Scalable Game Design: A Strategy to Bring Systemic Computer Science Education to Schools through Game Design and Simulation Creation

ALEXANDER REPENNING, University of Colorado Boulder
DAVID C. WEBB, University of Colorado Boulder
KYU HAN KOH, University of Colorado Boulder
HILARIE NICKERSON, University of Colorado Boulder
SUSAN MILLER, University of Colorado Boulder
CATHERINE BRAND, University of Colorado Boulder
IAN HER MANY HORSES, University of Colorado Boulder
ASHOK BASAWAPATNA, University of Colorado Boulder
FRED GLUCK, University of Colorado Boulder
RYAN GROVER, University of Colorado Boulder
KRIS GUTIERREZ, University of California, Berkeley
NADIA REPENNING, AgentSheets Inc.

An educated citizenry that participates in and contributes to Science Technology Engineering and Mathematics innovation in the 21st century will require broad literacy and skills in computer science. School systems will need to give increased attention to opportunities for students to engage in computational thinking and ways to promote a deeper understanding of how technologies and software are used as design tools. However, K-12 students in the United States are facing a pipeline for computer science education that is broken. In response to this problem we have developed the Scalable Game Design curriculum based on a strategy to integrate computer science education into the regular school curriculum. This strategy includes opportunities for students to design and program games and Science Technology Engineering and Mathematics simulations. An approach called Computational Thinking Pattern Analysis has been developed to measure and correlate computational thinking skills relevant to game design and simulations. Results from a study with over 10,000 students demonstrate rapid adoption of this curriculum by teachers from multiple disciplines, high student motivation, high levels of participation by women and interest regardless of demographic background.

Categories and Subject Descriptors: **K.3.1 [Computer Uses in Education]:** Collaborative learning; **K.3.2 [Computer and Information Science Education]:** Computer science education, Literacy, Self-assessment.

General Terms: computational thinking, computational thinking patterns, student motivation.

Additional Key Words and Phrases: STEM, game design, programming, Latent Semantic Analyses, elementary schools, middle schools.

ACM Reference Format:

Alexander Repenning, David C. Webb, Kyu Han Koh, Hilarie Nickerson, Susan Miller, Cathy Brand, Ian Her Many Horses, Ashok Basawapatna, Fred Gluck, Ryan Grover, Kris Gutierrez and Nadia Repenning. 2014. Scalable Game Design: A Strategy to Bring Systemic Computer Science Education to Schools through Game Design and Simulation Creation. ACM Trans. Comput. Educ.

1 INTRODUCTION

School curriculum in the United States is in the midst of an overhaul unlike any witnessed in previous generations. The widespread adoption of the Common Core Standards in the United States demonstrates a sea change among federal, state and local education policy toward the fulfillment of a national curriculum and assessment system for mathematics and literacy with science soon to follow. However, in spite of the calls for 21st century college and career readiness underlying these initiatives, computer science education is notably absent from any of the US curriculum standards for math or science [cf. NGA Center for Best Practices and CCSSO 2010].

In short, curriculum initiatives in the US supported by federal and state education policies have moved forward but have left computer science education behind.

At present, there is no unified model for computer science education in the United States [Gal-Ezer and Stephenson 2014]. From elementary through secondary schooling, there are few opportunities for young students to innovate, design, and produce using computer programming. The first glimmer of opportunity to explore programming often appears in after-school programs or summer camps designed for upper-elementary and middle grades students (ages 10 to 14 years old). However, these enrichment opportunities are more of the exception than the rule. These programs are neither systemic nor compulsory, and so only a relatively small percentage of children who have shown some initiative or interest are provided access to these piecemeal pathways into computer science. Most students complete their compulsory educations as users, not designers, of computer products. This leads to the prevailing perception that technology is a tool for computation, game play, word processing, or social networking. Most state curriculum standards have not mandated student design of computer programs or, more specifically, computer programming. Furthermore, most universities have not developed licensure programs for computer science education or related instructional methods courses to prepare future generations of teachers. As a result, there are fewer students entering Computer Science within our universities. The problem is well known: the computer science education pipeline in the United States is broken [Wilson et al. 2010].

Even though job growth in the IT sector has continued to increase, "the current reality is that most US students do not have the opportunity to take even one academic computing course in high school" [Cuny 2012]. US computer and technology companies have responded to this lack of appropriately educated American workers by lobbying the federal government to allow more tech-credentialed citizens of other countries to work in the US. In the April 11, 2013 article, "Tech Firms Push to Hire More Workers from Abroad," the New York Times reported that there were 124,000 applications for the temporary H1-B visas for skilled workers available for 2014. In response to this demand, the US Citizenship and Immigration Services office was forced to set up a lottery to hand out the 85,000 available visas. Tech companies in the US have pushed hard for a fast-track green card application process for foreign math and science graduates of US universities and for a doubling of the number of visas awarded through the H1-B program.

The need for computer skills is not limited to potential tech industry workers. Physicians, architects, scientists and other skilled professionals use computers regularly in the workplace. Business people rely on software to run and advertise their businesses. Although these jobs do not require large amounts of programming, a facility with computers and an understanding of how they can be used to record and manipulate information, as well as solve industry- or profession-specific problems, is very useful to these workers. Even in the service industry, computers are ubiquitous: keeping track of money, advertising on the web and collecting information about customers. The 21st century economy requires most future workers to be able to use computers as part of their jobs [Stephenson and Wilson 2012].

The various programs designed to address the inadequate number of American high school and college graduates moving into the computer science field are due, in part, to the relationship between student interest and the development of expertise. At present, the de facto reliance on individual student interest in elective high school and college computer science classes has attracted a limited range of students to the technology profession. Recent documents published by both Google and Apple show

that businesses are predominantly white and male. Less than 20% of the tech positions in both companies are filled by females or non-Asian minorities.

Students from lower socioeconomic backgrounds are significantly underrepresented in the tech worker population. Broadening participation in computer science and Science, Technology, Engineering, and Mathematics (STEM) education would benefit female and racial/ethnic minority students who otherwise might never have considered studying this material. It is important to expose these students to computer science in school because they may not know anyone who works in a technical field and often have no idea what a computer scientist does. Schools with more economically and racially diverse student populations tend to focus on preparation for the annual standardized exams, which assess student progress. In addition, such schools are often short on money and staff to support elective classes like computer science [Margolis 2008].

To increase curricular coherence in K-12 computer science education, the Computer Science Teachers Association recently published a revised version of the *K-12 Computer Science Standards* [Seehorn et al. 2011]. Yet, in spite of these CSE standards, as noted by Gal-Ezer and Stephenson [2014],

> "Where computer science is concerned [in the US], it is fair to say that it has been largely ignored by the public education sector. Instead, responsibility for promoting and supporting the teaching of computer science in schools has fallen almost exclusively to individual researchers and professional associations."

The urgency of this situation has resulted in a number of new initiatives trying to change the public perception of computer science through assertive marketing. Code.org, for instance, has gained national attention through a video, watched on YouTube over 10 million times, that features computer science celebrities such as Bill Gates and Mark Zuckerberg. More recently, during the Computer Science Education week in December of 2014, Code.org launched a site called the Hour of Code featuring about 20 different hands-on programming tutorials including one based on Scalable Game Design. The Scalable Game Design tutorial was used by about a quarter million participants during that week.

Another program designed to address the opportunity to learn problem has been the "CS 10K" initiative, which promotes the development of new high school computing courses [e.g., Astrachan and Briggs 2012] and seeks to implement these programs into 10,000 high schools taught by 10,000 well-prepared teachers. CS 10K is supported by the National Science Foundation and many corporate sponsors.

The Scalable Game Design research group proposes a solution to this broken CSE pipeline problem: introduce all middle school students to computer science by teaching computational thinking through a curriculum that employs the AgentSheets and AgentCubes programming environments so that students can program a complete game or science simulation in a relatively brief period of time (e.g., 5-10 hours). Scalable Game Design is a curriculum that starts with motivating middle school students first through game design and then advances students to the design and use of STEM simulations by leveraging the computational thinking skills acquired from game design (see appendix). Let's break this statement apart and look at the rationale behind each piece of this solution.

The SGD project works with middle school students, as this approach offers the most productive opportunity for increasing the number of high school graduates with basic technology skills as well as increasing the number of students pursuing advanced education in science, technology, engineering and math [Grover, Pea and Cooper 2014]. Middle school students were chosen because they are old enough to

read fluently and work independently; yet these students are young enough that they have not yet picked an educational direction for their high school years. They still have time to make the choice to take optional computer science and technology classes in high school. Furthermore, if a middle school student decides that a college degree is a worthwhile objective, that student has the opportunity to complete necessary courses in high school in order to qualify for admission to a college or university.

US middle schools often include some sort of required computer literacy class for all students. Adding a Scalable Game Design unit to this class means that every child has the opportunity to complete a motivating computer-based project by designing the look and programming the behavior of their own computer games. Our computational thinking tools are powerful enough that as students become more advanced, they can move beyond games to build simulations. We also would assert that student ownership of personally meaningful projects provides the best environment both for learning computational thinking and for building interest in further education in computer science and in STEM careers.

Teaching middle school students computational thinking rather than traditional programming achieves our goal of providing a computer-based learning activity that is creative and exciting for most students, while building the strong computational thinking skills needed in many classes such as mathematics and science courses. Computational thinking, defined by Cuny et al. [2010] as "the thought processes involved in formulating problems and their solutions so that the solutions are represented in a form that can be effectively carried out by an information-processing agent," provides a bridge for students between a natural language game description and the agent behaviors in their SGD computer games. In contrast, most programming languages used by professionals are powerful but their complex syntax requires significant effort to master for beginners. Teachers in the SGD program focus on teaching the computational thinking skill of translating a game description into agents and actions. Students' verbal descriptions of the games they are trying to create can be translated into if-then rules in order to control the behavior of agents in a game or simulation. The students learn to create and use groups of rules that form a computational thinking pattern (e.g. collaborative diffusion, collision, generate; see appendix). By using an environment that implements drag-and-drop structures, SGD enables students to build a game or simulation incrementally and test their program at each step. A project-first rather than a principles-first curriculum keeps students engaged as they learn how to think and design computationally while they build their games.

In the following sections this paper outlines our systemic computational thinking education model, describes the theory behind the Scalable Game Design strategy and presents data illustrating the efficacy of our approach.

2 A SYSTEMIC MODEL TO DEVELOP COMPUTATIONAL THINKING IN SCHOOLS

In the following sections we describe, as the main contribution of this paper, how game design can be employed as a *systemic strategy* to develop students' computational thinking skills, which they can later leverage to create science simulations. These sections describe the need and the process to move beyond programming environments, explain the idea of systemic computational thinking, outline the benefits of a project-first philosophy, suggest the relevance of student ownership to motivation and, finally, compare related work.

2.1 Beyond Programming Environments

Starting in 1995, our research at the University of Colorado with AgentSheets [Repenning and Ambach 1996] explored the idea of supporting kids with game design and simulation building through drag and drop visual programing interfaces. Other programming drag and drop programming languages including Alice [Conway et al. 2000] and Scratch [Resnick et al. 2010] followed. However, it became clear that while making programming more accessible was important, the real challenge was to create a systemic model of computer science education working in schools. Early on, we had many schools using this curriculum in middle school afterschool programs such as Friday afternoon computer clubs. During our visits to some of these clubs, we noticed that students were highly motivated and were able to create advanced games. However, it was also apparent that these clubs typically attracted the usual students who were already interested in computer programming. In 2003, one of the teachers of a Friday afternoon computer club felt he could teach game design to a regular class instead of just using it as an afterschool program. When we visited his class, scheduled during the regular school day, we realized that the ratio of boys to girls was 50:50 and the class also included traditionally underrepresented minority students, who were absent from the after school club. In informal discussions, most students confirmed that they never even considered participating in a Friday afternoon computer club but they did enjoy the game design activity very much and wanted to do more. This transition from self-selected students to assigned students changed the focus of the research to broadening participation and became the main insight into developing a strategy for a systemic model to increase student access to computer science education.

