Back to the Future Once More

Dan Ingalls*

September 4, 2000

Author's note

I began this chapter attempting to duly credit each contributor appropriately. It soon became clear that either the chapter would degenerate to an encyclopedia of credits, or that it would be full of unfair omissions. I have chosen instead to take the point of view that I write for the entire Squeak community that has worked tirelessly and selflessly to make Squeak what it is. "We" did this together, and so it will be reported here. Most of the important contributions are credited elsewhere.

1 Introduction

The purpose of this chapter is to update the paper "Back to the Future – The Story of Squeak, a Practical Smalltalk Written in Itself" (hereinafter simply "BTF"). As such, the bulk of the text follows the structure of that paper, with comments and new data presented in a parallel sequence. However, a mere update of Squeak's features and performance would not give a sense of the various forces, technical and social, that have guided the evolution of Squeak over the three and a half years since the paper was published. Therefore we begin with an overview of some of these forces and the effects they have had on the evolution of Squeak since the publication of BTF.

As documented in BTF, Squeak began as a simple bytecode interpreter or VM (virtual machine) written in Smalltalk and translated to C, together with a modernized version of the Apple Smalltalk image. The major changes were to extend the object memory to 32-bit pointers, and to extend BitBlt to a flexible color model. The innovation of translating the interpreter made Squeak a practical and portable Smalltalk while being entirely self-contained and self-describing. Throughout its life so far, Squeak has enjoyed the ministrations of both the core development team at Disney (hereinafter "Squeak Central") and

*Here is the raw text, with apologies for lack of beautiful formatting. The time has passed for acting on suggestions (other than directly to me), but many thanks to those who read early drafts and offered many improvements: Mark Guzdial, Alan Kay, David Lewis, John Maloney, John McIntosh, Chris Norton, Andreas Raab, Stephan Rudlof.

a large and active internet community of developers, academics, and recreational computer scientists. Key to the continuing synergy between these two groups have been Squeak's total openness and the complete portability of Squeak across all major computing platforms, including even simple chip sets with only a BIOS.

2 The Evolution of Squeak

BTF is mainly about the implementation of Squeak; how it began, how it was carried out, and how it performed. Very little is said about the Squeak image which, in 1996, was simply a classic Smalltalk-80 image, with extensions for color and sound, and the support for simulating and generating the Squeak interpreter.

A major difference between then and now, is that most of Squeak's evolution has taken place in the Squeak image (the Smalltalk system class definitions), rather than in the VM. Most of the changes to the VM have come in response to that evolution, to enable or optimize various new capabilities needed along the way. Of all the changes over these years, the areas of greatest impact to Squeak's place in the world so far have been graphics and networking, and the flexible cross-platform support for sound.

Based on experiences with Morphic and Fabrik, we felt a need for a more flexible and concrete graphics model than the existing MVC (Model, View, Controller) view framework, and so, over the next couple of months, we built a reasonable implementation of the Morphic interface used in the SELF system. This interface was then put to immediate use as the basis for the EToy experiment —a novice interface to Squeak that allows one to control Morphic objects with halos and active inspectors and to script them by assembling tiles that correspond to Squeak message fragments.

The new Morphic worlds soon cried out for an equivalent to MVC's StandardSystemView so that we could "live" (i.e. integrate all our work) in Morphic as we had lived before in MVC, and this led to the creation of Morphic parallels to the basic MVC views. Similarly our educational experiments in these new worlds indicated the need for arbitrary scaling and rotation of Morphic objects. Not having anticipated this need so soon, and not having the resources to implement a completely general graphics model at the time, we combined a general transform definition with Squeak's WarpBlt capability to produce the TransformationMorph. While TransfomrationMorphs are neither a perfect nor a general solution, as one critic has frequently admitted, "You guys sure got a lot of mileage out of that Warp thingy." The advantage of this approach is archetypal of exploratory programming. In a couple of weeks, there was a way to experience and experiment with general scaling and rotation in Morphic, and we could move on to the next most interesting problem. Moreover, when time actually allows us to rewrite Morphic with a general approach to transformations, we will have several years' worth of working software as examples of what

we want and how we need to use it. [1]

While WarpBlt actually provides a reasonable anti-aliasing of interior images, the Morphic canvas rendering model had no way to properly combine morphs with anti-aliasing. At this point, we took on the task of designing a new canvas model that would incorporate curve drawing and filling with proper anti-aliasing. This model became the Balloon 2D rendering engine.

