Children as Unwitting End-User Programmers

Marian Petre
Centre for Research in Computing
Open University, UK
m.petre@open.ac.uk

Alan F. Blackwell
Computer Laboratory
Cambridge University, UK
Alan.Blackwell@cl.cam.ac.uk

Abstract

Children who are active on the internet are performing significant design and programming activity without realising it, in the course of hacking little animations, game scripts and so on. What does such effortless learning suggest about how to support end-user programming? This paper presents observations of 'unwitting' design and programming activity by a small group of teenagers, aged 12–17. It analyses their adoption and appropriation of technology, and discusses how such practices are embedded in social networks.

1. Introduction: 'It's simple, it can't be programming'

Computing is a routine part of play for contemporary children, to the extent that they're hardly aware of their programming activities: this is just the sort of milieu Papert [7] proposed in 1980. Consider a conversation the first author had with her then-14-year-old son:

"Tell me about the programming you and your friends do."

"We don't do programming."

"What about that game you showed me with the running man?"

"Mum, that wasn't programming, that was simple."

In fact, the game he'd written required reasonably sophisticated programming within an authoring system. But because his activity was embedded in intrinsically motivated activity, it went unnoticed – it's interesting, therefore it's simple, therefore it can't be programming.

The Internet has made a variety of tools and components available to children, and they mine this vast resource with astonishing facility. 10- and 12-year-old girls who declare themselves averse to programming make animated emoticons to import into MSN dialogues. How does this 'unwitting' programming shape their models of software development?

In one sense, this phenomenon is not unique to software. Children have always appropriated, reconfigured and customized their toys. Papert was motivated by such behaviour in the philosophy of 'constructionism' that motivated the Logo programming language. However, there is a distinction between programming in the course of play, and programming as a matter of interest in itself. Logo (as suggested by its name) is, like Lego, Meccano, and similar toys a tool *for doing construction*. The distinctive feature of such toys is that the components themselves are made almost willfully uninteresting, forcing construction as the primary mode of play rather than simply one aspect. This implicit pedagogic intent is made explicit in Figure 1 – a toy from the second author's own childhood.

Figure 1: pedagogic intent of a construction toy

Toys that educate children are a preoccupation of aspirational parents, promising early preparation for professional life. As expressed by an engineering director at Rolls-Royce [12]: "the best engineers are those who are 'Meccano-minded'". Many products created for programming by children are presented as technical and educational, rather than toys. They seem designed either to be incorporated into the (explicit) educational agenda of a school curriculum, or the (implicit) educational agenda of the type of households that might in another generation have encouraged Lego and Meccano.

However, in the UK and elsewhere, today's children live in an environment where computing and the internet are ubiquitous (we might call them 'the web generation'). For this generation, programming is encountered not only through educational construction toys, but in the course of activities outside the parental and educational spheres of attention. Here children program, not in order to be

educated, but as end-users pursuing their own objectives, just as with adult end-user programmers.

1.1. Research Method

This paper reports observations on how children of 'the web generation' behave in their computing milieu. We aim to identify characteristics of their behaviour that might inform end-user programming research. The discussion is based in 'lightweight' empirical evidence: we draw lessons from longitudinal observations of our own children. This has clear drawbacks in terms of objectivity, but we believe is warranted by the benefit of a naturalistic, longitudinal perspective. Such 'parental observation' has previously been used to good effect in studies of design-like skill development where a longitudinal view is important (e.g., [3]).

The observations centered on two children, currently aged 12 (female) and 14 (male), and extended to their social groups (ranging in age from 11 to 17), over about 18 months. Direct observation of interactions was supported by examination of artefacts such as discussion logs, games, and program files (often demonstrated by the children), and by informal interviews with the children and with others in their social networks.

The children are typical of many children in the UK, where ICT is part of the national curriculum, and where even primary schools have significant computing resources. Schools typically keep computer labs open out of hours, in order to support children who do not have access at home. This study focuses on a single household, but it extends to the children's inter-linked social networks. These networks span several schools and are demographically diverse (economically, culturally, ethnically).

2. Programming and pre-programming activities

The variety of pre-programming and programming activities supported on the Internet is rich, from simple parameter tweaking, through variation and composition of components, to creation of simulations, animations, and games. Simple parameter tweaking – manipulating values and selections of pre-defined parameters in order to change the appearance, behaviour, or effect of an element in an environment – is commonplace, and a basic element of computer games such as The Sims. Internet-based examples include cyberpet tailoring in Bunnyhero Labs [5], and costume assembly in any of the multitude of dress-up games.

Many applications allow children to vary and compose components in order to make new configurations and designs. In effect, they are re-using and adapting library objects. Examples include the creation of 'skins' and composition of widgets in social networking environments such as Bebo.

Various authoring tools allow children to construct interactive simulations, animations, and games, in a manner that places far more emphasis on construction. One example is Game Maker [6], for authoring simple 2D games, which offers drag-and-drop composition of condition/action rules, a component library, and a textual programming language. Another is Kid's Programming Language (KPL) [9], a text-based general-purpose environment providing graphics, sound, and a powerful component library, within a 'learn by modification' philosophy. We also observed use of professional tools by children, such as the use of Flash to program emoticons for import into MSN.