After this transformational experience, we started to explore scale up models and created the Scalable Game Design curriculum, which includes two components: the professional development for teachers as well as a set of curricular materials for the classroom. The Scalable Game Design project started with a strategy developed from our CSE experiences in local schools over the past 15 years. The mission statement of the Scalable Game Design project is to "Reinvent computer science in public schools by motivating and educating all students, including women and underrepresented communities, to learn about computer science through game design starting at the middle school level." The key to finding a systemic solution to the CSE pipeline problem was to develop a strategy that addressed student motivation and pedagogy used in the classroom as discussed in [Repenning 2013].

In 2003, based on the success in one of their schools, a moderately sized district in the Rocky Mountain Region asked us to train their middle school educational technology teachers to implement Scalable Game Design in most of its middle schools. Over the next two years, the SGD project shifted from incidental exposure for a few dozen students attending the after school program to an initiative that could reach practically all middle school students in a regular class within a district. But just because this intervention worked in a tech-oriented university town did not mean that it would also work in other communities.

In 2008, with grant funding by the National Science Foundation, we were able to investigate how Scalable Game Design could be implemented in radically different communities including inner city, remote rural and Native American schools around the US. The project advanced much faster than expected. The SGD study was initially planned to include only about 1,300 students but because of the level of excitement from teachers and students it quickly expanded into a study that after five years, has involved over 200 teachers and 10,000 students across the US. Word-of-mouth advertising led to teachers beginning to teach other teachers in their

schools and region. Teacher use of Scalable Game Design also moved beyond educational technology courses to be implemented in math, science, English, art, social science and even foreign language classes. The project currently involves more than 25 school districts, including several large school districts that are involved in scale up efforts funded by the National Science Foundation and Google, as well as four international replication sites. The overall impact of Scalable Game Design is much larger as the majority of schools using Scalable Game Design are not participants of the study.

Over the duration of the project, we collected student data to document levels of motivation as well as emerging skills. Our analysis of the data provoked the exploration of particular patterns in student motivation. For example, why did we find different levels of motivation between boys and girls – measured by the degree that students wanted to continue these classes – in different schools? We found that pedagogy was the key to broadening participation. This was great news as we were able to not only report these findings to teachers but to include pedagogically oriented activities in our teacher professional development with the goal of focusing teachers toward a more guided discovery approach. We are currently investigating this theory of broadening participation [Webb et al. 2012] with a new large project and have started to explore issues of scaling up through online and blended teacher professional development. In summary, we have developed and tested a strategy to reach a large number of students early on in middle schools. But this also begs the question, when we make computer science education available through a broad spectrum of courses, what is it really that we are teaching?

2.2 Systemic Computational Thinking

The idea of a computationally enabled 21st Century workforce is appealing. For instance, the US President's Information Technology Advisory Committee recognized early on the urgency of multidisciplinary technology education [PITAC 2005]. However, if computation is so important, how can it be introduced in schools in order to reach an extremely large audience? The perception of most schools in the US is that while they are interested in adding computation to the curriculum, they are highly concerned about replacing existing topics. In other words, computer science education is considered a zero sum game. If one adds computer science to the curriculum, something else will have to go (e.g., one may have to replace foreign languages with programming languages).

The urgency of learning-to-program initiatives is justified by the alarmingly large and steadily growing gap between the supply and demand of a programming-skill-enabled workforce. Implicitly, the assumption made is that the people who are needed are programmers and that curricula and courses in schools should steer K-12 students towards a career trajectory that includes a computer science college education, resulting in a job as a programmer. This may not be the most productive model because it suggests that computer science education, at the K-12 level, is of a highly self-serving nature. This approach also clashes irreconcilably with the prioritization of tested subjects; that is, since computer science is not addressed on standardized tests, schools have little incentive to offer computer science courses. Consequently, this kind of computer science education model is not likely to be adopted by low performing schools because they, perhaps rightfully so, have to focus on their core educational mission which includes math, literacy and science courses.

A radically different model of computer science education would conceptualize computing skills at a *systemic* level. That is, instead of pursuing a professional programmer-bound career path, a systemic computer science education would

consider the impact computing has on a wide range of 21st century careers including STEM and social science careers. Computing is everywhere and goes far beyond the ability to use word processors or slide presentation applications [PITAC 2005]. Computing includes the ability to use programming to gather and process data, as well as visualize it.

Traditional K-12 programming courses are not likely to have systemic impact on STEM courses. In the 1980s, a number of early attempts to bring programming to public schools were declared as failures largely because research did not find compelling evidence of the expected transfer from programming to other disciplines. At the time Pea [1983] summarized these attempts most critically:

> "This idea – that programming will provide exercise for the highest mental faculties, and that the cognitive development thus assured for programming will generalize or transfer to other content areas in the child's life – is a great hope. Many elegant analyses offer reasons for this hope, although there is an important sense in which the arguments ring like the overzealous prescriptions for studying Latin in Victorian times."

While transfer may be the key to a systemic computer science education model, it is not clear what is really being transferred. In our experience we have found computer science education to be more effective when the focus is less on specific programming skills and more on abstract notions of programming. This is particularly true when these concepts can be mapped systemically onto existing STEM topics that are of immediate relevance to schools. Computational thinking is such a framework. The term was introduced early by Papert [1996] but later popularized by Wing [2006]. Computational thinking (CT) offers a more conceptual perspective of computation than programming. An instructional technology administrator of a large school district participating in the Scalable Game Design project put it most succinctly: I want to be able to walk into a classroom where students are designing games and ask a student: "...now that you can make Space Invaders, can you also make a science simulation?" This more practical notion of transfer became the ultimate benchmark for our research because it had the potential to answer difficult questions of how to create a systemic model of computer science education for public schools in ways that avoid zero sum games for curricula.

There are now almost as many definitions of computational thinking as there are academic papers on this topic. The International Society for Technology in Education (ISTE) and the Computer Science Teachers Association (CSTA) recently produced a operational definition in which computational thinking was described as a problem-solving process with multiple characteristics [CSTA et al. 2011]:

- Formulating problems in a way that enables us to use a computer and other tools to help solve them.
- Automating solutions through algorithmic thinking (a series of ordered steps).
- Generalizing and transferring this problem solving process to a wide variety of problems.

Our viewpoint for *systemic computational thinking* is based on a simple process consisting of two stages:

(1) Motivation and Skills through Game Design The first stage is to start with game design as a motivational activity to get students introduced to computational thinking concepts, which we call computational thinking patterns [Basawapatna et al. 2011]. Computational thinking patterns are not based on traditional programming

concepts such as loops and if-then statements, but on phenomenological notions [Michotte 1962] describing agent interaction patterns such as object collision that are relevant to games, STEM simulations and many more applications. The motivational aspect draws heavily on creating opportunities for student ownership in which students create 2D [Repenning and Ambach 1996] or 3D [Repenning et al. 2012] objects and worlds that are interesting and relevant to them. Programming brings these 2D and 3D worlds to life.

(2) Leverage Skills in Science Classes In the second stage students transition from games to science simulations, leveraging the computational thinking patterns they acquired during game design projects. As part of our Scalable Game Design study, we have found evidence of sustainability in two directions: students are highly motivated by this approach, and teachers without computer science backgrounds can teach these concepts. A large percentage of participating schools moved beyond the initial training materials transitioning from games design to STEM simulation design.

Compared to more traditional programming education, systemic computational thinking should not be understood as an educational trade-off between breadth and rigor. For instance, when designing games, even the youngest students using Scalable Game Design follow object-oriented design principles to analyze textual game specifications and, by identifying nouns and verbs, turn them systematically into working programs. They learn to employ notions of methods, use variables, create complex conditional code and debug highly interactive systems. In other words, students gradually become exposed to advanced topics in computer science in motivational contexts such as game and simulation design. However, to work well in schools computer science education needs to shift radically in terms of pedagogical approaches to reconsider the order in which skills and challenges are being balanced over time.

Systemic computational thinking is highly attractive to schools because it already aligns well with existing standards as it supports the use and creation of models and simulations while strengthening math and literacy skills. The goal of introducing computational thinking to schools, therefore, is not to replace existing subjects but to think of computational thinking as a new literacy that is useful for a wide array of subjects. Our experience to date suggests that, with appropriate scaffolding, diverse groups of students not only become interested in computer science but also can transfer computational thinking to a wide variety of disciplines including science, technology, engineering and mathematics (STEM), as well as language learning and art. Students who use programming to create a scientific model transform their computers into highly creative instruments that support scientific processes comparable to the way actual scientists engage in research. Therefore, computational thinking fundamentally transforms learning into a personal experience rather than an act of memorization or rote learning.

2.3 The Project-First Philosophy

Our philosophy stems from a project-first, as opposed to a principles-first, approach. Whereas a principles-first approach focuses on conveying theoretical aspects of a subject and learning necessary skills before students have opportunities to apply them, a project-first approach allows students to immediately engage in computer programming design experiences and to learn concepts as the need arises.

Early on it became clear to us that students did not sign up for computer science courses because of their negative perceptions of the subject. Asked in the context of a

typical computing course, one middle-school student summarized her perception of programming as "hard and boring," which does not suggest a workable tradeoff but instead a lose-lose proposition. The "hard" part is a cognitive challenge that requires a visual programming approach such as drag and drop programming. The "boring" part is an affective [Picard et al. 2004] challenge that relates back to motivation. Why should students really want to program? In our research we found student ownership [Repenning 2013] to be the key to motivation. When students are enabled to create artifacts that are personally meaningful to them, they are much more likely to be motivated. Once they have created shapes such as people, animals and cars and assembled them into a world, they are highly motivated to bring them to life through the process of programming.

The project-first approach affects motivation to the point where, at the middle school level with students as young as 10 years old, students essentially demand access to advanced programming concepts to build their games. For instance, students want to build artificial intelligence that allows the characters of their game to collaborate with each other and to engage in tracking, even if this means that they will have to use sophisticated mathematics concepts such as diffusion equations. A principles-first approach would not likely work for these students. Advanced concepts such as diffusion equations are not considered suitable material for middle school mathematics education. And yet, these very students want to learn these advanced topics because they are solving a problem relevant to them [Papert 1980].

In order to work well, the project-first philosophy requires appropriate scaffolding [Reiser 2004], and can be best described in the context of our theoretical framework called the Zones of Proximal Flow [Basawapatna et al. 2013]. The Zones of Proximal Flow framework is a combination of Csíkszentmihályi's Flow [Csikszentmihalyi 1997] theory with Vygotsky's Zone of Proximal Development [Vygotsky 1978] conceptualization. The essence of Scalable Game Design is that programming challenges and skills should be balanced and that there are different paths, some better suited than others, for students to acquire new skills and tackle more advanced challenges.

The Zones of Proximal Flow framework (Figure 1) illustrates the difference between the principles-first and the project-first philosophies. Imagine that a student has just created a simple game representing a combination of limited skills and minimal challenge (point A). This student wants to progress to point B by creating a more sophisticated game involving greater challenge and requiring additional skills. The traditional, principles-first route would suggest the abstract acquisition of facts and concepts without any concrete application of these principles. This path is likely to navigate the students into the "boredom" zone. Only later, perhaps many semesters later, will that same student finally have an opportunity to apply these skills in a meaningful project. By then, many of the skills may have been forgotten. The "project-first, principles just-in-time" path, in contrast, immediately engages the student in project work. The project will be challenging and is likely to push students to their threshold of understanding (the Zone of Proximal Development) but with the help of the teacher (and peers) they manage to learn the relevant concepts in an optimal way that is highly engaging.

A final but very important aspect of the project-first philosophy is the notion of ownership. Balancing challenges with skills is important but what motivates students to engage in challenges in the first place? Our observations of students using SGD units suggests that students' ability to create their own 2D or 3D characters build visually interesting worlds is an important motivational factor in developing students' interest in programming.

