The Parks PDA: A Handheld Device for Theme Park Guests in Squeak

Yoshiki Ohshima[*]
Twin Sun, Inc.
360 N. Sepulveda Blvd.
El Segundo, CA 90245 USA
Yoshiki.Ohshima@acm.org

John Maloney[†]
MIT Media Lab
E15, 77 Massachusetts Ave.
Cambridge, MA 02139 USA
JohnMaloney@earthlink.net

Andy Ogden[‡]
Strategy, Design, and
Development Consulting
1738 N. Bellford Ave.
Pasadena, CA 91104
aogden@earthlink.net

ABSTRACT

The Parks PDA is a lightweight, handheld device for theme park guests that functions as a combination guidebook, map, and digital camera. Together with a small team of artists and designers, we created a prototype Parks PDA and content for a three hour guest experience, including a camera interface, a hyper-linked guide book, three games, an animal spotters guide, a cross-referenced map, animated movies with lip-synched sound, a ride reservation system, and more. Over 800 visitors to Disney's Animal Kingdom™ theme park tested the Parks PDA over a two week period.

Developing the software for this test posed a number of challenges. The processor and memory of the target device were slow, the screen was small, and we had only three months of development time.

We attacked these problems using Squeak, a highly-portable, open source Smalltalk implementation. We ported Squeak to the target device and used it to provide nearly bit-identical behavior across four different platforms. This supported a cross-platform development style that streamlined the production of both software and content. We created a tiny user interface and application framework for pen-based devices and implemented a simple card-stack media editor and player using it. We isolated and addressed several challenging performance issues.

The project was completed on time and guest response was favorable. Looking back, we can identify seven aspects of Squeak that contributed to the success of the project. In fact, we feel that Squeak was the ideal tool for this job.

[*]Yoshiki was a student at Tokyo Institute of Technology and interning at WDI R&D while doing this work.

[†]John was at Walt Disney Imagineering R&D while doing this work.

[‡]Andy was at Walt Disney Imagineering R&D while doing this work.

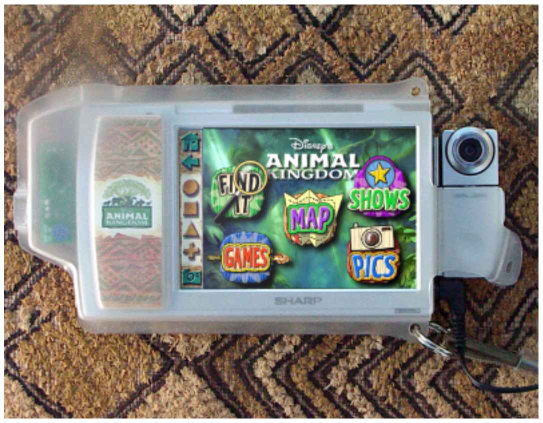

Figure 1: The PDA Showing the "Home Page". There are five hyper-link buttons on the screen. To the left of the screen there are seven silk-screened buttons. Attached to the right is the digital camera.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments—*design prototyping, debugging on embedded systems*; D.1.5 [**Programming Techniques**]: Object-oriented Programming—*Squeak*; H.3.2 [**Information Storage and Retrieval**]: Information Storage

General Terms

Design, Experimentation, Languages, Performance

Keywords

Handheld device, PDA, end-user software, multimedia data management, rapid software development, development environment

1. INTRODUCTION

By the late 1990's, computers were becoming small and light enough that you could carry one everywhere, even on vacation. At Walt Disney Imagineering R&D, the big question was, could you create a user experience on a handheld device that was both fun and useful enough to justify carrying that device in a theme park? For us, a color screen was essential. Unfortunately, theme park guests spend much of their time outdoors where an ordinary backlit color liquid crystal display (LCD) becomes illegible. In late 1999, Sharp announced a very compact personal digital assistant (PDA), the Zaurus MI-C1, that had a *reflective* color LCD display that worked well even in direct sunlight. The MI-C1 was one of the first PDA's to have such a display. The MI-C1 also supported a camera, another essential feature for us.

With a new medium, such as a PDA in a theme park, providing a new content is challenging since there are no prior examples to study. It is even possible that the medium itself has a fatal flaw. We decided to test this new medium by creating a working prototype with enough content to support three hours of use in Disney's Animal Kingdom™ theme park and to test it on 800 randomly chosen guests. The business mission of this test was "to rapidly evaluate the potential appeal of a PDA user experience for theme park guests that might ultimately take advantage of wireless connectivity and location-sensing technologies without having to wait for the maturation of those technologies."

The production team for this test consisted of nine creative staff – artists, designers, and show writers – plus two programmers. We had three months to create both the content and the software. We also needed to integrate hardware components such as a digital camera and a location sensing system. Using most languages and development processes, this would seem ambitious, perhaps even impossible. Fortunately, we had a secret weapon: Squeak[5]. Squeak is a highly-portable, open-source Smalltalk implementation with a modest memory footprint and an easily extensible virtual machine. Yoshiki Ohshima had already ported Squeak to an earlier Sharp Zaurus model with good results.

This paper explains how Squeak supported the object-oriented programming techniques and cross-platform development processes that made the project an overwhelming success. Towards the end of the paper, we identify seven properties of Squeak that were key to that success.

2. A PDA FOR THEME PARKS

The Parks PDA is a lightweight, handheld device for theme park guests that functions as a combination guidebook, map, and digital camera. Our prototype Parks PDA contained content for a three hour guest experience covering about a third of Disney's Animal Kingdom™ theme park. In addition to the hyper-linked guide book and camera, the Parks PDA contained three games, an animal spotters guide, a cross-referenced map, a number of animated movies with lip-synched sound, a simulated ride and restaurant reservation system, an online shopping application, and a short feedback survey.

2.1 The Guest Experience

This section describes guest's experience of using the Parks PDA. All the guests in a group (e.g., a family) typically shared a single Parks PDA device, either by standing close together or by passing the unit around.

Hosted Narrative
Upon entering a new "land" (part of the park) guests were automatically prompted to play a narrative hosted by an animated character who explains the "story" of that land. Since the earbud allowed only one guest at a time to hear the audio, a repeat button was provided to allow every member of the party to enjoy this experience.