At about this time two other projects sharpened our focus on 3D graphics. To begin with, we had just begun a collaboration to port the Alice 3D system to Squeak [2]. At the same time, for various internal reasons at Disney, we wanted to be able to demonstrate a virtual gallery of computing environments in full 3D. With the Balloon engine in place, we set about designing a 3D graphics model that could employ this high-quality rendering onto a general Morphic canvas. This project became the Balloon 3D engine.

As we used and enhanced the new 3D facilities, performance inevitably became an issue. The desire to make use of various hardware and software accelerators on different platforms led to a serious reworking of the BitBlt and WarpBlt primitives.

Also crucial to Squeak's coming of age was the implementation of decent network support. By the time BTF was published, Squeak had cross-platform support for client-server protocols for file transfer, world-wide web, and electronic mail. At OOPSLA 97 two Squeaks, one on a Mac, and one on a PC, were shown running Telemorphic, a network-integrated version of Morphic (simply using multiple hands) with a multi-user paint program, and a multi-user music sequencer application.

The immediate focus at Squeak Central was not on browser access, but rather on leveraging the network to distribute "updates" to anyone participating in the active development of Squeak itself, or managing their own collaborative development efforts. The update mechanism is a simple and effective approach to coordinated system development. Each update is a file containing Squeak source code and possibly executable expressions as well. On a server is a list of all updates published so far, and an index of the order in which they were published. Any Squeak system connected to the internet can automatically determine what files have been published since it was last updated, and can read them in, thus becoming current with the latest work at Squeak Central or with the image from which the system updates were issued. This mechanism immediately brought the entire Squeak community together and allowed anyone to follow the latest changes with almost no effort at all.

About the time automatic updating became practical, a Pluggable Web Server (PWS) was implemented in Squeak. This turned out to be a very high leverage piece of software. At about this time, Ward Cunningham's WikiWiki server was being used to coordinate various designs and projects in the Squeak community, and it suddenly became

[1] For more on this topic, see the chapter on Morphic

[2] For more on this project, see the chapter on Alice

clear that one could build a server based on Ward's WikiWikiWeb in almost no time on top of the PWS. Within a month or two, the first so-called Swiki server was operational, and it soon became a part of the Squeak general release. Many people downloaded Squeak just to get a free cross-platform Swiki server!

Over the next year, Squeak's mail (Celeste), browsing (Scamper) and Chat (IRCMorph) facilities became operational, along with FTP access in the FileList. Other interesting projects to date include Commanche, a high performance web server, and Nebraska, a much more flexible approach to multi-user (and remote headless) applications in Morphic. [3]

Many other factors played a role in the progress of Squeak over these four years, but they are beyond the scope of this summary. With the foregoing sketch as context, let us now return to BTF and bring the major topics up to date.

3 The Interpreter

We have been able to retain the original bytecode interpreter design, keeping the core of the virtual machine relatively simple and yet we have constantly improved its efficiency through care in compilation (register variables), strategies for garbage collection and interrupt handling, and other specific techniques covered under "Performance and Optimization" below. Also, numerous ancillary "pluggable" primitives, most of them simply compiled from Squeak, have added greatly to the core computational power of Squeak.

A number of limitations in the original Squeak interpreter have been improved. For instance, pluggable primitives allow for essentially unlimited primitive extensions, the maximum number of temporary variables is roughly quadrupled, the image format has been tested at over 2.5GB, and so on.

4 The Object Memory

Even more than the interpreter, the original design for Squeak's object memory has stood the test of time. The basic object format survives unchanged, with only one extension to support weak array references.

The feature most often criticized is the use of a special header format for 1-word headers. While this typically saves an extra word per object for approximately 90% of the objects in the system, it can require an extra memory access to check the class of such objects. We have not yielded to this criticism yet because 1) the hard work has already been done, so moving to a simpler design would not save work at this point, 2) cleverness can in many cases avoid the extra penalty for looking up the class, and 3) 4 bytes per object can be a significant savings in small systems.

