Teaching and Engaging with Debugging Puzzles

Michael J. Lee

A dissertation

submitted in partial fulfillment of the

requirements for the degree of

Doctor of Philosophy

University of Washington

2015

Reading Committee:

Andrew J. Ko, Chair

Katie Davis

Mark Guzdial

Julie Kientz

Program Authorized to Offer Degree:

Information Science

University of Washington

Abstract

Teaching and Engaging with Debugging Puzzles

Michael Jong Lee

Chair of the Supervisory Committee:
Associate Professor, Andrew J. Ko
The Information School

This dissertation describes Gidget, an online educational debugging game that is designed to engage and teach novices introductory programming concepts. Players solve puzzles throughout the game to help a robotic character complete its missions. These puzzles are actually debugging tasks, where players must inspect, modify, and test existing code to fulfill the goals that are written as test cases. Each level teaches a specific concept or set of concepts related to topics that are covered in introductory programming courses.

The dissertation defines the core principles that constitute an educational debugging game. Three controlled experimental studies show that the game is engaging to novice programmers. This was especially true when 1) the compiler/interpreter and its feedback messages were personified, 2) objects in the game attributed more purpose to the game goals, and 3) assessments were integrated seamlessly into the game. Another controlled experiment, where

participants were assigned to use one of three learning interventions, revealed that those who completed the Gidget game or an online tutorial on a website called Codecademy showed similar learning gains, with Gidget players doing so in about half the time.

Thousands of people have played Gidget through its development and public release. It has been shown to be appealing to a broad range of users independent of age, gender, education, or place of residence. A total of 68 teenagers from underrepresented groups in computing (i.e., females, and those from rural communities) took part in four, weeklong summer camps. With only about 5 hours of training playing through and completing the Gidget game, these teenagers were able to create a total of 210 of their own Gidget levels with minimal or no outside help. Furthermore, Gidget has attracted several thousands of players since its release. Registered players, composed of 54.8% males and 45.2% females, completed 0-37 levels playing between between 1 minute to 5.22 hours each.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I wish to thank my committee members for all their time and dedication. A special thanks to Dr. Andrew Ko, my chair and advisor, for taking me on as his first full student and spending countless hours helping me become the researcher I am today. His insight, patience, expertise, and concern for all of his students is admirable, and I could not imagine a better mentor. I would also like to give thanks to my committee members, Julie Kientz, Katie Davis, and Mark Guzdial for their service and taking the time to support me through my dissertation process.

I would also like to acknowledge the many other advisors I have been fortunate to work with throughout my academic career. First, I would like to thank Dr. David Kirsh for introducing me to research as an undergraduate student, and allowing me to gain invaluable experience teaching introductory programming and design courses. Next, I thank Dr. Robert Glushko, who demonstrated the importance of information organization and open standardization in service design. I also thank Dr. Kimiko Ryokai, who helped me identify my research interest and co-authored my first two research papers.

Finally, I would like to thank the many collaborators I worked with on launching the Gidget game online and with helping on summer camp outreach activities. At the University of Washington, this includes: Polina Charters, Fanny Luor, Michael Beswetherick, Nadav Ashkenazi, Steven Raden, Staffan Hellman, and Christina Xiao. At Oregon State University, this includes: Dr. Margaret Burnett, Dr. Irwin Kwan, Dr. Catherine Law, Faezeh Bahmani, William Jernigan, Amber Horvath, Jilian Laferte, Taylor Cuilty, and Sheridan Long.

This work was supported by the National Science Foundation under grants CNS-1240786, CNS-1240957, CNS-1339131, CCF-0952733, CCF-1339131, IIS-1314399, IIS-1314384, and OISE-1210205.

DEDICATION

To my parents, Sean Lee and Alice Lee,

for all their support and love.

1. INTRODUCTION

Programming is increasingly becoming an important 21st century skill. End-user programming is already quite prevalent in the workplace – some estimate that for every professional software developer, there are four non-professionals writing programs without any formal training or experience in programming (Scaffidi et al. 2005). In the USA alone, computer science related jobs are increasing at double the national average and are among the top paying fields (Bureau of Labor Statistics 2012), but there are not enough people trained to fill these roles. At current rates, it is estimated that there will be over one million unfilled computing jobs by 2020, which equates to a $500 billion opportunity (Bureau of Labor Statistics 2012). These numbers highlight the need to create more public interest in computing and acquiring the necessary skills to pursue these jobs not only through formal educational settings, but also through new types of discretionary educational resources.

In recent years, major efforts such as the Hour of Code and CS Education Week events have attracted millions of people, including celebrities and even the U.S. president, to try programming using many of the discretionary learning resources available for free online (Beres 2014). These resource include tutorial websites such as Codecademy (n.d.) and CodeSchool (n.d.), open-ended creative environments such as Scratch (Maloney et al. 2010) and Alice (Cooper, Dann, & Pausch 2000; Dann, Cooper, & Pausch 2011), and educational games such as Wu's Castle (Eagle & Barnes 2008) and LightBot (n.d.). Users of these systems report that they enjoy these informal resources more than traditional coursework because they allow for flexibility in how they learn, they provide a better sense of retention of the material (Boustedt et al. 2011), and they are more motivating, engaging, and interesting than traditional classroom courses (Cross 2006). Some of these attitudes can be attributed to these resources' use of game mechanics such as scaffolded materials, structured mastery learning, concrete goals, and extrinsic incentives such as badges (Young 2008). Furthermore, these online resources allow users to learn about programming in a safe environment at their own pace (Steffe & Gale 1995), which gives them the opportunity to clear up any of their negative misconceptions about programming or their ability to learn it, to something more positive (Charters et al. 2014).

1.1. The Problem

Unfortunately, these many online educational resources have three major issues: it is unclear 1) to what extent learners are engaged with the material, 2) whether they show measurable learning outcomes, and 3) who is actually using these types of resources. First, unlike traditional classrooms, learners in discretionary settings have the option to disengage with the content at any time. Traditional educational resources have peers and instructors that can help motivate or engage a struggling learner immediately, but most online resources do not. Therefore, knowing how to keep a learner engaged with the educational material is very important. If the learner decides the material is too boring, too easy, or too difficult, they may decide they do not like the subject, which may have long-lasting, negative consequences. Unfortunately, although there are many studies examining what learners find difficult and discouraging about learning programming, there are fewer works specifically examining what factors engage learners. This leads to a whole new set of pedagogical and design challenges, where players need to be sufficiently challenged to keep them interested and coming back, but not so much as to discourage them, all while actually teaching them. Knowing what engages learners, especially in the context of online learning, will be crucial in making effective educational tools.

