Learnable Computing with Kodu? Computational Thinking and the Semiotics of Game Creation Interfaces

Benjamin DeVane, University of Iowa
Kelly M. Tran, Arizona State University
Cody Steward, University of Florida
Brian LaPlant, University of Florida

Introduction

More research suggests that easy-to-use game creation software, some of which feature visual programming environments, could help interest young people in computing while supporting their nascent development of computational thinking practices (Games, 2008; Kafai et al., 2010; Berland & Lee, 2011).

This paper looks at an afterschool program's use of a tool called Kodu Game Lab to introduce middle-school students to basic elements of programming. It investigates the unique design of Kodu's visual programming interface, and asks how learners think computationally with unique visual programming representations. To do so, it analyzes data from a programming challenge presented to middle-school program participants, and argues that the *economy of activity* embedded in Kodu's interface design shapes and structures participants' understandings of basic programming concepts in particular ways.

Theoretical Framework: Computational Thinking & Game Design

Understandings of computational thinking emphasize a person's ability to abstract problem spaces and use algorithms to systematically frame solutions – practices fundamental to high-level computer science practice (Wing, 2008). While the core emphasis in computational thinking is the use of algorithmic procedures to solve problems, some scholarship places more emphasis on building learners' competencies with programming languages (Guzdial, 2008). Aiming to support the development of computational thinking practices among young people, recent research initiatives have employed game- and animation-creation environments.

These game creation environments, like Scratch (Resnick et al. 2009, Kafai et al., 2010), AgentSheets (Reppenning & Sumner, 1995), and IPRO (Berland et al., 2013), use intuitive end-user, visual programming languages to support the development of computational thinking practices. These high-level languages are often integrated within visually-rich interfaces that attempt to support and scaffold learners' understanding of programming principles like conditions, iteration, events and modularization (see Berland et al., 2013).

Kodu Game Lab is one-such game-focused, high-level visual programming environment, which was developed by Microsoft Research. Kodu distinguishes itself with a unique visual interface (described below), which provides users with point-and-click tools that afford the fashioning of three-dimensional game levels. These tools allow the use to place characters and objects in a world, program those characters using conditional logic, and employ a set of game-focused commands to craft in-world events. Early research suggests Kodu aids in the development of foundational computational thinking practices (Anton et al., 2012).

Programming interfaces as semiotic-material 'pillars' of computational literacy

This study understands programming languages to be a material pillar, to use DiSessa's (2001) language, of computational literacy. In framing computational thinking practices as a form of literacy, DiSessa writes that every type of literacy has three pillars: material, cognitive and social. He further argues for the importance of representational form in shaping literacy practices and material intelligence. People, in DiSessa's reckoning, do not "have ideas and then express them in the medium"; rather, people "have ideas *with* the medium" (2001, pg. 116).

Scholarship in systemic-functional linguistics on multimodal representations has similarly been concerned with the interconnectedness between semiotic, cognitive and social elements of the world (Kress & van Leeuwen, 2006; Lemke, 2009).The way that people recognize, produce and think with images is in part a product of the "design grammar" visual of representations – the organization of depicted elements into semiotic-cognitive meanings. What, then, is the design grammar of Kodu Game Lab? And how does it shape the way that users *have ideas with the tool*? In what respects does the design of this tool render aspects of computational thinking more or less "learnable" (see DiSessa, 1977)?

The ‘Design Grammar’ of Kodu Game Lab

Kodu has two modes, an edit mode in which the player creates the game, and a play mode in which they play the game they have produced. The edit mode has two types of interfaces: one type concerned with world level design in which users can place objects and character, shape and design the landscape, and place automated character paths; and the second focused on object programming in which the character can set the behaviors of an object (or character). The former set of interfaces is immediately visible to the user in the ‘default’ edit mode, while the latter must be accessed by right-clicking on an object and selecting ‘program.’ Characters and objects are each individually programmed like agents – they each respond to the world in a different way (see Figure 1).

Figure 1: Programming a Kodu bot to find and eat apples

Designed originally to be interoperable on personal computers and gaming consoles, Kodu’s programming interface uses a wheel menu that shows which conditions, actions or modifiers are available to be used. As of July 2013, the number of conditions and actions is large, with over 64 conditions and 111 actions available. At most twelve programming ‘primitives’ – conditions, actions or modifiers - are available to select on the screen at any one time, meaning that users have to navigate a multi-level hierarchy of wheel menu interfaces to find a primitive (see Figure 2).

