Smartphone Cross-Compilation Framework for Multiplayer Online Games

A. Puder and I. Yoon
San Francisco State University
Department of Computer Science
1600 Holloway Avenue
San Francisco, CA 94132
Email: {arno|yoon}@sfsu.edu

***Abstract*—Social networks and multiplayer online games have drastically gained in popularity over the last decade. Likewise smartphones have become interesting targets for extending social networks and multiplayer games thanks to their innovative features such as intuitive user interfaces. Mobile versions of games and social network applications often make use of special capabilities of smartphones such as GPS and accelerometer. Porting these applications to different smartphones incurs high overhead due to their different programming models. To facilitate this porting effort, we introduce XMLVM, a byte code level cross-compiler to overcome the heterogeneity of the different programming models used by various smartphones. Specifically, we show how XMLVM can cross-compile an Android application to the iPhone and the Palm Pre, thereby significantly reducing the porting effort. The cross-compiler is based on Java byte code instructions that are translated to high-level programming languages supported by the targeted smartphone. Compatibility libraries map the API between smartphones. As a proof of concept, we have implemented a strategy game the demonstrate the feasibility of our approach.**

I. Introduction

The most dominant culture trends of Internet users of these days can be characterized as "being connected all the time," via social networks and multiplayer online games (MOG) [4]. Recent innovations introduced by smartphones accelerate this trend and create synergies between those two domains. MOGs thrive over social networks and extend the user's playing experience by getting connected using smartphones anywhere they go. In addition, supporting smartphone versions for MOGs can utilize their unique features such as GPS and accelerometer. However, one obvious challenge is making a decision which of the popular smartphones to support. Several smartphones share the market, among them Google's Android, Apple's iPhone, and the Palm Pre. All devices allow the development of native applications that can take advantage of specialized user interface elements and hardware components. While similar in capabilities, smartphones differ greatly in the way native applications have to be written for them. Google's Android is based on Java with an Android-specific set of widgets, while Apple's iPhone only supports Objective-C as the programming language of choice. In fact, Apple explicitly prohibits virtual machines on the iPhone per license agreement. The Palm Pre on the other hand uses JavaScript for the development of native applications.

Developers targeting smartphones ideally want their applications to be available on as many platforms as possible to increase the potential dissemination. Given the differences in the way applications are written for smartphones, this incurs significant effort in porting the same application to various platforms. In this paper, we introduce a cross-compilation approach, whereby an Android application can be cross-compiled to both the iPhone and the Palm Pre. The solution we propose not only cross-compiles on a language level, but also maps API between the different platforms. Although Android, the iPhone, and the Palm Pre differ in their user interface style guidelines, we believe that games are an ideal candidate for cross-compilation. Games typically take over the complete screen and use few special purpose widgets. Since all platforms support almost identical animation and graphic capabilities, games can readily be cross-compiled. The benefit of our approach is that only skill set for the Android platform is required and only one code base needs to be maintained for all three devices.

This paper is organized as follows: Section II gives an introduction to Android, the iPhone, and the Palm Pre. We present versions of "Hello World" for all three devices to highlight the differences in their programming models. Section III presents our cross-compilation framework that can cross-compile Android applications to the iPhone and the Palm Pre. In Section IV, we discuss our prototype implementation of this framework as well as a game that was developed using our toolchain. Finally, Section V provides a conclusion and an outlook to future work.

II. Smartphone Overview

Table I provides a comparison of the Android-based HTC G1, the iPhone 3GS, and the Palm Pre. The intent of the side-by-side comparison is to show that although all three smartphones are relatively similar with respect to their hardware capabilities, they differ greatly in their native application development models.

Android is a mobile operating system running on the Linux kernel. It was initially developed by Google and later the Open Handset Alliance. Android is not exclusively targeting smartphones, but is also available for netbooks and settop boxes. The iPhone 3GS is a proprietary product by Apple. Its

TABLE I
SMARTPHONE COMPARISON.

	HTC G1	iPhone 3GS	Palm Pre
OS	Linux	Mac OS	Linux
CPU	ARMv6, 528 MHz	ARMv6, 600 MHz	TI-OMAP, 600 MHz
RAM	192 MB	256 MB	256 MB
Sensors	Accelerometer, GPS, compass.	Accelerometer, GPS, proximity, ambient light, compass.	Accelerometer, GPS, proximity, ambient light.
IDE	Eclipse	Xcode	Eclipse
Dev-Language	Java	Objective-C	JavaScript
GUI	Android	Cocoa Touch	WebOS
Virtual Machines	Allowed	Not allowed	Allowed
License	Open Source	Proprietary	Proprietary

user interface is called Cocoa Touch which is an extension of the Cocoa framework that is used on Apple desktop and laptop computers. The Palm Pre is a proprietary product as well. It is based on the Linux kernel similar to Android, but uses web technologies (JavaScript, HTML, CSS) as the foundation of its framework called WebOS.