Furthermore, although there are major efforts to attract more people to programming (Beres 2014) and a long history to make it more accessible to learners (Kelleher & Pausch 2005), educators struggle with understanding how to teach people programming (Guzdial 2014) in an effective and measurable way. Few (if any) of the many online resources report anything beyond the number of users that have signed up for their services and how many activities their users have completed. We do not know how long people interact with an activity, if they ever come back, or, most importantly, what they are learning, if anything. This lack of evaluation makes it unclear how useful these tools are beyond merely engaging learners for a brief period of time, which resources are actually successful at teaching coding, or what parts of these resources contribute to success or failure. Without this knowledge, we risk designing instructional tools that do not actually instruct learners (Garris, Ahlers, & Driskell 2002).

Figure 1.1. Screenshot of the (A) Gidget game start screen, and (B) main game interface.

1.2. A Solution

What if we could use the internet to effectively teach a wide range of people computer programming concepts at their own discretion and pace, while keeping them entertained? In this dissertation, I describe how my research does this using an online programming game called Gidget (see Figure 1.1). Gidget introduces programming (and debugging) to novices in a low-barrier, engaging way that produces measurable learning outcomes. Through my research, I explore how different design elements in a game affect people's engagement with the activity and measure their learning outcomes. This leads to my thesis:

> *An online game can engage and measurably teach programming concepts covered in a typical introductory computer science (CS1) course to a wide range of learners.*

More specifically, the research questions (and related sub-questions) that arise from this thesis are:

- RQ 1. Do players of an educational debugging game show measurable signs of engagement playing the game?
    - RQ 1.1. How does (compiler/interpreter) feedback affect players' engagement?
    - RQ 1.2. How do goals affect players' engagement?
    - RQ 1.3. How does explicit testing in the game affect players' engagement?

- RQ 2. Do players of an educational debugging game show measurable learning of programming concepts covered in a typical introductory programming (CS1) course?
    - RQ 2.1. To what extent are players able to transfer their understanding of fundamental CS1 concepts from the Gidget language to pseudo-code?
- RQ 3. Who is playing the educational debugging game?
    - RQ 3.1. Does the game appeal to underrepresented groups in computing?
    - RQ 3.2 What are the demographics of the people who are choosing the play the game?

In the following sections, I describe my approach, provide some definitions of terms used throughout the dissertation, list my contributions, and detail the contents of the following chapters.

1.3. Research Approach

The work outlined in this dissertation draws from practices in Human-Computer Interaction (HCI) research. I used a participatory design process involving representatives from several stakeholder groups to inform the design of the first version of the game. This was done to ensure the game would appeal to a wide audience, and included a middle school student, several high school students, a college student, a graduate student, a computer science educator, and a college graduate with a non-technical job.

Using an iterative interaction design approach (Frayling 1993; Zimmerman et al. 2007), I continued to modify and update the game as I conducted and finished more studies. My design decisions for each iteration of the game were based primarily on empirical evidence gathered from controlled experiments (i.e., A/B or A/B/C testing) that were conducted to answer the specific research questions listed in the previous section about the game's effect on its users. The game described in this dissertation is the result of the findings from these controlled experiments.

1.4. DEFINITIONS

This dissertation uses several terms that will be defined here for clarification. First, both the Gidget game and its eponymous protagonist will be referred to as Gidget throughout the paper – with the context clearly differentiating between the two. Seven design principles (detailed in Chapter 3) define what constitutes an educational debugging game and how Gidget fits these properties. In the game, players must solve puzzles (i.e., fix code defects) to pass each level. Defects, errors, and bugs all refer to some code in the game that results in a fault or failure, preventing the player from completing the level.

The studies described in this dissertation focus on a specific group of players. A *novice programmer,* the primary target audience of Gidget, refers to someone who does not have any experience (either formally or informally) with writing or reading computer code. Conversely, those with *any* programming experience are referred to as *experienced programmers,* and the extent of their experience or ability with programming is not distinguished since they are not the primary focus of this dissertation. The novice programmers from our studies will be primarily referred to as *learners*, but depending on the context, may also occasionally be called: *players*, *users*, *campers*, and *participants*.

1.5. CONTRIBUTIONS

This dissertation provides a number of contributions:

- A description of seven design principles that define the components needed to make an educational game that effectively engages and teaches introductory programming concepts to novices.
- Evidence that novice programmers are engaged with an educational game when:
    - The computer compiler/interpreter is personified.
    - The game goals are made more purposeful using specific types of data elements.
    - In-game assessments (i.e. exams) are added at the end of each subject module.
- Evidence that novice programmers can effectively and measurably learn introductory programming concepts using an educational game.

- Evidence that an educational programming game can attract a wide range of players.
    - Knowledge about who is attracted to play the game.

To summarize, this dissertation describes Gidget, a novel approach to teach programming through debugging puzzles in a way that appeals to a broad audience, engages its users, and shows measurable learning outcomes.

1.6. Outline of Dissertation

Figure 1.2 provides a visualization of the content in this dissertation, depicting the studies conducted to answer the research questions above. The studies described in Chapters 4 through 7, and parts of Chapter 8, were all previously reported elsewhere in peer-reviewed publications with myself as the first author and lead researcher. In these chapters, I use the inclusive pronouns *we* and *our* to describe the work for consistency and to acknowledge my coauthors' contributions.

- Chapter 2 – Related Work
    - Chapter 2 summarizes the works closely related to this dissertation, including different technologies used to teach programming, and the use of games as a medium to teach programming.
- Chapter 3 – The Gidget Game
    - This section describes the instrument used in all the studies detailed in this dissertation. It includes an explanation of the game's interface, its curriculum, the optional puzzle designer interface, automated data collection, and the relevant differences in the various iterations of the game throughout its development.
- Chapter 4 through 6 – Studies About Learners' Engagement

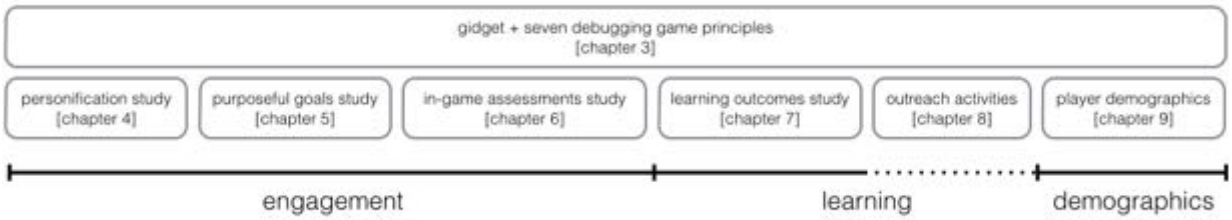


Figure 1.2. A map of the contributions in this dissertation and their corresponding chapters.