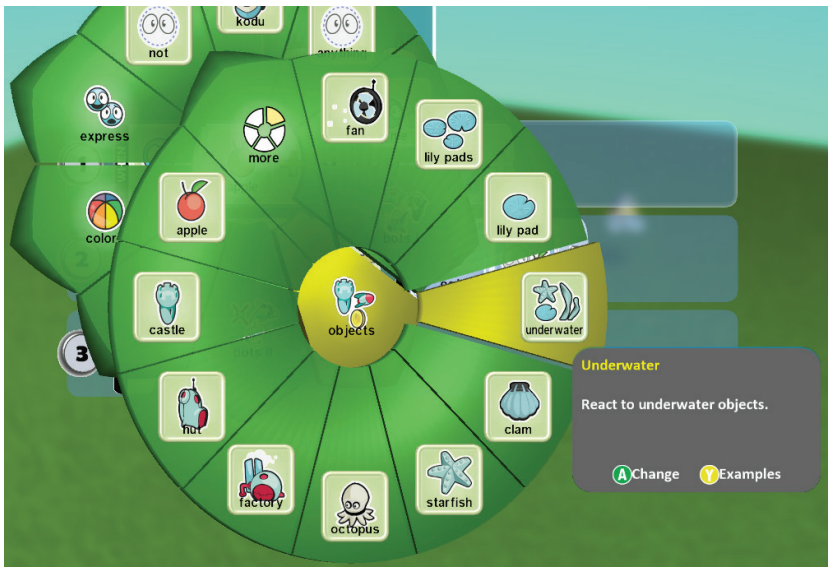

Figure 2: The programming interface shows a limited number of primitives at once

Data Sources and Research Context

Data analyzed is drawn from participants’ Kodu files that researchers archived, audiovideo recordings of participants’ activity, and researcher field notes from a nine-week after-school program at a middle school. Between 12-17 participants attending each session, and all participants were asked to commit to missing at most one session. Each session ran for one hour.

During the fourth week of the program, participants were challenged to solve a problem in the game world: Given two pieces of virtual land suspended in the air, how could they create a game where a character teleported from one piece of land to another? (see Figure 3) Participants were put into teams of two, and asked to collaboratively create a solution inside the Kodu programming interface. Six pairs of participants, eleven males and one female,

all between the ages of 10 and thirteen who had completed five introductory tutorials on object and character programming took part in the challenge.

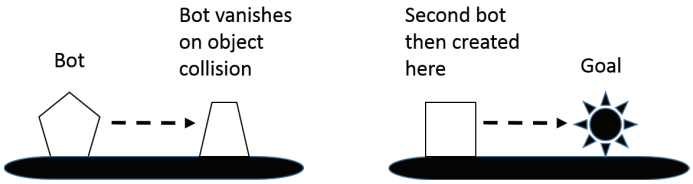

Figure 3: The solution to the challenge

In the starting condition, the following objects were already present in the virtual spaces: a) two game characters (called "Kodu bots"), one visible and one marked "creatable" – visible to creators but invisible in play; b) one object on each platform – a bumper and a rock; and c) a castle that marked the creator's final goal. Participants were told that there was no single "action" that would solve the problem, that they would have to be creative, and that they would need to use a second character present in the file.

The second character, a "Creatable" character, was invisible in game play until instantiated though it was visible and programmable in Kodu's edit mode. This challenge, which was a very difficult one for players in their fifth hour of game play, was a perhaps poorly conceived attempt to help participants understand that Kodu characters were instances of an object and not permanent figures in a game world as some participants seemed to think.

One correct solution to the problem is to create the illusion of teleportation: when the character hits an object on the first landform, it plays a vanish sound and becomes invisible. The sound event tells an identical second character-object to become visible on the second landform, giving the appearance of teleportation.

Methodology: Multimodal Discourse Analysis

This study employs multimodal discourse analysis (Lemke, 2012) to examine the design grammar of participants' game artifacts relative to the authoring environment, and utilizes D/discourse analysis (Gee & Green, 1998) to examine how meanings are generated relative to particular contexts of activity (e.g. the visual grammar of the game, the social system of an afterschool club). This study was conducts as part of a nine-week design-based research investigation (Collins, 1992). Verbal data is represented using a version of *naturalized transcription* in order to provide a verbatim account of talk as social action (Jefferson, 1984).

This paper presents data from a single case study, chosen using "typical text" selection methods. Like other forms of qualitative inquiry in which external validity is establish socially, through the accumulation and assessment of the collective body of research, this paper does not purport to demonstrate the external validity of its sample through its methods (Wodak, 2001). Rather, it adopts utilizes a purposive, convenience sample (Stake, 1995) as a means of iteratively developing constructs of inquiry relative to game creation, computational thinking and visual interface design.

Results

All groups failed to solve the difficult, in-game problem of 'teleporting' a Kodu bot from one floating island to another. But the nature of their attempted solutions is significant: Almost all participants' attempted solutions employed actions that were visually prominent in the environment's interfaces. The solution path of Chuck and Sean, presented below as a case analysis, was in many ways emblematic of participants' problem-solving activities.