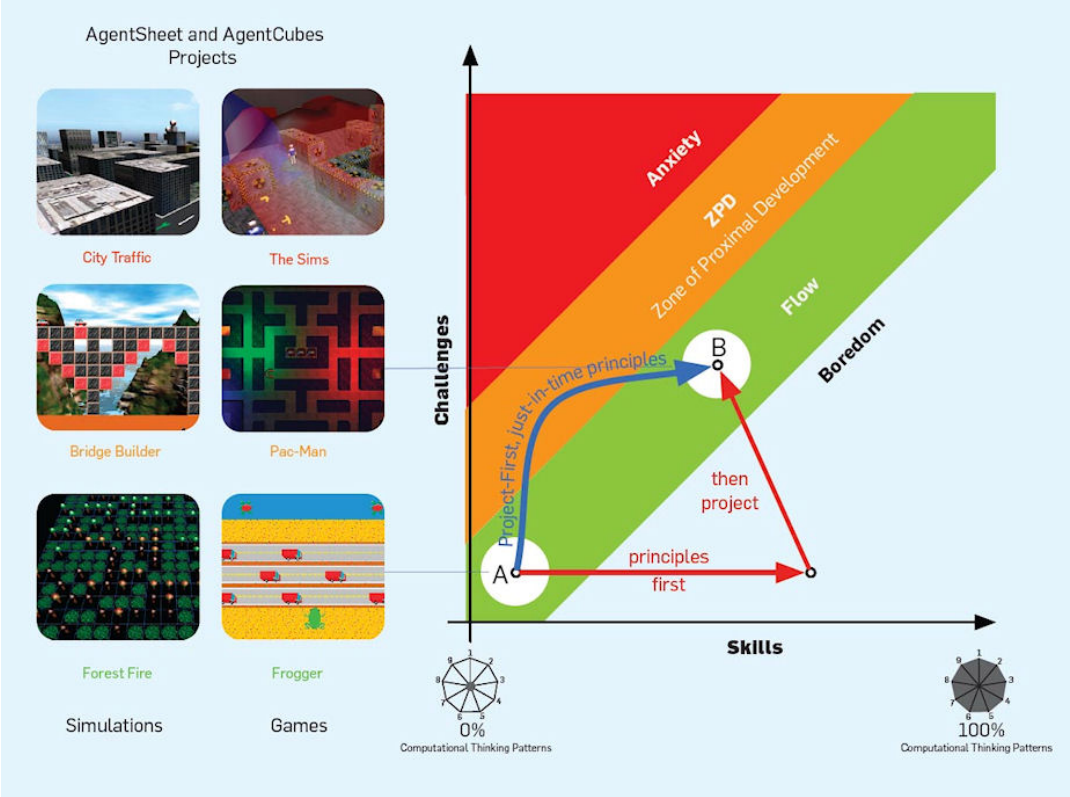

Figure 1: The Zones of Proximal Flow, a combination of Vygotzky's Zone of Proximal Development and Csikszentmihalyi's Flow.

2.4 Motivate through Ownership

The introduction of SGD often involves student completion of games using instructional materials that heavily scaffold the initial design, programming, and completion of a working game. This is seen as a necessary part of an instructional sequence in which students are learning a new interface along with related syntax, skills and concepts. However, at some point in the instructional sequence students should have an opportunity to apply the computational thinking patterns they have learned in a new context. To promote student design and ownership in computer programming, we encourage teachers to include a culminating activity at the end of the SGD unit in which students create their own game by defining the problem space and related objectives, making the agents used in their world, and deciding how agents will interact with each other. In this more student-centered project scenario, student interests and understanding of computational thinking patterns are more transparently revealed.

Yet, we have also found that even when teachers do not have the time to permit students to construct their own projects, some degree of student ownership still emerges in students' reflections on the SGD experience. As students begin to see how computer games are designed and adapted, they often apply their own signature on the recommended SGD sequence of games and simulations; students also begin to explore the relationship between programming methods and agent behaviors that are generalizable and see how their subtle adaptations are revealed in different variations on the same game or STEM simulation. Ownership of learning means

students can: make choices about what they will learn; become responsible for their own learning by defining goals that are meaningful to them; and evaluate their own progress, asking questions, finding information and making corrections when needed. Even when the teacher provides guidance and structures a complex activity, students set their own goals within that activity, increasing their motivation and interest in the completion of the activity [Hmelo et al. 2000].

Our evidence of increased student ownership through SGD is derived, in part, from students' responses to open-ended prompts that we included on an online student motivation survey administered by teachers to students after they completed their programming projects. From set of over 10,000 responses submitted by students to the prompt, "How did AgentSheets make you think differently about ways to use computers?" we randomly selected a sample of 700 responses and coded them for characteristics of student ownership. Included below are several representative responses that were coded for student ownership:

— "[SGD] is a great way to learn how to create your own games because when you build the game you also learn how and why the rules that you use work …. It made me think about the other complicated things about computers. Before I took this class, I never knew that making these games could be so complicated yet efficient." (female student)

— "I like soccer and since the world cup has just passed I would like to make a game on AgentSheets about the world cup where the players kick the ball and score it in the goal." (male student)

— "…gives me the chance to make games I have dreamed about, and wish other[s] had made so I could play them. I also get to be creative, and take responsibility for my own games and actions. The games on the website are also fun to play, and give me ideas for what I want to put on my game, or want to work towards." (female student)

Out of the 700 responses that were reviewed, 139 (20%) indicated some aspect of student ownership. We would argue that the accessibility of the SGD curriculum and software, and the opportunity for students to engage in relatively rapid design of a working game or simulation, contributes to greater student ownership of the process and increased interest in similar activities.

2.5 Related Work

The space of computer science education research is vast and, in its entirety, beyond the scope of this paper in terms of related work. One way to focus the discussion is to think about middle school computer science education approaches based on curricula and programming environments with explicit connections to computational thinking.

Alice [Conway et al. 2000] is a programming environment used often for 3D story telling. Werner has analyzed student creations with Alice and conceptualized the algorithmic thinking, programming, modeling, and abstraction based on evidence they find for students to work with events, alternation, iteration, parallelism, methods, and variables as computational thinking [Werner et al. 2009]. The Alice website does include a curriculum in which students with no programming background learn about object-oriented programming. Alice is similar to AgentSheets and AgentCubes by also using a drag and drop programming language. In contrast to AgentCubes, Alice does not include 3D modeling tools allowing students to create their own 3D shapes.

Scratch is a programming environment often used for creating animations [Resnick et al. 2009]. Resnik suggests that computational thinking is more than programming and is analogous to the argument that language literacy is more than

writing. Scratch features a drag and drop visual programming language. There is an active community website including ScratchEd where teachers can share lesson plans, however Scratch does not provide its own curriculum. The implicit learning model of Scratch is a social one in which members of the community learn from each other through the opportunity to share and extend projects. Studying Scratch users by analyzing their projects, Scaffidy and Chambers [2012] found some evidence that in many cases social skills of community members increased but the level of sophistication of their projects decreased over time.

Both Alice and Scratch could be used, in principle, to create science simulations but because of their lack of built in visualization tools, such as 2D or 3D plotting tools, and lack of integration to spreadsheets, it would be challenging to use them for computational science applications.

Computer Science Unplugged [Bell et al. 1998] is a curriculum that has been developed for many years and has the goal to teach computational thinking without using computers; instead, many of the activities include manipulatives such as cards or dice. The activities covered in the main book have explicit curriculum links to mathematics, English and technology. Scalable Game Design originally did not include unplugged activities. However, Scalable Game Design Brazil (SGD Brazil) started to introduce a highly popular unplugged activity based on a Frogger game. In this unplugged Frogger game the students play out the roles of the different characters in the game (Figure 2) such as the frog, trucks, turtles and logs. The researchers of the SGD Brazil project found this activity highly motivational for kids and also speculated that it could increase the students' understanding of spatial relationships relevant to computational thinking. A number of SGD USA teachers started to include unplugged activities for similar reasons as well as to address concerns of parents that computer science education results in students spending even more time behind computer screens sitting and not moving.

Computer Science Principles (CSP) is a curriculum currently being developed by the College Board and piloted by a number of schools [see Astrachan and Briggs 2012]. The College Board is the organization also in charge of conducting the Computer Science Advanced Placement test, which is for high school students taking Advanced Placement courses for college credit. This curriculum is based on a generic framework for "Big Ideas of Computer Science" that recognizes computational thinking as data, abstraction, creativity, impact, internet, programming and algorithms. CSP primarily focuses on computing concepts and does not prescribe how to employ specific tools to achieve concrete skill requirements.

Explore Computer Science (ECS) [UCLA 2011] is a year long high school course. For students to become "computational thinkers" in ECS they have experiences solving a wide range of problems and experimenting with a variety of solution strategies. Unlike Computer Science Unplugged and Scalable Game Design, even though CSP and ECS focus on computer science they provide minimal connections to non-computer science topics and are not designed to be used systemically.

A key difference between these tools and curricula compared to AgentSheets, AgentCubes and Scalable Game Design, from a computational thinking point of view, is an explicit focus on transfer. At the tool level, AgentSheets/AgentCubes are aimed for dual use in both game and STEM simulation design by including features such as 2D/3D plotting tools and integration with spreadsheets. At the curriculum level, Scalable Game Design strongly emphasizes specific notions of computational thinking, called computational thinking patterns that are measurable and are an explicit part of teacher professional development so that teachers can promote

transfer among their students. The next section provides details on how the Scalable Game Design project addresses this challenge.

3 THE SCALABLE GAME DESIGN STRATEGY

The Scalable Game Design strategy consists of four components: exposure, motivation, education, and pedagogy. Goals were created for each of these four components as detailed below. The goals are presented in a theory of change format [Ployhart & Vandenberg 2010]. That is, each introductory sentence in italics is in the format of <*approach*> so that <*outcome*>.

3.1 Exposure

Develop a highly adoptable middle school CT curriculum that can be integrated into existing computer education and STEM courses so that *a potentially very large and diverse group of children is exposed to CT concepts.* The small numbers of middle schools that offer programming-related after-school programs attract only a very small percentage of students. A successful computer club at a middle school may draw about twenty students, consisting almost exclusively of boys already interested in programming. As noted earlier, a curriculum-integrated approach has a much higher potential for systemic impact compared to many after-school programs. However, to reach this kind of exposure, school districts must see direct value in CT education and find ways to integrate it into existing courses. Alignment with standards is essential. The SGD curriculum has aligned game design activities with International Society for Technology in Education (ISTE) National Educational Technology Standards [1] (NETS), and aligned STEM simulation building with Common Core Standards[2] (CCS). The Next Generation Science Standards[3] (NGSS), which are still emerging, mentions CT explicitly. Many NGSS CT activities are based on students collecting, analyzing and visualizing data from simulations.

In order to broaden participation of underrepresented students in computer science education, it is essential to reach students at strategic times in their development as noted in [Peckham et al. 2007]. Middle school students, who reach the conclusion that math or science is not for them, are not likely to pursue advanced STEM courses in high school. In contrast to the US model of sending all students to the same type of high school, many European school systems sort students according to their aptitude into different tiered secondary schools. Further details regarding this approach can be found in the Darmstadt model [Hubwieser et al. 2011]. Given the limited CSE opportunities available to secondary students we believe it is important that CSE can reach all students through earlier school experiences. That is, computer science education at the secondary level should not be limited to the top tier and "gifted and talented" programs. Our experience with some of the most challenging schools has demonstrated that engaging students in computer science education can be truly transformational.