Digital Map and Index
Either the digital map or an on-screen index could be used by guests to locate and learn about attractions, shops, restaurants, and restrooms. All information was cross-indexed, allowing the guest to quickly jump between the map and the information pages.

Sign Posts
Guests came upon sign posts distributed throughout the park marked in a simple code consisting of four basic shapes. By entering this code using the silk-screened symbol buttons on the Parks PDA, the guest could get detailed information in the form of audio, text and images about the plants or animals in that location. Symbol codes were used to create a detailed botanical tour of one section of the park.

Ride Reservations
Guests could use their Parks PDA to make a reservation for one of the most popular rides. To make sure that they had a chance to take advantage of this feature, they were prompted with an invitation as they passed selected locations for the first time. If they made a reservation, then shortly before the reservation time, a reminder was presented, giving them a comfortable amount of time to get to the ride. When the guest and their party arrived at the ride, they showed their "electronic ticket" to the attendant and were allowed to bypass the queue line.

Games
While guests were waiting in the queue lines of other attractions they could pass the time enjoyably by playing interactive games themed to reinforce the attraction they were about to see.

Digital Picture Taking
Guests could use the PDA's digital camera feature to take pictures during their test session. The camera interface included an easy to use erase function. Guests could take an unlimited number of photos as long as they kept a maximum of 40.

Online Picture Viewing
Back home, guests could visit a website and enter their ID and password to access a page showing a map in the same graphical style as the one that appeared on the Parks PDA, but with thumbnails of their pictures placed around the map in the approximate locations where the photos were taken. The guest could either view full-sized versions of their photos on the screen or print them out. The map showing the thumbnails could also be printed as a record of their visit and an implicit invitation to return to the park.

2.2 User Testing

The Parks PDA was tested on over 800 guests over a two-week period. PDA's were distributed to guests who agreed to participate. After using the Parks PDA for two to four hours, the guest returned it and gave us their feedback about the experience.

Selection
Guests were chosen at random by Walt Disney World Market Research staff, shown an example Parks PDA, and invited to participate in the test. Most people visit theme parks with family or friends; we chose one adult member of each party to be our primary contact for the test.

Registration
Each guest was directed to a registration area where they

were assigned a unique ID and password and given a numbered Parks PDA. The guest name, ID, password and PDA number were recorded in a database. The guest name and ID were also downloaded into the Parks PDA via IrDA. Having the guest name and ID in the Parks PDA allowed the experience to be personalized and simplified tracking of their photos. The ID and password were written on a printed map given to the guest. This information allowed them to later access their photos online.

Instruction

Groups of eight to twelve guests at a time were then given a five-minute briefing on how to use the Parks PDA. Instructions included a scripted overview of the content features and how to use the PDA's graphical "finger push" interface. Guests were given suggestions for picture taking and shown how to wear the PDA and use the attached earbud for audio. The guests were then invited to explore the park with their PDA's and asked to return to the drop-off location three to four hours later.

Interview

At the end of the test session, guests dropped off their Parks PDA at a restaurant in the Africa land where a room was reserved for interviews. They were interviewed by Walt Disney World Market research specialists for approximately 20 minutes about their experience. They were also told how to go online to view the pictures they took with the Parks PDA.

Photo Upload

The Parks PDA devices returned by the guests were taken to a photo uploading station where the guest's photos and session log were uploaded to a laptop computer via the IrDA port. The photo files were later transferred to the website.

PDA Recharging

After uploading the guest photos, the Parks PDA was checked for damage and placed in a charging rack. Early the next morning, the screen was cleaned to remove any fingerprints, the earbud cushion was replaced, and the Parks PDA was checked for proper functioning.

3. THE HARDWARE PLATFORM

The hardware for the Parks PDA was the Sharp Corporation's Zaurus MI-C1 PDA, a device available only in Japan, where Sharp's Zaurus PDA's enjoyed a large market share. It's specifications are shown in Table 1. The original retail model had only a single expansion slot. Sharp, our partner in this experiment, designed and built 100 custom MI-C1 units with a second expansion slot, allowing us to use both the digital camera module and a 128MB Compact Flash memory card at the same time. Sharp also doubled the RAM of these custom units, from 8MB to 16MB.

The SH-3 CPU is a RISC processor with 16-bit fixed-width instruction set with low power consumption but only modest performance; our tests show that its arithmetic intensive performance is an order of magnitude slower than the 206MHz StrongARM. Access to the underlying hardware was made through ZaurusOS, a Sharp proprietary operating system based on a realtime microkernel called XTAL[2].

Along the left side of the screen, we created a strip of touch-sensible, silk-screened navigation buttons: **Home**, **Back**, **Camera** and four symbols. The Home and Back button function like their equivalents in a typical web browser. We dedicated a button to the camera because we wanted guests to be able to access the Parks PDA camera as quickly as an ordinary camera.

Table 1: The specification of Sharp MI-C1

dimension:	136mm × 80mm × 22mm
weight:	190g (incl. second CF slot)
CPU:	Hitachi SH-3 60MHz
display screen:	320 × 240, 16 bit color reflective LCD
Operating System:	Sharp proprietary OS called ZaurusOS
Memory capacity:	16MB DRAM
Sound Output Quality:	8-bit sample, mono, up to 22.5kHz sampling rate
Com Ports:	one serial port and one IrDA port
Expansion Slots:	2 Compact Flash type I slots.
Miscellaneous features:	Silk screened touch-sensitive area.
Key Performance Numbers	
Read throughput CF card:	440kb/s
Read throughput DRAM	10.3MB/s
Write throughput DRAM	5.7MB/s
Typical Battery Life:	10 hours; 1.5 hours with camera on

We plugged a proprietary IR location sensor (about 40 × 15 × 10mm) into the serial port. This sensor received location codes at various locations in the park. These location codes were used to help guests navigate and to trigger location-specific events, such as playing an animated welcome message when the guest first enters a new area.

We used the MI-C1's IrDA port to download guest registration information and to upload their photos and log files.

We built an impact-resistant plastic case to protect the unit, keep its attachments in place, and provide a themed look for a device that might otherwise look like a business tool. We also added a neckstrap with an integrated "earbud" style headphone to allow the user to hear audio output without disturbing other guests. We assumed that the stylus would be easily lost, so we discarded it. The UI was driven by pressing one's finger directly against the touch screen.