[3] For more complete coverage of these topics, see the chapter on Networking

One interesting capability has been added to the Squeak Object memory since the publication of BTF. This is the ability to extract and install image segments. One day, while musing about how to deep copy a structure without copying the entire world, it occurred to us that the garbage collector was in a position to solve this problem for us. The idea is to first mark a number of root objects, and then run the normal gc mark phase. Since gc marking stops at any marked object, the end result would be to mark every object in the system except those "in the shadow" of the root objects. The unmarked objects are therefore exactly the objects pointed to by the original roots, but not from anywhere else in the system.

Squeak image segments are produced by a primitive that accepts an array of roots, and produces an array of outward pointers from the segment into the rest of the image, and a binary object containing a copy of all the objects in the segment. The binary format is identical to that of a Squeak image, except that non-local pointers are represented as indices into the table of outpointers. Extraction of an image segment can be extremely fast, and installation is even faster. For example[4]:

- Segment size: 940K segment with 16403 objects, 6912 outpointers
- Time to extract: 264 milliseconds (mostly marking)
- Time to install: 43 milliseconds

The simplest application of imageSegments is to perform a fast deepCopy by extracting an image segment and then installing it again as a copy. Another application is segment swapping. For swapping, the extracted segment, is written to disk, and then the roots are converted into root stub proxy objects that will read the segment back from disk if they are ever touched. Special care is required to ensure that if a class is a root, it will still be successfully retrieved when a message is sent to one of its instances. When swapping segments, the table of outpointers is retained in the image as part of the root stub complex.

A third valuable application of image segments is for data export. In this case the binary segment is stored on a file along with a fully externalized representation of the array of outpointers. Such a structure can be transferred from one image to another and can be used for archival data storage as well. A fortuitous discovery about exported segments is that both internal and external pointers in image segments are quite local, and as a result, GZIP compression frequently achieves a factor of 4, as opposed to 2 or less on typical Squeak images.

5 Storage Management

Squeak's simple two-generation garbage collector has proven to be remarkably well behaved across a wide variety of applications. Its simple approach of incremental collection and compaction without maintaining free lists has provided both high performance and full utilization of available memory. It is still the case that a 600K Squeak image can fit

[4]Configuration: 8Mb image on a 400MHz Mac

in 1 megabyte along with the VM, and still be happy with the modest 200k of remaining available memory.

As various Squeak applications grow to larger sizes, we have been fortunate to see processor speeds also increase, leaving the typical latency for incremental collection and compaction well below the 10ms threshold that is critical to sound I/O and similar real-time response. A typical Squeak image with a substantial amount of content yields the following statistics:

- 20Mb old objects
- 2.2Mb young objects
- 2.8Mb free

During a 53 second run, there were 1400 incremental collections averaging 3.0 ms each, for an overall cost of around 8% [5]

Squeak now provides access to VM parameters, allowing one to trade latency (time required to perform and incremental GC) against overhead (% execution time spent in GC). Experimenting with a system similar to that above, and changing the quota of objects allocated between each GC, we found[6]:

Allocation quota	**Avg. latency**	**Avg. overhead**
2000	2 ms	11%
4000	3 ms	9%
8000	4 ms	6%
16000	6 ms	6%

In BTF we reported on Squeak's efficient bulk implementation of object identity reversal known as "become". The Squeak storage management system now provides both forms of become, symmetric and forwarding, both still being done in bulk if appropriate.

6 BitBlt and WarpBlt

The original design of BitBlt and WarpBlt survives relatively unchanged in the current Squeak release, but it is soon to be supplanted by a completely new implementation dubbed FXBLT. This new primitive responds to a number of forces in Squeak's evolution. First is the continued pressure for increased flexibility and low-cost setup when called by the balloon rendering engine. Second is the ability to take advantage of hardware acceleration. Third is the potential of improved performance in the absence of special hardware by applying some of the dynamic code generation techniques analogous to those used to accelerate the interpreter. Fourth is the need to reduce latency time which requires that large blts be interruptible. And last but not least is the desire to execute efficient transfers between bitmaps of differing

[5] Configuration: 20Mb image on a 400MHz Mac
[6] Configuration: 10Mb image on a 400MHz Mac

formats, including bits per pixel, bits per color, endianness, and even the order of color components.