3.2 Motivation

Create a scalable set of game design activities that begin with a low threshold and progress to a high ceiling so that *students with no programming background can*

[1] http://www.iste.org

[2] http://www.corestandards.org

[3] http://www.nextgenscience.org

produce complete and exciting games in a short amount of time and advance toward the creation of highly sophisticated games. Scalable Game Design has a uniquely low threshold (as described by [Papert 1993]) for teachers and students, making game design accessible across gender and ethnicity. We have developed a 35-hour professional development program in which we train teachers to have students build their first playable game from start to finish in about one week (e.g., 5 lessons x 45 minutes). The ability to create a playable game is essential if students are to reach a profound realization of, "Wow! I can do this!"

Figure 2. Motivation is based on a gradual process and comes through a variety of experiences.

A project-first approach requires highly accessible authoring tools, which are essential to support learner acquisition of computational thinking patterns. The AgentSheets programming environment, as described in [Repenning and Sumner 1995], is a low-threshold, high-ceiling end-user tool that uses rule-based [Repenning and Ioannidou 1997] and conversational programming approaches [Repenning 2011] for game and STEM simulation design. Figure 2 shows a simple Frogger-like game created in AgentSheets, a Frogger-based unplugged activity, a game design classroom activity and a sophisticated CityTraffic simulation created in AgentCubes (excerpted from [Repenning et al. 2012]). From our experience with the SGD curriculum, we claim that for systemic impact, the software used in K–12 schools needs to support student design of games *and* ways of modeling social and scientific phenomena in STEM.

Motivational levels, as measured by the expressed interest to continue with similar classes, are extremely high with this approach. The results from our survey (detailed in [Webb, Repenning & Koh 2012]) found that 74% of boys and 64% of girls want to continue with similar coursework. There is also almost no difference in "interest to continue" between white (71%) and ethnic minority (69%) students. In several schools, teachers have reported that after students complete SGD units, both boys and girls are so motivated by their experience that they go to the counseling office to put computers as their first elective choice. They also report a significant increase in the participation of girls in elective courses. As one computer teacher reported, "I used to only have 2 or 3 girls in my elective classes, now half of the class is girls."

Motivational levels were not only high for students participating in the study, but also for community college students who served, at the beginning of the project, as support for implementation sites. We found, through interviews of community college students, that these students decided to continue their computer science education beyond the community college as a direct result of their exposure to this project. Multiple students stated that they became motivated to pursue further computer science education, some even computer education paths, while helping middle school teachers create video games in summer institutes. Following one of the initial summer institutes, one community college student transferred to a 4-year university and three other students prepared to transfer the next academic year.

3.3 Education

Build computational instruments that analyze student produced projects for CT skills so that *learning outcomes can be objectively measured.* To be effective in a school context, a CT curriculum needs to include educational activities with measurable learning outcomes. We believe that it is critical to go beyond motivation research and study the educational value of game and simulation design, including the exploration of learning progression and transfer. One method we have found of abstracting end-user game design is through what we call Computational Thinking Patterns (CTPs). CTPs are design patterns that students initially learn in game design, but then, transfer to the creation of STEM simulations [Koh et al. 2010]. They are groups of behaviors (conditions/actions) given to one or more agents in games that are very similar to groups of behaviors you would use to create agent interactions in simulations. CTPs can be thought of as the units of transfer between game design and science simulations. We devised the Computational Thinking Pattern Analysis tool (CTPA), described comprehensively in [Koh et al. 2014] as an analytical means of extracting evidence of student use of CTPs directly from games and simulations built by students (Figure 3).

Computational Thinking Pattern Analysis is based on a *embedded assessment* framework inspired by Landauer and Dumais' Latent Semantic Analysis [1997]. CTPA is not meant to analyze the entire range of CT skills, but it can identify CT skills relevant to a large number of applications. CTPA detects program patterns expressing object interactions with phenomenalistic (described in [Michotte 1962]) interpretations such as collision or diffusion. For instance, a collision between a frog and truck expressed in a Frogger-like game is programmed in the same way that the collision between two molecules in a science simulation would be programmed. CTPA measures the presence of nine different computational thinking patterns, described in detail in [Basawapatna et al. 2011]. The CTPA graph is used to determine a number of performance measures including divergence, CT skill progression, and transfer.

Divergence The difference between a game defined by a tutorial (Figure 3, left, green) and a game built and submitted by a student (Figure 3, left, orange) is rendered visually through the difference in shape and can be quantified numerically as divergence [Benett et al. 2013] from the tutorial. Zero divergence suggests that a student built a game with identical CT patterns to a tutorial.

CT skill progression CTPA graphs can be aggregated over time to compute a CT skill progression that describes student acquisition and application of computational thinking patterns. This can be a powerful technique to longitudinally track student use of CT skills.

Transfer CTPA graphs can be employed to study transfer of computational thinking patterns. Our research has shown early indications of potential transfer as described in [Koh et al. 2010]. Short transfer, defined in [Bransford et al. 2000], can be assessed by teaching a computational thinking pattern in the context of one game and analyzing the expression of the same pattern in a different game. Far transfer, also defined in [Bransford et al. 2000], involves transfer of programming and modeling from a game context to a STEM simulation context. CTPA can be used to find correlations between source and target domains, but would have to be augmented with other methods to make stronger claims of transfer.

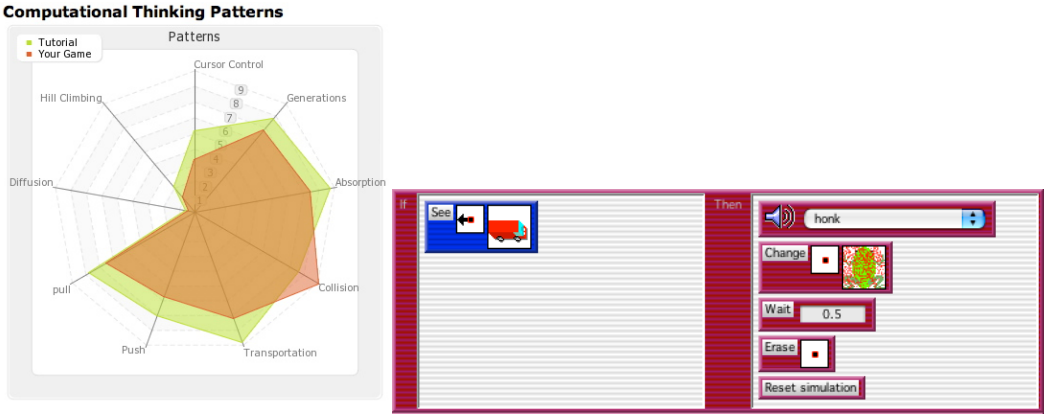

Figure 3: CTPA graph (left) of a student's Frogger game (orange) compared to tutorial (green), including patterns such as collision, implemented in an AgentSheets Frogger-like game as a rule (right) when the Frog meets a Truck.

3.4 Pedagogy

Systematically investigate the interaction of pedagogical approaches and student motivation so that *teachers can broaden student participation.* To accomplish our aim of building CT capacity and broadening participation, we envision a need for scaffolding that enables flow and proximal development within challenges that would otherwise be in the realm of unrealized learning, as defined in [Quintana et al. 2004]. Additionally, and of import to new populations of students and novice participants, scaffolding separates learners in ways that bring diverse repertoires of practice, as described by [Gutierrez et al. 2003], into play in the learning setting, positions all participants as learners in the setting, and enables *re-mediation* as defined by [Gutierrez, Hunter and Arzubiaga 2009]. In contrast to traditional 'remedial' approaches to improving learning, re-mediation involves re-designing contexts for learning that let students stretch their own levels of competence through the conscious and strategic use of a range of tools and forms of mediation, including

feedback. Teacher, peer, and resource-based scaffolding and mediation are part of Scalable Game Design.

With school sites implementing SGD across the US and more than 16,000 student-created games and simulations, we were able to explore a rich set of data, and found that scaffolding was the main common factor supporting student motivation across different school contexts, gender and ethnicity. Based on classroom observations, we partitioned the degree of scaffolding delivered in a classroom into three ranges (Figure 4), which correlate to student motivational levels and suggest gender-specific interpretations relevant to our theory of broadening participation [Webb et al. 2012]. The conceptual graph of gender-specific levels of motivation is based on a combination of student motivation data and classroom observations. These data suggest that pedagogy is essential to broadening participation and that too much scaffolding, defined as *hypermediation* by [Gutierrez and Stone 2002], may have a negative impact on the motivational levels of girls.

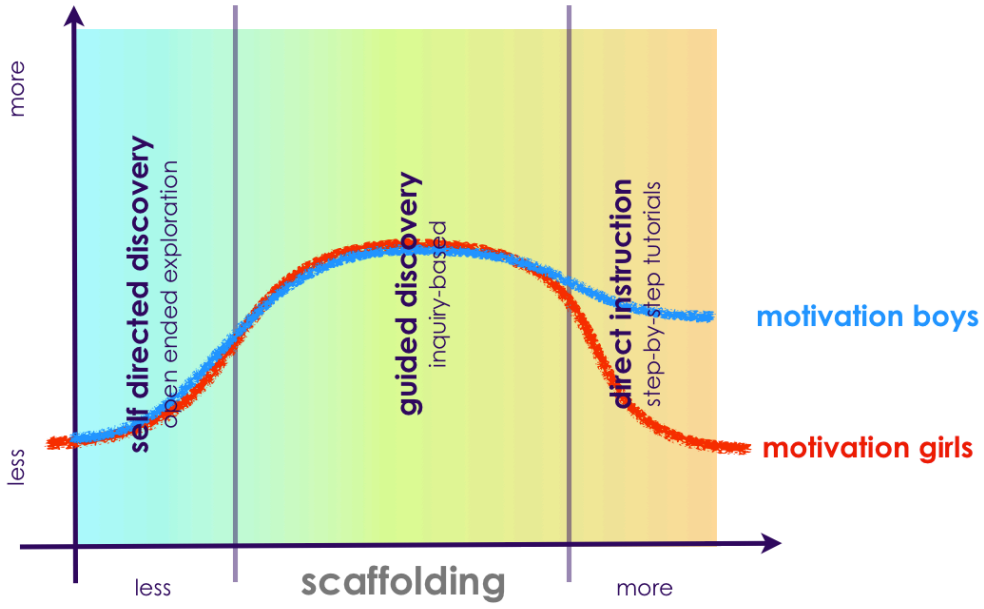

Figure 4: Scaffolding vs. Motivation

Teacher directed approaches to instruction, particularly when it involves hypermediation, appeared to polarize the motivational levels between boys and girls in our data. With direct instruction a teacher provides step-by-step instructions at a detailed level (e.g., "click this button", "paint the frog green"). This method of instruction was not motivating to either boys or girls. With guided discovery a teacher employs a more inquiry-based approach as defined by [Edelson et al. 1999] that includes student sense-making of the relationship between code and agent behaviors and meaningful classroom discussions ("what should we do?", "how can we do this?"). In classrooms exemplifying a guided discovery approach, the motivational levels of girls was statistically equivalent to the motivational levels of boys [Webb et al. 2012]. This is exciting because it suggests that broadening participation is not a question of hard-to-change factors such as school affluence. Even in cases where most literature would suggest a gender effect (e.g., if there were significantly fewer girls in a class), we found that the right level of scaffolding could negate these gender differences in student motivation.

3.5 Scalable Game Design Training and Support

For the most part, teachers using the Scalable Game Design curriculum receive training during our SGD Summer Institutes, which have been offered annually since 2009. During the subsequent school year, teachers are then expected to implement at least one unit from the SGD curriculum with their students.

3.5.1 Teacher Professional Development

Our primary approach to professional development has been through providing annual one-week summer institutes. This training is conducted in 3 blocks: 25 hours (3 days) beginner training focused on basic game design, pedagogy and computational thinking; 16 hours (2 days) intermediate training focused on STEM simulations; and 16 hours (2 days) advanced training focused on 3D designs of games and simulations. To date, more than 200 teachers of technology and content area courses from across the United States, as well as internationally, have participated in these institutes. Over time, however, the participant profile, content, and format have evolved as we have learned more about what approaches are most effective and how to most efficiently allocate our funding to support Scalable Game Design initiatives, based on teacher feedback about the summer institutes and teacher responses during the implementation of SGD units.