For developers it makes sense to target as many platforms as possible to increase the dissemination of their application. Targeting all three smartphones introduced here requires significant skill sets. Whereas Android uses Java as the development language, Cocoa Touch uses Objective-C and the WebOS JavaScript. These three languages are radically different. While Java features strong typing and garbage collection, the version of Objective-C used on the iPhone supports dynamic typing, but no garbage collection. JavaScript on the other hand is a dynamic, weakly typed, prototype-based language with first-class functions.

The same differences exist in the APIs and programming models defined by Android, Cocoa Touch, and WebOS. To better highlight the different programming environments, the following sections will show versions of "Hello World" for each of the three smartphones. The intent is to provide a brief introduction to the programming abstractions employed by the respective smartphone, but also to demonstrate the heterogeneity of smartphone application development.

A. Android

An Android application consists of a set of so-called *activities*. An activity is a user interaction that may have one or more input screens. An example for an activity is to select a contact from the internal address book. The user may flip through the contact list or may use a search box. These actions are combined to an activity. Activities have a well-defined life cycle and can be invoked from other activities (even activities from other applications). To write a "Hello World" application consequently requires the programmer to derive an application class from base class `Activity`:

```
1 // Java
2 public class HelloAndroid extends Activity {
3    @Override
4    public void onCreate(Bundle savedInstanceState) {
5        super.onCreate(savedInstanceState);
6        TextView tv = new TextView(this);
7        tv.setText("Hello World!");
8        setContentView(tv);
9    }
10 }
```

The main entry point of an activity is method `onCreate()` that also signals the creation of the activity. In case the activity was active at an earlier point in time, the saved state of the earlier incarnation is passed as an argument to `onCreate()`. Inside the `onCreate()` method, a `TextView` is instantiated and then made the main view of the activity via the call to `setContentView()`.

Besides a variety of widgets, Android also allows the declarative description of user interfaces. XML files describe the relative layout of a user interface which not only simplifies internationalization but also allows to render the user interface on different screen resolutions.

B. Cocoa Touch

The only official language offered by Apple for iPhone development is Objective-C, hence the "Hello World" application for this device is written in that language. Apple prohibits multi-processing per license agreement of its SDK (no background processes are allowed) which implies that the life cycle of an application is much simpler compared to Android. Applications have to be derived from base class `UIApplication` and the entry point is a method called `applicationDidFinishLaunching`. The following code demonstrates "Hello World" for the iPhone:

```
1 // Objective-C
2 @interface HelloWorld : UIApplication
3 - (void) applicationDidFinishLaunching:(NSNotification*) n;
4 @end
5
6 @implementation HelloWorld
7 - (void) applicationDidFinishLaunching:(NSNotification*) n
8 {
9    UIScreen* screen = [UIScreen mainScreen];
10   CGRect r = [screen applicationFrame];
11   UIWindow* window = [[UIWindow alloc] initWithFrame: r];
12   r.origin.x = r.origin.y = 0;
13   UILabel *title = [[UILabel alloc] initWithFrame: r];
14   [title setText: @"Hello World!"];
15   [title setTextAlignment: UITextAlignmentCenter];
16   [window addSubview: title];
17   [window makeKeyAndVisible];
18 }
19 @end
```

Square brackets in Objective-C denote a method invocation. The first argument represents the target of the invocation followed by the named arguments. Object instantiation always

happens in two steps in Objective-C: first sufficient memory is allocated via class method `alloc` and then followed by some variation of an `init` method that initializes the newly created instance. Since Objective-C is a strict superset of the C programming language, references to objects are regular C pointers.

In the program above, the dimensions of the main screen are retrieved (lines 9–10) first. Because of the top-level status bar of the iPhone, the origin of the `CGRect` returned in line 10 will be (0, 20). `UIWindow` represents the top-level window that the application will occupy. The `UILabel` is added as a sub-view to the `UIWindow` instance. The `makeKeyAndVisible` method makes the `UIWindow` instance the main window and makes it visible.