    - These chapters detail three different controlled experiments exploring the factors that affect players' engagement with the game. The studies in Chapters 4 and 5 primarily manipulate the presentation of textual and graphical information to see how it affects learners. The study in Chapter 6 manipulates the inclusion or exclusion of assessments (i.e., exams) throughout the game to see how it affects learners.
- Chapter 7 – Study About Learners' Learning Outcomes
    - Chapter 7 measures the pre-test and post-test scores of learners before and after playing Gidget (and two other learning activities) to see how it affects learners.
- Chapter 8 – Outreach & Public Deployment Demographics
    - Chapter 8 reports on two activities used to reach a wide range of players for Gidget. The first is an outreach activity using Gidget at summer camps for teenagers, specifically focusing on underrepresented groups in computing. The second is a public release of the game and a demographic overview of its players.
- Chapter 9 – Conclusions and Future Work
    - Before listing some final remarks, this chapter summarizes the dissertation as a whole and provides ideas about the future direction that this work can talk so that educational games can continue to be a relevant player in computing education.

2.RELATED WORK

2.1. TECHNOLOGIES TO TEACH PROGRAMMING

This research follows a long tradition of efforts to create programming environments for beginners (Kelleher & Pausch 2005). Many of these technologies have focused on increasing learner motivation by incorporating new factors to entice learners to explore computational activities. For example, Logo (Papert 1980) and EToys (Kay 1997) both created computational spaces for children to explore music, language, and mathematics; Light-bot (n.d.) pushed players to take the robot's point-of-view of the environment to successfully navigate through levels; Playground (Fenton & Beck 1989) and LEGO Mindstorms (Barnes 2002) had similar goals, enticing children with the modeling and simulation of phenomena from the world or actually enabling them to write programs that sense the world. These approaches and others like them seek to entice learners with their intrinsic curiosity about the world and its processes.

Other approaches have motivated children with opportunities for self-expression. Play (Tanimoto & Runyan 1986), My Make Believe Castle (Logo 1995), Hands (Pane, Myers, & Miller 2002), ToonTalk (Harel 1991), Stagecast (Smith, Cypher, & Tesler 2002), Toque (Tarkan et al. 2010) and others all focus on enabling learners to create novel animations and games. Similar efforts have been made at the college level with projects such as Georgia Computes! (Bruckman et al. 2009) and Game2Learn (Barnes et al. 2007), which encourages students to create and test their own games. Examples include Bug Bots (Chaffin & Barnes 2010) – a game where players attempt to repair robots by dropping tiles into a flowchart representing a computer program – and Virtual Bead Loom (Boyce & Barnes 2010) – a game where students are encouraged to learn looping functions to create bead artwork instead of placing beads one at a time. Other systems that have added to these self-expression goals the ability to share the content one has created. For example, MOOSE Crossing invites learners to create characters and spaces in a virtual, multi-user text-based world (Bruckman 1997); more recently, Storytelling Alice (Kelleher, Pausch, & Kiesler 2007) and Scratch (Maloney et al. 2010) have focused on enabling learners to tell and share stories. Kelleher et al. (2007) were one of the first to demonstrate that

opportunities and affordances for storytelling can significantly improve learners' motivation to program. My work follows these traditions, but provides learners with the story, allowing them to contribute to its progress by interacting with a character in a game.

While all of the systems discussed thus far aimed to increase motivation, several systems have aimed to lower demotivating factors in programming tools. Such approaches include simplifying the textual programming language syntax (Bruckman 1997; Papert 1980), designing languages that mimic how children describe program behavior (Pane, Myers, & Miller 2002), preventing syntax errors entirely by designing program construction interfaces that use drag and drop interactions (e.g., Kay 1997; Maloney et al. 2010) or form filling (Logo 1995; Smith, Cypher, & Tesler 2002) rather than text. Others have attempted to simplify the debugging of programs by enabling learners to select "why" questions about program output (Ko & Myers 2004; Kulesza 2009). My research follows the same vein as these projects, aiming to mitigate factors inherent to programming that would diminish motivation by changing the programming environment.

2.2. Using Games as a Tool to Teach Programming

This research also follows a long tradition of using games as a motivational tool to teach computer programming. Games have been used to teach programming as informal learning interventions, have shown to positively effect motivation (Garris et al. 2002; Cliburn 2006; Malone 1981; Gee 2003), and attract people to pursue computing education (Papastergiou 2009). Learners' motivation is of critical importance and can have a major impact on their learning (Farthing 1997; Armstrong et al. 1998). Moreover, motivation is crucial in programming education, where learners are required to actively apply their knowledge (Feldgen & Clua 2004). Therefore, understanding what motivates people to start and continue to learn programming can potentially lead to better quality learning experiences and new ways to attract people to programming. Gaming can be used to provide a low-pressure, non-threatening, and engaging medium to learn new skills such as programming (Griffiths 1997). Well-designed games could share the attributes of a good teacher: they provide immediate feedback of success of failures, assist in learning at different rates, and offer opportunities to practice (Gentile 2009). My

research follows the same vein as these projects, aiming to provide a low-barrier, safe environment that uses best practices in education to teach players computer programming with little or no human intervention.

Studies using the ARCS model (attention, relevance, confidence, and satisfaction) (Keller & Suzuki 1988) have shown that it is important to raise and maintain motivation of learners in the very early stages of learning computer programming, as this is the moment when motivation changes the most (Tsukamoto et al. 2008). Games have been suggested to maintain learners' motivation in programming through the early stages of learning. Learning by playing games is becoming increasingly recognized in research and educational practice for its their engaging properties (Garris et al. 2002; Gee 2003), with some empirical evidence showing that games can be effective tools for enhancing learning and understanding of complex subject matter (Cordova & Lepper 1996; Ricci et al. 1996). Moreover, gaming has been shown to be of interest to a broad range of people and not only to those who are already engaged in technological studies (Papastergiou 2009). Recent statistics reveal that the average gamer in the USA is 37 years old (with a mean of 12 years of gaming experience), 97% of youth play video games, 42% are female, and the number of people over 55 years old playing games is increasing (Ito et al. 2009; newzoo.com 2011; NPD 2011).

Games have a rich history in education. For example, in his studies, Clibrun (2006) found that when given the option to use a game or non-game assignment for the course covering the same topic, nearly 80% of the students opted to use the game, even though the average grade received was 6% lower than non-game assignments. Although this was the case, the majority of students still reported preferring using the game-based assignments, suggesting that games do indeed provide psychological motivation and increases course enjoyment, even though they may not improve students' scores (Cliburn 2006). In addition to preferring games as homework assignments, there is evidence that the use of web and game programming examples in place of classical programming examples in formal education settings have been found to be more motivating for novice programmers (Feldgen & Clua 2003). These findings demonstrate people's general preference towards games and game elements as substitutes for other activities, possibly making educational content and concepts more relatable or engaging to learn.

