there is no Squeak VM compiled with Emscripten, PyPy (which is based on the same RPython translation toolchain that R/SqueakVM uses) has been successfully compiled using Emscripten, and the same might work for R/SqueakVM also. However, both NaCl and Emscripten focus on whole-system translation, implement a C semantics, and are not intended for rich interactions with the JavaScript environment. In this regard, NaClSqueak and a hypothetical Emscripten-compiled Squeak VM come close to the Squeak Web Plugin. Their most direct advantage is near native performance through highly optimized JavaScript code, but they are difficult to debug and instrument from JavaScript. Additionally, NativeClient is only available to Chrome browsers.

Dynamic languages on dynamic languages Squeak itself can be run on top of Squeak within its *Simulator*. While being slow, this provides means of introspection into the Squeak VM. However, these means are limited to the C-oriented view of Slang; objects of the hosted Squeak image cannot be accessed as such. In SqueakJS, objects of the hosted Squeak are readily available.

Since Self 4.0, a Smalltalk implementation [18] is an optional part of the Self distribution. Here, Smalltalk is directly translated to Self bytecode, without loss of expressiveness; there is no difference between running Smalltalk code and running Self code in "self includes: Smalltalk".

Amber[5] is a Smalltalk dialect implemented in JavaScript. It includes a complete development environment and a Smalltalk-to-JavaScript compiler. Amber does not interpret bytecodes but parses Smalltalk source code. Hence, it is not possible to load pre-exisiting Squeak image files. The performance of Amber is in the same order of magnitude as SqueakJS. It is possible to directly call JavaScript from Amber Smalltalk code.

[3] `http://sourceforge.net/projects/potatovm/`

[4] `https://github.com/yoshikiohshima/NaClSqueak/`

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

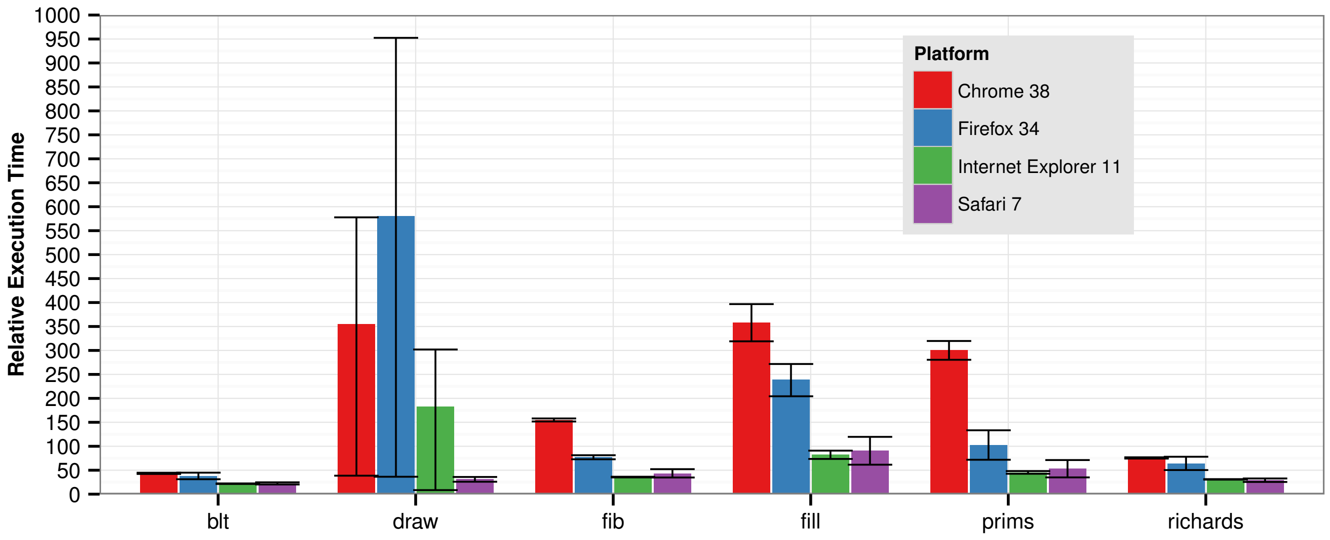


Figure 4. Benchmark execution time results. Each bar shows the arithmetic mean of 5 runs, normalized to the Squeak 3.9 interpreter VM.

Hobbes by Bykov is a VM for the original Smalltalk-80 written in VisualWorks Smalltalk and later ported by Ingalls to Squeak/ Smalltalk.

Whalesong [19] (Racket), Biwa[6] (Scheme), and Skulpt[7] (Python), among others, are JavaScript implementations of VMs for dynamic languages.

Other languages on top of JavaScript O'Browser [3] (OCaml), BicaVM[8] (Java), and Doppio[9] (Java), among others, are JavaScript implementations of VMs for static languages.

