Teaching kids to code - Bridging the gap between scientific research and practical experience.

Ramón Martín Huidobro Peltier

Supervised by Margit Pohl, Ao.Univ.Prof.Mag.Dr.

Abstract

After running an after-school activity to teach children to code by developing games on the computer for several years, previous and ongoing research in the field of teaching children as well as technologies to facilitate this is important.

This paper aims to compare scientific research in the above fields with the practical experience gathered.

Introduction

Since the introduction of higher-level computing research, getting it into the classroom has been the target of ongoing research.

7 years ago, we began an after-school activity to introduce children to programming through computer game development. This has run every year on a weekly basis since then.

When teaching children to code, a number of considerations different to those in higher education need to be taken [Papert, 1972].

Despite having no previous pedagogical training, we have gathered insights and adapted the activity based on the feedback and reception from the children and parents alike.

Moving forward, it's important to see what experts have done/are doing in order to face the challenges of teaching children not just programming, but also logical thinking.

In order to address this, this paper poses the following question and aims to answer it:

How is scientific research in teaching children and teaching children to code in particular observed through practical experience with running an after-school activity for children?

Overview of Established Scientific Research

Why teach children to code?

In the context of education, it is claimed that learning to code can provide skills that can be transferred to other areas of education, such as problem solving by developing heuristics, breaking problems down into smaller subproblems that can be resolved in a modular fashion, developing a foundation that helps in learning mathematics, understanding that there are solutions to problems that may not necessarily be best, but can be compared with others in terms of their costs/benefits, as well as making clear expressions of solutions that can be used in other contexts [Pea et al., 1987].

There has also been research showing that learning to interact with computers and programming them at a young age can help develop higher-order thinking. By learning to "debug" programs they've created, the former research has shown that children "learn to learn" [Clements et al., 1993].

There have been past claims that computers hinder the social interaction between children [Cordes et al.]. However, research demonstrated that the opposite of this is true. Solving problems using computers has been demonstrated to encourage children to work with their peers and garner positive reinforcement for this, as opposed to working on puzzles in real life [Clements et al., 2002].

Overview of Programming Teaching History

In his 1972 work on teaching children thinking, Papert ponders over introducing computer science concepts to grade school children. In this paper, he proposes the LOGO programming language, a set of tools to draw patterns using a set of instruction combinations. Papert and his team at the Massachusetts Institute of Technology (MIT) tested this and other approaches with children of around 12 years old, and found the following [Papert, 1972]:

> Within three months these children could write programs to play games… soon after that they worked on programs to generate random sentences.

Although the research was made at the time and advocated the teaching of programming in schools, the uptake has been slow. Brunner and Di Angelo, in their work on teaching Information Technology (IT) in Austria, study a case of teaching IT at the ninth-year level. This takes place after a shift in teaching focus over to attaining competences. As defined by Weinert [Weinert, 1999]:

> …the concept of competence refers to an individually or interindividually available collection of prerequisites for successful action in meaningful task domains. In the following, the individual aspect, which dominates the social and behavioral sciences literature, will

> be accentuated. An important reason for this focus on the individual perspective is the fact that schools are the primary 'educational settings' over the course of individual development. Each single student must acquire necessary competencies and required education as preparation for his or her later social and professional life.

This is also the definition used in Austria at the time of their work, and made into a standard in the EU and Austria in 2008 and 2011, respectively [Brunner et al.]. The aim of teaching specific competences to students is to eliminate the challenge of the "passive role" a student plays [Brunner et al., 2014].

Cognitive Science

Investigating the intricacies of introducing children to programming concepts requires getting into specific concepts of cognitive science, which is the study of how intelligent entities learn, how they are introduced to new concepts and understand them [Pritchard, 2013].

Jean Piaget, one of the leading researchers in cognitive science of the early twentieth century, was one of the first proponents of the concept known as constructivism. Pritchard defines this as follows:

> Constructivists view learning as the result of mental construction. That is, learning takes place when new information is built into and added onto an individual's current structure of knowledge,understanding and skills. We learn best when we actively construct our own understanding.

In his research, Piaget proposes that a child's learning is divided into stages [Ackermann, 2001] that occur as the child grows, until around the age of 11, when children can begin to think in an abstract form [Pritchard, 2013]. Before then, learning is more focused on motor skills, empathy, as well as physical logic [Pritchard, 2013].

Based on Piaget's research, Ackermann infers three points:

> 1. *Teaching is always indirect.* Kids don't just take in what's being said. Instead, they interpret what they hear in the light of their own knowledge and experience. They transform the input.
> 2. *The transmission model, or conduit metaphor, of human communication won't do.* To Piaget, knowledge is not information to be delivered at one end, and encoded, memorized, retrieved, and applied at the other end. Instead, knowledge is experience that is acquired through interaction with the world, people and things.
> 3. *A theory of learning that ignores resistances to learning misses the point.* Piaget shows that indeed kids have good reasons not

> to abandon their views in the light of external perturbations. Conceptual change has almost a life of its own.

Based on Piaget's research above, it's possible to infer that teaching children is one thing, but children learning what one teaches them is another.