Games also appeal to both genders, leveraging enthusiasm for entertainment and social relevance (Barnes et al. 2007), and appear to have equal benefits for both males and females. For example, a study found that despite males' greater involvement with liking and experiencing computer gaming, and generally having a greater initial knowledge of computers, the learning gains in the experiment were not significantly different from females', and that the game was equally motivating for both genders (Papastergiou 2009). These findings suggest that games can provide a neutral educational space for learning, where anyone, regardless of experience or gender, can benefit.

2.3. Game Based Learning, Gamification, and Related Theories[1]

In addition to prior works about games used in practice, there are several related theories of learning that explore how people can use games or game-based elements to effectively learn new things. In particular, game based learning (GBL) explores how games with defined learning outcomes contribute to pedagogy (Prensky 20015). This is often confused with "gamification," which is the use of game thinking and game mechanics in a *non-game* context to engage (and sometimes teach) users (Huotari & Hamari 2012). Both GBL and gamification have been used in efforts to teach programming. For example, Minecraft classes add directed learning tasks into playing the game (Schifter & Cipollone 2013; Short 2012; Zorn et al. 2013) and Scratch Online (n.d.) incorporates "favoriting" and "loving" uploaded projects, which rewards users by having popular projects featured on their front page. Prior work has also shown that summer camps using games or tools/activities with gamification are great at engaging their users (Bruckman et al. 2009; Webb & Rossen 2011; Zorn et al. 2013), but that all of these required instructional scaffolding by teachers for learners to succeed. Similar to these related projects, my research uses a game specifically designed with game based learning objectives to teach users computer programming in a low-barrier, safe environment – but with little or no human intervention.

Games and gamified elements may include a range of learning theories in their design: some constructivist (allowing learners to participate and experiment in non-threatening scenarios), some experiential (learning by doing), and some situated (providing relevant context or setting;

[1] Parts of this section have been adapted from my ICER 2015 publication (Lee & Ko 2015).

for multiplayer, learning takes place alongside social interaction and collaboration). Some games open-ended, creative games such as Minecraft (n.d.) and programming environments such as Alice (Kelleher, Pausch, & Kiesler 2007) and Scratch (Maloney et al. 2010), are largely unstructured and allow users to explore, tinker, and create content that is meaningful for themselves. These attributes align with constructivist theories of learning through hands-on experience (Steffe & Gale 1995) and constructionist ideas of learning through construction of meaningful projects (Papert & Harel 1991). My game uses the ideas from both constructivism and constructionism, where learners first construct knowledge for themselves through the experience of examining and solving programming puzzles throughout the game's curriculum, and further develop their knowledge by then creating their own projects through tinkering and exploration after completing the game.

3. THE GIDGET GAME

Gidget is a web application that is playable in a web browser (see Figure 3.1) and developed using the seven design principles outlined in Table 3.1[2]. We derived these seven principles by drawing from best practices in game design, educational technologies, learning sciences, help systems, and by observing our players interact with earlier iterations of Gidget. Table 3.1 defines each of the principles in more detail and lists the related studies (and chapter numbers) using Gidget. The description of the game in this chapter also highlights principles when applicable.

The game is motivated by a story: there has been a chemical spill from a factory and Gidget, a small robot capable of identifying and solving problems with programs, has been deployed to clean up the area (*P2-game;* Figure 3.2). Unfortunately, Gidget was damaged in transit, and is only able to provide code (Figure 3.1-A) that partially, but not completely solves each level's goals (*P3-fallible;* Figure 3.2). It is the player's job to help the robot by diagnosing and fixing

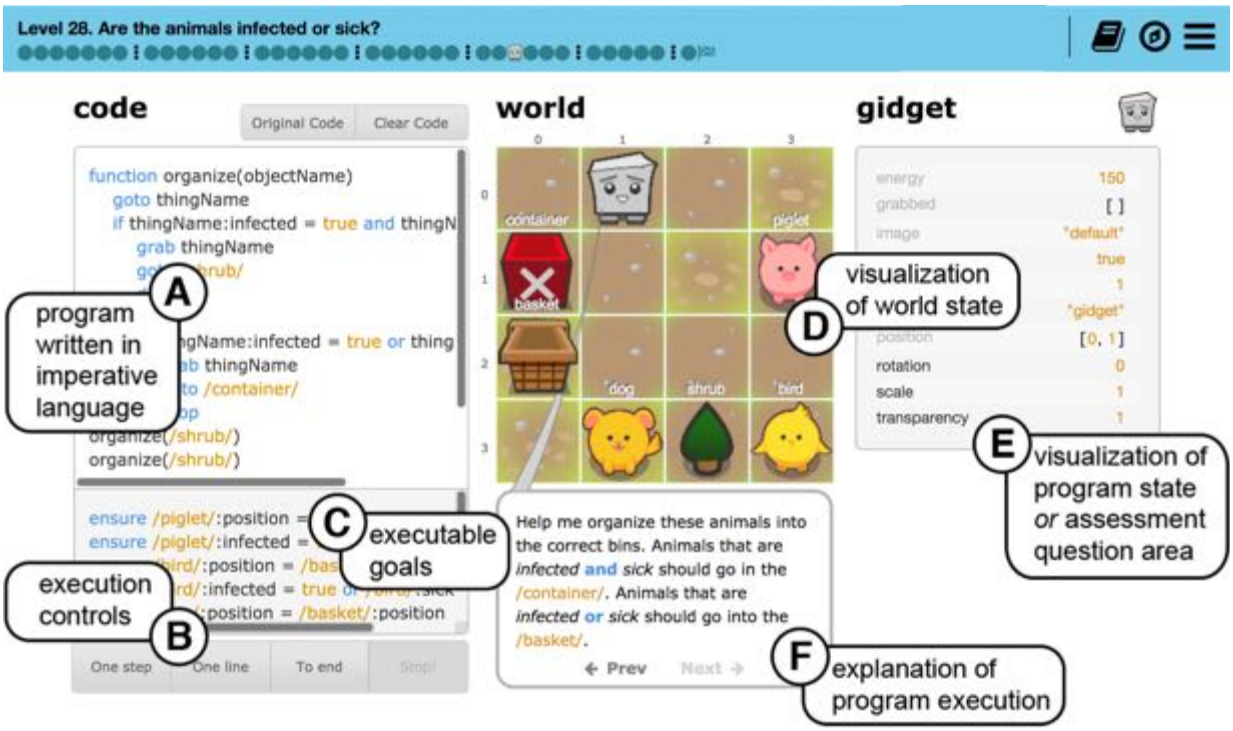


Figure 3.1. Screenshot of Gidget with added callouts on different interface elements.