C. WebOS

Native applications for the Palm Pre are developed using web technologies. Since the Palm Pre uses the WebKit rendering engine internally, all standard HTML, CSS, and JavaScript techniques are permissible. The Palm Pre introduces the notion of *stages* and *scenes*. A stage is the platform on which an application builds its user interface. A stage generally corresponds to an application window. A scene is a formatted screen for presenting information or a task to the user. Both stages and scenes have so-called *assistants* that determine their behavior. The following JavaScript code represents a "Hello World" program for the Palm Pre:

```
1 // JavaScript
2 function StageAssistant() {
3 }
4
5 StageAssistant.prototype.setup = function() {
6   this.controller.pushScene("hello-world");
7 }
8
9 function HelloWorldAssistant() {
10 }
11
12 HelloWorldAssistant.prototype.setup = function() {
13   this.controller.topContainer().innerHTML = 'Hello World';
14 }
```

It should be noted that the example above is a programmatic version of "Hello World." Since every stage has its own `index.html` file, a simpler version would have been to just write the text string "Hello World" into this HTML file. However, to better compare the Palm Pre's programming abstractions we have opted for a programmatic version. The entry point for the stage assistant is function `setup()`. The stage assistant above pushes a new scene called `hello-world` (line 6). Not shown in the example above is a configuration file called `sources.json` that associates the name of the scene `hello-world` with class `HelloWorldAssistant`. The scene assistant has its own entry point also called `setup()`. Via a Palm Pre specific function `topContainer()` the top-level HTML element is retrieved. The assignment of a string to property `innerHTML` is standard HTML.

III. Cross-Compilation Framework

As shown in the previous section, different smartphones differ greatly in their programming environment. Not only do they offer different APIs but they also require different programming languages. In this section we introduce XMLVM, a flexible, byte code level cross-compiler, that allows to translate an Android application to the iPhone and Palm Pre. The main benefit is that only one code base needs to be maintained. In Section III-A we give an overview of the XMLVM toolchain. Section III-B explains our byte code level cross-compiler and in Section III-C we outline the API mapping.

A. Toolchain

We chose Android as the canonical platform. This means that a developer only needs to be familiar with the Android system and can then cross-compile an Android application to other smartphones. There are several reasons for choosing Android. First of all, we believe that there is a wide skill set for the Java programming language and there are powerful tools to develop for Java. We view this as an advantage over JavaScript and Objective-C.

The design of Android itself offers various advantages. For one, Android was not exclusively designed for smartphones, but for a wide range of mobile systems. Android's API allows to explore the device's capabilities to give the application the chance to adapt accordingly. In a way, an Android developer is expected to deal with different device capabilities. This is not the case for both the iPhone and the Palm Pre that treat their devices as a homogeneous platform. Since Android applications can more easily adapt to different devices, it makes them ideal candidates to be cross-compiled to different platforms.

Apart from this technical advantage, we also prefer Android because there is less of a lock-in. Android is maintained by a consortia and a reference implementation is available under a permissive Open Source license. The Android SDK is available for different platforms and the API is well documented.

Figure 1 depicts the XMLVM toolchain. Starting from a Java program we first use a regular Java compiler to generate a class file (1). The class file is used as input to our toolchain and as a first step we generate an XML representation of the contents of the class file (2). On the basis of the intermittent XML file we generate code in the target language, which in our case is either Objective-C or JavaScript (3). The following two sections explain in detail the cross-compilation process as well as the API mapping via a compatibility library.

B. Byte Code Level Cross-Compilation

A unique property of our toolchain is that we cross-compile from byte codes to high-level programming languages. We make use of the byte code instructions introduced by the Java Virtual Machine [5]. In a previous project we have used a similar approach to cross-compile Java byte code to JavaScript for AJAX applications [6]. Using byte codes has several advantages. For one, byte codes are much easier to parse than Java source code. Several language features such as generics are already reduced to low-level byte code instructions. The Java compiler also does extensive optimizations to produce

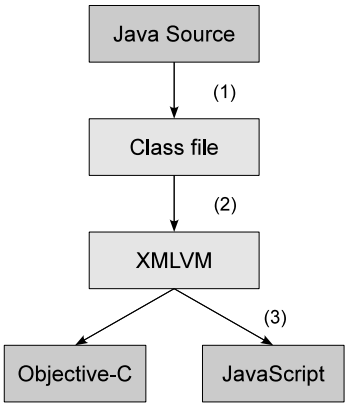

Fig. 1. Cross-compilation toolchain.