While the SGD Summer Institutes have always offered information regarding how to teach game design, computational science, and programming concepts, over the life of this project we have 1) continued to place great emphasis on pedagogical approaches that increase motivation for female and underrepresented minority students, 2) added information regarding the creation of three-dimensional games, and 3) increased support for STEM simulation design. Recently, we have also been able to offer introductory training for schools and school districts that are interested in implementing a train-the-trainer model, in which teachers trained by us become mentors for others.

As of 2014, the SGD Summer Institute included the following content:

— Complete development of Frogger, which we recommend to teachers as the starting point for all students. This game introduces four widely-used computational thinking patterns (i.e., collision, cursor control, generate and absorb) and also offers the opportunity to quickly learn how to use the programmatic interface.

— The design of additional games and STEM simulations, which familiarize participants with more computational thinking patterns, debugging strategies, and advanced skills.

— Implementation reports from experienced teachers. Teachers new to Scalable Game Design not only find these accounts inspiring, but are also made aware of potential instructional strategies and pitfalls.

— Information regarding computational thinking, how this skill is being increasingly recommended for inclusion in K–12 curricula through initiatives such as the Next Generation Science Standards, and how Scalable Game Design supports these efforts.

— First-hand experiences with demonstrations of alternative teaching styles and a discussion of the efficacy of the guided discovery approach.

— Time to plan and adapt lessons with the SGD team and colleagues.

— Project logistics and expectations for data collection.

Incentives for participating teachers include continuing education credit, a stipend upon completing their first classroom implementation (typically Frogger), and free software access for themselves and their students during each year of participation. In the past, we have also provided funds for travel and lodging expenses and a larger stipend, but we have found these extra offerings to be unsustainable going forward.

3.5.2 Instructional Approach to Support Scalable Design Curriculum

An ideal strategy to increase student access to computer science in a way that will be inclusive to female and underrepresented students may be to make it part of existing required courses with related contexts (e.g., computer power, technology exploration, or content area courses). In a typical classroom implementation, students complete either a game or a simulation during a one- to two-week module, and in many cases go on to develop more.

One of the major benefits of the Scalable Game Design curriculum is that students can master newly acquired computational thinking patterns by following appropriate pathways in which they build increasingly sophisticated games and/or simulations. In schools that adopt Scalable Game Design, instructional pathways have the potential to stretch across multiple years (See Appendix 1 for Scope and Sequence). Therefore, duration and content may differ for individual students.

The SGD curriculum builds students' CT capabilities by moving them from easier games (such as Frogger) to more complex games requiring additional CTPs such as Sims. Similarly, more experienced students move from these games to simulations, again starting with easier simulations having fewer CTPs such as Contagion, to more complex simulations such as simulating a forest fire.

The CT patterns a student uses to implement games are similar to the patterns they would use to implement an analogous STEM simulation. The only difference is that the agents who perform the interactions in the simulation have changed.

Our professional development events and materials emphasize the use of the previously discussed project-first approach. In the classroom, this means that a teacher starting an activity would begin by designing a game with students in the first hour of instruction, even if the students have no background in programming. Within this approach, students will experience the Zones of Proximal Flow framework in two ways. At some times, students will be in Flow, when the game or simulation authoring challenge matches their skill level. At other times, students will be in the Zone of Proximal Development, when they need outside help in order to meet the program challenges they face. In these situations, teachers can offer scaffolding to bring students back into Flow through a combination of approaches: class instruction, curricular materials from the Scalable Game Design Wiki, in-class peer student learning, software features that assist students in identifying problems with their programs, assessment instruments that make explicit the student skills obtained through programming, or through downloading and examining fellow classmates' projects. To avoid hypermediation during class instruction, a shift towards a guided discovery approach is suggested after the initial overview of interface features. This approach involves setting general goals for building the game or simulation while including discussions of why and how something should be done. Guided discovery still includes teacher directed instruction, but teacher regulation of mediation provides students more freedom to explore design options. Students' ability to personalize their design contributes to student ownership, which in turn enhances student motivation.

An alternative to utilizing tutorials and teacher-directed instruction for creating games or simulations is to have the students use a design-based approach. This can

be useful for both advanced and novice students because it allows them to gain ownership over whatever they create, and also helps them to work through their own complex ideas in a way that makes sense to them. Students learn to translate their thinking and ways of expressing their ideas into phenomena (CTPs) that can then be used to make a game or simulation. Using this approach, students are asked to write out the rules of their proposed game or simulation, either on a worksheet or in a digital format, describing what they want to create. From that statement, they then can identify the agents, as nouns, the agent behaviors, as verbs, and the situations in which the agents will interact. Using these initial parameters, they are then encouraged to elaborate on other aspects of the design, such as drawing out different depictions (states) for each agent, associating CTPs to agent behavior and interactions, and creating a representation for the worksheet/world in which the agents will interact.

This process has been successful with both older students working on advanced assignments as well as with elementary aged students who otherwise may lose interest in programming due to the difficulty and lack of enjoyment they may have with working through a tutorial with limited reading skills.

4 EVALUATION AND RESULTS

The Scalable Game Design project has developed a number of educational instruments to assess motivation and skill data of students and teachers. The following sections describe these instruments and provide evidence of sustainability in schools.

4.1 Computational Thinking Pattern Analysis

The Computational Thinking Pattern Analysis (CTPA) tool is designed to measure student learning skills and represent student content learning at the semantic level through phenomenological analysis in real time [Koh et al. 2014]. This concept uses a technique inspired by Latent Semantic Analysis (LSA) to analyze semantic meanings of a given context using several pre-defined subjects/phenomena.

4.1.1 Phenomenology: The Origin of Computational Thinking Patterns

The origin of Computational Thinking Patterns goes back to the exploration of computational thinking transfer between different application domains, and more specifically between game design and STEM simulations. The simplest level of investigation would be the program level at which students express computational thinking ideas through writing code. There are many different programming languages, exhibiting different degrees of accessibility that could be employed. Consider a programming language such as Python, which has been used to create applications such as games and STEM simulations. The study of transfer could investigate the use of language statements such as the IF, THEN, ELSE, or LOOP statements found in many programming languages. Finding similar use of these statements in game and STEM simulations could be considered an indicator of low-level transfer of some computer science skills. However, most researchers investigating the notion of computational thinking do agree that computational thinking is not identical to programming. Consequently, analysis of computational thinking may have to take place at a higher level that is programming-language independent. For computational thinking, it is not of the utmost relevance how something is expressed (e.g., programmed differently in Flash, Java, or C++) but what computational idea is being expressed. In general, to infer the intention for a given program is a challenge. However, for the relatively large universe of programs

controlling some kind of object on a screen – which is common to most games and simulations – one could try to analyze object interactions. What kinds of behaviors are objects exhibiting? Are they static? Are they moving around? If they are moving around are they moving in some straight direction or are they wandering around randomly? If there is more than just one object, how are they, if at all, interacting with each other? Are objects colliding, pulling each other, or tracking each other?

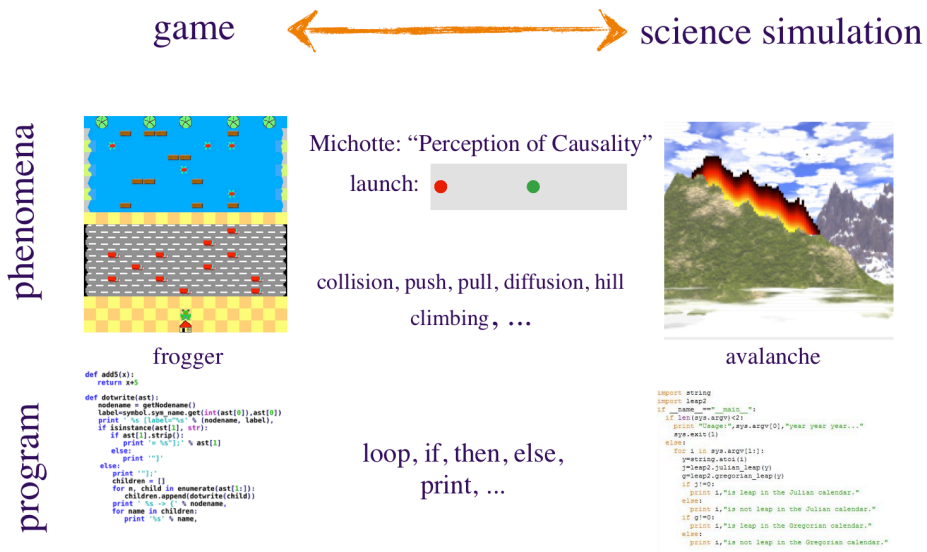

Figure 5: Phenomena: The appropriate level of analysis for Computational Thinking Patterns in SGD

In science, phenomenology is used to describe empirical observations of occurrences. This is useful for capturing object interaction. Phenomenology can be employed to describe a wide variety of phenomena including mechanical phenomena and scientific phenomena. For instance, a mechanical phenomenon can be the study of the motion of objects. In his seminal work on the perception of causality, Michotte [1962] demonstrated how peoples' interpretation of mechanical phenomena is not a cognitive process, but a perceptual one. Michotte explored a range of mechanical phenomena between abstract objects including objects colliding and launching each other. We have adopted and extended the set of phenomena suggested by Michotte to build a basis for a set of computational thinking patterns relevant to game design as well as to science simulations (Figure 5).

Using this phenomenalistic analysis for thousands of games and simulations developed using the SGD curriculum over the years, we created a list of basic and advanced Computational Thinking Patterns (see appendix). This gradually expanding set of computational thinking patterns was first used in a number of object oriented design and game design courses offered at the undergraduate level in computer science. Over time, the list of patterns as well as their explanations were refined to the point that computational thinking patterns also proved accessible to much younger and significantly less experienced audiences including middle school students with minimal computer science background.

Initially, computational thinking patterns were intended to provide a theoretical framework to conceptualize object interactions. This framework allowed students to dissect game descriptions and to describe the computational thinking patterns they found. Each computational thinking pattern would not only describe the phenomenon but also include instructions about how patterns could be operationalized. For

instance, how would one program an object collision in a certain programming environment? These operationalized descriptions range from concrete instructions such as code (e.g., how to implement a collision in Flash) to a more theoretical treatment such as a UML sequence diagram representing object interactions in a programming language-agnostic way.

4.1.2 Computing Computational Thinking: Computational Thinking Pattern Analysis

To compute computational thinking and automatically measure student learning outcomes, we built a cyberlearning infrastructure called the Scalable Game Design Arcade to collect and analyze games and simulations created by students using the Scalable Game Design curriculum. The submitted game is analyzed in terms of the Computational Thinking Patterns it uses, and results are displayed on a graph (Figure 6, right, orange spider graph). If there is a tutorial for a submitted game/simulation, then the CTPA graph also depicts the computational thinking patterns of the tutorial (Fig 6, right, green spider graph). The relationship between the two graphs is an indication of how close the submitted game is to the tutorial suggested code for the game.

CTPA compares a given game/simulation with nine pre-defined canonical computational thinking patterns: cursor control, generation, absorption, collision, transportation, push, pull, diffusion, and hill climbing. CTPA compares the given game/simulation with each canonical CTP (Figure 6, left) to produce the corresponding values in the CTPA graph (Figure 6, right). The calculated value represents how each CTP is similar to a given game/simulation. A higher value means greater similarity. If a specific CTP is used frequently in the implementation of the game/simulation, then the similarity between that pattern and the game/simulation is greater.