Seymour Papert, who worked with Piaget during the mid-20th century [Ackermann, 2001], expanded on the above concept and came up with constructionism. In his own words [Papert et al., 1991]:

> Constructionism–the N word as opposed to the V word–shares constructivism's connotation of learning as "building knowledge structures" irrespective of the circumstances of the learning. It then adds the idea that this happens especially felicitously in a context where the learner is consciously engaged in a constructing a public entity, whether it's a sand castle on the beach or a theory of the universe.

In other words, constructionism adds onto constructivism by stating the importance of context in learning. This can be not only dependent on time and place, but also on the media through which children learn.

One other difference between the two schools of thought, as related by Ackermann, is that children, according to Piaget, learn to detach themselves from concrete problems and their physical manifestations, and instead gain knowledge by thinking in abstract terms. Papert, on the other hand, argues that in order to best learn, children learn best by "diving into" problems and gaining knowledge from those concrete experiences that they can then apply to other problems.

Papert argues that it's important to set aside assumptions about learning with children:

> While we can "see" that children learn words, it is not quite as easy to see that they are learning mathematics at a similar or greater rate. But this is precisely what has been shown by Piaget's life-long study of the genesis of knowledge in children. One of the more subtle consequences of his discoveries is the revelation that adults fail to appreciate the extent and the nature of what children are learning, because knowledge structures we take for granted have rendered much of that learning invisible.

Working in groups

In professional software development, arguments are made for and against working in pairs, otherwise known as pair programming. This involves two software developers sitting at the same computer, one doing the actual programming and the other making observations. This is often referred to as the driver and

navigator, respectively. In their work on the topic, Cockburn et al. present the following advantages of pair programming:

- many mistakes get caught as they are being typed in rather than in QA test or in the field (continuous code reviews);
- the end defect content is statistically lower (continuous code reviews);
- the designs are better and code length shorter (ongoing brainstorming and pair relaying);
- the team solves problems faster (pair relaying);
- the people learn significantly more, about the system and about software development (line- of-sight learning);
- the project ends up with multiple people understanding each piece of the system;
- the people learn to work together and talk more often together, giving better information flow and team dynamics;
- people enjoy their work more.

Game-based learning

The relationship between programming and computer games has been actively researched. In their work, Johnson et al. quote Thomas W. Malone as follows:

> In some senses, computer programming itself is one of the best computer games of all. … The "player" gets frequent performance feedback (that is, in fact, often tantalizingly misleading about the nearness of the goal). … Self-esteem is crucially involved in the game, and there is probably the occasional emotional or fantasy aspects involved in controlling so completely, yet often so ineffectively, the behavior of this responsive entity. Finally the process of debugging a program is perhaps unmatched in its ability to raise expectations about how the program will work, only to have the expectations surprisingly disappointed in ways that reveal the true underlying structure of the program.

Johnson et al. talk about the positive effects on children of "Educational Games". Regardless of whether they have an educational or entertainment-based focus, games been shown to have value towards learning. This is especially important, given games are widespread in countries like the United States [Johnson et. al, 2016].

The positive effect of these games in education include [Johnson et al., 2016]:

- Cognitive development. This can be measured by the time necessary to accomplish a task or the accuracy thereof.
- Increased motivation.

- Skills that can be transferred to other areas, such as communication, leadership or self-efficacy.
- Improved behaviour or attitude.

Computational Thinking

At a first glance, computational thinking is, as implied by its name, thinking like a computer. As outlined by Wing [Wing, 2006]:

> Computational thinking involves solving problems, designing systems, and understanding human behavior, by drawing on the concepts fundamental to computer science. Computational thinking includes a range of mental tools that reflect the breadth of the field of computer science.

One emphasis Wing makes is that computational thinking is "a way that humans, not computers, think". This pushes the notion that it's humans who dictate the way of computing. Given that software is abstract, it's important not to be constrained by the current paradigms.

Given Wing's emphasis on the importance of teaching people, not just children, computational thinking, there have been tools developed or being developed to help introduce children to computational thinking. One such example is Light-Bot, a game suggested by the Code.org initiative. The purpose of the game is to program a robot with simple commands to reach a goal. As evaluated [Gouws et al., 2013], it helps children understand abstractions in robotic movement.

Computational thinking is helping scientists in biology, economics, and many others expand their fields, and this will only continue to expand [Wing, 2006].

Overview of technologies used or developed by researchers

LOGO

LOGO is a project designed in the late 1960's with teaching children to interact with computers in mind.

By using a series of commands to start drawing, stop drawing, move forward or backward and rotating, children can build programs to build drawings. This is represented to the child as a turtle that moves around the screen, as described further by Papert et al. [Papert et al., 1971]:

> At any time the turtle is at a particular place and facing in a particular direction. The place and direction together are the turtle's geometric state.

According to them, their goal to make this approachable to children is achieved using a carefully designed turtle language [Papert et al., 1971]:

> For example, we can type LEFT 90 on the console keyboard and thereby cause the turtle to rotate 90°.

This, like games, has the advantage of representing results immediately.