4. THE SOFTWARE PLATFORM: SQUEAK

Squeak is a highly portable, open-source implementation of Smalltalk-80 that provides nearly bit-identical operation across platforms. The Squeak virtual machine (**VM**) implementation consists of three parts. The *bytecode interpreter*, the *object memory system* and a collection of *primitives*. The bytecode interpreter and the object memory are written in Squeak itself and translated into ANSI C code. They are *exactly* the same on all platforms. Most of the primitives are also portable, but a small handful of primitives are OS-specific, including primitives for display output, touch-screen input, clocks, file system access, sound, and so on. These primitives and a small amount of startup code must be ported to make Squeak run on a new platform [4].

Squeak uses a *virtual image* model. A snapshot of the state of Squeak's object memory can be saved in an *image file* that can later be resumed from the point at which the snapshot was taken. Since Squeak runs on a platform-independent virtual machine, the image file format is free from actual hardware dependencies. A Squeak image file can be resumed and run on any computer for which you have a Squeak virtual machine, even if the OS, processor, or even the byte-ordering is different from that of the platform that wrote the image file.

4.1 Squeak VM extensions

In addition to porting Squeak to the Zaurus, we also added a number of primitives to access Zaurus-specific hardware

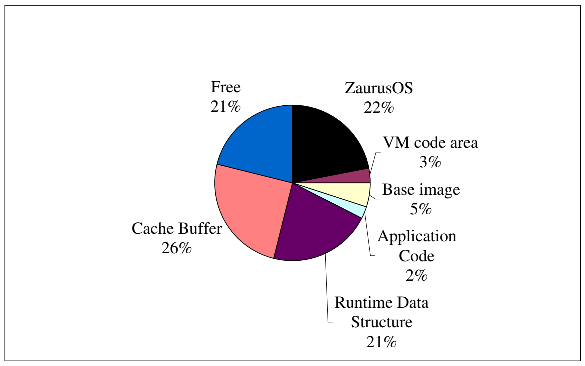

Figure 2: Runtime Memory Usage of the 16MB DRAM. "ZaurusOS" includes the work area of JPEG library and the camera, and the native stack. The "Free" area satisfies the transient memory needs of the applications.

features including:

- camera control and image capture
- audio output
- power saving and sleep
- reading the current battery level
- checking the amount of free space on the Compact Flash card
- configuring the Zaurus silk-screened button strip.

We also added primitives improve the performance of key graphic operations including:

- scaling and alpha-blending 16-bit images
- JPEG image compression and decompression

The ability to port and easily extend the virtual machine was essential, and Squeak's open-source, portable design made it easy. Only about 6700 lines of C and 2000 lines of Smalltalk-80 glue code were needed to accomplish the port to the Zaurus and to implement the custom primitives.

4.2 Memory Footprint

Memory is a limited resource on handheld devices. Squeak helped us use that resource efficiently.

We stripped down a version 2.4c Squeak image by removing classes and methods that we didn't need. This "base image," containing only 204 classes and 4872 methods, was 835k bytes[1]. Our application code added 88 classes and 1897 methods. This code and it's associated objects added 134k bytes of CompiledMethods (these are mostly bytecodes) and 267k bytes of additional objects to the base image.

[1]The version 3.5 release image of Squeak distribution is more than 10MB in size and contains more than 1800 classes and over 41000 CompiledMethods.

The virtual machine also takes up space. The core of the Squeak VM, the interpreter and the object memory compile into about 150k bytes of SH-3 code. However, native libraries such as the JPEG decoder/encoder add hundreds of kilobytes to the executable. With all the necessary libraries, the final virtual machine executable file size was close to 500k bytes.

Figure 2 shows the breakdown of the memory usage at runtime. The total DRAM capacity on the device was 16MB. We allocated 12MB for Squeak heap, leaving 4MB for the OS and the Squeak virtual machine. As shown in the figure, the relative overhead for the Squeak virtual machine and the base image is quite small. This allowed us to use memory space for working with large graphics files and, as we'll see, for a large image cache that was important for overall performance.

5. SOFTWARE ARCHITECTURE

5.1 Applications

The core Parks PDA application is an interactive information retrieval program based on hyper-linked pages, similar in concept to the World Wide Web. A Page contains zero or more links to other pages, and the user navigates through the content by following these links. We call this core application **Stacker**, since it models a stack of cards (pages).

Many of the pages and links on the World Wide Web are textual, but we wanted the Parks PDA experience to be primarily graphical. Thus, every page of our content is a full-screen graphic image and every hyper-link is a graphical button. Using pre-rendered, full-screen images gave the artists complete control over the visual experience; there were no tool bars, window, or menus that might remind guests of daily business tools or frustrating computer experiences.

However, high-quality graphics require memory. Each full-screen image is a 320 × 240 pixel, 16-bit color bitmap. Some pages had smaller images or animated sprites in front of the full-screen image. While the content for a Parks PDA might ultimately be delivered over a wireless network, for this test we kept the content on a 128 megabyte Compact Flash storage card. This allowed us to create a user test with the maximum quality graphics; our only limitation was how much content would fit onto the Compact Flash card.

In addition to Stacker, the Parks PDA software includes thirteen other applications as shown in Table 2. Navigating among these applications is completely seamless to the user.

To help us analyze usage patterns, we logged guest actions as they used the Parks PDA. Each log entry includes a time stamp, the action taken (e.g., page transition, photo taken, or application switch), and additional action data such as the destination page or application.

