TouchDevelop: Programming Cloud-Connected Mobile Devices via Touchscreen

Nikolai Tillmann, Michał Moskal,
Jonathan de Halleux, Manuel Fähndrich

Microsoft Research
One Microsoft Way, Redmond WA 98052, USA
{nikolait,micmo,jhalleux,maf}@microsoft.com

August 17, 2011

Abstract

The world is experiencing a technology shift. In 2011, more touchscreen-based mobile devices like smartphones and tablets will be sold than desktops, laptops, and netbooks combined. In fact, in many cases incredibly powerful and easy-to-use smart phones are going to be the first and, in less developed countries, possibly the only computing devices which virtually all people will own, and carry with them at all times. Furthermore, mobile devices do not only have touchscreens, but they are also equipped with a multitude of sensors, such as location information and acceleration, and they are always connected to the cloud.

TouchDevelop is a novel application creation environment for anyone to script their smartphones anywhere – you do not need a separate PC. TouchDevelop allows you to develop mobile device applications that can access your data, your media, your sensors and allows using cloud services including storage, computing, and social networks. TouchDevelop targets students, and hobbyists, not necessarily the professional developer. Typical TouchDevelop applications are written for fun, or for personalizing the phone.

TouchDevelop's typed, structured programming language is built around the idea of only using a touchscreen as the input device to author code. It has built-in primitives which make it easy to access the rich sensor data available on a mobile device. In our vision, the state of the program is automatically distributed between mobile clients and the cloud, with automatic synchronization of data and execution between clients and cloud, liberating the programmer from worrying (or even having to know about) the details. We report on our experience with our first prototype implementation for the Windows Phone 7 platform, which already realizes a large portion of our vision. It is available on the Windows Phone Marketplace.

1 Introduction

The way in which we interact with computing devices is changing:

- most devices have some form of internet connectivity, and are almost always connected to the cloud via cellular networks, with short interruptions,
- instead of using keyboards and mice as the point input devices, advanced touchscreens are becoming more common,
- many people now carry a smartphone or some other small mobile device with them at all times,
- mobile devices are often equipped with sensors that give auxiliary information about the environment such as orientation, acceleration, and location,
- even small mobile devices such as smartphones are now typically equipped with comparatively powerful batteries, graphic processors, high-resolution screens, and main processors, which rival small ordinary laptops or even PCs which were state-of-the-art just a few years ago.

Applications for such devices are typically written in the traditional way: Using traditional programming languages, traditional compilers, traditional development environments augmented with extra libraries to access the touchscreen, sensors, network, etc., and using a traditional PC with a mouse, keyboard, and a big monitor.

What has not yet happened is the shift to use the new mobile devices themselves to write applications for those devices. This might not seem desirable for professional developers who spend the majority of their time writing code and can afford a traditional PC development set up. On the other hand, automating little tasks, performing simple computations, or retrieving personal data stored on a mobile device with a query expression, with minimal set up and possibly on-the-go, offers practical value and can also serve as an excellent entry point into computer science education for students as well as adults.

We propose a new programming environment and language which makes it possible to write applications on mobile devices: TouchDevelop. The aim of TouchDevelop is to bring the excitement of the first programmable personal computers to the now ubiquitous mobile devices. TouchDevelop engages users by allowing them to write scripts that show and manipulate things that connect with their personal interests and possessions, such as the music and pictures stored on their own mobile device, the sensors of the phone, and the friends in their social networks.

TouchDevelop comes with a typed, structured programming language that is built around the idea of only using a touchscreen as the input device to author code. TouchDevelop has built-in primitives which make it easy to access the rich sensor data available on a mobile device. In our vision, the state of the program is automatically distributed between mobile clients and the cloud,

with automatic synchronization of data and execution between clients and cloud, liberating the programmer from worrying (or even having to know about) the details.

The contributions of this paper are as follows:

- We motivate the idea of programming on a mobile device itself.
- We describe the TouchDevelop language.
- We give an overview over the TouchDevelop environment: the program management, the code editor, the runtime environment.
- We report on our experience with the first prototype implementation of TouchDevelop, which already realizes a large portion of our vision, and illustrate its practically in a case study with an example.

TouchDevelop is available on the Windows Phone Marketplace; the TouchDevelop website[1] contains the latest information, documentation, and instructional videos to get started. Within a week of being launched, TouchDevelop became one of the 100 most popular (most downloaded) applications on the Windows Phone Marketplace (out of over 13,000 available apps on 4/14/2011), and the 3rd most popular app in the general "productivity" category. This ranking indicates that TouchDevelop serves a purpose — a large percentage of all Windows Phone users want the ability to write and run scripts on their phone.

Our currently available prototype does not yet implement all features which we envision for TouchDevelop. We will indicate in the text when an aspect will only be realized in the future, and we give a summary of all current limitations in Section 5.

2 Principles, Design Decisions, Target Audience

All design decisions of the TouchDevelop programming environment are motivated by one main principle: the programs are created on a mobile device, and they are consumed on a mobile device. By a *mobile device* we understand a pocket-sized computing device with a touchscreen; a typical laptop does not fit this description. This principle has large implications on the design of the language, the program editor, and the execution environment.

Main principle, part 1: All programming and relevant content authoring must be done *on a mobile device*. We assume that a separate laptop or PC is not physically accessible to the user. While exploring hybrid mobile-PC programming environments is certainly interesting, we have decided to focus on exploring how far the mobile-only experience can be pushed, at least for now. Notice that in general non-essential input devices tend to die; witness for example the PDA stylus and tablet PC pen, and the declining popularity of slide-out keyboards for phones. If it is possible and effective to only use a

[1] http://research.microsoft.com/TouchDevelop

mobile device to program certain applications, then other, more complicated approaches are likely to be eventually replaced by the single-device approach for those tasks.