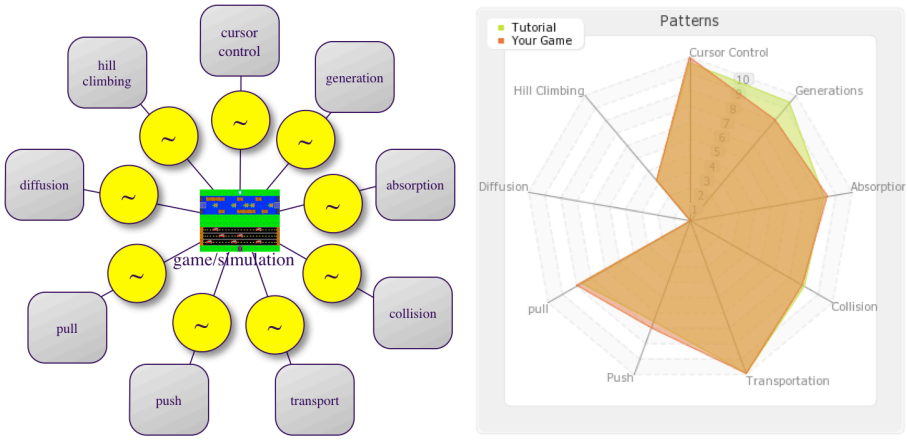

Figure 6: Using LSA-inspired similarity (denoted as on the left), a game submitted to the Scalable Game Design Arcade gets compared to nine canonical Computational Thinking Patterns. A CTPA graph showing the similarity values for each pattern is produced (right).

4.1.3 Significance: Measurable Learning Outcomes and Early Indicators of Transfer

By automatically breaking down complex programs into the constituent parts that specifically enable students to create not just games, but also simulations, CTPA is a first step for measuring computational thinking learning outcomes. Teachers and students can also use CTPA for assessing whether a desired learning goal was met. The ability to automatically detect computational thinking patterns by analyzing student creations can give educators and learners greater insight into what concepts

are understood and what concepts still need to be learned. This information has been used to inform the development of Scalable Game Design curriculum that supports a more natural and fluid progression of student exposure to computational thinking in a way that balances skills and challenges [Csikszentmihalyi 1993].

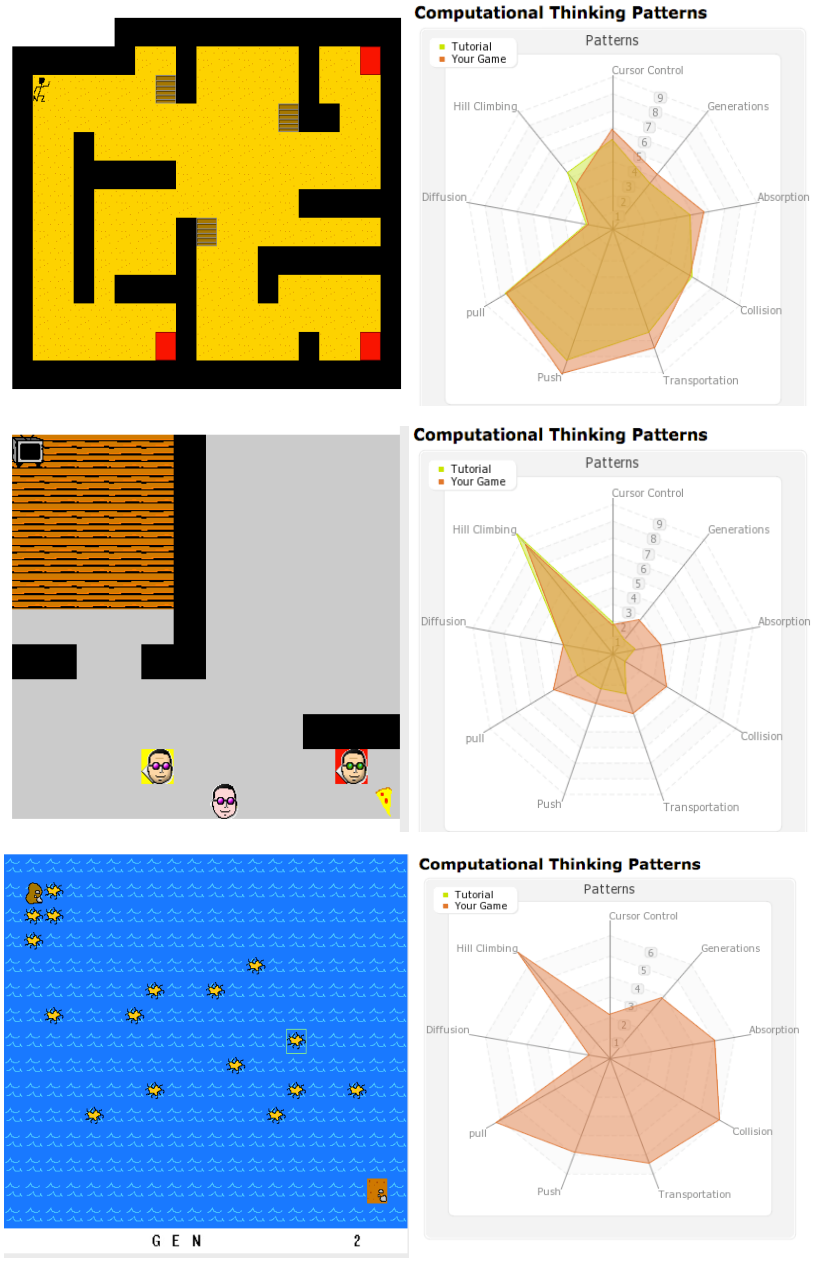

Figure 7: A Sokoban game screenshot and its CTPA graph (top). A Sims game made by the same student and its CTPA graph (center). A science simulation based on Chaos theory made by the same student and its corresponding CTPA graph (bottom). The combined CTPA graph of Sokoban and Sims are very similar to the CTPA graph of the science simulation, offering some early indications of transfer of skills from game design to computational science.

However, the really intriguing aspect of the CTPA lies within the idea that CTPs are common, shared elements between games and other STEM disciplines and that they are identifiable and measurable. If students can build games using CTPs, it might be possible that they can apply these same patterns to the implementation of STEM simulations. And if we can measure these patterns in the resulting artifacts, we can show transfer of skills between domains. For example in Frogger, the frog is transported by a log that is floating on a river. The implementation of this log transporting the frog is the same way one might implement a red blood cell transporting iron in a science simulation. In Figure 7, three projects were made by a single student in a Scalable Game Design class. In chronological order, the student implemented Sokoban and Sims games as well as a science simulation based on Chaos theory. The shape of the CTPA graph of this student's science simulation is similar to a combined CTPA graph of the Sokoban and Sims games that the student made before creating that science simulation. This can be interpreted that this student used the programming skills learned from making Sokoban and Sims to create a science simulation.

While further work is needed to distinguish correlation from causation and investigate the role of teachers in scaffolding concepts to be transferred, having any concrete indications of transfer is an important step for computer science in general and computer science education in particular. People often assume that any programming experience leads to transfer; however, as previously pointed out by Roy Pea, this is typically not the case [Pea 1983].

The specific nature of the units of transfer in CTPA, namely the Computational Thinking Patterns, and the fact that we can identify and analyze them make it possible to begin to show that transfer is actually happening. This makes CTPA a unique tool in that it is among the first tools that are able to automatically compute computational thinking.

4.2 Computational Thinking Skill Progression: Demonstrated & Comprehensive Skill Scores

While the semantic information from individual games/simulations is only a piece of student learning development, it could provide a measurement for a student's entire skill progress. Representing semantic meaning in measureable units to visually demonstrate student learning trends can benefit students and teachers directly. This approach could also indicate possible curriculum failings at a fundamental level.

The value of each axis on the CTPA Graph represents the proportion of implemented knowledge for a given computational thinking pattern within a game/simulation. The sum or average of these values is interpreted as the student's skill in designing the game/simulation, much as an average GPA from several classes is understood to represent a student's overall learning. That is, the nine computational thinking patterns are target-learning categories. A sample game accompanies each tutorial, and we created a target score for each CT pattern in each game by running the sample game through CTPA. Thus, the CTPA Graph illustrates how well students meet the target-learning goal in each assignment or group of assignments. Within the CTPA, a one-time assignment analysis is referred to as a Demonstrated Skill Score. Learning that takes place over time through several assignments is referred to as a Comprehensive Skill Score. Both Demonstrated and Comprehensive Skill Scores are calculated from the length (norm) of a vector of the nine computational thinking patterns reduced to one dimension (unit).

The Demonstrated Skill Score (1) and Comprehensive Skill Score (2) are calculated using the following equations.

$$\text{Demonstrated Skill Score } (n) = \frac{\sqrt{\sum_{i=1}^{n} (P_i)^2}}{\sqrt{n}} \tag{1}$$

$$\text{Comprehensive Skill Score } (m) = \frac{\sqrt{\sum_{i=1}^{n} [\max_{j=1}^{m} (P_{i,j})]^2}}{\sqrt{n}} \tag{2}$$

In these equations, P is a computational thinking pattern, n is the number of computational thinking patterns on the CTPA Graph, and m is the number of submitted assignments. Those equations are derived from the formula for the length of a vector.

The Demonstrated Skill Score (1) shows a student's programming skill as of when the game was submitted, while the Comprehensive Skill Score (2) shows a student's progress in skill acquisition over time. Each Skill Score is the normalized CTPA vector length. That is, a value of 0 indicates that the values of all the Computational Thinking Patterns are 0 (no evidence of any skill). A value of 1.0 indicates maximal values of all the Computational Thinking Patterns (100% evidence of all skills). For the Comprehensive Skill Score calculation, we make the following assumption to track students' skill progression: if there is a skill that a student has learned and demonstrated accurately at least once, then that skill is available for the student to use for the entire duration of the course even if it is not used again. In other words, a maximum value of any given game represents its creator's (student) best achieved level in CT pattern implementation. Consequently the maximum value is selected in this equation.

A CT skill score, Demonstrated Skill Score or Comprehensive Skill Score, can be used to track an individual student's CT skill progression or the entire class' CT skill progression. For this project, we explored two entire class' CT skill progressions (one middle school class and one college class) as computed with CTPA tool. The Demonstrated Skill Score shows a student's programming skill score as of when the game was submitted, while the Comprehensive Skill Score shows a student's progressed CT skills over time.

To develop the summary graph in Figure 8, we used 268 games and simulations from 30 college students and 73 games from 33 middle school students. From these projects, we calculated the average Comprehensive skill scores of each game/simulation for the entire class submission in one academic semester and computed CT skill progressions for each one. Figure 8 shows these CT skill progressions and the comparison between them.

Students from the middle school class sections (using SGD for up to 8 weeks) and the semester-long college class increased their knowledge and understanding of programming and game design. These improvements are reflected in their calculated skill scores over time. Middle school and college students are expected to learn each new game's computational thinking patterns. Since each new game has patterns that are specifically needed to complete it, skill scores increase with newly learned games.

A game of Frogger requires five computational thinking patterns that the students must learn. As Figure 8 illustrates, the two classes overlap over time although the CT skill scores are quite different for each class. This gap between their CT skill scores can be explained by the completeness of the different Frogger implementations. College students completed Frogger entirely, but more than half of the middle school students uploaded an incomplete Frogger game (i.e. missing transportation or some other patterns). The same completeness factor holds for the uploaded Sokoban, Pacman and Space Invaders games.