Interestingly, while the latency issue is in some cases the most critical one, it is, at the same time amenable to high-level solution by recognizing large blts and breaking them into smaller ones outside of the primitive operation. While we have not applied ourselves seriously to the task of reducing latency in Squeak, we have implemented a limited-latency interface to BitBlt which did in fact cure interference of large blts with music generation. The extended primitives for text display and line drawing are also potential latency problems but, being optional, they can simply be eliminated at the cost of somewhat slower display of text and lines.

7 Smalltalk-to-C Translation

The core translator has remained relatively stable since the original release of Squeak. Probably the most significant change introduced since that time is the ability to compile independent primitive modules in conjunction with Squeak's "pluggable primitive" facility.

The pluggable primitive facility allows a method to specify a named primitive implementation. When such a method is executed for the first time, the interpreter attempts to load a module of that name from the directory in which the interpreter exists. If it is not found, then execution proceeds following the normal rules for primitive failure. If the module is found, then it is loaded, and the specific primitive name is sought within that module. If found, then the appropriate code is executed as a primitive in Squeak. A module can be defined both internally and externally. This means that an interpreter can be shipped with many intensions built in so that no additional plugins are needed, yet if a new version is present, it will override the code in the interpreter. Note that, after the first lookup, subsequent references to pluggable primitives are resolved with essentially no overhead, whether the module is present or absent (fast failures can be important).

The ability to compile optional plugins from Squeak has spawned a number of extensions of great value to various applications of Squeak. Each of the following plugins enables a significant capability for Squeak applications:

- Balloon 2D vector graphics engine
- Squeak3D 3D rendering engine
- JPEG decoder Fast JPEG decoder
- FFT Fast fourier transform
- FFI Foreign function interface
- KLATT Speech synthesis
- SoundCodecPrims 10:1 ADPCM sound compression/decompression
- LargeIntegers Fast implementation of LargeInteger arithmetic
- GZIP Fast GZIP data compression/decompression

The fast LargeIntegers have enabled practical DSA encryption, and the GZIP compressor is used in many places to save space in Squeak and its external files.

The Foreign Function Interface has enabled a number of interesting experiments and real-world applications, including control of a large real-time 3D simulator with multiple display screens, a Quicktime toolbox controller capable of displaying QT movies in Morphic, and an interface to the FreeType toolbox. [7]

8 Sound

Most of Squeak's sound support is in Squeak itself. However, as described in the original BTF paper, a few primitives are necessary to achieve reasonable performance. The original specification of the primitives has changed somewhat in order to allow fine-grained control over envelope parameters with relatively little computational overhead.

Much more has been done with music in Squeak since the BTF paper. The release image includes a MIDI score player (ScorePlayerMorph) with pan, gain, and mute controls as well as the ability to change the orchestration. A MIDI score piano roll (PianoRollScoreMorph) can display the piece being played in real time, and it supports some very limited editing facilities as well. Sampled timbres can be recorded from a microphone or from other sources, cleaned up and looped in the sample editor (WaveEditor). FM timbres can be edited in the envelope editor (EnvelopeEditorMorph), and tested with an on-screen keyboard (or with another music application). Beyond these primitive but useful facilities, the SIREN system, described elsewhere in this volume, is a mature application devoted to the creation and manipulation of musical scores in Squeak.

Merely making sound available in a cross-platform and interactive manner has spawned many other interesting capabilities. For instance, the Squeak release includes a real-time speech synthesis facility capable of reading any text intelligibly. It can even sing "Silent Night" as a duet with animated singing faces.

Squeak's 30-line FFT became one of the first pluggable primitives, and a showcase for fast access to Floating-point arrays. Running in Squeak, this routine performed a 4096-point Fourier transform in about 580 milliseconds. The plugin computes the same transform in 3 milliseconds. The FFT plugin has enabled Squeak to display real-time sonograms (SpectrumAnalyzerMorph) and to perform limited speech recognition.

9 Code Size and Memory Footprint

The size of the kernel interpreter has grown very little, but a number of added primitives have increased the size of the interpreter module

[7] For more on pluggable primitives, see the chapter on Extending the Squeak Virtual Machine

by about 50% since the figures were reported in BTF. The Object memory has grown very little since it was first written. BitBlt and the other related graphic routines have nearly doubled in size as a result of the enhancements and other experiments alluded to above.

It is still possible to produce a practical Squeak that will run (interpreter, image, and adequate free space) in one megabyte. Some Squeak releases require massaging to produce an adequately small image (700K) to fit within this constraint.