Scratch

Scratch is a platform developed at MIT intended for addressing the needs of children aged 8 to 16 [Maloney et al., 2010]. Careful consideration took place in creating the user interface for children:

- A single-window user interface. By not having the workflow split between, for example, a text editor and the terminal, children have only one place where the game development take place and learn not to manage tools but first focus on the primary objective, which is logic.
- A "block"-based user interface. Instead of writing code, children drag around and connect "blocks" of code, similar to LEGO.
- Having the code be live. By having the code be "live" the children can click on a block of code and it will instantly run. This allows debugging to be instantaneous.

With the goals above, Maloney et al. have a solution where they avoid error messages as much as possible [Maloney et al., 2010]:

> When people play with LEGO® bricks, they do not encounter error messages. Parts stick together only in certain ways, and it is easier to get things right than wrong. The brick shapes suggest what is possible, and experimentation and experience teaches what works. Similarly, Scratch has no error messages. Syntax errors are eliminated because, like LEGO® bricks, blocks fit together only in ways that make sense. But Scratch also strives to eliminate run time errors by making all blocks be failsoft. Rather than failing with an error message, every block attempts to do something sensible even when presented with out-of-range inputs.

Minecraft

At the time of writing, Minecraft remains one of the most popular games played by children. Originally, Minecraft is a video game played in a virtual sandbox, where players are incentivised to build and create their own worlds and experiences.

As an interesting extension of the last section on LOGO, community modifications to Minecraft have allowed for programming environments to exist within the game.

One example comes from Wilkinson et al. [Wilkinson et al., 2013], who used ComputerCraft as a tool to run a workshop where children were introduced to programming inside the Minecraft game, using the LUA scripting language. Their workshop yielded the findings that children were more active and encour-

aged to work in an environment that not only were already familiar with, but also enjoyed.

Robots and Internet of Things

In their research, Papert et al. [Papert et al., 1971] were already proposing robots as a way to teach children computer programming concepts in 1971.

In recent years, robots and the Internet of Things have become ubiquitous in teaching environments. Computers such as the Raspberry Pi have in recent years become a staple in UK classrooms and workshops, opening the doors to hardware programming learning.

In their investigation testing the effectiveness of "tangible" hardware programming against "graphical" software-based programming, Zhu et al. concluded that the children had more fun, had a higher interest, and felt more confident in their coding abilities having used hardware-based interfaces. Interestingly, at the same time, they found that the children agreed that "coding is hard" at a higher rate when working with hardware.

Sonic Pi

Sonic Pi is a more recent and different approach to introduce children to computer programming.

In their research when creating Sonic Pi, Aaron et al. used Papert's work on LOGO as inspiration to make a concept accessible to children. Using domain-specific language (DSL) techniques offered by modern programming languages, they created Sonic Pi, which is a DSL as well as a development environment for creating music.

As inspired by the name, Sonic Pi is meant to be used on Raspberry Pi computers, but is available on other platforms as well.

After having conducted a trial of the platform in a classroom, Aaron et al. conclude that Sonic Pi is an effective way to quickly introduce children to these concepts.

Methodology

Our practical experience took the form of an after-school activity, titled "Computer Game Programming".

Aim

The aim of Computer Game Programming is to introduce children to the basic concepts of programming by creating games.

The reason games were selected as the medium for coding is because of the straightforward nature of games, their appeal to children, as well as the instant gratification that games offer when features are added or bugs are fixed.

Structure

As an after-school activity, Computer Game Programming is intended to be a mixture of a club, workshop and dedicated lesson. There are two groups of children the activity is run with. Primary students code 2D games, whereas secondary students code 3D games. Although a majority of time is dedicated introducing the children to concepts, they also are encouraged to play around with what they've built and discover things this way.

The activity takes place once a week, as implied, after school, and for one hour. Over the session, the group will recap on what took place the previous week briefly in the form of asking the children to put up their hands if they know the answer. For example, a week after learning about coordinate systems, a few minutes would be dedicated to recapping how these worked with examples, such as showing an asteroid displayed at different sections of the game window. One by one, the kids would be asked to tell the group which asteroid had which coordinates.

The activity is divided into modules over the year. Specifically, working on one different game at the same time for each set of concepts. This has been tweaked slightly over the years, as will be described in detail in a later section.

Technologies

This section will comprise a overview of each technology used, how these have changed over the years, and why these were selected as a medium to teach children the core concepts of programming:

Coding games in a text editor and running them in the terminal

This was done using the `gosu` games framework and writing games using the Ruby programming language in the `Atom` text editor.

Advantages:

- Using this programming environment is identical to how professional software developers work at the time of writing. By using the same tools, the children are not only introduced to programming, but also to central computer science concepts, such as working with Unix/Linux.
- Given that the `gosu` framework provides window and image drawing as well as keyboard event catching, children need to program all their game logic by hand. This includes physics engines (gravity), sprite drawing, displaying windows, etc.