Main principle, part 2: The resulting program should run on a mobile device, leveraging the computing capabilities, sensors and cloud connection. The programming and execution devices can be the same. The programming language should be Turing complete.

However, we do not require that every possible program that could potentially run on a mobile device can indeed be created with TouchDevelop — besides legal and licensing restrictions which might make this impossible, we expect that once the program reaches a certain size, complexity, or has high performance requirements, a program creator will likely devote more resources to the project and would shift his or her efforts over to a bigger development environment form factor, possibly a traditional PC.

In other words, our target audience is everyone who might traditionally have been able to write a BASIC program on a regular keyboard and ordinary PC. This includes students and hobbyist programmers.

Moreover, we were aiming for a language and runtime system where features are not only fast to type-in once the user knows about them, but where features are also easily discoverable, in order to help novice programmers navigate the available functionality without training.

3 TouchDevelop Language

TouchDevelop uses a simple, imperative, statically typed programming language. We decided against using an existing, well-known programming language for three reasons. First, languages with similar scope tend to be dynamically typed which greatly complicates autocompletion features (see Sections 3.1 and 4.2). Second, the lack of a precise explicit keyboard with around one hundred separate keys dramatically changes the trade-offs to be made when designing the concrete syntax. Note that some recent design decisions in the evolution of C# (and VisualBasic) were made only in order to enable autocompletion in the code editor; for example, while the established SQL notation for a query usually begins with "SELECT column FROM table", in the SQL-inspired LINQ notation of C# (and similarly VisualBasic) the order is surprisingly reversed: "from x **in** table select x.column"; as a result, after typing the code to assign to x elements of the table, the editor can give autocompletion support with the inferred table type while typing the later selection expression x.column; this would not be possible in the traditional SQL order. In a similar spirit, we will base design decisions in TouchDevelop on the ability to accurately select relevant program elements with an imprecise finger tapping gesture on a touchscreen. Third, we wanted to approach the question how program state should be managed by the runtime system with a fresh mind which addresses the challenges of the mobile execution environment of a cloud-connected device.

Even though accommodating these three properties greatly influenced cer-

string ::= "Hello world!\n" | ...
number ::= 0 | 1 | 3.1415 | ...
local ::= x | y | z | foo bar | ...
action ::= frobnicate | go draw line | ...
global ::= phone | math | maps | ok button | chat roster | ...
property ::= vibrate | sin | add pushpin | disable | add | ...
exprs ::= *expr* | *exprs* , *expr*
parameters ::= | (*exprs*)
op ::= + | − | / | ∗ | ≤ | ≥ | > | < | = | ≠ | **and** | **or** | ||
expr ::= *local* | *string* | *number* | **true** | **false**
| − *expr* | **not** *expr* | *expr op expr*
| *action parameters* | *expr* → *property parameters*
block ::= *stmt* | *block stmt*
locals ::= *local* | *local* , *locals*
stmt ::= *expr* | *locals* := *expr* | **do nothing**
| **if** *expr* **then** *block* **else** *block*
| **while** *expr* **do** *block* | **foreach** *local* **in** *expr* **do** *block*
| **for** 0 ≤ *local* < *expr* **do** *block*
type ::= **Number** | **String** | **Song** | **Nothing** | **Phone** | ...
formals ::= *local* : *type* | *local* : *type* , *formals*
formals-opt ::= | *formals*
returns-opt ::= | returns *formals*
action-def ::= **action** *action* (*formals-opt*) *returns-opt block*

Figure 1: Abstract syntax of the TouchDevelop actions.

tain details of the language design, in fact the abstract syntax, typing rules, and the basic semantics of the language are fairly similar to corresponding fragments of mainstream programming languages like Java or C#. This lets the user transfer expertise between TouchDevelop and a regular programming environment, in both directions. Moreover, it allowed us to focus on the details specific to touchscreen input and the mobile execution environment, instead of the high-level programming language design.

The TouchDevelop language is a mix of imperative, object-oriented, and functional features. The imperative parts are most visible: users can update local variables, and state of global objects. Object-orientation is dictated by autocompletion requirements—properties of objects are an easily accessible and intuitive concept. However, for the sake of simplicity, the language itself does not have built-in constructs which would allow users to define new types or properties. Functional features come in form of a query language akin to LINQ (Section 3.4) which we plan to incorporate in the future.

A TouchDevelop script consists of a number of actions (procedures) and the global state (global variables and user interface (UI) elements). The UI elements have various properties (e.g., position, color, content), and define bindings of particular events (e.g., button clicked, text changed) to the actions. Global variables define their type, sharing scope, as well as their current value. Global state is edited via various property dialogs and persisted either on the phone or in the cloud. We discuss the storage of shared global state in the cloud later in Section 3.3.

Figure 1 describes the abstract syntax of TouchDevelop actions. We freely use spaces in identifiers—this is justified by our expression input model, which treats identifiers as atoms. The body of an action is a block of statements, which in turn contains nested statements and expressions. Expressions include the usual assortment of literals, arithmetic and Boolean operators, as well as the string concatenation operator (||). Actions can define local variables (the first assignment is the definition point), call other actions, as well as call properties of objects. The TouchDevelop runtime system defines a number of data types (e.g., **String** or **Song**; **Nothing** is the equivalent of the C void type), each with a number of properties (e.g., $\rightarrow$ index of or $\rightarrow$ artist). Additionally, we define singleton objects (e.g., phone of type **Phone**), also with properties (e.g., phone $\rightarrow$ vibrate or math $\rightarrow$ sin). Types of singletons are not exposed to the user (i.e., one cannot make an action returning a **Phone**).