[2] This table and chapter have been adapted from my VL/HCC 2014 publication (Lee et al. 2014).

the faulty code in each level (*P1-debug*) to satisfy each level's mission goals (*P4-goals;* Figure 3.1-C) in the form of assertions about the game's world state.

Gidget uses an imperative, Python-like programming language designed specifically for the game (the complete language grammar is described in the appendix). The language supports dynamically typed-variables, Boolean operators and expressions, conditionals, mathematical operators, loops, objects, functions, and domain-specific keywords for the game characters to interact with their world. These interactions primarily include finding things in the world (Figure 3.1-D), going to them, checking their properties, and carrying them to other places on the grid. In some cases, objects have their own abilities, which Gidget can call as functions. After each execution step, the effect of these commands are shown in the 'program state' panel (Figure 3.1-E) and explained by Gidget (Figure 3.1-F) to reinforce the semantics of each command (*P5-*

Table 3.1. Seven design principles for an educational debugging game

Principle	Description
P1-debug	**Debugging First:** Encourage learners to learn programming concepts by debugging existing programs before creating new programs. Unlike many other educational technologies where creation occurs immediately (Kelleher, Pausch, & Kiesler 2007; Maloney et al. 2008), our approach provides nearly complete, but broken programs for learners to debug and fix before moving onto the more demanding task of creating new puzzles from scratch.
P2-game	**Game-oriented:** To make the environment be engaging to those who want to be entertained by solving puzzles (Cao et al. 2013, Lee & Ko 2011, Lee & Ko 2012, Lee, Ko, & Kwan 2013), not just engaging to those who want to learn programming, it should feel like a game, drawing upon games' combination of interactivity, story, and objectives to benefit learning (Gee 2003).
P3-fallible	**Computers as helpful but fallible:** Frame computers as helpful but fallible collaborators. This is in contrast to other educational environments, which often frame the compiler, interpreter, development environment, and other programming tools as all-knowing, authoritative figures, which can be discouraging for novice programmers (Lee & Ko 2011). The study described in Chapter 4 supports this principle.
P4-goals	**Embedded goals:** Give learners an explicit goal as scaffolding (Ram & Leake 1995). Provide one specific game goal – debugging faulty code – so that learners are focused and not distracted by additional objectives that can be distracting and negatively affect performance (Anderson et al. 2011). The study described in Chapter 5 supports this principle.
P5-instruction	**Embedded instruction:** Provide embedded instruction, with specific learning objectives, a planned curriculum, and an explicit, sequenced set of instructional materials and tasks (Ellis 2005, Lee, Ko, & Kwan 2013). This contrasts with open, creative environments, where learners are left free to explore at will (Kelleher, Pausch, & Kiesler 2007, Maloney et al. 2008, Monroy-Hernandez & Resnick 2008). The study described in Chapter 6 supports this principle.
P6-help	**Scaffolded help:** Deliver, on request, in-game help, including "Idea Garden" (Cao et al. 2012, Cao et al. 2013) help that provides incomplete examples, problem-solving strategies, and higher-level programming concepts to enable learners to help themselves
P7-gender	**Gender inclusiveness:** Females represent 42% of all video game players in the USA (ESA 2011), but are seriously underrepresented in computing fields (NCWIT 2010). We aim at this problem by building on best practices for reaching both males and females (e.g., Burnett et al. 2011, Subrahmanyan 2007, Werner, Hanks, & McDowell 2004), such as avoiding competitive objectives and using a gender-neutral protagonist.

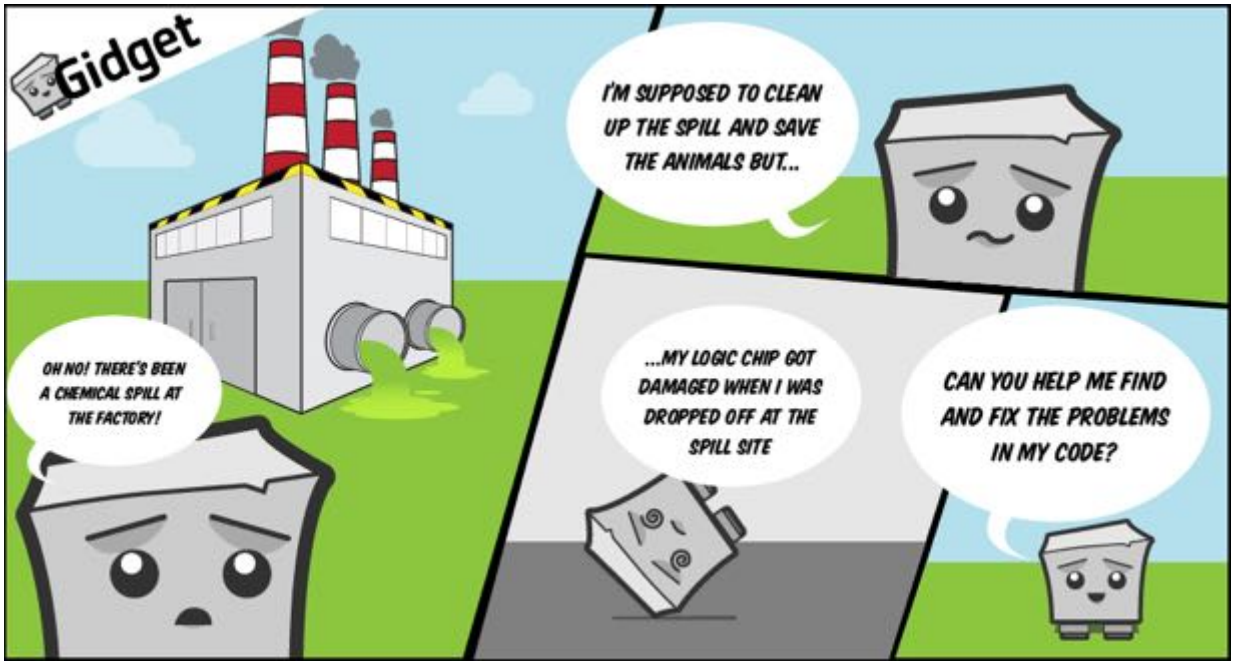


Figure 3.2. Screenshot of Gidget's story when the game is first started.

instruction). Each step costs Gidget 1 unit of 'energy' (displayed at the top, right of Figure 3.1-E), which forces players to consider how to write efficient programs that can be solved using the allocated amount of energy. Different levels may start with different amounts of energy, and restarting the level resets the energy units back to its original value.

