Neither Grasshopper nor Ant: learning from DIY coding and from gaming [WIP]

Ioanna Iacovides and T R G Green
Computer Science Dept
University of York
jo.iacovides@york.ac.uk : thosgreen@gmail.com

Abstract

'DIY' coding for fun and gaming have more in common than seems to have been noticed, both notationally (no abstractions, no juxtaposability, etc) and in the user experiences (challenge, breakdowns, breakthroughs). We argue that some familiar claims about end-user programming, such as a need for domain-specific languages, do not apply to DIY coding, and that lightweight optional features such as 'abstraction by accretion' could help both DIY coders and gamers when the project grew too big.

1. Introduction

> *You are baffled. You've tried all the obvious things and they didn't work. What about ... here's another possibility ... Ah! Success! Suddenly you understand how this bit works! – And now, on to the next challenge ...*

Question: what were 'you' doing? Surely that vignette could apply equally well to certain kinds of gaming or to certain kinds of coding: games that present a series of obstacles to be surmounted, or to 'design it yourself' coding – small-scale end-user development. The user experiences of gaming – challenges, breakdowns, mental absorption ('flow'), and learning by breakthroughs – closely match the authors' experiences of using DIY tools. At present gaming and small-scale coding are regarded as entirely different worlds analysed in very different ways, but if we are right, they are similar in many ways. Which means that insights from one world should apply equally to the other world.

To code is to engage with an information artefact, and there has been a certain amount of exploration of the properties of coding notations (such as programming languages and their relatives) and the development environments. Playing a game where the player explores a world is also a form of engagement with an information artefact, though less obviously so; the player has to discover the choices, the pitfalls, the rewards etc, using a highly specialised interaction system. These can be analysed using similar concepts, and we shall choose (surprise surprise) the cognitive dimensions framework (Green 1989, Green and Church, in prep.).

Games are more conventionally approached via the user experience (UX), of course. Apart from the anecdotal vignette above, Iacovides et al. (2014; 2015) have provided a detailed analysis of the gaming experience that highlights how gameplay involves iterative cycles of breakdown and breakthrough. The same analysis can be applied to coding. Thus, we can explore both coding and gaming in terms of both notational analysis and UX analysis, our conjecture being that there will be a substantial overlap, overlooked until now. In short, we wish to argue that the coder and the player engage in very similar strategies, for similar reasons.

2. What kind of coding? What kind of games?

Of the many genres of coding, our focus is on small-scale, single person, coding by one person, not necessarily as part of their job, small-scale, with little eye to the future, maybe doing it as much for fun as for anything else: e.g. a frequent flyer might write some code for their smartphone to record when the biggest shocks occurred during luggage transport on a long haul flight, not because they really needed to know but for curiosity. Or someone might decide that although they could perfectly well do <insert task here> by hand, it would be more entertaining to write a script, even though the overall time might well be much longer. There is a spectrum between the person building a tool for no other reason than because they need it, and the person who intrinsically enjoys the coding experience and looks around for an excuse, like a home woodworker casting about for a suitable project. This is a subgenre of 'end-user programming' or 'end-user development' that has received little attention in the research literature. We shall refer to it [unless someone finds a better name] as 'Develop It Yourself' coding, or DIY, and especially 'DIY for fun'.

There are also many different types of games. For the purposes of comparison, our focus is on single-player digital games that individuals play for entertainment and leisure. These range from puzzle games played on mobile phones, to larger open world PC games with multiple quests. From solving the MC Esther inspired puzzles in Monument Valley, to exploring which potions will be most useful to use when battling a specific monster in Witcher 3, these games include multiple challenges that need to be overcome. In each case, the player is intrinsically motivated to play, and to learn how to do better after they fail.

2. Notational aspects

Taking gaming first: in a typical role-playing game the player controls a 'hero' with some kind of quest that has to be achieved by evading or fighting other characters, collecting some objects and avoiding others, and finding a route through various locations to a goal. The control of a game is an 'action language', in the terminology of cognitive dimensions, since gaming uses a transient medium rather than a persistent one like coding. In the cognitive dimensions framework we distinguish between various broadbrush types of activity, and in this type of game the major type of activity might be seen as searching ('How do I get to the goal?') or as exploratory understanding ('How does this game work?'): depending on whether the game-play feels more like mapping a space, or more like trying to understand the internals of an obscure device like the Antikythera Mechanism (https://en.wikipedia.org/wiki/Antikythera_mechanism).