Actions can have multiple return values. We support a multiple-assignment statement *locals* := *expr*; while most assignment statements are likely to assign only a single expression to a single local, the multiple-assignment statement can and may only be used with an expression that represents a call to an action with multiple return values of the same arity. The motivation to allow multiple return values for an action is to support a refactoring that extracts multiple statements into a separate new action. The type representing multiple return values is not exposed to the user.

3.1 Type System

We decided on a statically typed language, since type information is necessary to provide highly useful context-sensitive code completion options to the user. Except for parameter and return types of actions and global variable types, the user does not have to provide type annotations, as types are usually inferred. The type system is fairly standard.

Every expression has at most one type. Every built-in property, singleton, and operator has a fixed type defined by the TouchDevelop runtime system. User-defined actions have their return type explicitly defined. Locals are introduced by assignments, and thus their type is inferred from the type of initialization expression. They remain in scope until the end of the block in which they are defined.

Arguments in calls (to actions, properties, operators, assignment etc.) are expected to have the declared type, however every type can be implicitly converted to **String** or **Nothing**.

3.2 Execution Model

The semantics of the language is a standard eager call-by-value semantics.

The execution is reactive—actions are run in response to events. Events can be raised by user input (i.e., manipulating a UI element), passage of time, or updates of the data stored in the cloud.

TouchDevelop uses cooperative multi-threading. Normally, actions are executed single-threaded, however calls that block (e.g., fetching data from the web or waiting for user input) allow other actions to run. The blocking calls are marked visually in the TouchDevelop editor, to make the user aware that global data can change at this point. Moreover, there are some properties the user might want to invoke which require the application to suspend, and later resume, e.g. when the user wants to take a picture via the camera application. These are also marked visually. See Section 4.4 for a longer discussion of this concept, called *tombstoning* in the Windows Phone platform.

3.3 Shared State in the Cloud

In the future, TouchDevelop will support shared state for scripts, which can be thought of as special global variables that live in the cloud, shared by multiple application instances. An application can run on multiple devices in a common application domain; in this case, all application instances operate on the same shared global variable, and they can observe each other's state changes. The distributed state evolution is realized via Concurrent Revisions [1]: a mobile device forks from the shared state when its connection to the cloud is interrupted, and its updates are merged back into the shared state when the connection is restored. To save battery power, these events can be further postponed and bundled.

```
expr ::= ... | current
type ::= ... | [type]
stmt ::= ... | local := expr apply transformers
transformers ::= transformer | transformers transformer
transformer ::= distinct | reverse
  | top expr | bottom expr
  | where expr | order by expr | transform to expr

top_artists := media → songs apply
  where current → duration > 30
  order by current → rating
  bottom 20
  transform to current → artist
  distinct
  top 5
```

Figure 2: The proposed TouchDevelop query language and an example.

According to the semantics of Concurrent Revisions, merging of data is performed by a resolution strategy for competing updates from different devices; the basic resolution strategy is "the last update wins", but this can be refined by further annotations from the user, e.g. the user can indicate that an integer shared global variable is a "counter", in which case changes are additive. In this way, possible merge strategies are limited, but merging never fails, and conflicts are resolved automatically, maintaining a simple programming model as far as the programmer is concerned.

An application domain in the cloud is associated with its creator; if the creator permits, then direct friends of the creator in a social network are allowed to attach to this application domain.

3.4 Query Language

Figure 2 defines syntax for queries which we plan to implement. We introduce a sequence type, a special expression to refer to the current element, and an **apply** statement, which takes a sequence and a number of transformers. The editor treats the transformers in the query like statements, using a similar user interface.

Operations like average, sum, median, minimum, maximum, count, all, and any are available as properties on the sequence type. We might allow their usage as the last transformer.

3.5 Intentional Limitations

The TouchDevelop language is intentionally limited, in order to be easy to understand by non-expert programmers.

No user-defined types As mentioned before, we do not let the user define new types or write new properties. There is also no other encapsulation mechanism beyond the concept of a script. However, while there is no language-intrinsic mechanism to define custom types, externally realized and possibly user-configurable data sources and state can extend the universe of available types.

Restricted libraries TouchDevelop provides a set of built-in primitives and types to interact with the phone sensors, screen, web, etc. While all relevant aspects of the mobile device are exposed in this way, we often expose the underlying Windows Phone APIs only in a simplified way to TouchDevelop scripts, preferring easy-of-use over expressivity, efficiency, and performance. In the future, additional libraries could be added via extension mechanisms.

No error handling For simplicity, when a rare unrecoverable error occurs, e.g. a web service cannot be contacted, the script simply stops. This is typical for small applications on mobile devices. Also, structured error handling would add another layer of complexity to the language which might be too difficult to master for many of our target user group.

Some kinds of "errors" are in fact rather common. For example, a call to the built-in media → choose picture property may fail when the user cancels the request to select a picture from the phone's picture library. To deal with such situations, every data type has a special invalid value, which indicates the absense of a proper value. The property is invalid allows the user to test for the presence of this special value. As a convention, all built-in properties return invalid when invoked with an invalid argument.

4 TouchDevelop Environment

Together with the language itself, the programming environment has been designed around the idea of working on a mobile device with a touchscreen, typically with a 4" diagonal on a smartphone.