Disadvantages:

- If the purpose is to gently introduce children to programming, having them write dozens of lines of code before they can play around with their game configuration can be overwhelming. Some children would have to follow along by copying what we did, never really understanding why we were doing what we did and getting frustrated in the process.
- As a result from the last point, few concepts stuck between weeks, leading to the need for lots of repetition.
- Having to type out code by hand, especially lots of code, can lead to typing errors. It was tricky to truly drive home how important it is, for example, for variable name spelling to be used consistently in a script, leading otherwise to errors.
- Debugging Ruby errors from stack traces was often frustrating, leading to a lack of motivation for the kids to figure out what went wrong with their code.
- The syntax of Ruby, using the keyword `end` to denote the end of a block (that is, an `if` statement, loop, class or method declaration, for example) led to situations where a child might have added too many `end` keywords or worse, forgotten to add some. As described above, the children were oftentimes not motivated to read the error messages, which meant that a lot of time was spent by the instructor trying to find this problem and solve it.
- Platform limitation: When attempted, this programming environment was only usable on Mac OS X, due to the implementation of Ruby on Windows not being flexible enough, and MP3's for music not being possible on Linux. This was not a big issue in the context of the activity itself. However, if the children wanted to work on their games at home, this would also require a tricky installation.

Stencyl (http://www.stencyl.com/)

Stencyl is a commercial program on Mac, Linux and Windows. It uses a drag-and-drop code block user interface, and a graphical level / scene editor:

Advantages:

- By employing a graphics-based interface (as shown in Figure 1) instead of writing code, onboarding the children to the basic concepts, like `if` statements or loops, was a lot faster than when using the previous solution. This also eliminates the issues brought up by typing errors, as well as `end` keywords, since this is all taken care of for the children.
- Stencyl comes with a built-in game engine. This includes, but isn't limited to, gravity, character animation, collision detection, and much more.

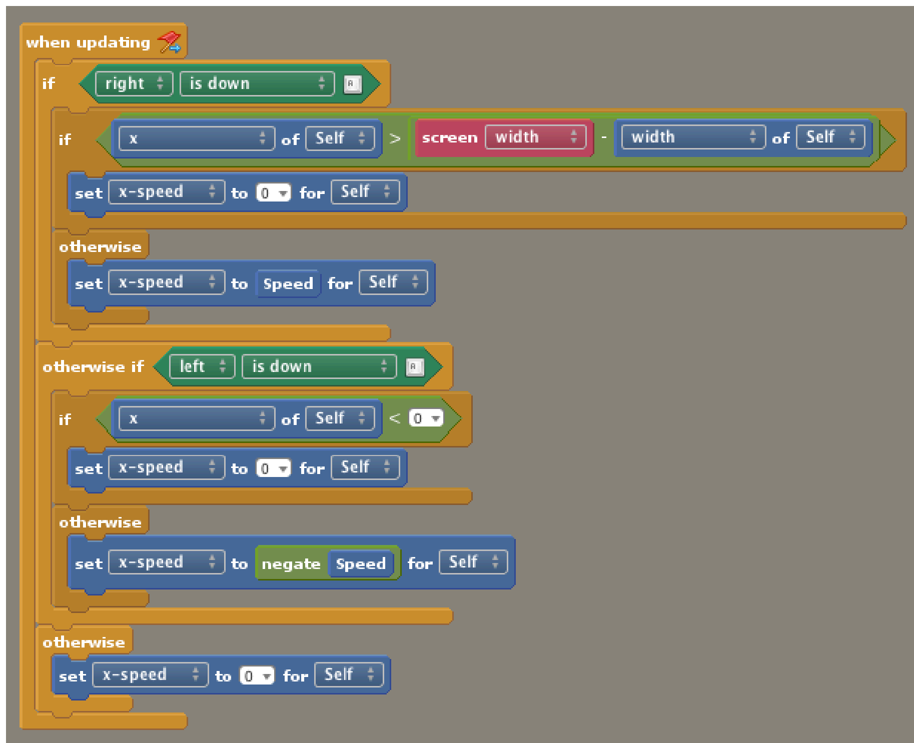

Figure 1: Stencyl block code editor

Removing these means that the group can focus on learning the basics without being too limited.

- By using a graphical scene editor, children are immediately privy to how the coordinate system works, as they can see the coordinates of characters change when dragged around. This can be seen in Figure 2.
- Unlike the previous solution, making games with Stencyl is confined to a single-window environment, as shown in Figure 1. Doing this avoids the confusion of switching between the text editor and the terminal.