5.2 UI Considerations for Handhelds

Several considerations arose when designing the user interface for the Parks PDA. First, a handheld device has different display and input affordances than a desktop, laptop, or even a tablet computer. For example, it's screen is far too small for overlapping windows to be effective. Furthermore, theme park guests should be enjoying the park around them; we didn't want to burden them with complex UI interactions such as scrolling, window manipulation, or menu selection. Finally, the MI-C1, like most PDA's, uses a

Table 2: The applications in Parks PDA

Name	Description
Stacker (main app)	player/editor of hyper-linked pages
Map	zooming, hyper-linked park map
Talking Mask	lip synched talking virtual host
Restraunt Reservation	make priority seating reservation
Online shopping	buy park exclusive goods
Fastpass	get FASTPASS® ride reservation ticket remotely
Bugboms	an artillery game
Camera	digital camera interface and photo album
Kilimanjaro Quiz	trivia game
PT Flee's Word Circus	a hangman like game
Symbol Code Entry	enter four digit code to get info
Survey	online survey
Guest Registration	register the device to the guest
Uploader	upload photos, log files, and survey data via IrDA

touch screen that can only track the pen (or a finger) while it is in contact with the screen. Techniques such as roll-overs and changing the cursor to indicate the active user interface elements cannot be used, since they depend on being able to track the pointing device while the "mouse button" is up.

For these reasons, we adopted a simple button-pressing convention for all user inputs, similar to touch-screen interfaces found on information kiosks. We used drop shadows and a 3-D look as a visual cue that a graphical object was a pressable button. Users pressed these screen buttons with their finger. Since a fingertip is less precise than a stylus, our screen buttons had to be large, with buffer space around them to avoid accidentally pressing an adjacent button. Even with these precautions, user testing revealed that the buttons were not quite large enough for some users with exceptionally large fingers. Overall, we were happy with our decision to omit the stylus, even though it meant that screens collected finger prints and had to be cleaned daily.

The software for the Parks PDA was built using two object-oriented frameworks: PenSprites, a stripped-down user interface framework, and PDAApplications, an model to switch applications similar in spirit to Java Applets. A third package manages images and other media. The remainder of this section describes these three software components.

5.3 PenSprites

PenSprites is simple, lightweight user interface framework that, like Morphic[12][7], combines the roles of display, user input, and autonomous behavior (animation or "stepping") into a single class, Sprite. As in Morphic, it is easy to create new kinds of Sprites with custom appearance and/or user input behavior by making a subclass of Sprite and adding just a few methods. However, PenSprites is much simpler than Morphic: PenSprites consists of only 18 classes and 323 methods, and it takes up only 50k of space. In contrast, a recent version of Morphic has over 400 classes and 10,000 methods, and consumes several megabytes. While we could have created a stripped-down version of Morphic for this project, it was quicker to write PenSprites from scratch. The resulting framework is tiny and precisely tailored to handheld, touch-screen devices.

At the core of PenSprites are two key classes, Sprite and Stage. A Sprite knows its own position and bounds, can draw itself, handles pen and keystroke input, can perform periodic activities, and can do simple animations involving position, size, and transparency. The class Sprite defines default behaviors for all these things, although in many cases that default behavior is to do nothing. A Stage holds a collection of Sprites, manages display updates, dispatches user inputs, and processes any ongoing Sprite animations and periodic activities. A Stage also handles any pen and keyboard events that aren't handled by a Sprite.

Both Sprite and Stage are designed to be specialized by subclassing. When building a PenSprites application, one creates a subclass of Stage to hold the application state and behavior and subclasses of Sprite for each new kind of graphical object used by that application. For example, the BugBoms game defines its own subclass of Stage and eleven subclasses of Sprite. However, BugBoms is a fairly complex game with several screens and many custom widgets; most applications are made up of Sprites from a small library of reusable widgets.

When PenSprites application is running, it's Stage repeatedly invokes the following method:

```
doOneCycle
    self processInputs.
    self processActions.
    self updateDisplay.
```

The methods processInputs, processActions, and updateDisplay, as well as doOneCycle can all be overridden and specialized by an application stage. For example, PDAMoviePlayer overrides doOneCycle to check the millisecond clock and advance to the next frame of the movie if necessary.

5.4 PDA Application Switcher

Although using the Parks PDA is a seamless experience, its implementation actually consists of fourteen applications, each of which is a kind of Stage (see Table 2). The main application is a hyper-linked card stack containing navigational screens and the bulk of the content. Certain pages in this "main stack" are placeholders that link to the other applications.

An instance of PDAApplicationSwitcher manages this suite of applications. At any given moment, there is one active application that processes user inputs, runs animations and other ongoing activities, and updates the display. However, PDAApplicationSwitcher has its own version of the Stage interaction loop that calls the active stage's doOneCycle and also:

1. checks for ride reservation state changes
2. checks the battery level
3. polls the location sensor
4. returns to home card after a few minutes of idle time
5. optionally, puts the device to sleep when idle

Of course, one could also use threads to perform these background tasks, but threads can be tricky to reason about and debug. Furthermore, tasks are not always independent. One might want to conserve processor cycles by omitting battery level checks and location sensor polling during a performance sensitive animation sequence. It's easy to implement such policies in PenSprites.

The application switcher allows the code for the Parks PDA to be factored a number of small applications that can be independently implemented and tested. Each application retains complete control over its own screen display, user input, and background activities. Care was taken to avoid extra screen updates when switching between applications, so application switches are fast and appear seamless to the user. Many of the applications are quite small, some as few as one or two methods.

The application switcher was implemented fairly late in the development process, after many of the applications had already been written. It is notable that it required few changes to the existing applications. Most applications were converted by merely inserting the class PDAApplicationStage above them in the class hierarchy. We attribute the ease of making this change to the clean design of the PenSprites framework, OOP in general, and Smalltalk-80 in particular.

5.5 Managing the Media

A multimedia application uses many images and sounds, which require a lot of space. The media for a large application like Parks PDA cannot all fit into RAM at once; it must be kept in external storage until needed. One way to to do this is to store every piece of media – or “asset” – as a separate file. However, for an application with hundreds of assets, managing file names and versions can be a tedious and error-prone business. Furthermore, all those files must be dealt with when moving the application among machines or sharing it among team members. Some file systems have trouble handling large numbers of files. Finally, when space is tight, the amount of space lost due to rounding file sizes up to the device’s block size can be significant.