4.1 Multiple-screen User Interface

Windows Phone apps use the metaphor of the browser going through the pages of the app. The user taps on UI elements to navigate to new pages, and uses the hardware back button to return to the previous page (usually saving whatever state was modified, which is slightly different from the basic browser paradigm). In this way, the TouchDevelop editor is used to "browse" through scripts stored on the device. The first page shows a list of scripts currently available on the mobile device. Each script has its own page with a list of actions and global variables (in the future, the script's UI elements and events will be also available at this level). Selecting an action brings up the action editor page, and selecting a statement there shows the expression editor. Finally, selecting certain literals

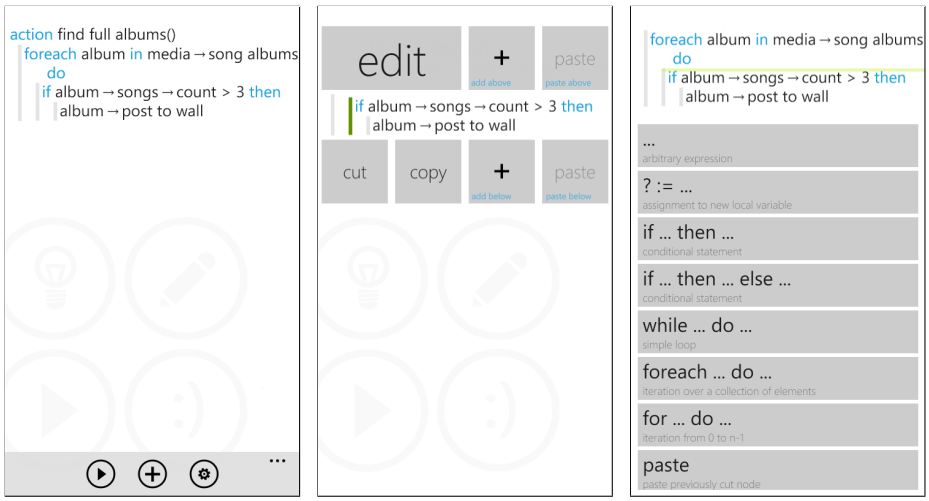

Figure 3: TouchDevelop action editor. (left) Main action view; note that return type is hidden, as it is **Nothing**. (middle) Statement editing options. (right) Inserting a new statement above the existing **if** statement.

(e.g., string or date-time literals) in the expression pops up specialized literal editors.

4.2 Editing Actions

TouchDevelop offers a structured editor for statements and autocompletion-enabled token-level editor for expressions. Full statements can be moved around by cut/copy and paste, and new statements can be inserted and edited. Limiting these operations to full statements was done in interest of expediting and simplifying the editing process. As a side effect, it is impossible to construct a program which is syntactically invalid at the statement level. On the other hand, expressions are entered as strings of tokens and it's perfectly possible to have syntax or type errors there. We hope that confining syntax errors at the relatively small expression level makes it reasonably easy for our users to correct syntax errors.

The main action view shows the source code of a single action. The font is easily readable, but not big enough for precise selection (left screen shot in Figure 3). The user can tap anywhere on the action to select the statement to operate on. If the selection was not exactly what the user intended, it can be adjusted by sliding the text up or down. The selection brings up a set of buttons with available operations (middle screen shot): editing the expression in current statement, copy and paste operations, as well as inserting a new statement above or below. Tapping the insertion button brings up a list of available statements (right screen shot).

Each statement directly contains at most one expression. For **if** and **while**

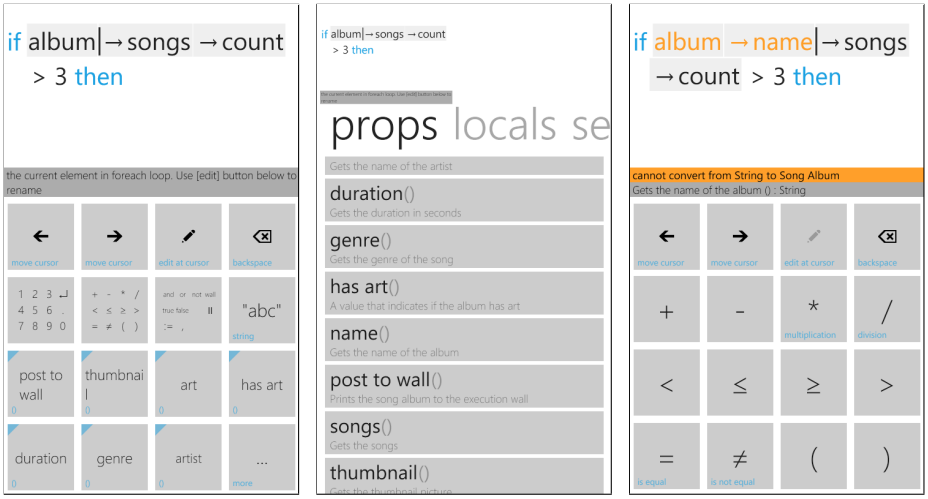

Figure 4: TouchDevelop expression editor (calculator). (left) Editing condition in if-then statement; help shown for the expression before cursor. (middle) The "..." button shows more properties of Album with descriptions. (right) Expression with error in orange, error message for expression before cursor; also virtual keypad in operator mode.

statements it is the condition, for **for** the upper bound (the bound variable is inserted automatically), etc. Inserting a statement or tapping the "edit" button on it brings up the calculator view (Figure 4). The calculator provides a virtual on-screen keypad. It has multiple keypad modes—the left screen shot shows the main keypad with autocompletion suggestions and options of switching to other keypads; the right screen shot has the operator keypad. The on-screen keys are big enough to be tapped without mistakes. Because there is a keypad, there is also the cursor. It can be moved by tapping on the expression screen at the top, or when precision is needed, using on-screen cursor keys.

Editing expressions happens at the level of tokens: the user inserts and deletes tokens, some of which can be parentheses or operators. Operators and literals are available in one of the keypads. Recently used properties, global and local variables, as well as actions are shown on the main keypad. Tapping the "..." button opens up the full list, divided into categories (middle screen shot). Editing string literals and renaming local variables are the only places where the regular virtual on-screen (or possibly hardware slide-out, if available on a particular mobile device) keypad is used. In particular, the keypad is not used for entering property names.[2]

The expression is parsed and type-checked after each tap on a key. The type information is used to provide error messages, autocompletion suggestions, and

[2]We can use the keypad to filter the full property list if so desired. Still it will be impossible to make a typo in an identifier.

context-sensitive help. When expression editing is complete, the hardware back key is used to return to the action editor.