Now we turn to DIY-for-fun, three DIY systems in particular: the old Macintosh system 'HyperCard', the very familiar spreadsheet, and Twine, an IDE for text adventure writers. These three are sufficiently different to provide a representative sample, we hope, and all three have been very successful in their own spheres. They have important features in common; some of those features one might expect to be quite disadvantageous, so a rethinking is needed.

(1) The information (code instructions and data) is divided up among little cells. The contents of these little cells can only be inspected by opening them, and only one can be opened at a time – which is exactly like visiting one game location at a time to find out what's there: no **juxtaposition**, in CDs terms.

(2) The cells can only communicate via global variables (for HyperCard and Twine) or data-flow links (in spreadsheets), and the IDEs do not offer any support to reveal which cells set or use what data. This, too, parallels the game task, where one location might contain a squirrel and another might contain the Famous Oak and the player has to have fed the squirrel before reaching where Famous Oak grows, but there is nothing manifest to connect the two. In cognitive dimensions terms, these systems generate **hidden dependencies**.

(3) Perhaps crucially, there are **no abstractions**. The three coding systems contain no arrays or lists or other complex data structures, no parameters to functions, etc. There are therefore no aggregate operations or definitions. For example, a simple calculator in HyperCard would include 10 buttons labelled with the digits 0-9, and each individual button would have its own packet of code saying something like this:

```
on mouseup
  get the short name of me
  do something with that name
end mouseup
```

There is no way to declare that buttons 0-9 belong to a class of 'digit buttons', and that all digit buttons have the same set of properties. Much the same was true of spreadsheets until recently – certainly they achieved their grand success, in the days of VisiCalc and Lotus 1-2-3, without any such abstractions: the contents of any cell, whether a value or a formula, had to be manipulated individually rather than as a group of similar cells. It is also of true of Twine, in which the game world is made up of locations (called 'passages'), each containing its own individual bit of code, with no class structure to group them by.

Exactly the same is true of the game-play. Each character, object and location is *sui generis*, of its own kind and no other, and there is no way to interact with them in any way except individually; but one could imagine games where the 'hero' could organise other characters into groups and locations into suites, and recruit a group of elves to build a bridge over every river or search a suite of rooms. The player would have to decide whether the cost of creating groupings was likely to pay off later in the game; the cost would be a form of '**attention investment**' (Blackwell 2002), and the need to make the decision would be another example of premature commitment. Games and coding systems where the player could create such groupings would be termed '**abstraction-tolerant**' in the CDs framework, in contrast to the existent **abstraction-hating** nature.

The lack of abstractions has many important consequences. Since the code must be repeated there are opportunities for slips, and since the buttons, locations, cells etc are all individual, if the coder wishes to change the code they have to make the same change many times rather than redefining a button object. In the same way, if a player wishes to repeat the game-play (perhaps after have been 'killed' and needing to restart), each individual location and choice must be revisited. This is an **error-prone** structure, and if the structure needs to be modified in the future it will require much work, '**repetition viscosity**' in CDs terms. But swings and roundabouts: the syntactic load for learners is markedly reduced, and the **start-up** effort is minimal – the user can get straight on with the job in hand. With no abstractions, **premature commitment** (being forced to make a choice before you're ready to do so) is non-existent, and any component can be added or edited at any time, perhaps put on one side to be used later.

3. The user experience

As noted above, user experience of DIY coding is characterised by challenges, breakdowns, mental absorption ('flow'), and learning by breakthroughs. That is not how software engineers and professional programmers proceed, but DIY coders are not software engineers and usually have no relevant training – indeed, an interesting study by Blackwell and Morrison (2010) highlights many of the differences, not just in training but in the work context of end-users and software engineers. For the DIY coder, progress is far from smooth. Ko et al. (2004) identify barriers in "design, selection, coordination, use, understanding, and information ....[of] any element of a programming system's language or accompanying libraries that can be used to achieve some behaviour." These are the barriers that lead to learning by breakthrough.

Nevertheless DIY coders appear to get intrinsic satisfaction from the process – or rather, some do: Aghaee et al (2015) report evidence that identifiable and distinct motivational factors in end-user programming are associated with particular psychometric personality traits.

Similarly, the game user experience can also be characterised by challenges, breakdowns, mental absorption ('flow'), and learning by breakthroughs. Building on the work of Sharples and colleagues in evaluating mobile learning technologies (Sharples 2009; Vavoula & Sharples, 2009), Iacovides et al (2015) illustrate how, players experience cycles of breakdown and breakthrough in an attempt to overcome in-game challenges. Breakdowns and breakthroughs can occur in relation to action (e.g. problems with the controls vs performing a new attack); understanding (e.g. not knowing what to do next vs figuring out a solution a puzzle); and involvement (e.g. getting frustrated vs experiencing satisfaction). Iacovides and colleagues (2015), in an in-depth analysis of a range of different games, shows how minor breakdowns are a regular part of the gameplay experience, but when these are overcome, via breakthroughs where the player has learnt how to improve, and particularly when players feel responsible for their own progress, they lead to a sense of satisfaction that increases overall enjoyment.