To avoid these potential problems, we implemented a simple media storage system that stores a collection of assets in a single file. Assets are stored sequentially in the file as records of the form:

```
<asset ID> (3 bytes)
<type ID>  (2 bytes)
<size>     (4 bytes)
<serialized asset data...>
```

A new asset is added by assigning it a unique ID and appending a new record to the asset file. An existing asset can be updated by appending a record containing a new version of that asset to the file. When an asset file is opened, its assets are scanned sequentially while constructing a dictionary mapping asset ID’s to file offsets. Versions of the same asset stored later in the file merely update that asset’s dictionary entry. When the scan is complete, every dictionary entry points to the most recent version of its asset. An asset file can be compacted to remove all but the latest asset versions to reclaim space.

Asset files greatly simplified media development. It was easy to keep the assets together, it was easy to move them onto a Compact Flash memory card for testing on the device, and it allowed artists to work together more easily. Initially, each artist had one or more asset files for their sections of the project. As the project progressed, these asset files were merged into a small number of master asset files. Artists would often take turns working on an asset file, each making some changes before passing the file on to another artist. In fact, this process worked so smoothly that we were able to make additions and changes to the content while user testing was underway.

Table 3: Compression Comparison on MI-C1

Format	File Size (bytes)	Read (r) (msecs)	Decode (d) (msecs)	r + d (msecs)
RAW	153600.0	362.99	1.18	364.17
RLE	124593.2	277.93	82.02	360.01
JPEG	18881.8	43.81	695.49	739.30

Table 4: Compression Comparison on iPAQ

Format	File Size (bytes)	Read (r) (msecs)	Decode (d) (msecs)	r + d (msecs)
RAW	153600.0	123.47	0.24	123.71
RLE	124593.2	90.80	11.62	102.42
JPEG	18881.8	15.07	44.15	59.22

6. PERFORMANCE TUNING

Adequate performance is critical for a good guest experience. A key goal is to minimize the latency between a button press and the moment a response appears on the screen. Most of the Parks PDA content consisted of one full-screen image per card. Thus, in response to a button press the system typically had to fetch and display a full-screen image.

The time for this operation is determined by 1) reading a screen image from Flash memory, 2) decoding that image if necessary, and 3) updating the display. These operations were all implemented as primitives written in C. As with many media-rich applications, performance is so completely dominated by the media-manipulation primitives that the speed of the bytecode interpreter is almost irrelevant.

Compared to desktop and laptop computers, mobile devices have slow processors, narrow data paths, and small or non-existent caches. To our surprise, we found that accessing Flash memory on the Zaurus MI-C1 was slower than reading from a disk drive on a desktop computer. This is due to an expensive error-correcting code computation. Thus, the effective read speed of Flash memory depends on the speed of the CPU. On the Zaurus MI-C1, data can be read from a Compact Flash card at about 440k bytes/sec.

The next two sections explain how we achieved adequate performance even on a device with modest computational power.

6.1 Data Accessing Speed

A full-screen image (320x240 pixels, 16-bit color) consumes about 154k bytes in uncompressed form. Thus, at 440k bytes/sec, it takes roughly 1/3 second to read an uncompressed image from the Compact Flash card. If the image is stored in compressed form, the reading time will decrease but that savings may be exceeded by the image decompression time. One must make empirical measurements on representative content to determine the best compression strategy for a given device.

We considered three compression options: no compression (RAW), Squeak run length encoding (RLE), and JPEG encoding. Run length encoding works best for content with areas of solid color, whereas JPEG works best on continuous-tone images such as photographs and natural textures. Both compression schemes are implemented using fast VM primitives. Table 3 shows the tradeoffs among these options for the Zaurus MI-C1. The numbers are the average of the *real* content of the Parks PDA, which contains 415 full screen images created by professional designers.

As the table shows, using RLE compression doesn't save significant time, but it does yield a modest 19% space savings. JPEG compression significantly reduced the contents size and hence the reading time, but unfortunately decoding JPEG images was unacceptably slow on the MI-C1. We thus choose RLE compression for the Parks PDA.

Table 4 shows how the tradeoff between read time and decompression time depends on the actual device. On the Compaq iPAQ 3600 implementation of the Parks PDA, using JPEG compression is the best choice. The high speed of JPEG decoding on the iPAQ is due in part to our use of a fast JPEG library from Intel: the *Intel Integrated Performance Primitives* (**IPP**) [6].

The final step is to display the decompressed image on the screen. Compared to reading and decompressing the image, this operation is relatively quick: 52.4 milliseconds on the MI-C1 and 10.5 milliseconds on the iPAQ.

6.2 Caching and Prefetching Assets

Unfortunately, image compression did not solve our performance problem. We decided to use 4 megabytes of DRAM as a cache for images. Images in the cache would not need to be read or decompressed, resulting in a much faster response time.

The cache was simple: decompressed images were cached using an LRU replacement algorithm. We implemented this cache as a subclass of AssetFile so no client code had to be changed. A cache hit returned the cached asset immediately; a miss called the normal asset file read operation, then added the newly-read asset to the cache before returning it. This worked well since frequently used pages such as the home page and the map page were often cached. Response to the "back" button was also fast since recently visited pages would also be in the cache.

However, we realized we could do even better. We decided to anticipate upcoming asset requests and *prefetch* these assets into the cache using time that the processor would otherwise be idle. Prefetch predictions were made by examining all links on the current page. Since users often spend several seconds viewing a page, there is time to prefetch images for several of these possible next pages. Once the user clicks on a link, the prefetch queue is cleared and a new prefetch cycle is started. We arbitrarily chose to limit prefetching to the first three outgoing links.

One might imagine that prefetching would always be successful if the user paused long enough. However, links on the page are not the only way to navigate to another page. The user can enter a symbol code to jump to an arbitrary page. A page reached via symbol code entry is much less likely to be in the cache. Another case is a page containing a large number of outgoing links. While many pages had four or fewer links, certain index pages (e.g., the page listing all the restaurants) had many more.

How well did caching and prefetching work? After the test, we analyzed 289 log files taken during the last four days of our user tests. In an average session, caching and prefetching resulted in cache hits 77.1% of the time over 336.3 page visits.

There are several questions one might ask. First, what is the breakdown of the cache effectiveness between caching and prefetching? Second, what would have happened if the cache size was smaller or bigger? To answer these questions, we built a simulator to replay user actions from the log file and gathered data for different combinations of caching and prefetching.