The action editor also offers the possibility (Figure 3 left, at the bottom) of executing the currently edited action, adding a new statement at the end of the action (to save one tap), and editing action properties (e.g., the name, formal parameters and return type; in the future we plan to have event bindings here) using a dedicated dialog.

4.3 Discussion of the Editor

We decided against a Tactile Programming paradigm [9] and a Visual programming language [8], as we wanted to enable the authoring of BASIC-like programs. Instead, our language is similar to a traditional text-based programming language, but with a structural editor, where the extent of enforced structure was chosen to make each editing step possible with the press of a finger on a touchscreen. Such a tapping gesture is not as precise as a mouse-click, but instead needs dozens of spare pixels in each direction to buffer against errors due to the imprecision of the gesture recognition.

At the statement level, the language editor is structured. At the expression level, the editor is unstructured. However, at the token level, the editor is structured again. In other words: You cannot make syntax errors at the statement level, but you can at the expression level; however, individual identifier names are treated as atoms, and they cannot be misspelled. This set of decisions is a usability compromise, largely driven by the overall design of our code editor.

The distinction between the statement-level structured editing and unstructured (yet misspelling-free) expression editing comes from observation of the usage of modern IDEs like Eclipse or Visual Studio. Users seem to be happy to close braces immediately after opening them, and only later fill them in, because otherwise autocompletion gets confused. On the other hand, people rarely write particular expressions that way. Finally, autocompletion tends to correct (or preserve) all misspellings. In our experience, moving this model to the mobile device seems to work well.

We currently use the tap-to-select model with the option to cancel the selection if an unintended statement gets selected. Also, the user can scroll and expand the selection after the initial selection.

We have experimented with a few other models and decided against them:

- Pinch to zoom the view and then tap to select. This is the standard model in other phone applications. However, we found that continuous zooming and changing of the viewing area is confusing.

- Dragging a button like “edit” or “add statement” to the place where it should be executed. While dragging, the “drop place” is highlighted and updated depending on the current position of the finger until it is released, thus allowing high precision. However, the entire operation takes a long time and requires long-distance finger movements.

- Tapping and holding for 500ms to bring up a cursor for selecting the statement above the finger. This cursor can be then dragged and the selected statement is highlighted. Then the menu with options appears. This mode resembles how individual characters are selected in text boxes on the Windows Phone. However, for frequest use, we found that the initial hold period (500ms) was too long and impossible to avoid, because the action view needs to be scrollable which also happens with the drag gesture. In a variation of this approach, the view was zoomed when the selection cursor was active.

- Two-tap-selection—the first tap zooms the view, the second tap selects. This allows for very precise selection without the need to frequently cancel an unintended selection, but it adds an additional tap.

These models were unsatisfactory, mostly because the involved gestures are slow to execute (multiple taps, tap-and-hold), or they change the viewing area too often and drastically (pinch), or they involve far finger movements (drag). Selecting statements is a core part of TouchDevelop experience and therefore needs to be very fast and precise.

The calculator features on-screen keys much bigger than the usual on-screen keyboard used for text input. They are big enough to be tapped very quickly without mistakes. Additionally, the operator keypads go away after a tap, while the number keypad does not, since usually only a single operator is needed at a particular position, but numbers consist of several digits. Finally, the most often used identifiers valid in the current context are immediately available on the main keypad, greatly speeding up expression entry. As an example, entering the program from Figure 3 takes a minimum of 20 taps (including all navigation), which is about 35 seconds for one of the authors, whereas typing it using a regular laptop keyboard takes about 25 seconds. The first line takes 5 taps, second 11, and the last one 4. It would be significantly longer if the user did not like the default name for the local (songalbum) and decided to rename it; renaming involves typing the text for a new identifier name on a regular on-screen keyboard, which is independent of TouchDevelop. Thus, it seems the speed of expression entry can be still improved (particularly, some of the view transitions in our current implementation are a bit slow), but given a bit of training a performance can be achieved that is in the same order of magnitude as regular keyboard program entry. This speed indicates that the transition of program development from a traditional keyboard/mouse set up to a touchscreen is feasible, especially given the comfort of a mobile device.

4.4 Interpreter

At the core of TouchDevelop's runtime system sits an interpreter which operates on a byte-code representation directly derived from the abstract syntax tree of the script code. An important feature of the interpreter is that its state (and thus by extension the state of the user's program) is serializable in between any

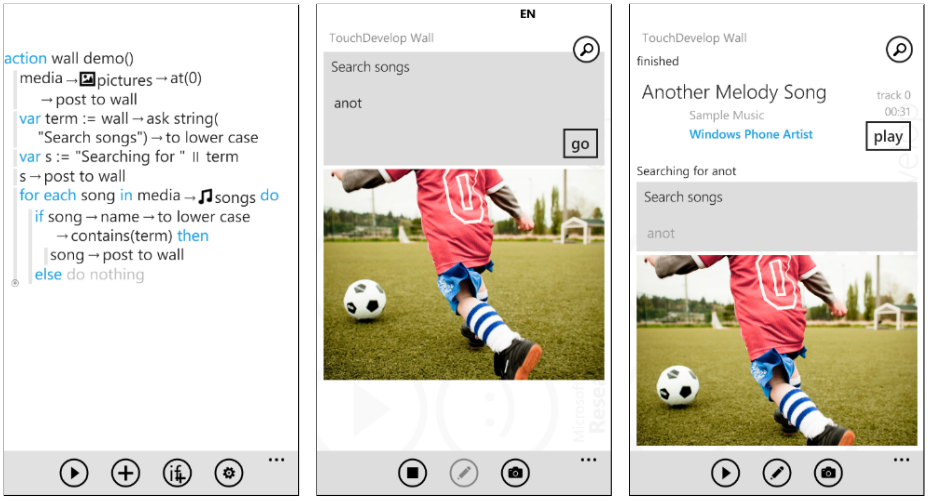

Figure 5: Capabilities of wall. (left) Action that pops up input box and posts a few things to the wall: a string, a song, and a picture. (middle) Input box. (right) The wall after executing the action.

two byte-code instructions, and during those built-in instructions which require *tombstoning*.