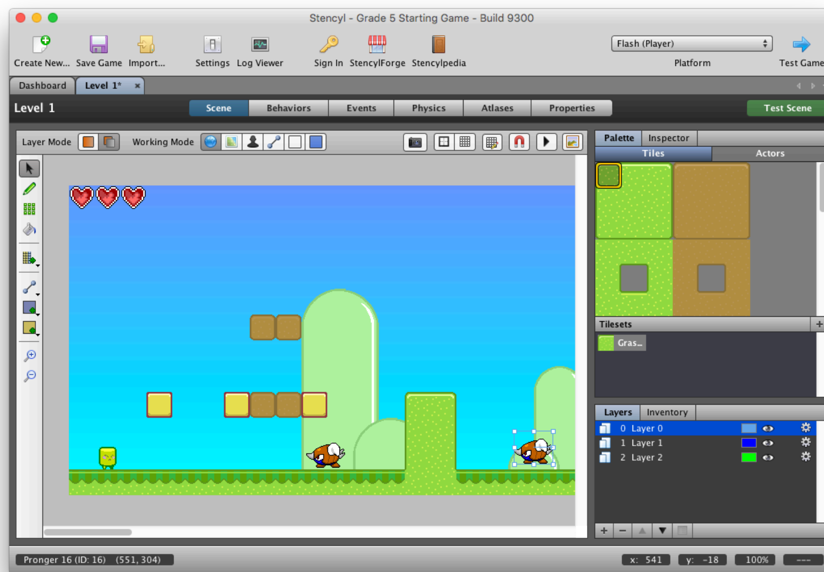

Figure 2: Stencyl Scene Editor

Disadvantages:

- Stencyl's error messages can sometimes be obscure. Being built on top of Java, when say an input is missing or invalid in one of the blocks, the error thrown has on several occasions confused the children.
- Some features are notably missing or not as clear to children. Things such as implementing a menu screen or credits scene need to be done directly with Stencyl, which isn't as immediately obvious. This disadvantage does have the hidden benefit of encouraging the children to work with these limitations, however.

Unity 3D

Unity 3D is, as implied by its name, an engine for programming and designing 3D games as well as 2D ones. It's widely used in the independent game developer industry. Its user interface can be seen in Figure 3.

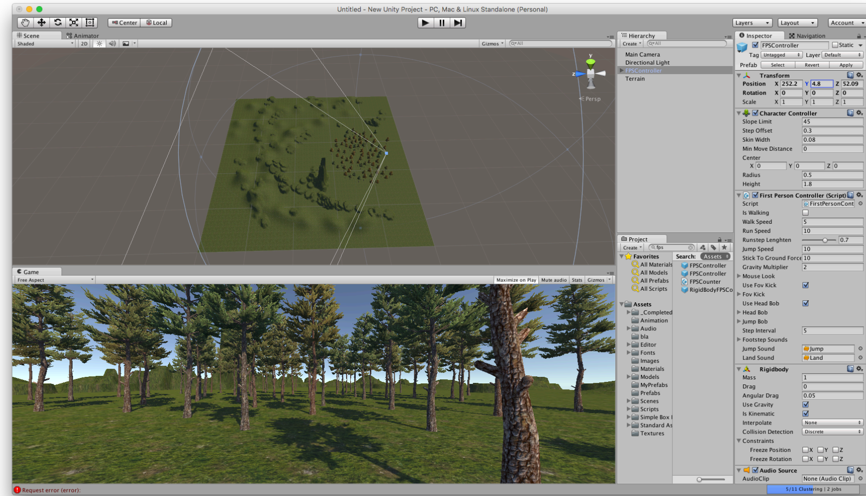

Figure 3: Unity 3D

Advantages:

- The first advantage is immediately apparent. The children are generally thrilled to move from a 2D game design environment to a fully 3D one.
- As with the solution of using Ruby, this is a fully professional development environment. The key difference here, however, is that this is an integrated development environment, or IDE, meant for game development.
- Unity allows to write the scripts used in-game in a variety of programming languages. I decided to go for Javascript, since at the time of writing it's one of the more widely-used programming language.
- Unity 3D has a thriving online community, with lots of active development going into the engine, as well as open source code, in-game assets (such as 3D character or layout models, sounds, music as well as code), and more.
- By having a scene editor similar to Stencyl's, the children can be gently introduced to the game development environment and learn coding at the same time without becoming frustrated.

Disadvantages:

- Having the children be introduced to a large IDE at the same time as learning the concepts of code can be tricky. At the beginning, they can

be overwhelmed and some parts can be forgotten between weeks.

- Having a resource-intensive 3D engine can make development a little slow on older, less-powerful machines. Since the kids most of the time bring their own laptops, this can be frustrating for them when compiling the game can take a few minutes whilst for others it's a matter of seconds.
- Some of the children were distracted from the programming aspect of 3D game development and found themselves more excited about 3D modelling. This isn't necessarily a disadvantage, because exposing children to all aspects of computing is important. However, this is not the purpose of the after-school activity.

Changes over the years

The program of the activity, that is, the order in which the concepts are introduced to the children has changed, depending on both the technology used, as well as the difficulties experienced the previous year.

Specifically, as outlined in the previous section, the use of the Ruby for coding required building a lot of the game engine from scratch. That is, drawing characters on screen, input recognition, physics, etc.

It was therefore decided on the first year to begin by making a "Text Adventure". Text adventures, also known as interactive fiction computer games, more common in the 80s, are games displayed only in text, normally in a console terminal, where commands are written to the game in plain text. A classical example of this is Zork (https://en.wikipedia.org/wiki/Zork).

The advantage of the above is that the children would not need to concern themselves with complicated concepts like coordinates, graphics, animation, physics, etc. and could use their imagination to create any adventure they want to.