To aid the players with debugging, the game offers four execution controls: *one step, one line, to end,* and *stop* (*P1-debug;* Figure 3.1-B). These controls function similarly to conventional breakpoint debuggers, allowing players to run parts of the program or all of it, halt the program, and edit code at any time. The *one step* button evaluates one compiled instruction, displaying text explaining the execution of the step. The *one line* button evaluates all steps on one line of the code, just as a breakpoint debugger does, jumping to the final output of that line. The *to end* button evaluates the entire program and the goals, animating each step in quick succession. If any errors are encountered, the program execution pauses, feedback is provided, and the player is given the option to restart the level or continue execution. The *stop* button allows the player to halt the program and edit code during any part of the execution. When the learner uses *one step* or *one line*, Gidget provides a detailed explanation of the execution of each statement in the program, highlighting changes in the runtime environment. This serves as the

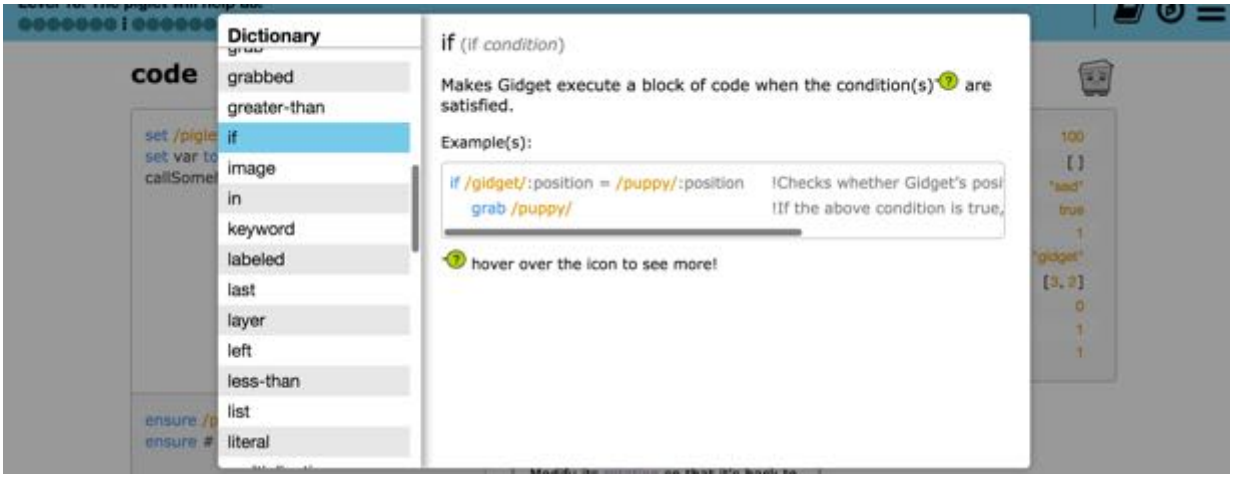


Figure 3.3. Screenshot of Gidget's dictionary/glossary.

game's primary instructional content, explicitly teaching the language syntax and semantics (*P5-instruction* & *P6-help*).

The game features several forms of scaffolded help to assist learners to succeed on their own (*P1-help*). On first load, the game shows an interactive tutorial that goes over the major interface elements to help learners begin playing Gidget. The help system also includes an in-game reference guide that provides explanations and examples of each command in the language, along with information about programming concepts such as variables, functions, the stack, and loops. The reference guide is available as a standalone dictionary/glossary (Figure 3.3) or as tooltips that appear when hovering over tokens in the code editor (Figure 3.4-A). The game also uses the Idea Garden help system (Figure 3.4-B; Cao et al. 2011) to detect context-sensitive

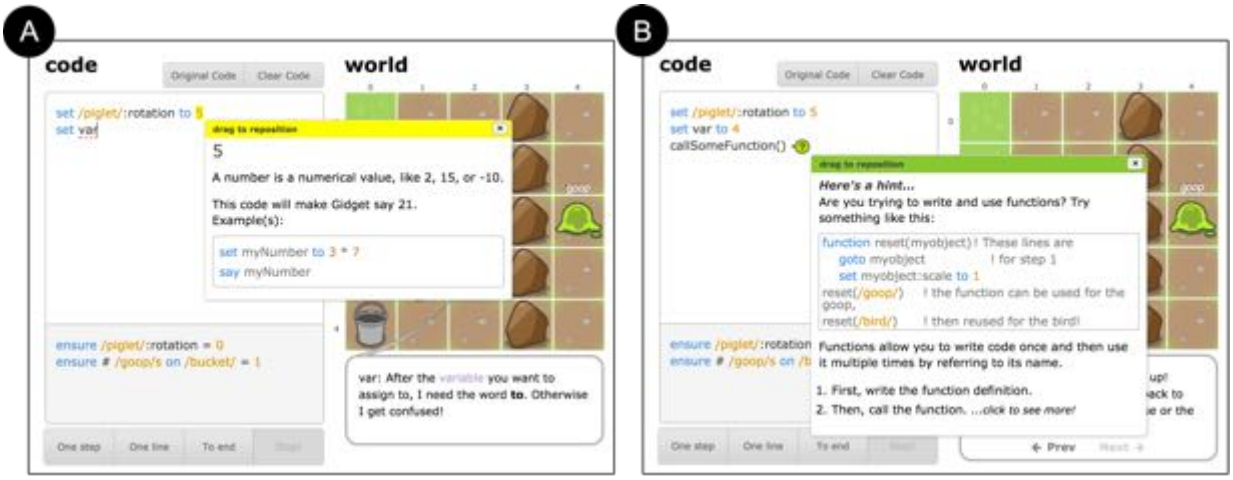


Figure 3.4. Screenshots of Gidget's (A) Tooltips, and (B) IdeaGarden help tools.

programming anti-patterns related to the learners' code (Jernigan et al. 2015), and the AnswerDash help system (Figure 3.5) which allows players to click on any part of the interface to ask questions about it or read responses to others' queries. Finally, the game's code editor provides keystroke-level feedback about syntax and semantics errors, underlining erroneous code in red and explaining the problem in Gidget's speech bubble (Figure 3.6).

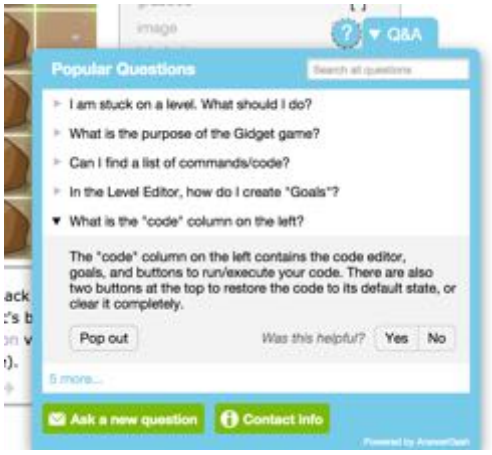


Figure 3.5. Screenshot of the AnswerDash interface in Gidget.