Tombstoning on the Windows Phone platform refers to a concept where an application has to persist its program state, die, be launched again later, and restore the program state to appear as if the application was running the entire time. This concept can be triggered by an external event, e.g. on an incoming phone call, or when the user pressed the off-button of the phone. When the external event is over, e.g. when the phone call is finished, or when the user turns the phone on again, then the application must resurrect itself. The intention of this concept is to save memory and processing power, and focus on single activities. This concept can also be triggered by an internal event, e.g. when the current application wants to take a picture using the phone's camera.

By making its state serializable, the TouchDevelop interpreter can continue execution seamlessly after tombstoning. In our experience, properly handling tombstoning is a tedious and error-prone duty when developing Windows Phone applications in the traditional way. Within the capabilities of TouchDevelop scripts, all of this is handled automatically by the interpreter.

4.5 TouchDevelop Wall

Just as a traditional text-based application usually has a console output stream, TouchDevelop has the concept of the *wall* to which any value can be posted during script execution. Some values are stateful, i.e. they have reference semantics, e.g. pictures, which can thus effectively function as an updatable canvas. Similarly, when a sound or song is posted, it can be played via an embedded button

(Figure 5). Similarly, posted videos can be played, maps can be zoomed, links followed, etc. The wall offers a very simple, yet rich, way of interacting with the script and the content it manipulates.

5 Prototype Implementation Limitations

Our currently available prototype does not yet support the following features:

- Attaching of actions to events; in the initial prototype, the only way to run an action is via an explicit "run" button.
- Persistent UI elements
- Synchronization of shared global script state with the cloud
- Functional query statements

We plan to implement these features in future releases.

Even with these limitations, interesting applications can be written, e.g. using an explicit event-handler loop in the code; see Section 6.2 for an example. In fact, the users of our prototype seem to be quite happy with the current programming model; the most frequent feature request after our prototype release was not related to the language or the editor, but instead the ability to "pin" a custom script to the phone's start screen, in order to appear as a regular application that can be launched with a single tap. That feature will be available with the Mango release of Windows Phone.

6 Case Studies

6.1 Samples

The TouchDevelop app comes with various sample scripts; each main action of a script is up to two "pages" long, or roughly 40 lines. The following list of samples illustrates the diverse capabilities of our initial prototype implementation of TouchDevelop.

- hello world / go — print text to the wall
- classical algorithms / fac 5 — recursive calls to helper routines with parameters and result, computing factorial of 5
- classical algorithms / fib 5 — recursive calls to helper routines with parameters and result, computing the fifth Fibonacci number
- goodies / ask duckduckgo — query a search web service
- goodies / beatbox — record multiple sounds from user via microphone, play in loop triggered by button taps, realized with an explicit event loop

- goodies / search songs — search song library on phone based on song name
- goodies / themify picture — user selects picture, graphical effect is applied to picture pixel by pixel via two nested loops
- goodies / vibrato morse — user enters a text, which is then turned into a sequence of phone vibrations in morse code
- simple graphics / go draw random dots — draw random dots on a canvas
- simple graphics / go draw line — draws a line on a canvas, pixel by pixel
- simple graphics / go draw random circles — draws circles on a canvas, pixel by pixel
- simple graphics / go draw spiral — draws a spiral on a canvas, pixel by pixel
- senses samples / go record microphone — records a sound from microphone
- senses samples / go snap camera — takes new picture via phone camera
- web samples / links — post links to the web to the wall
- media samples / find short songs — finds all songs shorter than 180s

6.2 Writing a Script

In this section, we will illustrate how one can build a new script from scratch. We will implement a well-known game, where two paddles are visible on the screen. Here, one is controlled by the user, and one is controlled by the computer. A ball is bouncing between the two paddles. The user can tilt the phone to control the location of the paddle on the screen, and the user gets feedback via the phone vibration when the ball is missed. To detect tilting of the physical phone device, this game uses the built-in accelerometer sensor of the phone.

We start by adding a new action with the (+) button at the script level; this brings us into the action editor. Then we add statement after statement with the (+) button at the action level. Now we get presented a list of different statement kinds to choose from, similar to the right screen in Figure 3. For example, to add the second statement (at line 3), we tap the assignment-statement kind, which will implicitly introduce a new local variable pic to hold the result. In the expression editor, we can easily add the right-hand side of the assignment by tapping on media in the quick-selection list that shows available services in the current context.