The issue, however, is that this was often met by ambivalence by the children, who wanted to see more exciting graphics. There was also a language issue, this being a multicultural school, having children write in their own spoken language meant that these were difficult to debug, especially given that this was their first foray into programming in general.

To combat this, the next year was started with a pre-made engine in Stencyl. In this example, a 2D platforming game engine was provided with some already-made components, such as the main character, enemies, tiles, etc. programmed and included in a few levels. The last level, however, is incomplete, encouraging the children to finish creating the level and set them off.

The above is purposefully done to encourage the children to try things out and at the same time have fun. They also are very gently introduced to the concepts of programming by gently lifting the veil over how a game works by adding components to an existing game. This is done by adding new enemies, a second

player for multiplayer gameplay, items to collect, displaying their score, adding new movement behaviors for enemies, such as flying, and so on.

After this first game, the children are then to design a space-faring game from scratch. An example on Figure 4 is using the internet character Nyan Cat.

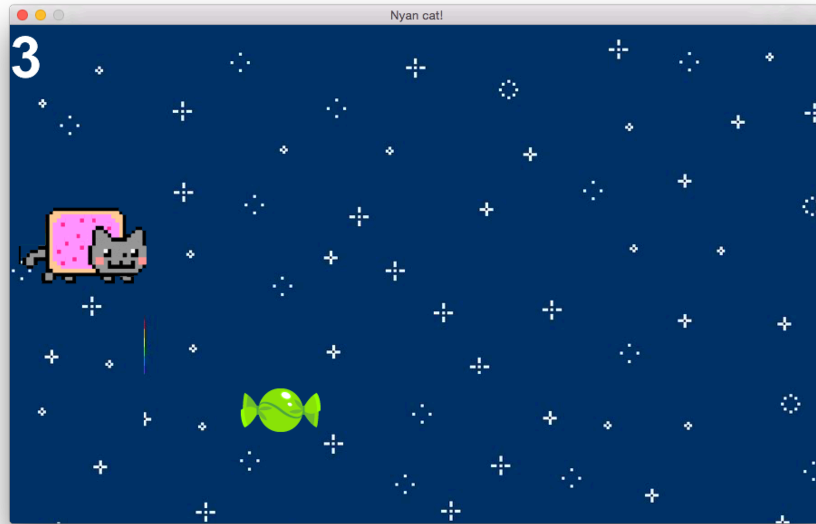

Figure 4: Nyan Cat in Space

Since the concept of the game is simpler than that of a 2D platforming game (that is, no (or very little) physics are needed to be coded in space!), the children are instructed to create the basic up and down movement of the player character, the movement of the asteroids they're flying past, and the collision detection of these. With these three components, the children have created a full-fledged game in 2-3 sessions. They are then encouraged to explore how they can add more components, such as enemies that move in a zig-zag pattern.

In one year, there was a third game made as a group, at the height of the popularity of the game "Flappy Bird" (https://en.wikipedia.org/wiki/Flappy_Bird), where the children are encouraged to re-create the game from scratch once again and add their own twist to it. This encourages the children to write their own physics engine, as well as come up with twists to a game they knew well at the time. This was successful, albeit time consuming, which ate into time that was perhaps better spent on the final game.

The final game the children produce in the year is a game of their own making. Given the skills they've picked up over the year, this has been a good starting skillset to begin making their own creation.

Discussion

From the start, it has been the goal of the activity to let children build their own understanding of computer logic and how this works. This ties in with the bases of both Piaget's constructivism and Papert's constructionism.

By starting from a pre-built game and working its way to encouraging the students to build their own game from scratch allows them to pick up the intricacies of programming in a piecemeal fashion.

Addressing the responses and needs from the children has led to changes in how the information is delivered to them. For example, the blog posts were introduced to allow the children who worked faster to work at their own pace, allowing the instructors to help those who needed it.

However, constructionism argues that the context in which the children receive the information is key.

This begs the question: Why learning to program games?

There are quite a few options for things children could learn to code. As explained above, there's LOGO, Minecraft modding, music with Sonic Pi, and others. Other, larger companies are also trying to address this. Apple Inc., for example, has introduced "Swift Playgrounds": an Integrated Development Environment (IDE) built from the ground up for teaching children to code using the iPad is wholly different from using a computer. Foregoing the computer altogether for a tablet, puts the new aptitude in a context more familiar to them and makes it easier to transfer the information to them, as argued by Papert.

Going back to the previous question, however, a few arguments come up as to why games were chosen.

Instant gratification

Being able to code something up and immediately test and/or debug it is essential in learning to code, as argued by Malone [Johnson et al., 2016]. This is especially prevalent when making games. When you add a feature or fix a bug, you can quickly see the fruit of your work. If you introduce a bug, this is also quickly visible.

Something noteworthy about bugs, however, is that there are often amusing results. Children are especially receptive to breaking things. This is not meant in a negative sense, however. When creating a game where you needed to dodge enemies to get to the goal, it's very common for the kids to try and push the limits of what's possible. For example, placing 999,999 (or how many times they feel like pressing the `9` key on the keyboard) enemies on the screen would cause their computers to hang or even freeze. Worst still, the game would be almost unplayable. After getting a few good laughs out of it, however, the children would understand that not only is it key to good game design to have

a playable concept that isn't unfairly difficult, but also how computers process this.