Table 5: Cache hit rates for various policies

Setting	Hit Rate
1. Original policy	77.1%
2. Caching only	62.6%
3. Double prefetching	78.1%
4. Double cache, double prefetching	85.1%
5. Half-sized cache, no prefetching	52.1%

Table 5 shows the result of this simulation. The table shows five cache policy variations, including the one we used in the actual test: 1) the original policy, 2) caching only (no prefetching), 3) doubling the number of links examined during prefetching (to 6) with the same the cache size, 4) doubling the number of links examined during prefetching and also doubling the cache size (to 8 megabytes) and 5) caching only with half the cache size (2 megabytes) and no prefetching.

These results show that, while caching alone was quite effective, prefetching added a significant 14.5% percent to the hit rate. Doubling the number of links examined during prefetching with the same cache size would not have improved the hit rate significantly, but doubling both the cache size and the prefetching (had we had the memory to do so) could have improved the hitrate by 8%. Cutting the cache size in half would have decreased performance significantly. On the whole, our cache size and prefetching parameters worked well for the amount of memory that was available. We didn't have any usage data when we set these parameters; we were just lucky.

As showed earlier in section 3, reading and decoding a typical full-page asset takes more than 360 milliseconds, a delay that feels noticeably sluggish. When the image for a page is in the cache, as it is 77% of the time, the response time is dominated by the display update time. At only 52.4 milliseconds, this feels quite snappy. Based on our surveys, users were entirely happy with the performance of the Parks PDA.

7. RAPID DEVELOPMENT

All the media content and most of the software for the Parks PDA was created by a team of 9 artists and two programmers in about three months. Considering the number of features, the integration with hardware, and our lack of prior experience developing software for handheld devices, this was extremely fast. Some of that speed can be attributed to the innate efficiency of the Smalltalk-80 programming environment. However, key aspects of the development process were enabled by features unique to Squeak. This section explains how Squeak supported rapid development.

7.1 Put Artists in Control of Content

It was clear from the beginning that if the artists depended on the programmers to assemble content then the programmers would be the bottleneck. One of the artists pointed out that much of the desired system behavior could be modeled as a stack of cards with links for navigating to other cards. We thus implemented a simple card-stack editor called "Stacker" that allowed artists to create cards, import artwork as images attached to these cards, and add links between cards. All artwork for cards, as well as the card list itself, was stored using asset file format described in section 5.5.

A key point is that the software that ran in the guest's Parks PDA was exactly the same software that was used for development, but with a flag set to disable editing. Thus, as the artists used Stacker to develop content, they were also testing the final software that would be used by guests.

7.2 Bit-Identical Cross-Platform Operation

We distributed the Stacker program as a Squeak virtual image set up to run Stacker on startup. Because the Squeak virtual machine had already been ported to Linux, Mac OS, and Windows, this one virtual image could be run on the artists' desktop computers regardless of what kind of computer and OS they had– and all three of these platforms were used. The same Stacker image could also be run on the Zaurus MI-C1, so artists could simply put Stacker and their current stack on a Compact Flash card to preview their artwork as it would be seen by guests. This was important because contrast and color differences made artwork "read" differently on the MI-C1.

While it didn't surprise the artists, it's actually unusual for a program like Stacker to operate virtually bit-identically across such a wide range of hardware and OS platforms. Squeak achieves this feat as the result of a number of design decisions, especially the fact that Squeak's class library was built to be as self-sufficient and platform-independent as possible.

Take fonts, for example. Many programming systems rely on the fonts of the underlying operating system. When an application is moved from one system to another, some font may not be available, so a substitute is used. This causes changes in the appearance and spacing of text, which effects the placement and layout of user interface elements. Squeak, in contrast, implements its own text display and every Squeak image carries it's own fonts, so the appearance and letter spacings are bit-identical on all platforms.

As another example, Squeak hides platform differences in graphics output by defining its own virtual display screen. All graphical operations on this virtual screen behave identically on all platforms right down to the pixel representation and byte ordering. There is only one place that knows about the native frame buffer representation: the C code in the virtual machine that copies pixels from Squeak's virtual screen to the hardware frame buffer.

One might think that maintaining a virtual display screen would adversely impact performance and memory footprint. However, smooth animation requires the use of an off screen display buffer to avoid flashing, and Squeak's display serves as that buffer. Thus, assuming the application uses double-buffering, there is scarcely any additional cost.

7.3 Simulate Missing Hardware

In some cases, an application needs hardware that's unique to a given platform, such as the Zaurus camera and silk-screened button strip. In this case, our approach was to simulate these hardware features when running on other platforms to support cross-platform development. For example, when running on a desktop computer, the Zaurus silk-screened buttons were simulated by a set of virtual buttons to the left of the virtual PDA screen.

We actually developed and tested most of the camera application on a laptop computer. After all, none of the logic for capturing and reviewing photos cares about the content of those photos, so any image of the right size will work as well as a real photo. However, the ZaurusCamera class uses primitives specific to the Zaurus PDA platform that fail on any other platform. So, for development purposes, we created a subclass of ZaurusCamera called DummyZaurusCamera that replaces those primitives with methods that emulate camera behavior in simple ways. For example, the shutter button is emulated by the enter key and taking a picture with the camera is emulated by grabbing a snapshot of the screen.

Using DummyZaurusCamera, we were able to exercise the camera application on any desktop computer; we did not need to download it to a PDA. This allowed us to iterate through the testing cycle quickly, allowing us to work in tight collaboration with a graphic designer, making and testing changes together as we refined the look and feel of the user interface.

Overall, the small investment of time required to simulate the unique hardware features of the PDA paid off many times over in saved development time. As a bonus, we can demo the Parks PDA application to large groups using a laptop computer and video projector.

7.4 Embedded Development Environment

Squeak, like other Smalltalk-80 systems, has a built-in development environment that allows every method in the system to be viewed, edited, and debugged at runtime. While it is possible to jettison this programming environment to recover several hundred kbytes of memory, we had sufficient RAM to retain the development environment in the virtual image we ran on the device – even when we deployed the application for the user tests. That decision paid off in many ways.