Now we see the middle screen of Figure 6 in the expression editor. Then we tap on create picture in the quick-selection list which has now adjusted to show everything we can do with media. By default, the picture size (480, 800) is entered (the full size of the standardized Windows Phone 7 screen). We choose to reduce the resolution to (480, 648) so it is fully visible on the screen given the TouchDevelop header and buttons. Editing the size is easily done with a

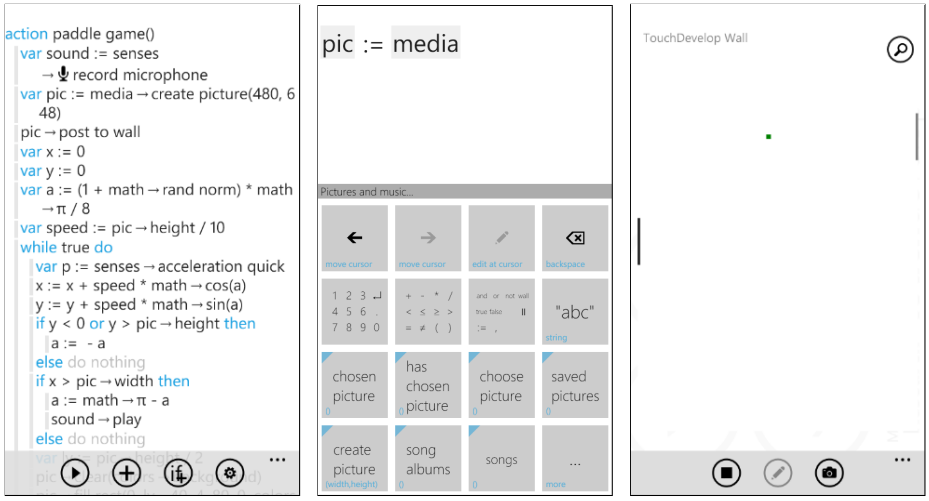

Figure 6: Paddle game example. (left) First part of game code in action editor (middle) Editing assignment in expression editor (right) Game while executing

calculator-like user interface to edit numbers. Overall, from the action editor, just four taps were needed to create a new picture, and a few more taps in the calculator to edit the desired picture size. A tap on the hardware back button brings us back to the action editor. Next, we want to show this new picture on the wall. For that, we again press the (+) button to add another action, then tap to select an arbitrary-expression statement kind. In the expression editor, we tap on pic in the quick-selection list, then we tap on the ... button to get more options, and tap on post to wall from the list of available properties. A tap on the back button brings us back to the action editor.

All other statements are added in a similar fashion, usually just taking a few taps for each, where each tap targets a comparatively large surface area which is easy to hit properly with a big finger. No tedious navigation to individual characters is necessary. In fact, the most difficult part of programming with TouchDevelop is usually entering of raw text either to enter a string literal, e.g. to show a message to the user, or to rename an identifier, if the developer chooses to do so. Entering text is done with the standard Windows Phone keyboard control, which works well, but still requires pressing of individual character buttons which are smaller than all other TouchDevelop button surface areas.

A screenshot of the beginning of the paddle game code in the action editor is shown on the left screen in Figure 6.

Listing 1: Paddle game

```
1 action paddle game()
2   sound := senses → record microphone
3   pic := media → create picture(480, 648)
4   pic → post to wall
```

```
5   x := 0
6   y := 0
7   a := (1 + math → rand norm) * math → pi / 8
8   speed := pic → height / 10
9   while true do
10    p := senses → acceleration quick
11    x := x + speed * math → cos(a)
12    y := y + speed * math → sin(a)
13    if y < 0 or y > pic → height then
14      a := −a
15    if x > pic → width then
16      a := math → pi − a
17      sound → play
18    ly := pic → height * (1 + p → y)
19    pic → clear(colors → background)
20    pic → fill rect(0, ly − 40, 4, 80, 0, colors → foreground)
21    pic → fill rect(pic → width − 4, y − 40, 4, 80, 0, colors → gray)
22    pic → fill rect(x − 4, y − 4, 8, 8, 0, colors → green)
23    if x < 0 then
24      a := math → pi − a + (0.5 − math → rand norm) * math → pi / 8
25      x := 0
26      sound → play
27      if y < ly − 40 or y > ly + 40 then
28        phone → vibrate(0.4)
29    pic → update on wall
30    phone → sleep(0.1)
```

Listing 1 shows the full game source code. Line 2 records a sound that will be played later when the ball bounces on the left or the right against the wall. As discussed, line 3, 4 create and post a canvas called pic. Line 5, 6 declare the coordinates of the moving ball, initially x=0 and y=0, starting at the upper left corner. Line 7, 8 declares the angle variable a which controls the direction of the movement of the ball, initially a random value in a range that ensures that the ball moves towards the right wall. Line 8 introduces a helper variable that governs how fast the ball flies. In line 9, a **while** ... **do** loop starts. The body of the loop is visually nested. Within the loop, line 10 queries the accelerometer for the current acceleration vector – a three dimensional value. Without any further forced-induced acceleration, this vector always includes the gravitational force, and thus, it can be used to determine the orientation of the otherwise steady device. In particular, the y coordinate of the vector is used later in line 18 to compute the current vertical position of the user's (left) paddle, based on whether the device is standing upright (p → y=−1) or in a horizontal position (p → y=0). Line 13, 14 let the ball bounce when it hits the upper or lower wall, by adjusting its direction angle a. Note that the body of the **if** statement is automatically indented, just like the loop body. Line 15-17 let the ball bounce when it hits the right wall, adjusting the direction angle and playing the user-provided sound. Line 18-22 update the picture by basically redrawing it from scratch, drawing filled rectangles for the left paddle, the green ball, and the right

paddle. Line 23-26 let the ball bounce when it hits the left wall, adjusting the direction angle, playing the user-provided sound, and vibrating when the ball does not hit the user's (left) paddle depending on its vertical position. Before ending the loop, on line 29, we use pic → update on wall to make the changes made to the picture visible on the screen. Finally, at the end of the loop in line 30, the game briefly sleeps to provide a steady game speed.

The right screen in Figure 6 shows the game while executing: The canvas pic, which changes over time, is visible on the wall. On the left, the user's paddle is visible; in the middle is the flying ball, and on the right is the computer's paddle. The user can stop the execution of the script at any time.