In related work, Iacovides et al (2014) also investigated the different strategies that players use to tray and overcome breakdowns. These strategies include:

- *Trial and error*: where the player tries out an action to see what, if anything, may happen e.g., what happens if I jump on the moving platform?
- *Experiment*: here the player builds on previous knowledge to develop an informal hypothesis and test it out in the game e.g., if I use this potion, it will increase charisma and make me more likely to persuade an in-game character.
- *Repetition*: where the player tries the same action again e.g., practising an attack until you are sure how it works.
- *Take the hint:* games often provide instructions or hints, in this case the player decides to do what the game suggests e.g., pressing the action button to use the lift when a pop-up screen tells you to do so.
- *Stop & think:* when a player decides to pause gameplay to consider what they are doing, or even look up external resources for in-game help e.g., getting stuck and looking at a walkthrough online to find out how to proceed.

Apart from perhaps *repetition,* these strategies seem remarkably similar to how users might engage in DIY programming. In both cases an exploratory approach is adopted, trying out different actions to see what works, adjusting actions based on the knowledge gained from doing and any in-system guidance, even resorting to external help when more significant difficulties are encountered.

4. Reasons for success

Overlaps and correspondences between these two apparently dissimilar domains have now become visible. The player of a typical game, seeking to understand the world and create a path to success, is doing something akin to the activity of exploratory design that a DIY programmer engages in, despite them interacting with rather different interfaces. In the process, they are able to overcome the breakdowns they encounter and to learn via achieving breakthroughs in understanding. For both, the satisfaction that results in being able to overcome breakdowns and feel responsible for that progress is key to their continued engagement.

5. And so ....

Gaming is fun. DIY coding is fun. The notations and the activities are similar. What can we learn?

The conventional wisdom is that systems for use by end-users should be domain-specific (as already noted), should avoid challenges and breakdowns, and should encourage a software-engineering work-style, with an eye to the future use of the code. It would also be reasonable to argue that hidden dependencies should be avoided. But as far as DIY-for-fun coding goes, we suggest that all of those are mistaken. HyperCard was wildly successful despite not being domain-specific. We think part of the fun in DIY-for-fun is the experience of challenge, breakdown, and breakthrough, just as in gaming. As for 'an eye to the future', we believe that the DIY coder takes little stock of the future; they are coding for *now*, partly because they enjoy it. Looking ahead and considering best practices, proper testing, etc simply isn't what they came for. (Who does unit testing for fun?) The DIY coder can remember enough of the code to be able to cope; for example, a hidden dependency that might prove troublesome in the future can be accepted for 'just for today' because they can remember the linkage, at least for now. In practice, these features, especially the hidden dependencies, are likely to lead to 'mature disfluency' (Green 1995) as projects grow – difficulties that increase exponentially with the size of the code.

One is reminded of the ant and the grasshopper: the ant sedulously stores food, against the arrival of winter, but gets no fun from doing it; the grasshopper gets lots of singing and dancing, but starves in winter. DIY coders get their fun by assuming all those contingencies that software engineering guards against will happen to others, not to them: a bit like betting that the sun will not rise tomorrow.

In a very different domain, Blackwell et al. (2003) found that of six professional administrative workers interviewed, only one "ever created any directories on the computer hard disk, and this was a computer enthusiast who had multiple computers at home. ... Others achieved impressive mnemonic feats rather than experiment with unfamiliar facilities of the computer – one secretary kept 358 working files in a single directory, and was able immediately to find the file she needed, despite the historical use of eight-character filenames." (p. 537) Yet these workers created extensive paper-based abstractions: desk drawers, shelves, baskets, paper spikes, diaries and address books, filing cabinet drawers, and so on. "The problem is not one of abstraction capability, but of the unsuitability of computer user interfaces for abstraction management," they say. Rather than struggle with those interfaces, they kept a lot of material in the head and hoped for the best, like the grasshopper.