Having the development environment available on the device had many uses during development. First, it was easy to evaluate expressions to test pieces of code in isolation. This was especially valuable for measuring performance, allowing the programmer to quickly isolate performance bottlenecks and compare the performance of alternate implementations. It was also useful when debugging hardware specific VM features, such as the camera support code and optimized graphics primitives. For this, the PDA VM was connected to a remote debugger running on a PC. Break points were set in the C code of the camera support code and Smalltalk expressions were evaluated one at a time to see what happened. The Smalltalk expressions could be edited to pass different arguments to the new primitives to test different cases and the results could be seen immediately. For example, one could evaluate an expression to grab a frame from the camera and display it on the screen. This interactive testing was much faster than it would have been to write and compile tests in C. Since we could use the results of earlier tests to decide what to try next, we were able to make quite a bit of progress in isolating problems before recompiling the VM. Since it took many minutes to build a new VM and install it on the device, this technique saved a great deal of time.

As mentioned, the Squeak development environment, including the debugger, was so small that we decided not to bother removing it from the final application. This turned out to be an unexpected life-saver during user testing.

The day before the user test began, we tested the Parks PDA on a group of park employees who had no prior exposure to the project. This testing revealed only a few minor problems that were easy to fix. After making these changes and loading them onto all 100 Parks PDA devices, we were confident that the first day of user testing would go smoothly. We were very wrong.

Within half an hour, guests began bringing back PDA's that weren't working. Oddly enough, other guests were having no problems. The symptoms did not seem to be related to anything we had changed. However, because the debugger was still available, we were able to examine the execution stack, look at variables, and evaluate expressions. After examining about six units, a pattern emerged: one of the asset files was corrupted. The essential clue was that the same exact error appeared at the same file location in all six units. This was clearly not a coincidence. We deduced that one of the laptops we had used to copy asset files onto Compact Flash cards had somehow corrupted the master asset file and that error had been copied to all the Compact Flash cards created on that laptop. We even knew which computer was probably responsible, since one of us had had trouble reading files from the master Compact Flash card. We added some code to verify the structure of asset files at startup time, and re-copied the master file set onto all 100 devices, this time avoiding the suspect laptop, and never had this problem again for the remaining eight days of user testing.

In most other programming languages, the error handler's capability is limited. When an unexpected error occurs during execution, the best that may happen is that the call stack is displayed or written to a file and the program quits. Unfortunately, the call stack alone may not be enough to diagnose a problem.

It is not exaggerating to say that having the full debugger and development environment available at deployment time saved the user test from disaster. The debugger allowed us to find the crucial clues that pointed to a file corruption problem during Compact Flash card duplication. The problem was not due to a bug in the software at all, but without any hints about the real nature of the problem we might have spent days fruitlessly trying to reproduce the "bug" under laboratory conditions, rapidly using up our two-week window of opportunity for the user test.

8. SEVEN KEYS TO PROJECT SUCCESS

A number of factors contributed to making Squeak an especially effective vehicle for the Parks PDA project. While some of these factors are a consequence of Smalltalk-80's clean, object-oriented semantics, most of them have little to do with the language; they stem from Squeak's open implementation and flexible packaging. In practical applications, such "details" can often make or break a project.

Pointer safety
Smalltalk-80 programmers take point-safety for granted, yet this property of the language allowed us to fearlessly make changes to the software in response to newly discovered needs. For example, during the first week of the user test, we added a clock to the home screen, a set of hardware diagnostics, and a blinking reminder that a ride reservation was due, in addition to fixing a number of minor bugs. We dared to do this because code changes had local effects. When adding a feature, we did not worry that we might introduce a wild pointer bug or storage leak that would break some completely unrelated–and perhaps critical– part of the system.

Portable, open-source virtual machine
As we've shown, one key to our rapid development of both software and content centered on our ability to do cross-platform development. Squeak already ran on over a dozen platforms, including Windows, MacOS, and Linux. But because our chosen device was new and ran the proprietary Zaurus operating system, we needed to be able to do our own port of Squeak to it. Fortunately, Squeak's interpreter-based virtual machine is extremely portable. It typically takes only a few weeks to get it running on a new platform.

Small memory footprint
Squeak's relatively modest memory footprint–roughly 500k bytes for the virtual machine and about 800k bytes for the base image, including the full development environment and debugger–left us plenty of room for application code and media. In fact, we were able to devote many megabytes of RAM to a media cache and we were not even tempted to remove the development environment.

Low-latency garbage collector
Predictable response times and smooth animations are essential for a quality multimedia experience. Large garbage collection pauses would significantly detract from the user experience. Squeak has an efficient increment garbage collector that runs often but usually takes under 20 milliseconds. Such short pauses are unnoticeable even during animation and user interaction. Although a full garbage collection can take a second or two, full-GC's happen very rarely in practice.

Platform independence
Cross-platform development could have been a disaster had the application behaved differently on the target device. Thanks to Squeak's deep commitment to platform independence, the operation of the software was bit-identical on all platforms. The only differences were unavoidable ones, such as processor performance and color/contrast differences between computer monitors and the reflective LCD screen.

System-level control and custom primitive support
Having complete control over the system as if Squeak were it's own operating system allowed us to manage interactions with hardware devices, control system sleep mode, and use knowledge of user activities to control the allocation of processor cycles. In addition, Squeak's open virtual machine allowed us to add primitives to accelerate performance-critical operations.

Self-contained development tool and debugger
The ability to evaluate expressions and do small amounts of programming directly on the device saves development time. The programmer can get much more testing and debugging done before having to perform the time-consuming process of copying a new version of the application from the development machine to the device. Bugs that arise during such testing can be tracked down using Squeak's built-in debugger. In fact, having Squeak's built-in debugger available even after we'd deployed the "finished" application may have saved the entire project. Having the debugger on board

is like using a seatbelt; you hardly ever need it, but putting it on after a collision does no good at all.

9. BUILDING A PARKS PDA TODAY

Mobile computing technology has evolved rapidly since our Parks PDA user test in November 2000. This section discusses recent innovations and how we might use them if we were to create a new Parks PDA today.

First, PDA processors and displays have improved dramatically. While a few handheld devices such as the Apple Newton and the Palm Pilot were available in 2000, practically none of them had outdoor-viewable color screens. In fact, we believe that the Sharp MI-C1 was probably the only product then available that combined a reflective LCD screen, a camera, a compact and lightweight form factor, and support for an external memory card. Today, many PDA's have outdoor-viewable color displays, and most of them also boast high performance/low-power processors such as the Intel XScale chip. These devices would make it much easier to achieve acceptable performance.