Gidget's graphics, text, and game goals were all designed to be gender-inclusive (*P7-gender*). The game's story integrates socially relevant themes (i.e., cleaning a chemical spill and saving animals), helping a partner, and provides challenge through puzzles—all of which have been shown to appeal to both genders (Reinecke, Trepte, & Behr 2008). Throughout the game, Gidget does not use gendered pronouns and remains androgynous, allowing the learner to decide how to characterize the game's protagonist to their individual preference. Gidget also avoids game mechanics, like achievements or competition, that would possibly disengage females (Yee 2006). Following the premise that language impacts culture, it eschews violence-oriented terminology (e.g., players "remove" a game object instead of "destroying" it; players "run" or "stop" a program instead of "executing" or "killing" it) (Misa 2010). Finally, its collection of scaffolded help offers information in the "selective" and "comprehensive" style statistically favored by males and females, respectively (Meyers-Levy 1989).

Figure 3.6. Gidget's syntax highlighting (left) and explanation of the error (right).

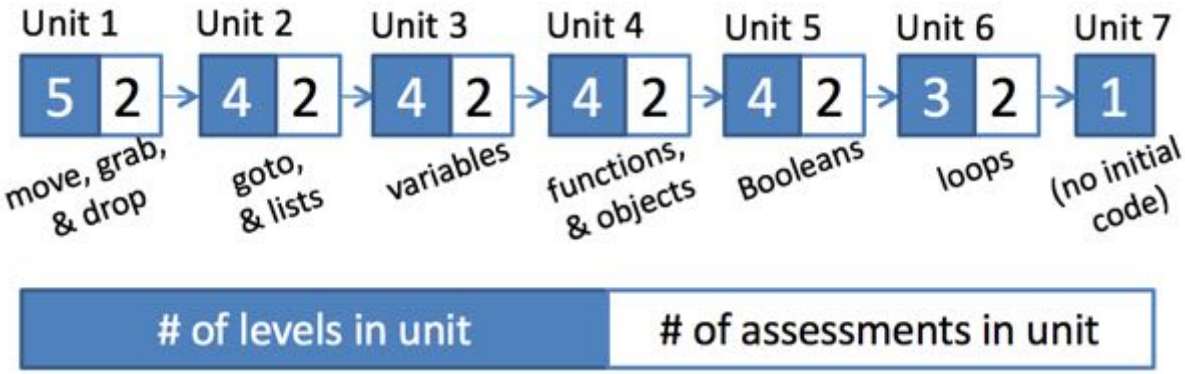


Figure 3.7. Map of the curriculum's units, topics, and number of levels.

3.1. The Game Curriculum

The game consists of 7 units with a total of 34 levels (see Figure 3.7). Each game level teaches a particular programming concept (*P5-instruction*), challenging the player to find and fix the defects in each level's program so that it passes the provided goals, which are executable test cases (all of the levels are described in detail in the appendix). Following the mastery learning paradigm (Pear 2004), each of the game's levels is designed to be passable only if the learner has grasped a particular concept in the game's programming language. Unit 1 focuses on moving Gidget and other objects around in the world by using simple keywords such as *up, down, left, right, grab*, and *drop*. Unit 2 furthers the ideas from the previous section, introducing the *goto* keyword, and working with lists. Unit 3 introduces variables, types, and values. Unit 4 presents the declaration and use of functions and objects. Unit 5 shows how to use Boolean values, expressions, and logic. Unit 6 focuses on loops. Finally, Unit 7 does not teach any new concepts, but challenges the player to write solutions from scratch to satisfy the level's goals. The last two levels in each unit are designed to be a cumulative overview, requiring the learner to recall and use the keywords and concepts covered in that unit.

Each level starts with Gidget briefly explaining the level's objective and providing hints about which concepts to use. The order of units and the sequence of levels was designed iteratively based on curricula found in CS1 textbooks (Deitel & Deitel 2005; Felleisen et al 2001; Lewis & Loftus 2005; Tew 2010; Zelle 2004), pilot testing with novice programmers, and

collaborators'[3] cumulative experience teaching CS1 courses. A list of overall learning objectives drove the creation, consolidation, and refinement of the levels (Bjork 1999). Each level was designed to address one or two specific learning objectives related to the language syntax or semantics. The sequence of levels was also influenced by the game story, and by the language itself (since certain keywords and concepts are easier to understand once other concepts have been learned). This sequence was validated by testing with participants in-person and online by observing that the order of levels was not a barrier in their progress through the game. Additionally, we validated the curriculum as engaging to online adult participants (*P2-game;* Lee, Ko, & Kwan 2013) and that it positively affected their attitudes towards programming, regardless of age, gender, or level of education (Charters, Lee, Ko, & Loksa 2013).

3.2. The Gidget Puzzle Designer[4]

Learners are given access to the game's puzzle designer once they complete the curriculum (see Figure 3.8). The puzzle designer allows them to create, save, modify, and share new levels using the Gidget language. The puzzle designer is an interface that allows the player to write code for new levels' behavior, add introductory text to the level, change the size of the world, set the goals and original code for the level, and view the usable graphics and sounds in the game (all these are listed in the appendix). It also introduces the concept of event handling (i.e., having objects in the game wait for a condition before running a code block), which was not covered in the game curriculum. In addition to creating levels from scratch, learners can also click on the puzzle designer's option menu to look through all the levels they had passed in the game's curriculum. The puzzle designer provides an option to duplicate these levels so that players can look through and modify the level's initial code, broken code, and solution code.

3.3. Automated Data Collection

The game automatically logs several user interactions within the game for each player. In addition to the total number of levels completed, the game logs the following for each level: the

[3] In addition to myself, collaborators included Information Science and Computer Science professors.

[4] More details about the Gidget Puzzle Designer can be found in Chapter 7.2.3.