Chuck and Sean's solutions: Bigger, faster, more explosions

First Solution: Level manipulation. Chuck and Sean initially sought to craft a solution to the programming challenge posed them that did not employ programming at all. Instead, they attempt to manipulate 'physical' properties of the virtual landscape and the objects within it. The first tried to flood the space with water, presumably so that their bot character could swim across the space between the floating islands. When their attempt to swim between the islands failed, they made their character bot bigger, believing that it might aid in the endeavor. These two actions – flooding with water and increasing character size – require little navigation of interfaces from the default edit screen. The water tool is immediately visible, requiring one click, and the size increase tool requires two. Neither requires use of the programming interface.

Second Solution: Exploding object. In their second set of solution attempts, Chuck and Sean focused on inserting an object – an exploding mine – into the world that would do the work of transporting their character bot for them. After placing the mine in the world, they switched to play mode, maneuvered the character-bot into it, and nothing happened. The two then increased the size of the mine, but then realized that they would have to program the mine to make it explode:

(0:05:35.2)C: Change size. Bi::ig °mine° (.) Per::fect.

(0:05:40.2)C: Ok so when- ((Opening Programming Screen))

(0:05:41.0)S: when (.) when bump. when [bump.

(0:05:44.5)C: bump]. it will- ((Searching primitive wheel menu))

(0:05:48.1)S: when bump into-

(0:05:48.9)C: it will- (2.0) °it will° (4.6) Oh! ((Searching wheel menu))

(0:06:04.2)C: it will- (3.9) it- (.) it will- ((Searching wheel menu))

(0:06:09.9)C: it will shoot? it will (.4) °shoot°.

(0:06:18.9)C: (hhh) (1.1) °cruise°.

(0:06:23.3)S: Alright? let see what happens?

In overlapping turns of sometimes self-directed talk, Chuck and Sean came to the joint realization that they will have to program the mine with the condition “WHEN BUMP KODU BOT, THEN [ACTION]” to try out their strategy. However, they were unsure what the [ACTION] should be.

The two spent almost forty seconds searching the wheeled primitive interface for an action to take. Much of this time was occupied by overlapping, self-directed talk, with long pauses as they sought to complete the conditional-like sentence “when bump into bot, it will explode” while trying to program that very sentence into the interface. However, they could only find the primitive action “shoot” instead of explode, so they chose to program the mine to shoot missiles upon colliding with a character bot. When they entered the “play mode”, the mine shot a missile into their bot upon collision, destroying it.

Third Solution: Launcher Creation. In their third solution attempt, Chuck and Sean tried to create a launcher that would catapult them between the two pieces of land, a more complicated iteration of their mine gambit. They conjectured that a large object, programmed to launch character bots on contact, would sling the character with sufficient velocity to make it to the other piece of land. To test out this theory, Chuck inserted a large castle-tower on the starting landform, prompting a moderator (M) to ask him what he was doing.

(0:07:05.1)M: Ok so you’re creating a big castle over there?

(0:07:06.5)C: ° >Yeah so we haven’t - I haven’t - I have an idea for an action-°

(0:07:09.6)M: What’s your theory?

(0:07:10.8)C: >My theory is (.) the castle’s big enough that we’re probably<

(0:07:14.0)C: We’re going to put a launcher that will be strong enough tolaunch us

(0:07:16.9)C: Cause that launcher down there is too weak.

(0:07:19.7)M: Oh (.) well that’s a theory

(0:07:21.0)S: >I HAVE- I have [an idea<

(0:07:22.0)C: >°Stop] stop stop°< we already- we already-

(0:07:24.1)S: The size! The size of the launcher.

Prompted by the moderator, Chuck outlined his theory that it is the size of objects that has to do with how far they can “launch” – a name of one of a catapult-like Kodu primitive – characters far enough to reach the other island. Sean, who seemed not to have been listening to Chuck, acted like he had reached the same conclusion independently.

After their launcher failed to propel their character across the gap, Chuck and Sean then tried launcher-based solutions that involved increasing the size and speed of their bot, solutions that also failed to accomplish their goal. Sean then pointed out that their launchers did not actually launching the bot. This observation frustrated both of them, and led them to abandon trying to formulate their own solutions. Instead they spied on others and mimicked their attempted solutions, and eventually ‘cheated’ by building a land bridge to between the pieces of land.

Discussion

Research has indicated that game creation tools, like Kodu with accessible and structured visual programming interfaces can support young people’s development of programming fundamentals and computational thinking practices. But questions remain about the manner in which these tools carry out this undertaking: How does the visual *design grammar* of a tool afford and constrain certain modes of problem-solving practice? What is the relationship between what is visible in an interface and what is learned?

The three solution paths of Chuck and Sean, which we understand to be typical of those in the club, gradually began to employ programming primitives that were further from the default edit screen. While their propos ed solutions may seem naïve, they grew more willing to craft solutions that used programming primitives that were less perceptible to the new Kodu user that were deeper in the layered menu interface (see Table 1).