Cellular telephones have become powerful computing devices. Some high-end cellphones sold in Japan have a two-inch QVGA color screen, a mega-pixel digital camera, and a faster processor than the MI-C1. As a hardware platform, such a cellphone is a good candidate. Unfortunately, the software environment available on cellphones is closed, making it difficult or impossible to control cellphone peripherals such as the digital camera or to write primitives in a low-level language to enhance performance. This situation may change as cellphones and PDA's converge.

At the time of our test, GPS was available, but not in a small form factor. It has now become much smaller and less power-hungry, but typical GPS systems still don't provide resolutions down to the one to five meters that would be ideal for theme park applications. Furthermore, GPS often doesn't work well when the sky is partially obscured by foliage or buildings, and it can take minutes to get enough data for an accurate fix. Thus, we believe that our proprietary, IR-based system remains the cheapest and most reliable location-sensing solution for theme parks. Several other mobile guide efforts such as Active Badge[3] and Cyberguide[8] are based on an IR system similar to ours.

What about wireless? The Parks PDA did not have wireless communications, although we pretended that it did to simulate the user-experience of ride and restaurant reservations. While the 50 to 200 kbits/sec bandwidth available through current cellphone networks could be used to provide informational updates, instance messaging, and ride reservations, it would require significantly more bandwidth to retrieve all the Parks PDA graphical content wirelessly. Today, we might consider using an 802.11 network for this purpose, although it is not entirely clear how well this would scale. Theme parks pack many people into a small area and even if only few percent of them carry Parks PDA's, the numbers can get large. Meanwhile, Flash memory has fallen in price so it may still be best for graphical content to be pre-loaded onto the PDA.

We might need to consider alternatives to Squeak for the software platform. For devices with insufficient memory to use Squeak (under 1.5 megabytes) such as PalmOS devices and a cell phones, options include Java J2ME[11] or PocketSmalltalk[1]. However, since many of the success factors discussed in section 8 do not apply to these languages, we would have to budget more time for development.

Actually, Java J2ME is the only way to program most programmable cell phones. Unfortunately, the J2ME virtual machines for cell phones are not open, so we would not be able to add primitives to access camera hardware or perform high-performance graphic operations.

One option for more open platforms would be to port the virtual machine for Python[9] or Ruby[13]. To support multimedia, we would write our own cross-platform graphics library, possibly using a portable GUI library such as Qt or Qt/Embedded[10]. Our goal would be to create a tool with bit-identical operation across platforms as good as Squeak. Of course, it would be much easier to just use Squeak!

10. CONCLUSIONS

The Parks PDA project was an ambitious undertaking. Part of the challenge involved creating a production-quality multimedia experience on a new device while interfacing to a camera and other special hardware. Cross-platform development of both software and content was essential for rapid development. Squeak supported cross-platform development with essentially bit-identical operation between three different desktop platforms and the device itself.

One of the big lessons from this project is that using a bytecode interpreter need not result in low performance, even in a performance-hungry multimedia application. Most of the time is spent in a few key operations, such as reading, decompressing, and displaying images. It is easy to write primitives in C to perform these operations at maximum speed. In fact, Squeak made it so easy to pinpoint and fix performance bottlenecks that we're confident that the performance of the Parks PDA software is very close to the maximum performance limits of the hardware for the key operations.

We were fortunate that we had already ported Squeak to the Zaurus platform before the project began. Yet, if we had to do a similar project again, we would be glad to invest the two to four weeks necessary to port Squeak to a new platform, knowing that this investment would pay off many times over for the rest of the project. If we had to use a different language, we believe that most of the key success factors that we discovered using Squeak could be available in Python, Ruby, or Java J2ME, assuming that one could extend the virtual machine. We'd avoid C++ due to its lack of pointer safety and true garbage collection.

Squeak turned out to be the perfect tool for the Parks PDA project. It allowed us to move quickly and to explore completely new territory on a handheld device. What's more, Squeak made the Parks PDA as much fun to develop as it is to use.

Acknowledgements

We thank all the team members of Parks PDA project, the engineers at Sharp Corporation, and all cast members at Disney's Animal Kingdom™. We especially thank Alan Kay and his team for creating Squeak and Squeak's open source community for their ongoing contributions to it.

11. REFERENCES

[1] E. Arseneau. Pocketsmalltalk. `http://www.pocketsmalltalk.com`.

[2] AXE, Inc. XTAL. `http://www.xtal.org`.

[3] A. Harter and A. Hopper. A distributed location system for the active office. In *IEEE Network*, volume 8, 1 1994.

[4] Ian Piumarta. *Porting Squeak*, chapter 8, pages 215–262. Prentice Hall, 2002.

[5] D. 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 *Object-Oriented Programming, Systems, Languages, and Applications*, pages 318–326, 1997.

[6] Intel Corporation. Intel Integrated Performance Primitives. `http://www.intel.com/software/products/ipp/`.

[7] John Maloney. An Introduction to Morphic: The Squeak User Interface Framework. In *Squeak: Open Personal Computing and Multimedia*, chapter 2, pages 39–68. Prentice Hall, 2002.

[8] S. Long, R. Kooper, G. D. Abowd, and C. G. Atkeson. Rapid prototyping of mobile context-aware applications: The cyberguide case study. In *Mobile Computing and Networking*, pages 97–107, 1996.

[9] M. Lutz. *Programming Python*. O'Reilly & Associates, 1996.

[10] Mathias Kalle Dalheimer. *Programming with Qt, 2nd Edition*. O'Reilly & Associates, 2002.

[11] Roger Riggs and Antero Taivalsaari and Mark VandenBrink. *Programming Wireless Devices with the Java 2 Platform, Micro Edition*.

[12] R. B. Smith, J. Maloney, and D. Ungar. The Self-4.0 user interface: Manifesting a system-wide vision of concreteness, uniformity, and flexibility. In *Object-Oriented Programming, Systems, Languages, and Applications*, pages 47–60, 10 1995.

[13] Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly & Associates, 2001.