3. Learning by tinkering

In using these products, the children did not set out to make something from scratch, and they certainly did not set out to 'learn programming'. They were offered toys, played with them, broke them, fixed them, and modified them. The means of play was programming, but the unwitting programming was undertaken as tinkering [1] with an existing artifact.

Many freeware environments are scaffolded for tinkering. They come with libraries of accessible example programs that do interesting things, and encourage experimental modification and re-use of their components. Tutorial examples offer appealing graphics and sound functions while introducing programming basics. They have usable help or suggestion systems. Importantly, they broker opportunities to share programs that users create.

Eventually, the children's expectations exceed what's available, and they begin to question constraints, to pose 'why' and 'why not' questions about the available examples. They deconstruct what's offered and seek more powerful and flexible tools. Some of environments offer a reasonably graceful transition between competencies, with effective scaffolding through visible links between actions and effects; libraries that include high-powered examples and techniques; both simple and advanced tools for changing library components, and possibly multiple notations (e.g. starting with drag-and-drop interaction, then bridging to textual notation, perhaps a general-purpose programming language).

Ultimately, the children start tinkering even where not invited, for example modifying commercial games to tweak their performance or bypass controls. The games culture promotes such activities through the circulation of 'cheats', release codes, and alterations to program files. Our observation is that children generalize from these examples. Published alterations, combined with the other experiences of modification and experimentation in other

environments, embolden them to examine program files and to think of them as things that can be altered.

A 'can-do' context is required for children to be bold to tinker: exploring, altering, and combining available material, posing 'what if' questions and experimenting. This requires confidence and comfort, as when one of our children, aged 5 and testing the limits of a calculator, made a display that contained only the letter 'E'. When asked what the E was for, she suggested "Excuse me", a contrast to the adult concern with "Error". Children clearly benefit from households where technology is assumed to provide an opportunity rather than a threat.

What conceptual and operational models are they developing as they learn by tinkering? Novices, if not offered adequate operational models of programming, may draw on other experience to create idiosyncratic or erroneous ones [10]. Nevertheless, children engaged in unwitting programming are able to discuss their goals, actions, and artifacts. They can identify and adapt components for re-use; recognize and generalize from patterns in different examples, and explain what things do, what they are for, and why they are designed that way (although the explanations may not be in conventional language). However their model of programming does seem dominated by assembly of components, rather than, for example, algorithm generation.

What impact does programming-as-play have on their models of software development processes? These environments tend to protect users (appropriately) from issues and tasks associated with large-scale software engineering. However, they are able to reason about modifications and consequences, and about interactions between components. They are able to diagnose unexpected behaviours by systematic reasoning and experimentation. Occasionally, they encounter the need to restructure programs. They spontaneously introduce disciplines such as version control, naming conventions, design for re-use and systematic debugging.

But they still don't think they're programming.

4. Playing, not learning programming

This rich milieu allows children to learn programming by composition while they tinker or play. But they have not set out to learn programming. Their motivations lie elsewhere, in social interaction, games, making cool stuff to share, and so on. Children are accomplishing this on their own, without adults, simply by reference to online material and to other children.

They learn by trying things out. When they engage with a new environment, children go straight to the examples – like adults, they don't read the tutorial unless they're already convinced that there's something worth doing [2]. They don't read the tutorial until they get stuck. And they don't read the tutorial if there's a friend (or a discussion forum) they can ask instead. As a last resort, they get an adult to read the tutorial instead [8].

The environments that appear to be successful with children are those offering useful instructive examples as a springboard to things they actually want to do; that provide immediacy of results and effects, that provide a forum for sharing and publishing successes; and that offer 'room for growth' by considerable progression beyond the basics to more advanced concepts and tools.

The environments that appear least successful are those that are hard to download and install ("I got bored. I gave up."); that have a cumbersome interface; that don't offer an easy way to start tinkering via modification of interesting existing programs; that require too much effort to produce early results; or that are too constrained to support increasingly sophisticated use.

There are many available products, designed to make programming accessible to children, that satisfy these requirements [4]. Yet much online content is still structured like traditional programming education, assuming that the user is motivated to learn programming, just as physical construction toys presumed an educational agenda. Even on the web, material intended to introduce children to programming is surprisingly traditional, for example starting with definitions of programming constructs. Take this example, from *Java for Kids* [11]:

> *"You are about to start a new journey. Writing programs that ask a computer to do certain tasks is fun and rewarding. ... You will learn what Java is and why you might want to learn Java. You will download and install the Java development software and download and install the software that will help you create Java programs. Once the preparation is done, you will run your first Java application to check that you have prepared properly. Let's get started."*

From there, the book is a standard Java tutorial, with added cartoons.