Solu-tion Number	Action	Action Type	Interface Layer Depth
1	Character size in-crease	Object/environment manipulation	1
1	Flooding level with water	Object/environment manipulation	1
2	Programming mine to shoot missiles	Object condition programming	2
3	Programming castle to launch characters	Object condition programming	3
3	Increasing the size of the castle	Object/environment manipulation	1

Table 1: Interface Layer Depth of Solution Actions

However, at the same time, the hierarchical and layered nature of the interface may have been an influence on the willingness of Chuck and Sean to pursue overly-simple solutions. Kodu makes creating game actions easy because of its extensive primitives, but it may in turn encourage a reliance on using existing primitives instead of thinking creatively.

Conclusion

The larger question emerging from this study is: What is the nature, to paraphrase DiSessa, of having ideas and thinking with Kodu and other highly-visual programming environments? This qualitative study suggests that thinking about computational problems with Kodu Game Lab is bound up with the activity of exploring layered visual interfaces. Given the very-developed visual interfaces of programming tools like Scratch, AgentSheets and IPRO, it is also likely true that the understanding and interpreting of interfaces is intertwined with computational thinking with those tools as well.

While researchers focused on computational literacy have always been concerned with the relationship between thinking and representation, historically they employed simple textual programming languages, rather than visual programming interfaces that are complex visual semiotic systems. How then do design grammars of visual representation and interaction support new modes of computational thinking for learners, while perhaps

constraining others? What constellations of semiotic, cognitive and social relationships are engendered by new ways of representing programming?

References

Anton, G., Hatfield, D., Ochsner, A., Shapiro, B., & Squire, K. (2013). Studio K: Tools for game design and computational thinking. In Rummel, N. Kapur, M. Nathan, M. & S. Puntambekar (Eds.) Proceedings of the 2013 International Conference on Computer-Supported Collaborative Learning (CSCL2013). Madison, WI.

Berland, M., & Lee, V. R. (2011). Collaborative strategic board games as a site for distributed computational thinking. *International Journal of Game-Based Learning*, *1*(2), 65.

Berland, M., Martin, T., Benton, T., Petrick Smith, C., & Davis, D. (2013). Using Learning Analytics to Understand the Learning Pathways of Novice Programmers. *Journal of the Learning Sciences*, 1–36.

DiSessa, A. A. (1977). On“ Learnable” Representations of Knowledge: A Meaning for the Computational Metaphor. *LOGO Memo* 47. Cambridge: MIT.

DiSessa, A. A. (2001). *Changing minds: Computers, learning and literacy*. The MIT Press.

Games, I. (2008). Three Dialogs: a framework for the analysis and assessment of twenty-first-century literacy practices, and its use in the context of game design within Gamestar Mechanic. E-Learning, 5(4), 396–417.

Gee, J., & Green, J. (1998). Discourse analysis, learning, and social practice: A methodological study. Review of research in education, 23(1), 119–63.

Guzdial, M. (2008). Education Paving the way for computational thinking. *Communications of the ACM*, *51*(8), 25–27.

Jefferson, G. (1984). Transcription notation. In J. Atkinson & J. Heritage (Eds.), *Structures of social action: Studies in conversation analysis*. New York: Cambridge University Press.

Kafai, Y. B., Fields, D. A., & Burke, W. Q. (2010). Entering the clubhouse: Case studies of young programmers joining the online Scratch communities. *Journal of Organizational and End User Computing (JOEUC)*, *22*(2), 21–35.

Kress, G. R., & Van Leeuwen, T. (2006). *Reading images: The grammar of visual design (2nd ed.).* New York: Psychology Press.

Lemke, J. L. (2012). Multimedia and discourse analysis. *Routledge Handbook of Discourse Analysis*, 79–89.

Papert, S. (1980). *Mindstorms: children, computers, and powerful ideas*. New York: Basic Books.

Repenning, A., & Sumner, T. (1995). Agentsheets: A medium for creating domain-oriented visual languages. *Computer, 28*(3), 17–25.

Resnick, M., Maloney, J., Monroy-Hernández, A., Rusk, N., Eastmond, E., Brennan, K., Silverman, B. (2009). Scratch: programming for all. *Communications of the ACM, 52*(11), 60–67.

Stake, R. (1995). *The art of case study*. Thousand Oaks, CA: Sage Publications.

Stolee, K. T. (2010). Kodu language and grammar specification. *Microsoft Research whitepaper, Retrieved September*, *1*.

Wing, J. M. (2008). Computational thinking and thinking about computing. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, *366*(1881), 3717–3725.

Wodak, R. (2001). The discourse-historical approach. In R. Wodak & M. Meyer (Eds.), *Methods of critical discourse analysis* (pp. 63–94). London: Sage.