efficient byte codes. To illustrate our approach, consider the following simple Java class:

```
1 // Java
2 public class Math {
3   public static int factorial(int n) {
4     return n == 0 ? 1 : n * factorial(n - 1);
5   }
6 }
```

Class `Math` has one static method called `factorial` that recursively computes the factorial of an integer. The source code is first compiled to a Java class file via a regular Java compiler. The binary class file is then fed into our XMLVM tool. Internally, XMLVM generates the following XML document based on class `Math`:

```
1 <vm:xmlvm ...>
2   <vm:class name="Math" ...>
3     <vm:method name="factorial" ...>
4       <vm:signature>
5         <vm:return type="int" />
6         <vm:parameter type="int" />
7       </vm:signature>
8       <vm:code>
9         <jvm:var name="n" id="0" type="int" />
10        <jvm:iload type="int" index="0" />
11        <jvm:ifne label="0" />
12        <jvm:iconst type="int" value="1" />
13        <jvm:goto label="1" />
14        <jvm:label id="0" />
15        <jvm:iload type="int" index="0" />
16        <jvm:iload type="int" index="0" />
17        <jvm:iconst type="int" value="1" />
18        <jvm:isub />
19        <jvm:invokestatic class-type="Math"
20                          method="factorial">
21          <vm:signature>
22            <vm:return type="int" />
23            <vm:parameter type="int" />
24          </vm:signature>
25        </jvm:invokestatic>
26        <jvm:imul />
27        <jvm:label id="1" />
28        <jvm:ireturn />
29      </vm:code>
30    </vm:method>
31  </vm:class>
32 </vm:xmlvm>
```

The above XML contains the same (and no further) information than the Java class file. The reason our tool is called XMLVM is because the structure of the class file as well as the byte code instructions generated by the Java compiler are represented via appropriate XML tags. On the top-level, there are tags to represent the class definition (line 2), method definition (line 3), and the signature of the method (line 4). The children of tag <vm:code> (line 8) represent the byte code instructions generated by the Java compiler for method `factorial()`.

In the following we give a brief overview of the byte code instructions generated for method `factorial()`. Since the Java VM is a stack-based machine, operands are pushed and popped from a stack. <jvm:iload> (*integer load*) loads the integer parameter n, that was passed as an argument to method `factorial()`, and pushes it onto the stack. Instruction <jvm:ifne> (*if not equal*) pops an integer off the stack and performs a jump to another location if that integer is not equal to 0. <jvm:iconst> (*integer constant*) pushes an integer constant of a given value onto the stack. <jvm:goto> performs an unconditional jump while instruction <jvm:isub> (*integer subtraction*) pops off two integers and pushes their difference back onto the stack. <jvm:invokestatic> invokes a static method, which in this case is the recursive invocation of method `factorial()`. The parameters to a method have to be pushed prior to the invocation of the method. The Java VM also offers other types of invocations, such as the invocation for virtual methods or invocations of method defined via an interface that are not further discussed here. <jvm:imul> (*integer multiplication*) behaves similar to the aforementioned <jvm:isub> instruction, except that it pushes the product of the two integers onto the stack. The instruction <jvm:ireturn> (*integer return*) finally leaves the scope of method `factorial()`. The integer return value is popped off the stack.

Once an XML representation of a byte code program has been generated, it is possible to cross-compile the byte code instructions to arbitrary high-level languages, by simply mimicking the stack machine in the target language. The runtime stack is based on an array and push and pop operations modify that array accordingly by maintaining a stack pointer via a dedicated helper variable. When mapping byte code instructions to Objective-C, the base type of the array representing the stack is based on the following union:

```
1 // Objective-C
2 typedef union {
3     id     o;
4     int    i;
5     float  f;
6     double d;
7 } XMLVMElem;
```

The runtime stack inside a Java VM only supports object references, integers, floats, and doubles. Shorter primitive types such as bytes and characters are sign-extended to 32-bit integers. With the help of the union `XMLVMElem`, it is possible to use XSL stylesheets [7] to produce code in the target language. In the following we show how the aforementioned byte code instruction <jvm:imul> is mapped to Objective-C source code:

```
<!-- XSL template -->
<xsl:template match="jvm:imul">
  <xsl:text>
    _op1.i = _stack[--_sp].i;  // Pop operand 1
    _op2.i = _stack[--_sp].i;  // Pop operand 2
    _stack[_sp++].i = _op1.i * _op2.i; // Push product
  </xsl:text>
</xsl:template>
```

The definition of helper variables `_op1`, `_op2`, and `_stack` are based on union `XMLVMElem`. Variable `_sp` represents the stack pointer. These variables are automatically generated by other XSL templates during the code generation process. With the help of these helper variables, the effect of individual byte code instructions can easily be mapped to the target language. A pre-decrement of variable `_sp` represents a pop while a post-increment represents a push operation.