Diverse game genres

Even though it's unlikely that the entire group of children will be satisfied with the game provided and/or proposed during the activity, different types that on the one hand are extensible so that the children can use the aesthetics (that is, graphics and sound) of their choice, but also in gameplay are proposed. This way, different tastes can be accommodated and the majority of children are pleased with the games they create.

However, teaching games also has its drawbacks. Most notably is that of managing expectations. Something encountered over two first year or two of teaching kids to code games is the tricky realization that it would be a lot of work. There have been children that stopped showing up to the activity after realizing that it wasn't as simple as they believed it would be. Having a pre-made example and asset set, along with ongoing conversations about time and financial resources has helped combat, however.

In short, having game programming be the medium through the children was selected as the most effective, as it's a popular medium for children and has space for self-expression and fun during development.

With regards to the specific technology used to teach the children, Stencyl's similarities to Scratch are very apparent (given that Scratch is open sourced with the GPL v2 license (https://github.com/LLK/scratch-flash) this technically isn't an issue, but not the purpose of this research in any case). So why not used Scratch to teach the children, especially when the school the activity was offered already had Scratch as part of the curriculum?

Even though they're very similar by look, Stencyl offers extensions to Scratch, such as mobile platform support, a Java-based background, and more. This, coupled with the already-familiar look, make the transition smooth into making sophisticated games. This again ties in with the importance expressed by constructionism of context.

Although there have been multiple changes to the structure of the activity, the children retention rate has usually been consistent. The children are entitled to drop the activity at any time during the school year, and out of a group of twenty, there's been an average of 3 dropouts. These are normally replaced with new students who are brought up to speed with the rest of the group. Some children drop out quickly when realizing that the activity would involve logical or mathematical thinking and turned off by this. As written by Papert:

> It is deeply embedded in our culture that the appreciation of mathematical beauty and the experience of mathematical pleasure are accessible only to a minority, perhaps a very small minority, of the human race.

This is not, however, to say that this is the only reason this happens. Some children are not interested anymore or don't react well to the teaching style offered in the activity.

When it comes to helping each other out, the children tend to have a preference to work in groups, or in pairs. This also, as noted by Fung, helped foster their learning. It has happened in the past that for the sake of getting the activity running as smoothly as possible that the team would solve problems for the children so that they're not stuck and/or frustrated and make sure that everyone is moving forward. The major problem with this is that the children were no longer thinking on their own and tended to just wait for an instructor to help them and until then didn't advance when an issue would come up.

To combat the above, we came up with the idea to have the children who were moving along more quickly to help those struggling. This freed up time for the instructors to help those truly struggling, and gave those with more time something to do. A number of noteworthy things came from this:

- The struggling children were no longer waiting for instructors to come over and help them. They would take their laptops to, or ask the kids who are finished to come over and help them with their bugs.
- By having the children who were finished read the code of others, they were observing the way the other children thought logically. Being exposed to the code of others helps share perspectives and shows different ways to solve similar problems [Skorkin, 2010]. This also leads to discussions on how to do this, which the children often engaged in.
- By the end of the year, as previously mentioned, the children were encouraged to work on their own projects. One other liberty they were given is to work in pairs or groups. After we introduced this idea to help each other out, we found that the children on average tended to prefer working in groups.

Conclusions and future work

When the activity was initially designed, it was unfortunately done so with little to no research into how to teach children from a scientific point of view, but rather an intuitive one. Doing this in a dynamic way and responding to feedback, however, has led the team to similar, if not the same conclusions as those presented during this research of this paper. Adapting to the needs of the children has helped lead the team down this path and the research has also helped to explain how to scientifically derive this.

This being said, there are a number of takeaways from having compared and contrasted the activity with established scientific research that can be implemented in future variations of the activity.

For starters, it would be important to spend more time understanding the im-

portance of computational thinking. Last year, we had the children play the game Light-Bot. This was due to a technical difficulty which meant we could not start the activity as intended. However, we found that the children not only enjoyed the game but had a much easier time understanding concepts such as procedures.

It also would be good to place a higher priority on understanding how programs work. We would like to spend more time in the command line, not making games but rather making a game out of programming. For example, as described by Papert, it would be interesting to try having the children generate sentences.

Paying more attention to the responses of children as they learn is something we'd like to emphasize moving forward. What they pick up is equally as important as that which they don't as easily. Furthermore, the panic that sets in when a child struggles or becomes frustrated is something we never handled all too well. However, as a software developer, this is a state one finds themselves often in, and the payoff of solving a problem is big. Furthermore, this helps them learn how to tackle future problems. As previously cited from Clements et al., it's important that children "learn to learn" [Clements et al., 1993].