For a plethora of languages, including Dart, CoffeeScript, TypeScript, the Google Web Toolkit (Java) and others, source-to-source compilers exists that emit JavaScript code as result of the compilation process. Browsers then execute JavaScript, rather than the code of the source language directly.

6. Conclusion and Future Work

We have presented our experience with implementing SqueakJS, a fully compatible implementation of a Squeak VM that runs in modern Web browsers. We were able to develop SqueakJS quickly using a rapid prototyping approach to VM development. We started from the earlier Potato design, which was small enough to be a tractable experiment, yet complete enough to be motivating. From Potato, SqueakJS inherits its mapping of Squeak objects to native JavaScript objects that can be inspected and manipulated using other JavaScript tools. This allowed us to take advantage of mature JavaScript tools for development and debugging and made a Web-appropriate adaption of file and image-file management possible.

We also designed and implemented a hybrid GC scheme to allow for Squeak object enumeration without a dedicated object table, while delegating as much work as possible to the JavaScript GC. SqueakJS rarely has to do manual garbage collection, and even then our algorithm provides good performance, pausing only in our setup for about300 ms with a typical object memory of 300,000 objects.

We have found that the Squeak VM approach to provide a virtual environment without direct access to platform libraries was useful for our implementation. The Squeak plugin interface that abstracts from many platform-specific details was implementable even in the JavaScript environment.

Some non-essential primitives have fallback code in Squeak, so they can initially be skipped during the implementation, and only be added as an optimization. This made implementing easier. It would have been nice if all non-essential primitives had working fallback code in image, so that implementing Squeak runtime support on new platform can be carried out even faster.

There are different directions for future work. Firstly, we want to make SqueakJS feature-complete so it can be used as a drop-in replacement for a regular Squeak VM. Weak references and finalization are still missing, as are many modules, e.g., to allow SqueakJS to play sounds or perform network requests. Secondly, to improve performance, we are planning to add a JIT that compiles Squeak methods to JavaScript. We will investigate separating event-processing and bytecode interpretation; currently, the constant switching between both imposes a performance trade-off. Another significant increase in responsiveness would come from improving BitBlt performance. One idea would be to run BitBlt in a separate process to utilize multiple cores. It might also be possible to re-implement it using GPU processing via WebGL. Lastly, we would like to package SqueakJS in a way to be easily used on any web page, transforming it from a novelty to a serious tool. This also means improving compatibility with a wider range of web browsers on both desktop and mobile platforms.

SqueakJS is still a very young project, but already performs well enough to be usable, and is complete enough to run both old and modern Squeak images.

References

[1] C. F. Bolz. *Meta-tracing Just-in-time Compilation for RPython*. PhD thesis, Mathematisch-Naturwissenschaftliche Fakultät, Heinrich Heine Universität Düsseldorf, 2012.

[2] C. F. Bolz, A. Kuhn, A. Lienhard, N. D. Matsakis, O. Nierstrasz, L. Renggli, A. Rigo, and T. Verwaest. Back to the Future in One Week—Implementing a Smalltalk VM in PyPy. In *Self-Sustaining Systems (S3)*, volume 5146 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 2008.

[3] B. Canou, V. Balat, and E. Chailloux. O'Browser: Objective Caml on Browsers. In *ACM SIGPLAN Workshop on ML*, pages 69–78. ACM, 2008.

[6] `http://www.biwascheme.org/`

[7] `https://github.com/skulpt/skulpt/`

[8] `https://github.com/nurv/BicaVM/`

[9] `http://int3.github.io/doppio/about.html`

[4] A. C. Davison and D. V. Hinkley. *Bootstrap Methods and Their Application*, chapter 5. Cambridge, 1997.

[5] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, 1983.

[6] M. Guzdial and K. Rose. *Squeak—Open Personal Computing and Multimedia*. Prentice Hall, 2002.

[7] I. Hickson. Web storage. *W3C Recommendation*, 2013.

[8] D. H. Ingalls, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *Conference on Object- Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, pages 318–326. ACM, 1997.

[9] A. Kay. Squeak Etoys, Children & Learning. Technical Report RN-2005-001, Viewpoints Research Institute, 2005.

[10] J. Lincke, R. Krahn, D. Ingalls, M. Röder, and R. Hirschfeld. The Lively PartsBin: A Cloud-Based Repository for Collaborative Development of Active Web Content. In *Hawaii International Conference on System Science (HICSS)*, pages 693–701. IEEE, 2012.

[11] J. Maloney. Morphic: The self user interface framework. *Self*, 4, 1995.

[12] J. Maloney, L. Burd, Y. Kafai, N. Rusk, B. Silverman, and M. Resnick. Scratch: A Sneak Preview. In *Conference on Creating, Connecting and Collaborating through Computing (C5)*, pages 104–109. IEEE, 2004.

[13] S. Marr and T. D'Hondt. Identifying a Unifying Mechanism for the Implementation of Concurrency Abstractions on Multi-language Virtual Machines. In *Conference on Objects, Models, Components, Patterns (TOOLS)*, volume 7304 of *Lecture Notes in Computer Science*, pages 171–186. Springer, 2012.