10 Performance and Optimization

Table 5 in BTF documents gradual improvements in the efficiency of Squeak's interpreter that achieved an eight-fold improvement over the course of nine months. When that table was written, we had reached version 1.18, and we felt we had squeezed about as much as possible out of a classical bytecode interpreter. We can now compare the 1.18 interpreter with the 2.8 interpreter in use at the time of this writing [8].

- Squeak 1.18: 17.7 million bytecodes/sec; 907 thousand sends/sec
- Squeak 2.8: 36.1 million bytecodes/sec; 1155 thousand sends/sec

It is gratifying to note that, whereas we thought we had reached the limit of what could easily be done to improve Squeak's performance, we still managed to double the bytecode speed and to squeeze an additional 27% in the time it takes to perform a send. The latter was achieved by doggedly reducing the code executed on each allocation and release of a context, especially eliminating the need to nil out all fields of a context before use by noting the stack pointer in garbage collection. The improvement in bytecode speed came from attention to register allocation in the generated C code, streamlined flow through arithmetic primitives, and the introduction of an "at-cache". The at-cache keeps a small cache (8) of recent objects that can respond primitively to at: and at:put:, along with their size and format (byte, word, pointer or bits). This cache enables fast treatment of these messages for such objects, and also similar speedups for next and nextPut: for streams whose contents satisfy the same conditions.

It must also be noted that during the 3 years that has elapsed between these two releases, comparable computer speeds have increased fourfold. One could conclude that it is better to put one's time elsewhere than optimization, since next year's silicon will make your efforts seem trivial. However, on slow machines such as PDA's, every cycle still counts, and on fast machines the results are multiplicative so it is still important progress.

An entirely separate assault on performance was mounted in 1998. The first attempt, code named JITTER, achieved a significant performance gain (bytecode speed, send speed). However overall benchmarks never made it to the level we sought (200-400% overall), and the design seemed to suffer from a number of tuning sensitivities.

[8] Configuration: 10Mb image on a 400MHz Mac

Almost before the first JITTER came to life, another design had sprung up in its place. Begun half a year later, and dubbed J3, this design has a better approach to pointer mapping and cache management, and works from a table-driven model of code generation. The J3 interpreter has demonstrated the level of performance we had hoped for (a factor of 3 in bytecode speed, and 6 in send speed), with several optimizations yet to be tried. It is our hope, looking forward, that J3 technology will soon make it into the mainline Squeak releases.

11 The Squeak Community

BTF is a technical paper, and most of the foregoing information serves to bring the reported results up to date. The section entitled "The Squeak Community" is really about the success of Squeak's portability and the remarkable achievements of a couple of outside contributors after Squeak's release. At this point it seems appropriate to balance the technical reports with some of the less technical factors that have made Squeak and the Squeak Community what they are.

If you were to spend time with the principal authors of Squeak, you would find them to be technically competent, but that is not what has made Squeak what it is. What sets this group apart is a further interest in simplicity, leverage and synergy. Look at Smalltalk itself. A completely different model for computation —sending messages to objects— looks simple and effective. We carry this through an entire system design and find enormous leverage from polymorphism and inheritance in this model. Then extending this out to the world of graphics, it turns out that many things that used to be hard are easy, and this feeds back, making the core of the system even more concrete and accessible; this is the synergy part. It's more than synergy —it's fun.

Thus Squeak, beginning as a Smalltalk system, was already earmarked for the adventurous, but with the added meta-circularity of including its interpreter within itself. Published with complete source code available free over the internet, Squeak was bound to attract an interesting community.

Difficult though it may be, it is probably worth trying to characterize the Squeak community. The two major orientations can best be characterized as "research", and "development". The research group consists of workers similar to the Disney team, whose primary interest in Squeak is as a vehicle for research, a malleable tool that can easily be adapted to serve a wide variety of investigations. The academic members in this group find in Squeak a vast laboratory of interesting experiments in computer science, many comprising the laboratory itself. The commercially oriented developers, on the other hand, see in Squeak a royalty-free Smalltalk base that runs on every major computing platform and can be easily ported to bare chip sets. Both constituents of the Squeak community also share certain fun-loving characteristics. They enjoy the art of programming, they are motivated to improve an open-source facility, and they see it as a sport to

reach results comparable to commercial implementations.