Looking forward, Papert ponders:

> The computer by itself cannot change the existing institutional assumptions that separate scientist from educator, technologist from humanist. Nor can it change assumptions about whether science for the people is a matter of packaging and delivery or a proper area of serious research. To do any of these things will require deliberate action of a kind that could, in principle, have happened in the past, before computers existed. But it did not happen. The computer has raised the stakes both for our inaction and our action. For those who would like to see change, the price of inaction will be to see the least desirable features of the status quo exaggerated and even more firmly entrenched.

Although it has not been possible to continue the after-school activity this year, it has been possible to offer individual tutoring that follows the same structure. It has been possible to carry through a lot of the knowledge gathered, as well as allow the child to work together with the instructor to solve problems together nonstop. Although the concept learning together with their peers is lost, I try to behave as one at times.

References

Gouws, Lindsey Ann, Karen Bradshaw, and Peter Wentworth. "Computational thinking in educational activities: an evaluation of the educational game light-bot." Proceedings of the 18th ACM conference on Innovation and technology in computer science education. ACM, 2013.

Johnson, Chris, et al. "Game Development for Computer Science Education." Proceedings of the 2016 ITiCSE Working Group Reports. ACM, 2016.

Weinert, Franz E. Concepts of competence. OFS; US Department of education, National center for education statistics (NCES), 1999.

Papert, Seymour. "Teaching children thinking." Programmed Learning and Educational Technology 9.5 (1972): 245-255.

Papert, Seymour, and Idit Harel. "Situating constructionism." Constructionism36.2 (1991): 1-11.

Clements, Douglas H., and Julie Sarama. "The role of technology in early childhood learning." Teaching Children Mathematics 8.6 (2002): 340.

Bers, Marina Umaschi, and Michael S. Horn. "Tangible Programming in Early Childhood." High-tech Tots: Childhood in a Digital World 49 (2010): 49-70.

Clements, Douglas H., and Bonnie K. Nastasi. "Electronic media and early childhood education." Handbook of research on the education of young children (1993): 251-275.

Pea, Roy D., and Karen Sheingold. Mirrors of Minds: Patterns of Experience in Educational Computing. Ablex Publishing Corporation, 355 Chestnut Street, Norwood, NJ 07648, 1987.

Brunner, Markus, and Monika Di Angelo. "Competence Orientation in Vocational Schools–The Case of Industrial Information Technology in Austria." International Conference on Informatics in Schools: Situation, Evolution, and Perspectives. Springer International Publishing, 2014.

Papert, Seymour, and Cynthia Solomon. "Twenty things to do with a computer." (1971).

Gouws, Lindsey Ann, Karen Bradshaw, and Peter Wentworth. "Computational thinking in educational activities: an evaluation of the educational game lightbot." Proceedings of the 18th ACM conference on Innovation and technology in computer science education. ACM, 2013.

Ericson, Barbara, and Tom McKlin. "Effective and sustainable computing summer camps." Proceedings of the 43rd ACM technical symposium on Computer Science Education. ACM, 2012.

Winslow, Leon E. "Programming Pedagogy—a Psychological Overview." ACM SIGCSE Bulletin 28.3 (1996): 17-22.

Pritchard, Alan. Ways of learning: Learning theories and learning styles in the classroom. Routledge, 2013.

Ackermann, Edith. "Piaget's constructivism, Papert's constructionism: What's the difference." Future of learning group publication 5.3 (2001): 438.

Skorkin, Alan. “Why I Love Reading Other People’s Code And You Should Too.” SKORKS, 19 May 2010, www.skorks.com/2010/05/why-i-love-reading-other-peoples-code-and-you-should-too/.

Cordes, Colleen, and Edward Miller. “Fool’s Gold: A Critical Look at Computers in Childhood.” (2000).

Maloney, John, et al. “The scratch programming language and environment.” ACM Transactions on Computing Education (TOCE) 10.4 (2010): 16.

Inc., Apple. “Swift Playgrounds.” Swift Playgrounds - Apple Developer. N.p., n.d. Web. 13 Mar. 2017.

Papert, Seymour. Mindstorms: Children, computers, and powerful ideas. Basic Books, Inc., 1980.

Cockburn, Alistair, and Laurie Williams. “The costs and benefits of pair programming.” Extreme programming examined (2000): 223-247.

Fung, Dennis. “Promoting critical thinking through effective group work: A teaching intervention for Hong Kong primary school students.” International Journal of Educational Research 66 (2014): 45-62.

Wilkinson, Brett, Neville Williams, and Patrick Armstrong. “Improving Student Understanding, Application and Synthesis of Computer Programming Concepts with Minecraft.” The European Conference on Technology in the Classroom. 2013.

Aaron, Samuel, and Alan F. Blackwell. “From sonic Pi to overtone: creative musical experiences with domain-specific and functional languages.” Proceedings of the first ACM SIGPLAN workshop on Functional art, music, modeling & design. ACM, 2013.

Zhu, Kening, et al. “How different input and output modalities support coding as a problem-solving process for children.” Proceedings of the The 15th International Conference on Interaction Design and Children. ACM, 2016.

Wing, Jeannette M. “Computational thinking.” Communications of the ACM 49.3 (2006): 33-35.