[14] N. Mehta, J. Sicking, E. Graff, A. Popescu, J. Orlow, and J. Bell. Web storage. *W3C Candidate Recommendation*, 2013.

[15] E. Miranda. The Cog Smalltalk Virtual Machine: Writing a JIT in a High-level Dynamic Language. In *Workshop on Virtual Machines and Intermediate Languages (VMIL)*, 2011.

[16] D. Ungar and S. S. Adams. Hosting an Object Heap on Manycore Hardware: an Exploration. *SIGPLAN Notices*, 44(12):99–110, 2009.

[17] L. Wassermann. Tracing Algorithmic Primitives in R/Squeak-VM. Master's thesis, Software Architecture Group, Hasso-Plattner-Institut Potsdam, 2013.

[18] M. Wolczko. self includes: Smalltalk. In *Workshop on Prototype-Based Languages (co-located with ECOOP), Linz, Austria*, 1996.

[19] D. Yoo and S. Krishnamurthi. Whalesong: Running Racket in the Browser. *SIGPLAN Notices*, 49(2):97–108, 2013.

[20] A. Zakai. Emscripten: An LLVM-to-JavaScript Compiler. In *SPLASH'11 Companion (Wavefront)*, pages 301–312. ACM, 2011.

Appendix

For reference, we present the raw data for our benchmarks in Table C and Table B. Due to operating system restrictions, it was impossible to run all Squeak platforms on the same machine. Therefore, the numbers for Safari are normalized to Squeak on OS X while all numbers of the other platforms are normalized to Squeak on Windows.

Table A lists all exact version numbers of the platforms used.

	Full version
Squeak 3.9	SqueakVM 3.9
Potato 0.1	Potato (formerly JSqueak) 0.1
Firefox 34	Firefox 34.0a1 (2014-07-23)
Chrome 38	Chrome 38.0.2002.0
Internet Explorer 11	Internet Explorer 11.0.9600.17207
SqueakOSX	Squeak 4.2.5beta1U
Safari 7	Safari 7.0.5 (9537.77.4)

Table A. Exact platform versions

Benchmark	Safari 7		SqueakOSX	
	mean	error	mean	error
blt	539.600 ms	± 46.126	24.000 ms	± 1.640
draw	1251.600 ms	±126.036	40.200 ms	± 5.660
fib	790.200 ms	±175.777	18.200 ms	± 0.392
fill	254.000 ms	± 81.655	2.800 ms	± 0.392
prims	266.400 ms	± 99.794	5.000 ms	± 0.000
richards	5393.000 ms	±757.060	185.200 ms	± 3.124

	Safari 7		SqueakOSX	
	mean	error	mean	error
Bytecodes	16655888	4051326	782878362	13360834
Sends	565604	82545	21313809	490523

Table B. Benchmarks and speeds, raw numbers (run on OS X on Intel Core i7-4850HQ CPU @ 2.30 GHz)

Benchmark	Chrome 38		Firefox 34		Internet Explorer 11		Potato 0.1		Squeak 3.9	
	mean	error	mean	error	mean	error	mean	error	mean	error
blt	1121.800 ms	± 19.584	982.800 ms	± 188.140	564.000 ms	± 8.129	252.400 ms	± 19.418	25.800 ms	± 0.733
draw	3337.800 ms	± 86.112	5451.600 ms	± 44.438	1717.400 ms	±51.153	0.000 ms	± 0.000	9.400 ms	± 7.244
fib	2912.000 ms	± 16.883	1448.400 ms	± 82.329	671.000 ms	± 7.094	150.200 ms	± 5.382	18.800 ms	± 0.392
fill	1147.400 ms	± 18.988	765.000 ms	± 72.171	263.800 ms	± 2.659	5201.400 ms	± 82.381	3.200 ms	± 0.392
prims	1745.000 ms	± 22.079	594.600 ms	± 198.044	263.800 ms	± 1.143	391.800 ms	± 408.781	5.800 ms	± 0.392
richards	13816.400 ms	±116.125	11684.000 ms	±2844.545	5643.000 ms	±45.448	8249.000 ms	±6761.833	182.600 ms	± 3.136

	Chrome 38		Firefox 34		Internet Explorer 11		Potato 0.1		Squeak 3.9	
	mean	error	mean	error	mean	error	mean	error	mean	error
Bytecodes	2296769	8200	8186019	214742	15157722	82399	22126742	245474	657239930	17551026
Sends	135157	625	737809	19835	591498	1307	2685416	62465	20096527	702311

Table C. Benchmarks and speeds, raw numbers (run on Windows on Intel Core i5-4300U CPU @ 2.80 GHz). Note that Potato could not run the **draw** benchmark due to its incomplete BitBlt implementation, and that the **fill** and **richards** benchmarks, both of which put much pressure on the GC, perform very badly on Potato.