As curators of this particular facility, it has been a constant challenge to decide where best to put our effort in the growth, refinement and (hopefully) simplifications that shape the future of Squeak. A classical planner would ask what is our market, what are our strengths, what is the competition, and what comprises, therefore, our best "product" opportunity.

Now we have to look at one more important aspect of this community – the role of Squeak Central. Thus far, Squeak Central has "enjoyed" a central position in shaping the evolution of Squeak. While we have always had an egalitarian attitude toward the community as a whole, we are not a neutral player in the process. Squeak was delivered to the world because (1) we felt that Smalltalk was the most malleable and highest productivity environment to serve as a vehicle for our investigations in personal computing, and (2) we felt that only by making it open and free would it garner the kind of intellectual participation needed to become a serious computing environment.

There is therefore a distortion in the original characterization of our "market" – a primary drive toward support for what we see as the computational and interactive needs of Squeak Central's "vision" at any given time. We have tried to deal with this asymmetry as much as possible in the manner of a benevolent dictatorship. We make most of the decisions about what is or is not included in the system, but we also try to maximize the synergy with the rest of the Squeak community.

Now we can return to the forces at work in Squeak's "market" as so defined. A number of Squeak's attributes are items that everyone agrees are important. These include cross-platform support, color graphics, and open source distribution. Many others are not. For instance, developers care more about ANSI compliance than anyone else, with academics running a close second. This is because they want to be able to import software from other systems, and they want to be able to use or ship their products with other Smalltalk systems besides Squeak. This can be a matter of performance, or one of co-operating with existing software that is tied to a particular Smalltalk host. For similar reasons, developers care a lot about including MVC, but researchers do not care so much, because many experiments turn out to be simpler to build and easier to demonstrate if written using Morphic.

Squeak Central may not care at all about either ANSI compatibility or support for MVC because it does not import code from other systems, and it does not use MVC in any of its work. However it is clear that these features are both very important to synergy in the Squeak community. Squeak Central benefits constantly and directly from work done by other members of the community, so concerns such as these are given high regard in all difficult decisions.

As long as the Squeak community is split between developers and researchers, between novices and experts, there will be tension surrounding the makeup and presentation of the system. As long as it makes sense, we try to keep the various forces in balance and maximize the synergy in the results.

12 Future Work

It is gratifying to see that almost everything we spoke of accomplishing in BTF's "Future Work" section has been done. Given that the scope of BTF was limited to a Smalltalk implementation, this is not surprising. When we look forward from here, in the broader context of general purpose multimedia computing, the challenges seem somewhat more daunting.

Interestingly, one of the future items in BTF was "to supplant the MVC graphics model with a new one along the lines of Morphic and Fabrik." That has all been done, and has successfully spawned numerous exciting graphical applications including the EToy environment, music editors and a complete 3D gallery of independent Squeak projects. Unfortunately Morphic grew uncontrollably in response to the differing and demanding masters it was asked to serve, and we are back at the same place a generation later. To paraphrase from four years ago, we now plan to supplant the Morphic graphics model with a new one that incorporates what we have learned so far, that will be simpler and better factored, and that will lend itself better to the support of multiple environments in 3D and multiple users in those environments.

Much remains to be done to make Squeak more approachable and productive for various classes of users, especially for novices or "internet" programmers. We hope to see in the near future a synthesis of EToy-style environments with the current "serious" software development tools. We also hope to introduce types at both the novice and expert levels, but in a way that is not encumbering to the simplicity and immediacy of coding in Squeak.

At the time of this writing, we are working to bring the various just-in-time compilation experiments into the Squeak mainline release and to support them on all platforms. Improved performance is important to researchers and developers alike, and it should augment the convergent forces in both sides of the Squeak community.

However circuitous the path may seem from time to time, Squeak is destined to follow its authors' vision of the real potential of personal computing. As a personal information appliance it will be simple and convenient. It will be able to store and manipulate many kinds of data from text and numbers to images and sounds and rules for their behavior. Access to different kinds of objects will be simple and uniform, and the system itself will be equally accessible to inspection and manipulation. From whatever perspective you choose to investigate the content or the system itself, the rules for its behavior will be immediately accessible and admit of change and experimentation. All will be so well organized and documented that the motto for learning Squeak would be, "The System is the Curriculum".

All we have to do is organize and document the system as though it were a curriculum.