If these ideas are right, then trying to encourage a software-engineering outlook in DIY coders is pointless (and perhaps patronising). Yet they run the risk that sometimes the project will grow too big for their head, and perhaps some of the effects could be mitigated. For example, it might be possible to add juxtaposability, or audit trails (as in Excel), or to devise a form of 'abstraction by accretion' akin to the way that some popular systems, including WhatsApp, address books, Facebook, and most vector-based drawing programs work, by selecting some items and declaring them to be a group. This can be done at any time, so there is no enforced attention investment and no risk of premature commitment, and there are usually no possibilities of subgroups, so the abstraction management is trivial; yet some of the advantages of abstractions are gained – for messaging, one can decide it would be useful to create an address group called "the people I do X with" and add names to that list as and when convenient, after which a message can be sent to the entire group in a single action. This not only brings the obvious advantage of aggregate actions on all the items at once, but also the useful side-effect of identifying the items concerned as a group, to help a future reader to understand the structure. Yet the grasshopper can go on singing happily, ignoring the option of forming any abstractions, until the project has become too big and Something Has To Be Done. Just how 'abstraction by accretion' would work would need to be explored. The important thing is that grasshoppers could go on having fun until things got too difficult, and at that point a nearly-effortless move could be made, towards just as much antlikeness as was necessary and no more.

6. References

Aghaee, S., Blackwell, A.F., Kosinski, M. and Stillwell, D. (2015). Personality and intrinsic motivational factors in end-user programming. *Proceedings of IEEE Symposium on Visual Languages and Human Centric Computing (VL/HCC 2015)*, pp. 29-36.

Blackwell, A.F. (2002). First steps in programming: A rationale for Attention Investment models. In *Proceedings of the IEEE Symposia on Human-Centric Computing Languages and Environments*, pp. 2-10.

Blackwell, A.F., Hewson, R.L. and Green, T.R.G. (2003) Product design to support user abstractions. In E. Hollnagel (Ed.) *Handbook of Cognitive Task Design*. Lawrence Erlbaum Associates. ISBN 0-8058-4003-6, pp. 525-545.

Blackwell, A. F. and Morrison, C. A logical mind, not a programming mind: Psychology of a professional end-user. (2010) Proc. PPIG 2010

Friedhoff, J (2014) Untangling Twine . *DiGRA '13 - Proceedings of the 2013 DiGRA International Conference, vol 7: DeFragging Game Studies*. August, 2014. ISBN / ISNN: ISSN 2342-9666

Green, T. R. G. (1995) Looking through HCI. In Kirby, M. A. R., Dix, A. J. and Finlay, J. E. (Eds.) *People and Computers X. Cambridge University Press*

Green, T. R. G. (1989) Cognitive dimensions of notations. In R. Winder and A. Sutcliffe (Eds), *People and Computers V.* Cambridge University Press

Green, T R G, and Church, L E *Notations: Life and Times (title TBC*). Unseen University Press, in preparation.

Iacovides, I., Cox, A.L., McAndrew P., Aczel, J., & Scanlon, E. (2015). Game-play breakdowns and breakthroughs: Exploring the relationship between action, understanding and involvement. *Human Computer Interaction*, 30 (3-4), 202-231.

Iacovides, I., Cox A.L., Avakian A., & Knoll, T. (2014). Player strategies: Achieving breakthroughs and progressing in single-player and cooperative games. In *Proceedings of the first ACM SIGCHI annual symposium on Computer-human interaction in play, CHI Play 2014*, pp. 131-140. New York, NY, USA, ACM.

Ko, A. J., Abraham, R., Beckwith, L., Blackwell, A., Burnett, M., Erwig, M., Scaffidi, C., Lawrance, J., Lieberman, H., Myers, B., Rosson, M. B., Rothermel, G., Shaw, M., and Wiedenbeck, S. (2011) The state of the art in end-user software engineering. *ACM Computing Surveys* 43, 3, Article 21 (April 2011), 44 pages. DOI = 10.1145/1922649.1922658 http://doi.acm.org/10.1145/1922649.1922658

Ko, A. J, Myers, B. A. and Aung, H H (2004) Six learning barriers in end-user programming systems. *IEEE Symposium on Visual Languages - Human Centric Computing*, 2004, pp. 199-206, doi: 10.1109/VLHCC.2004.47.

Sharples, M. (2009). Methods for evaluating mobile learning. In G. Vavoula, N. Pachler, A. Kukulska-Hulme (Eds.) *Researching mobile learning: Frameworks, tools and research designs*, (pp. 17–39). Oxford: Peter Lang Verlag.

Vavoula, G. N., & Sharples, M. (2009). Meeting the challenges in evaluating mobile learning: A 3-level evaluation framework. *International Journal of Mobile and Blended Learning* 1(2), 54–75.