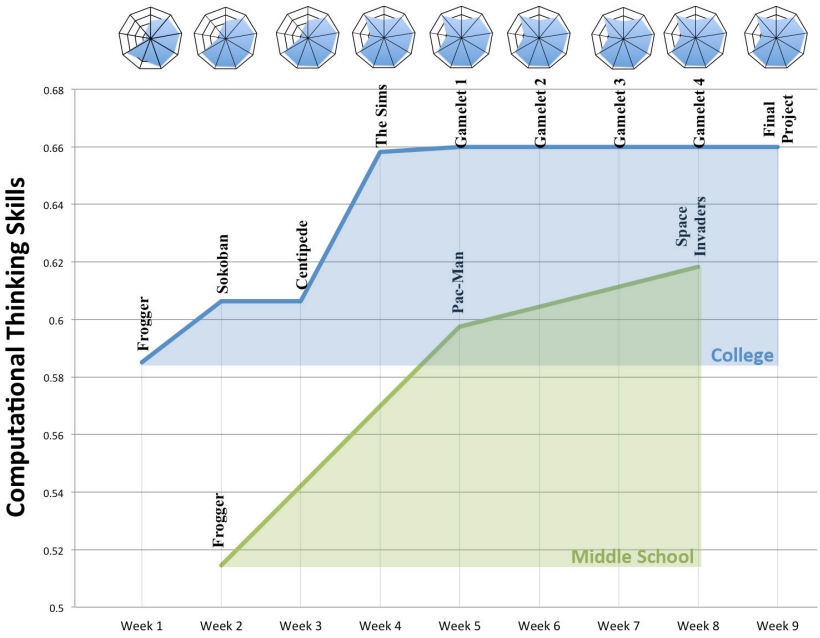

Figure 8. CT Skill Progression Comparison between College and Middle School Students with Comprehensive Skill Scores. A value of 1 would be 100% coverage of all computational thinking skills.

The time used by middle school students to move from one game to the next was over three times as long as that for the college students. Since the game learning progressions and sequence between the groups are very similar, the learning timeframe would appear to be a significant difference between the two groups. However, the game curriculum imparted enough programming skill and comprehension to allow middle school students to complete their design submission goal at a level similar to college students despite substantial differences in educational level and age—the middle school students simply took longer. The point is not that it takes middle school students longer than college students to accomplish the design of a similar video game, but that these middle school students were able to create a video game that approached the level of college students.

The CTPA's ability to compute CT skill progressions by measuring computational thinking pattern knowledge acquisition over time is a substantial step forward in showing the worth of game/simulation-programming courses and their associated visual language.

4.3 Teacher Interviews

Given the general absence of opportunities for teacher learning in computer science education, we designed Scalable Game Design Summer Institutes to support teacher implementation and develop a broad community of computer science educators. Considering the need to increase opportunities to engage students in computer-based game and simulation design, it is worth noting that in more recent summer institutes the content areas of teachers has been quite diverse. Based on the responses from 37 teachers to the most recent summer institute survey, the percentage of IT teachers participating was less than 50%. Even though the SGD project initially targeted CS teachers, the majority of summer workshop participants are now math, science and arts teachers who integrate these activities into their regular curriculum. Reaching a diverse teacher audience to engage students in computer science education is an important strategy to increase student access to computer programming. Teachers who teach across the gamut of required and elective content areas need to see how computer-based design opportunities can serve as powerful learning experiences for their students, with respect to both IT education and the goals specific to their content area.

To reveal the impact of the institutes on teachers' conceptions of computer-based design, we asked teachers to describe computational thinking in online surveys. Some representative responses from the most recent summer institute include:

> "Computational thinking is an analytical thinking practice that creates patterns of thought within the user that are applicable to many real-world situations."
>
> "Computational thinking is having students identify problems and develop solutions to these problems while reevaluating continually the success of their decisions."

In post-institute surveys for all summer institutes, many teachers noted the value of computational thinking patterns and how similarities between agent behaviors and subsequent programming requirements could be used to support the transition from game design to simulation design. As one science teacher from the 2012 Summer Institute described: "I've learned that students can, through simulations, learn the scientific process through the manipulation of variables. I've also discovered that there are many simulations that can be manipulated to teach a variety of scientific concepts."

4.4 Systemic Reform and Sustainability

Evidence of sustainability includes the probability for implementation sites to advance to program new projects and advance from game design to simulation design.

4.4.1 Sustainability: Probability to Advance

Over the last 3 years, 72 different types of games and simulations were collected from 46 participating schools. All 46 schools submitted at least one project (game or simulation) to the SGDA, and 37 schools submitted two projects or more. Also, 30 schools and 23 schools submitted 3 and 4 projects or more, respectively. Interestingly, as depicted in Figure 9, these numbers show that a ratio of 4-to-5 appears to be a threshold to move forward: 81% of the schools that submitted at least one project also submitted two projects or more, and 80% of this second group submitted three projects or more.

Considering teachers' short training timeframe and the limited financial support after the first module was implemented, a 80% success rate implies some degree of potential sustainability. Also, this result may indicate that many project schools have

successfully helped students follow the Zone of Proximal Flow, spanning students' problem solving skills over multiple problem domains.

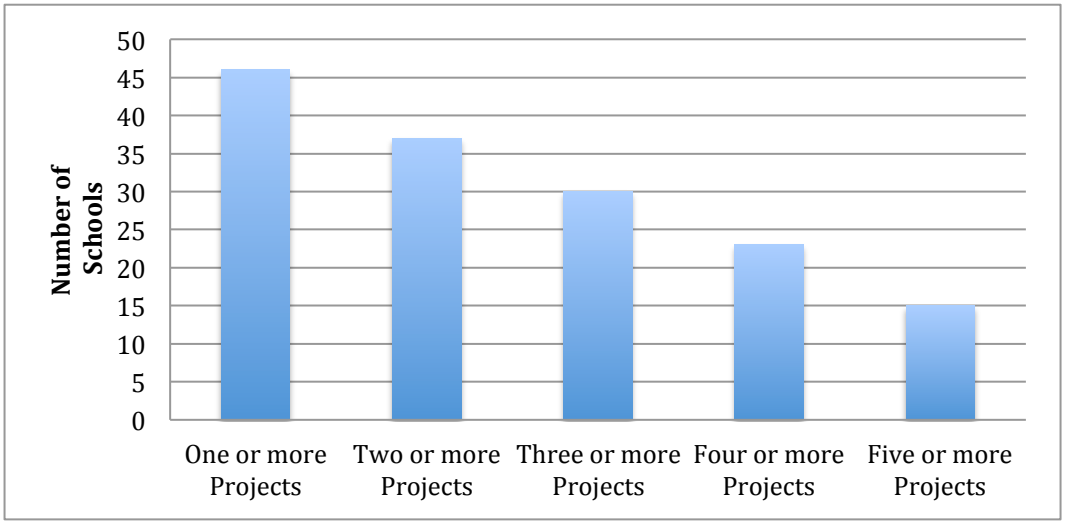

Figure 9: School sustainability of Scalable Game Design. Number of projects submitted to the arcade in the past three years per school.

4.4.2 Sustainability: Advancing from Game Design to Simulation Design

The core benefit of computational thinking education lies in bridging problem solving skills between two or more different problem domains [Wing 2006]. Through our project, we have witnessed many project schools that were able to extend and transfer their students' learning abilities and problem solving skills to the next level of problem domains; that is, from game implementation to simulation implementation. This seems to be related to the sustainability of computational thinking education since the learning continues toward a higher level of problem domain instead of stopping at the same level. For example, recall the student who created a science simulation based on chaos theory using computational thinking patterns learned from implementing Frogger and Sokoban. In spite of the lack of further financial incentive for a more advanced level of implementation, 43% of project schools successfully moved toward simulation implementation.

5 DISCUSSION AND CONCLUSION

This article has described Scalable Game Design as a strategy to introduce students to computer science through game and simulation design. To successfully address pipeline issues, CS projects must create sustainable programs that challenge and motivate students to engage in computational thinking. Furthermore these projects must be accessible by teachers and must support training in a way that enables teachers to grow into more complex engagement with the material. Finally, it must broaden participation for women and minorities, and reach students at a variety of grades and abilities. The data presented suggest that this strategy is not only highly motivational for students but, when combined with the inquiry-based teaching approaches, can significantly broaden participation of women and underrepresented students in computer science education. Its scalable nature allows students using the SGD curriculum to start as early as elementary school (at about the age of 7). This approach also allows students to progress to more advanced topics. The curriculum is

universally accessible across diverse school settings. An important aspect of Scalable Game Design is to provide the tools and opportunities for students and teachers to take the next step and transition from the design of games to STEM simulations. This is a quite daunting transition, but we found that a relatively high percentage of schools (41%) have already accomplished this transition. An important feature of SGD is its accessibility to teachers: with relatively modest professional development and with modest computer science background, teachers can quickly implement SGD curriculum in their schools. In addition, many teachers report transformational changes in their classrooms for their students and their own teaching practice.

The SGD approach has demonstrated that teachers and students are eager to engage in computer science education when the proper resources and supports are available. Unlike most professional development opportunities where only 10-15% of teachers actually implement changes as found by [Richardson 1998], SGD support has resulted in more than 80% of the teachers implementing SGD past the initial year, and more than 15% of teachers returning for additional training to strengthen and expand their skills. Even without a national curriculum for CSE in the US, teachers recognize the value of computer programming and opportunities for students to design games and STEM simulations.

Given the recent advocacy for computational thinking in the Next Generation Science Standards and STEM education literature, computer science education is being viewed as a cross-disciplinary necessity in ways that were not part of US education policy discussions less than a decade ago. Nevertheless, there is much work to do to mend the CSE pipeline in the US. Through a ground-up approach as opposed to a top-down method often employed by K-12 CS programs, SGD provides a way for teachers to integrate CSE in the regular school curriculum and help restore the CSE pipeline. It offers the curricular support desperately needed for CSE and allows for relatively easy transitions into other aspects of CS and STEM topics.

ACKNOWLEDGEMENTS

This material is based upon work supported by the National Science Foundation under Grants No. 1138526, 0833612 and 1312129. We like to thank the National Science Foundation, Google, AMD Changing the Game, Shodor Foundation, SRI International, CSTA and the National Center of Women in Computing (NCWIT) for funding, support and collaboration. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

(PITAC), 2005. Report to the President: Computational Science: Ensuring America's Competitiveness.

(UCLA) Exploring Computer Science. 2011.

Astrachan, O., & Briggs, A. (2012). The CS principles project. *ACM Inroads, 3*(2), 38-42.

Basawapatna, A., Koh, K. H., Repenning, A., Webb, D. C., & Marshall, K. S. 2011. Recognizing computational thinking patterns. In *Proceedings of the 42nd ACM technical symposium on Computer science education* 245-250. ACM.

Basawapatna, A. R., Repenning, A., Koh, K. H., & Nickerson, H. 2013. The zones of proximal flow: guiding students through a space of computational thinking skills and challenges. In *Proceedings of the ninth annual international ACM conference on International computing education research* 67-74. ACM.

Bell, T. C., Witten, I. H., & Fellows, M. 1998. *Computer Science Unplugged: Off-line activities and games for all ages*. Computer Science Unplugged.

Bransford, J. D., Brown, A. L., & Cocking, R. R. 1999. *How people learn: Brain, mind, experience, and school*. National Academy Press.

Conway, M., Audia, S., Burnette, T., Cosgrove, D., & Christiansen, K. 2000. Alice: lessons learned from building a 3D system for novices. In *Proceedings of the SIGCHI conference on Human factors in computing systems* 486-493. ACM.

Cope, P., & Walsh, T. 1990. Programming in schools: 10 years on. *Journal of Computer Assisted Learning*, *6*(2), 119-127.

Csikszentmihalyi, M. 1997.Finding flow in everyday life. BasicBooks, New York.

Csikszentmihalyi, M. and Rathunde, K. 1993. The measurement of flow in everyday life: Towards a theory of emergent motivation. University of Nebraska Press, Lincoln.

CSTA, NSF and ISTE. 2011. Operational Definition of Computational Thinking for K–12 Education. Computer Science Teaching Association, National Science Foundation, International Society for Technology in Education.

Cuny, J. 2012. Transforming high school computing: a call to action. *ACM Inroads, 3*(2), 32-36.