These approaches miss the fact that many contemporary children's introduction to programming is unwitting. It is a by-product of children pursuing their own goals. They are not motivated to program, and they are not attending a class or a workshop to be taught something. They're just 'messing around' with friends.

5. Support via social networks

A crucial feature of the programming activities we observed is that they are embedded in social interaction and supported by children's social networks. How is it that 10- and 12-year-old girls who 'hate programming' are making animated emoticons to import into online dialogues? Because they don't consider it programming; they see it as social activity. They impress or challenge each other; learn together and teach each other; share

ideas and insights; and program cooperatively, passing examples back and forth on-line.

Little of this activity begins in school: association with school is a social 'kiss of death' for this age group. Occasionally, the children followed leads from discussion fora or exploratory internet searches. More often, they followed leads from friends or family, usually because someone said 'See what cool thing I can do with this'. Children aren't introducing each other to programming – they're introducing each other to toys and to play.

Socialising drives tinkering:

"Look what I made."

"Cool, the asteroids explode! Can you make the comets explode too?"

"Yeah, OK"

"But there should be a cloud of debris…"

"I don't know how to do that."

"Can't you do it like the path on that bouncing ball thing you showed me?"

Much of the programming activity we observed was individual, but some was collaborative, often conducted remotely by passing artefacts back and forth online. Individual activity was socially rewarded when resultant artefacts were shared, and re-use by a friend constitutes explicit praise. Children spur each other on.

Sharing led to discussions about how an artefact might be improved, or what else might be possible. This provided a mechanism for adjusting understanding and correcting conceptual and operational models. Children critiqued and assisted with each other's artefacts, giving them additional opportunities to reflect on design and construction. Setbacks and frustrations were addressed by reference to the network: Does anyone know how to do this? Has anyone solved this? Questions were met with support and encouragement. Sharing and support also extended beyond the children's immediate social network, to fora and publication on relevant websites, with additional expertise and alternative explanations.

6. Conclusions

Many contemporary children engage in a computing milieu where they unwittingly learn end-user programming skills. The observations reported here identify several important characteristics of this activity:

1. Programming is not the goal. As far as children are concerned, they're not learning programming, but playing and socializing.
2. They are learning by tinkering: examining and modifying existing artefacts to make new variants.
3. Children do this in the society of other children; adults have little involvement. They are learning within a social network, not an educational context.

The way this activity is embedded in and supported by social interaction, the apparent ease with which children engage in tinkering, and the kind of environments they find congenial to do so, offer lessons for teaching and supporting end-user activity. Educators have long discussed a 'studio' approach to teaching programming. It seems that, in this social and online context, children have implemented it themselves.

7. Acknowledgements

Our gratitude to the children. Thanks to Mary Beth Rosson, and participants in the 2007 Dagstuhl Seminar on End-User Software Engineering, for comments on earlier material. Marian Petre is a Royal Society Wolfson Research Merit Award Holder.

8. References

[1] Beckwith, L., Kissinger, C., Burnett, B., Wiedenbeck, S., Lawrance, J., Blackwell, A. and Cook, C. (2006). Tinkering and gender in end-user programmers' debugging. Proc. CHI'06, pp. 231-240.

[2] Carroll, J.M. & Rosson, M.B. (1987). Paradox of the active user. In J. M. Carroll (Ed.) *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*, Bradford Books, pp. 80-111

[3] Fenson, L. (1985) The transition from construction to sketching in children's drawings. In N.H. Freeman & M.V. Cox Visual order: The nature and development of pictorial representation. Cambridge University Press, 374-384.

[4] Kelleher, C., and Pausch, R. (2005) Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. ACM Computing Surveys, 37 (2), 83-137.

[5] Lee, W.A. (2004) Bunnyhero Labs. Accessed 15 March 2007 at http://bunnyherolabs.com/about.php

[6] Overmars, M. (2007) Game Maker, v. 7. YoYo Games. Accessed 15 March 2007 from http://www.gamemaker.nl/

[7] Papert, S. (1980) *Mind-storms: Children, Computers, and Powerful Ideas*. New York: Harvester Wheatsheaf.

[8] Rode, J.A., Toye, E.F. and Blackwell, A.F. (2005). The domestic economy: A broader unit of analysis for end user programming. Proc. CHI'05, pp. 1757-1760

[9] Stagner, J. (2005) Kid's Programming Language. Morrison Schwartz, Inc. Accessed 15 March 2007 from http://www.kidsprogramminglanguage.com/

[10] Taylor, J. (1999) Analysing novices analysing Prolog: what stories do novices tell themselves about Prolog?. In: P. Brna, B du Boulay and H. Pain (eds) *Learning to Build and Comprehend Complex Information Structures: Prolog as a Case Study*. Ablex.

[11] Tylee, L, (2003) *Java for Kids*. KIDware Accessed 15 March 2007. Download via http://www.download.com/Java-for-Kids/3000-2415_4-10197217.html

[12] Workthing Ltd. (2005) Skillset. Accessed 15 March 2007 from http://www.workthing.com/career-advice/breaking into/graduate/breakeng_skillset.html