The same technique is used for generating JavaScript code. The resulting code has a substantial blowup in terms of code size because of the way we mimic the stack machine in the target language. This effect is partially mitigated for Objective-C since the resulting Objective-C source code still needs to be compiled to machine code. We have observed that in many instances the compiler (Xcode uses gcc) is able to optimize the inefficient code.

Since JavaScript is interpreted, the same kind of optimization is not possible. However, in our trials we have noticed that computational intensive operations (e.g., graphic manipulation) are usually done by special purpose hardware and not the application itself. The application usually only triggers these computational intensive operations so if the application itself is less efficient, it does not weigh as heavily.

Another point worth mentioning is memory management when targeting the iPhone. The iPhone SDK uses a version of Objective-C that does not feature a garbage collector. The implication is that the programmer is responsible for correct memory management. Objective-C uses a simple reference counting mechanism to assist the programmer with this task. The Objective-C code generated by XMLVM makes use of this reference counting mechanism. While this assures a simple and efficient memory management scheme, it is not as general purpose as a garbage collector. In particular, the code generated by XMLVM cannot properly garbage collect circular data structures. This impediment needs to be considered when writing an Android application that is to be cross-compiled with XMLVM.

C. API Mapping

The purpose of mapping an API is to make an application that uses API_1 make use of API_2 instead. In our case we need to make an Android application that uses the Android API, use the Cocoa Touch API instead. There are two basic ways to accomplish this. The first approach is to modify the application in such a way, that it uses the Cocoa Touch API instead of the Android API. E.g., if an Android application instantiates the class `android.widget.Button` (which represents a button widget), to make it instantiate class `UIButton` (the corresponding Cocoa Touch button widget) instead. This kind of transformation of the application requires a deep analysis of the way the application uses the Android API and change it to use the Cocoa Touch API instead.

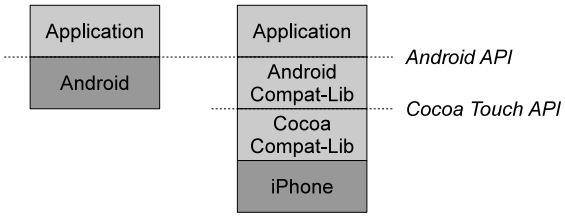

Fig. 2. Android Compatibility Library.

A second approach to achieve API mapping is via a compatibility library. Here the original Android application is unchanged, however, there is a compatibility layer that only uses Cocoa Touch API and offers the Android API. Figure 2 visualizes this approach. In some sense the compatibility layer acts as a virtualization solution, because the Android application is not aware that it is indeed running on a non-Android platform.

Both approaches have their advantages and disadvantages. Changing the Android application directly to make use of a different API is difficult. In some cases it is even impossible. If, for example, the Android application makes use of reflection to call Android API, it might not be possible in all generality to detect such usage. Changing the application is an undecidable problem. Adapting the API is therefore only feasible if constraints on the API usage are imposed. However, the benefit of this approach is that the resulting cross-compiled application runs efficiently.

The downside of a compatibility library on the other hand is the added overhead, both in terms of size and runtime efficiency. The advantage is a relatively simple way to map APIs. In XMLVM we have opted to implement a compatibility library as shown in Figure 2. Using this approach, the compatibility library can be implemented in either language: Java or Objective-C. If Java is used as the implementation language, it will also need to be cross-compiled to Objective-C. In this case we have opted to make use of XMLVM's Java API for Cocoa Touch and implement the compatibility library in Java. The following code excerpt shows a typical pattern of this compatibility library:

```
// Java
package android.widget;

public class Button extends View {
   protected UIButton button;
   // ...
   public void setOnClickListener(OnClickListener l) {
        final OnClickListener theListener = l;
        button.addTarget(new UIControlDelegate() {
            @Override
            public void raiseEvent() {
                theListener.onClick(Button.this);
            }
        }, UIControl.UIControlEventTouchUpInside);
   }
}
```

The code above shows how an Android button is mapped to a Cocoa Touch button. Class `android.widget.Button` essentially acts as a wrapper for a `UIButton` that is referenced through member `button`. Class `Button` is API-compatible with the official published Android API. The code above demonstrates how the setting of a click listener is delegated from the Android button to the `UIButton`. When the application calls `setOnClickListener()`, the anonymous class `UIControlDelegate` from the Java version of Cocoa Touch is instantiated with an overridden `raiseEvent()` method. Whenever the user pushes the iPhone button, the click event is delegated from `raiseEvent()` to the `onClick()` method that is implemented by the application.