Cuny, J., Snyder, L., & Wing, J. M. 2010. Demystifying computational thinking for non-computer scientists. *Unpublished manuscript in progress, referenced in http://www. cs. cmu. edu/~ CompThink/resources/TheLinkWing. pdf.*

Edelson, D. C., Gordin, D. N., & Pea, R. D. 1999. Addressing the challenges of inquiry-based learning through technology and curriculum design. *Journal of the Learning Sciences*, *8*(3-4), 391-450.

Gal-Ezer, J. and Stephenson, C. 2014. A tale of two countries: Successes and challenges in K-12 computer science education in Israel and the United States. ACM Trans. Comput. Educ. 14(2), 8:1 - 8:18.

Garber, M. 2013. Ridiculously Long Men's Room Lines at Tech Conferences: A Photo Essay. The Atlantic.

Grover, S., Pea, R., & Cooper, S. 2014. Remedying misperceptions of computer science among middle school students. In *Proceedings of the 45th ACM technical symposium on Computer science education* (pp. 343-348). ACM.

Gutiérrez, K. D., Hunter, J. D., & Arzubiaga, A. 2009. Re-mediating the university: Learning through sociocritical literacies. *Pedagogies: An international journal*, *4*(1), 1-23.

Gutiérrez, K. and Stone, L. 2002. Hypermediating literacy activity: How learning contexts get reorganized. Information Age Publishing, Greenwich, Conn.

Gutierrez, K. D. and Rogoff, B. 2003. Cultural Ways of Learning: Individual Traits or Repertoires of Practice. Educational Researcher, 32(5) 19-25.

Hmelo, C. E., Holton, D. L., & Kolodner, J. L. 2000. Designing to learn about complex systems. *The Journal of the Learning Sciences*, *9*(3), 247-298.

Hubwieser, Peter, Michal Armoni, Torsten Brinda, Valentina Dagiene, Ira Diethelm, Michail N. Giannakos, Maria Knobelsdorf, Johannes Magenheim, Roland Mittermeir, and Sigrid Schubert. 2011. Computer science/informatics in secondary education. In *Proceedings of the 16th annual conference reports on Innovation and technology in computer science education-working group reports*, 19-38. ACM

Koh, K. H., Basawapatna, A., Bennett, V., & Repenning, A. 2010. Towards the automatic recognition of computational thinking for adaptive visual language learning. In *Visual Languages and Human-Centric Computing (VL/HCC'10 IEEE Symposium on* 59-66. IEEE.

Koh, K. H., Repenning, A., Nickerson, H., Endo, Y., & Motter, P. 2013. Will it stick?: exploring the sustainability of computational thinking education through game design. In *Proceeding of the 44th ACM technical symposium on Computer science education* 597-602. ACM.

Koh, K. H., Basawapatna, A., Nickerson, H., & Repenning, A. 2014. Real Time Assessment of Computational Thinking, In *Visual Languages and Human-Centric Computing (VL/HCC'14 IEEE Symposium)*. IEEE.

Landauer, T. K., & Dumais, S. T. 1997. A solution to Plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological review*, *104*(2), 211.

Margolis, J. 2008. *Stuck in the shallow end: Education, race, and computing*. MIT Press.

Michotte, A. 1962. The perception of causality. Methuen, Andover, MA.

National Governors Association Center for Best Practices & Council of Chief State School Officers. 2010. *Common Core State Standards*. Washington, DC: Authors.

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

Papert, S. 1993. *The Children's Machine*. Basic Books, New York.

Papert, S. 1996. An exploration in the space of mathematics educations. *International Journal of Computers for Mathematical Learning*, *1*(1), 95-123.

Pea, R. 1983. LOGO Programming and Problem Solving. In *Proceedings of the Paper presented at symposium of the Annual Meeting of the American Educational Research Association (AERA'83)*, "Chameleon in the Classroom: Developing Roles for Computers".

Pea, R. D. 1983. Chameleon in the Classroom: Developing Roles for Computers, Logo Programming and Problem Solving. In *Proceedings of the American Educational Research Association Symposium* (AERA'83).

Peckham, J., Stephenson, P. D. and Harlow, L. L. 2007. Broadening Participation in Computing: Issues and Challenges. In *Proceedings of the 12th Annual Conference on Innovation and Technology in Computer Science Education (ITiCSE"07).*

Picard, R. W., S., P., Bender, W., *et al.* 2004. Affective Learning — a Manifesto. *BT Technology Journal*, *22*(4), 253-269.

Ployhart, R. E., & Vandenberg, R. J. 2010. Longitudinal research: The theory, design, and analysis of change. Journal of Management, 36(1), 94-120.

Quintana, C., Reiser, B. J., Davis, E. A., Krajcik, J., Fretz, E., Duncan, R. G., ... & Soloway, E. 2004. A scaffolding design framework for software to support science inquiry. *The Journal of the Learning Sciences*, *13*(3), 337-386.

Reiser, B. J. 2004. Scaffolding complex learning: The mechanisms of structuring and problematizing student work. *The Journal of the Learning Sciences*, *13*(3), 273-304.

Repenning, A. 2011. Making programming more conversational. In *Visual Languages and Human-Centric Computing (VL/HCC), 2011 IEEE Symposium on* 191-194. IEEE.

Repenning, A. 2013. Making Programming Accessible and Exciting. *IEEE Computer*, *46*(6), 78-81.

Repenning, A., & Ambach, J. 1996. Tactile programming: A unified manipulation paradigm supporting program comprehension, composition and sharing. In *Visual Languages, 1996. Proceedings., IEEE Symposium on*102-109. IEEE.

Repenning, A., & Ioannidou, A. 1997. Behaviour processors: layers between end-users and Java virtual machines. In *Visual Languages, 1997. Proceedings. 1997 IEEE Symposium on* 402-409. IEEE.

Repenning, A., Ioannidou, A. and Zola, J. 2000. AgentSheets: End-User Programmable Simulation. Journal of Artificial Societies and Social Simulation, 3(3).

Repenning, A., Smith, C., Owen, R., & Repenning, N. 2012. AgentCubes: Enabling 3D Creativity by Addressing Cognitive and Affective Programming Challenges. In *World Conference on Educational Multimedia, Hypermedia and Telecommunications* (Vol. 2012, No. 1, 2762-2771).

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., & Kafai, Y. 2009. Scratch: programming for all. *Communications of the ACM*, *52*(11), 60-67.

Richardson, V. (1998). How teachers change. *Focus on basics, 2*(C), 1-10.

Scaffidi, C., & Chambers, C. 2012. Skill progression demonstrated by users in the Scratch animation environment. *International Journal of Human-Computer Interaction*, *28*(6), 383-398.

Seehorn, D., S. Carey, B. Fuschetto, I. Lee, D. Moix, D. O'Grady-Cuniff, B. Boucher Owens, C. Stephenson, and A. Verno. 2011. *CSTA K-12 Computer Science Standards*. CSTA Standards Task Force.

Stephenson, C., & Wilson, C. 2012. Reforming K-12 computer science education… what will your story be? *ACM Inroads, 3*(2), 43-46.

Vygotsky, L. S. 1978. *Mind in society: The development of higher psychological processes*. Harvard university press.

Webb, D. C., Repenning, A., & Koh, K. H. 2012. Toward an emergent theory of broadening participation in computer science education. In *Proceedings of the 43rd ACM technical symposium on Computer Science Education* 173-178. ACM.

Werner, L., Denner, J., Bliesner, M., & Rex, P. 2009. Can middle-schoolers use Storytelling Alice to make games?: results of a pilot study. In *Proceedings of the 4th International Conference on Foundations of Digital Games* 207-214. ACM.

Wilson, C., Sudol, L. A., Stephenson, C., & Stehlik, M. 2010. *Running on empty: The failure to teach K-12 computer science in the digital age*. Association for Computing Machinery. Computer Science Teachers Association.

Wing, J. M. 2006. Computational Thinking. *Communications of the ACM*, *49(3), 33-35*.

APPENDIX

Scope and Sequence

Recommended scope and sequence for creating Games in Scalable Game Design:

	Games						
Computational Thinking Patterns	Frogger	Journey	PacMan	Sokoban	Ultimate PacMan	Space Invaders	Sims
Collision	N	P	P	P	P	P	P
Absorb	N	P	P		P	P	
Generate	N	P	P			P	
Transport	N						
Hill Climbing		N	N		P	P	P
Diffusion		N	N		P		P
Polling		N	N	P			
Pull				N			
Push				N			
Seeking					N		P
Collaborative Diffusion					N		
Script						N	
Multiple needs							N
Perceive/Act Sync							
Simulation Readiness							
Agent Attributes		N	N		P	P	P
Simulation Properties		N	N	P			P
Resources Available							
Curricula Available	X	X	X	X			
Tutorial Available	X	X	X	X	X	X	X

Key:
N: New skill based on scope and sequence implementation order
P: Previously encountered skills
X: Available on the Scalable Game Design wiki
F: Future resource under construction

Scope and Sequence (Continued)

Recommended scope and sequence for creating Simulations in Scalable Game Design:

	Simulations						
Computational Thinking Patterns	Forest Fire Simulation	Contagion Simulation	Predator-Prey - Basic	Predator-Prey - Intermed.	Predator-Prey - Advanced	Heat transfer simulation	Impulse wave generation
Collision	N	P	P	P	P		
Polling	N	P	P	P	P		
Perceive/Act Sync	N						P
Absorb			N	P	P		
Multiple needs			N	P	P		
Hill Climbing					N		
Diffusion					N	P	
Seeking					N		
Collaborative Diffusion					N		
Pull							
Push							
Generate							
Transport							
Simulation Readiness							
Agent Attributes	N	P	P	P	P	P	
Simulation Properties	N	P	P	P	P	P	
Resources Available							
Curricula	F	F	F				
Tutorials	X	X	X				
Sample Project Available	X	X	X	X	X	X	X

Key:

N: New skill based on scope and sequence implementation order
P: Previously encountered skills
X: Available on the Scalable Game Design wiki
F: Future resource under construction

Computational Thinking Patterns (Defined)

Generation To satisfy this pattern, an agent is required to generate a flow of other agents; for example, a bullet leaving a gun (ie: the gun agent generates a flow of bullets) or a car appearing from a tunnel are both examples of generation. In predator/prey models generation is used to create offspring.

Absorption This is the opposite pattern of generation. Instead of an agent generating other agents, an agent absorbs a flow of other agents (i.e. a tunnel absorbing cars). For example, absorption is used to program a predator eating its prey.

Collision This pattern represents the situation when two agents physically collide. For example, a bullet or a missile hitting a target creates a collision situation wherein the agents must react to being collided with. In the game Frogger, for example, if a truck collides with a frog, the frog must be "squished."

Transportation Transportation represents the situation when one agent carries another agent. For example, a turtle in Frogger carries the frog as it crosses the river. In ecological simulations, transportation can be used to have bees carry pollen, for example.

Push The push pattern is the pattern we see in the game of Sokoban. A player in Sokoban is supposed to push boxes to cover targets. As the player pushes the box in Sokoban, the box moves towards the direction (up, down, right or left) it is pushed.

Pull This pattern is the opposite pattern of push. An agent can pull another adjacent agent or any number of agents serially connected to the puller. For example, you can imagine that a locomotive pulls a large number of railroad cars.

Diffusion You can diffuse a certain value of an agent through neighboring agents with a diffusion pattern. For example, a torch agent can diffuse the value of heat through neighboring floor tile agents. The closest eight neighboring floor tile agents to the torch agent will have the highest value of heat, and tile agents that are further away from the torch agent will have a lower heat value.

Hill Climbing Hill climbing is a searching algorithm in computer science. A hill climbing agent will look at neighboring values and move toward the one with the largest value. Hill climbing can be found in the game of Sims or Pacman. In the game of Pacman, Ghosts chase Pacman by following the highest value of Pacman's scent that is diffused throughout the level. As with the torch above, the floor tiles around where Pacman is currently situated have the greatest scent value.