The initial development of this game by the first author stretched over a time span of approximately two hours, incrementally adding more game aspects (left paddle, flying ball, right paddle, vibration, sound) and testing them. Since the program is small in size, errors were easy to spot and fix. In fact, the biggest intellectual effort in writing this game was to figure out how to realize the physically-correct ball bouncing against the different walls. A novice TouchDevelop programmer will most likely spend much more time than two hours for this example, but we expect that much of this additional time will be spent familiarizing with the programming environment, which will be amortized when developing further applications.

This game is quite simple, and is meant to illustrate how easy it is to create conceptually engaging applications with TouchDevelop, using only a touchscreen to enter the program, and leveraging the sensors of the phone. Much larger and more refined applications can be created with TouchDevelop.

7 Related Work

TouchDevelop aims at making programming easier and accessible for non-expert programmers, a topic that has been studied widely for decades; an extensive overview can be found in [5]. Some notable graphical programming environments are Scratch [7], Alice [2], and Greenfoot [6]. Alice and Greenfoot target older students than Scratch. Alice and Scratch are visual programming languages which shield the user from syntax errors, while Greenfoot uses Java as its programming language. All of them neither specially target mobile devices, nor are designed for the touchscreen. In fact, to our knowledge, TouchDevelop is the first general-purpose programming environment that specifically targets touchscreens as the main input device.

Closely related to TouchDevelop's approach are structured code editors, e.g. the early Cornell Program Synthesizer [10]. Structured editing has been used previously mainly as a way to prevent certain syntax errors. TouchDevelop's editor is not strictly structured, as it allows arbitrary token-strings at the expression-level, but each token is treated as an atom again. All of the structured editing in TouchDevelop is motivated by the ability to easily edit the code on a touchscreen, where elements are typically selected with a big thumb, and not for the sole purpose of preventing syntax errors. The GNOME [4] envi-

ronment, for example, even prevented the programmer from moving the cursor while a syntax error was present.

The TouchDevelop language could be considered a Visual programming language [8], since it is edited graphically, but the interaction with the graphical elements is quite limited; the TouchDevelop language still very much resembles a traditional, text-based programming language, with a specialized editor and automated reformatting and annotation of program text.

Although based on touchscreen-focused program editing, TouchDevelop is also different from Tactile programming [9] in that TouchDevelop does not make it easy to cross the boundaries between the application world and the programming world; both worlds are still strictly separated, as in a traditional programming model.

Dasher[11] is another new user interface that aims at efficient input of character-based text based on language modeling and driven by continuous two-dimensional gestures which could come from a touchscreen. In TouchDevelop, we do not face so much the problem of continuous text input, but rather have to deal with highly structured programs.

Another visual programming platform that targets novice programmers is App Inventor for Android[3], which enables the creation of applications for the Android platform. However, for program development, App Inventor requires a regular (big and powerful) PC with a full Java runtime environment; it does not run on Android itself.

Programming on tablet PCs with a stylus has been studied before [3], using ordinary programming languages, compilers, and development environments, using a pen. While the mobility of the device was appreciated, the students felt that this arrangement was ineffective for programming tasks.

8 Conclusion

Incredibly powerful and easy-to-use smartphones are going to be the first (and possibly only) computing devices which virtually all people will own, and carry with them at all times. TouchDevelop is a novel application creation environment for anyone—in particular students and hobbyist programmers—to program and script their phone, directly on the phone, enabling fun, interesting and novel combinations of phone sensor data (e.g. location) and the cloud (via services, storage, computing, and social networks). TouchDevelop is available on the Windows Phone Marketplace where it became one of the 100 most downloaded apps (out over $13,000$ apps, on 4/14/2011) within one week of being launched. Users gave TouchDevelop an average rating of 4.7/5, confirming our proposition that scripting on a mobile device is possible and even desirable.

[3] http://appinventor.googlelabs.com/about/

Acknowledgements

We would like to thank all researchers and developers in the Research in Software Engineering group at Microsoft Research who helped to shape TouchDevelop in countless discussions.

References

[1] S. Burckhardt and D. Leijen. Semantics of concurrent revisions. In G. Barthe, editor, *ESOP*, volume 6602 of *Lecture Notes in Computer Science*, pages 116–135. Springer, 2011.

[2] S. Cooper. The design of alice. *Trans. Comput. Educ.*, 10:15:1–15:16, November 2010.

[3] S. H. Edwards and N. D. Barnette. Experiences using tablet pcs in a programming laboratory. In *Proceedings of the 5th conference on Information technology education*, CITC5 '04, pages 160–164, New York, NY, USA, 2004. ACM.

[4] D. B. Garlan and P. L. Miller. Gnome: An introductory programming environment based on a family of structure editors. *SIGPLAN Not.*, 19:65–72, April 1984.

[5] C. Kelleher and R. Pausch. Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM Comput. Surv.*, 37:83–137, June 2005.

[6] M. Kölling. The greenfoot programming environment. *Trans. Comput. Educ.*, 10:14:1–14:21, November 2010.

[7] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. *Trans. Comput. Educ.*, 10:16:1–16:15, November 2010.

[8] J. V. Nickerson. Visual programming. In *PhD thesis*. New York University, 1994.

[9] E. Repenning and J. Ambach. Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. In *Proceedings of the 1996 IEEE Symposium of Visual Languages*, pages 102–109. Press, 1996.

[10] T. Teitelbaum. The cornell program synthesizer: a syntax-directed programming environment. *SIGPLAN Not.*, 14:75–75, October 1979.

[11] D. J. Ward, A. F. Blackwell, and D. J. C. MacKay. Dasher - a data entry interface using continuous gestures and language models. In *Proceedings of the 13th annual ACM symposium on User interface software and technology*, UIST '00, pages 129–137, New York, NY, USA, 2000. ACM.