IV. Prototype Implementation

A prototype implementation based in the ideas described in this paper exist. We make use of BCEL [1] and JDOM [2] to parse Java class files and build up the XMLVM files. Saxon [3] is used as the XSL engine to apply the stylesheets that are responsible for the code generation. The implementation of the Java to Objective-C/JavaScript cross-compilation is almost complete owing to the simplicity of Java byte code instructions and the way they are mapped to high-level languages. Our tool does not offer the same kind of completeness for the API mapping. Considering that both the Android and the Cocoa Touch API consist of thousands of methods, XMLVM currently only maps approximately 5% of the API. However, the currently supported API already allows for complex applications.

XMLVM has been used by other Open Source developers to cross-compile Android applications to the iPhone that have been published on the Android Market as well as Apple's App-Store. As a proof of concept, the authors have implemented a smartphone version of the strategy game called Xokoban. It is based on the game Sokoban where the player has to move items to designated target areas. Some simple rules confine the movement of the player which have to be taken into consideration when solving a particular level. Xokoban has been developed in Java making use of the following Android API:

- 2D animation.
- Alert views, buttons, checkboxes.
- Accelerometer and swipe interface.
- Saving/loading of preferences.

XMLVM is capable of cross-compiling these features to both the iPhone and the Palm Pre. Figure 3 shows a screenshot of the cross-compiled version of Xokoban running inside Apple's iPhone emulator. UI elements such as alert views and widgets are mapped to their native counterparts on the respective platform. A prototype implementation of XMLVM and Xokoban are available under an Open Source license at `http://xmlvm.org`.

V. Conclusion and Outlook

The popularity of smartphones makes them attractive platforms for MOGs. However, while smartphones have nearly identical capabilities with respect to their hardware, they differ substantially in their programming environment. Different programming languages and different APIs lead to significant overhead when porting applications to various smartphones. MOGs are not bound by user interface guidelines in the same way widget-driven applications are and therefore can be more easily cross-compiled. In this paper, we have demonstrated that a cross-compilation framework is feasible, thereby significantly reducing the porting effort. Only one code base needs to be maintained.

The byte code level cross-compilation introduced in this paper mimics the stack-based machine in the target language. This kind of code generation is not efficient and only suitable for applications that are not computationally intensive. We plan to introduce a stack- to register-based conversion within XMLVM to make the generated code more runtime efficient. The register-based machine will be described itself via appropriate XML-tags so that all code generating backends benefit from this optimization.

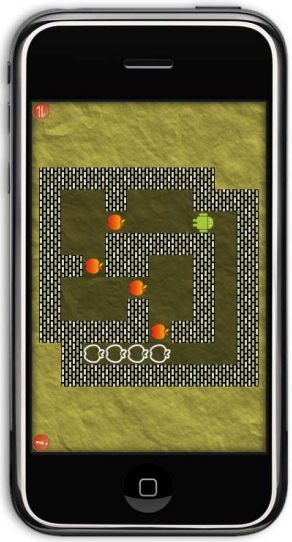

Fig. 3. Xokoban screenshot.

References

[1] Markus Dahm. Byte code engineering. Java Informations Tage, pages 267–277, 1999.

[2] JDOM. *Java DOM-API*, 2004. http://www.jdom.org/.

[3] Michael Kay. *Saxon: The XSLT and XQuery Processor*. http://saxon.sourceforge.net/.

[4] Kevin Li and Scott Counts. Exploring social interactions and attributes of casual multiplayer mobile gaming. In *4th international conference on mobile technology, applications, and systems and the 1st international symposium on Computer human interaction in mobile technology*, pages 696–703, New York, NY, USA, 2007. ACM.

[5] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Pub Co, second edition, April 1999.

[6] Arno Puder. A Cross-Language Framework for Developing AJAX Applications. In *PPPJ*, International Proceedings Series, Lisboa, Portugal, 2007. ACM.

[7] W3C. *XSL Transformations*, 1999. http://www.w3.org/TR/xslt.