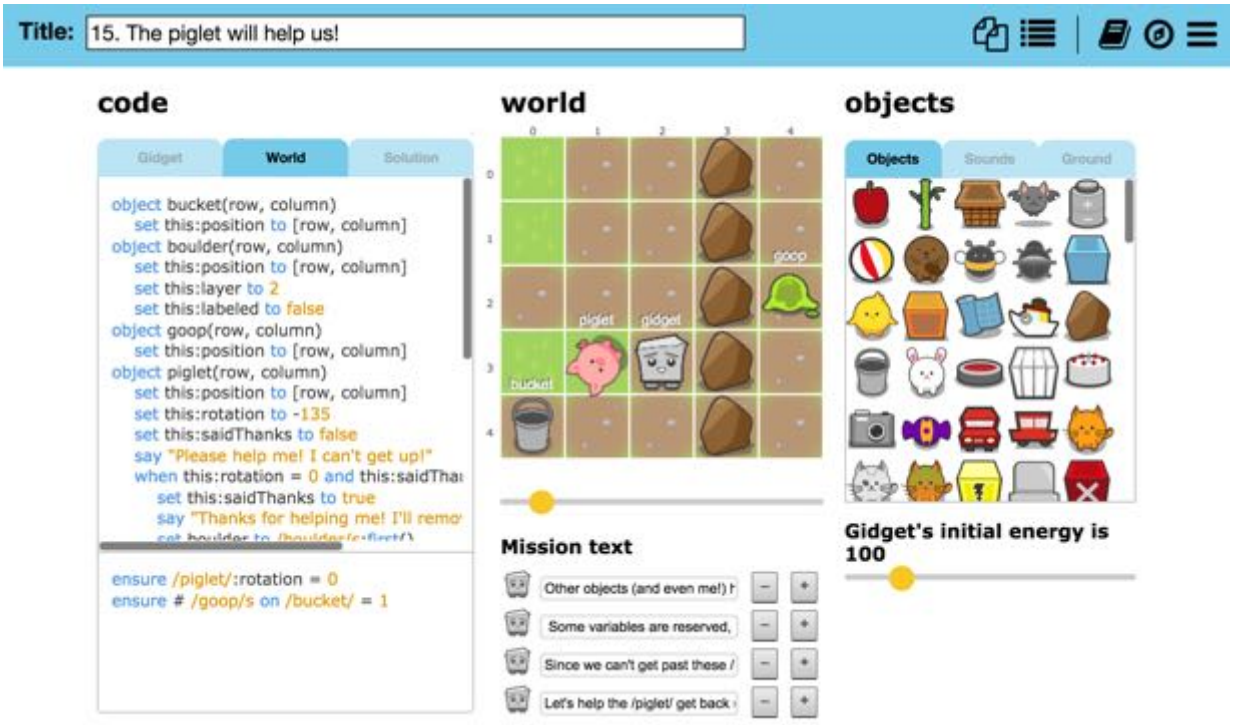


Figure 3.8. Screenshot of the Gidget Puzzle Designer editing one of the default levels.

time spent on the level, the number of times a tooltip appeared, the number of times the dictionary was used (and which term was looked up), the number of times the tutorial was triggered (and the number of steps/slides viewed), the mouse cursor position (and time spent dwelling on certain interface elements), the total time spent editing code, all the code versions, actions within the code versions (e.g., deletion events, cut/copy/paste events, and undo events), when and how many times each of the execution buttons were pressed, and each time the "clear code" and "restore original code" buttons were pressed.

Throughout several studies, the game also collected data from users in the form of questionnaires that were administered at the end of the game or when the user decided to quit. In addition to demographic data (e.g., gender, age, location, education), the questionnaires asked about people's prior programming experience, attitudes towards programming (before and after playing the game), what they liked most and least about the game, whether or not they felt compelled to help the robotic character while playing the game, and whether or not they would recommend the game to a friend.

3.4. VARIATIONS IN VERSIONS

In addition to the experimental manipulations for the controlled experiments discussed in later chapters, Gidget went through several revisions throughout its development (for example, Figure 3.9 shows the first version of Gidget). This section lists the major differences among the various Gidget versions that are relevant to the material covered in this dissertation:

- The studies described in Chapters 4 and 5 used a simpler imperative programming language that had 7 basic commands that allowed the game's protagonist to find, identify, pick up, move, compare, and drop objects around in the world. This was replaced by the more expressive language modeled after Python that was described earlier in the first section of this chapter.
- The studies in Chapters 4 and 5 had a total of 16 levels, where the first 9 focused on teaching the 7 basic commands in the robot's syntax grammar and variations. The subsequent 9 levels taught useful design patterns for composing these commands to achieve more powerful behaviors. This was replaced by the curriculum described in Chapter 3.1.

Figure 3.9. Screenshot of the first version of Gidget using the original language.

- The studies in Chapters 4 and 5 used a slightly different set of execution buttons. Gidget's earlier version of the *all steps* button functioned like the current *to end* button, executing all of the level's code and goal while animating each step in quick succession. Moreover, Gidget's earlier version of the *to end* button functioned exactly like the latest *all steps* button, but only showed the final state of the code execution and goal checking without animating any of the intermediate steps. The studies in Chapters 4 and 5 (which used the earlier simpler programming language) did not include the *stop* button.
- The code pane for the studies in Chapters 4 and 5 included a button labeled "?" on the top-right portion of the code pane, which opened up a syntax guide (also labeled as "cheat sheet") that described the different language commands and syntax. In subsequent studies using the updated language, this functionality was replaced by a dictionary button that opened up a glossary of terms, definitions, and examples (see Figure 3.3).
- The studies in Chapters 4, 5, and 6 displayed a 9-slide tutorial of static images when a player first started the game. This was replaced by the interactive tutorial described in the first section of Chapter 3 and covered the same material.
- The study in Chapter 6 removed several of the execution/communication visualizations in the game. For example, the interface integrated the execution buttons into the code pane and no longer included the players' speech bubble and character avatar. Moreover, Gidget's speech bubble originated from the character in the world pane instead of having a separate and redundant image of the character.
- The study in Chapter 6 introduced the level progress indicator/map in the interface, allowing players to see the total number of levels and how far they had progressed in the game.
- The current version of the game uses a completely updated set of consistently-styled character images and updated interface aesthetics (see Figures 1.1 and 3.1).
- The current version of the game no longer includes any questionnaires.
- The current version of the game allows players to save their progress by creating user accounts. The account creation process asks for players' name, email address, age, and gender. Google analytics collected additional aggregate information including returning vs. new visitors, and location (e.g., city, state, country) of users.

4. EFFECT OF PERSONIFIED FEEDBACK ON ENGAGEMENT[5]

This chapter describes the first of three studies that addresses RQ1 – do players of an educational debugging game show measurable signs of engagement playing the game? More specifically, how does compiler/interpreter feedback affect players' engagement playing the game?

4.1. Background and Motivation

For most beginners, the experience of writing computer programs is characterized by a distinct sense of failure. The first line of code beginners write often leads to unexpected behaviors, such as syntax errors, runtime errors, or program output that the learner did not intend. While all of these forms of feedback are essential to helping a beginner understand what programs are and how computers interpret them, the experience can be quite discouraging (Ko, Myers, & Aung 2004; Ko & Meyers 2009) and emotional (Kinnunen & Simon 2010).

These findings have significant implications for computing education. To many learners, error messages are not perceived as actionable facts, but as evidence that they are incompetent and that the computer is an all-knowing, infallible authority on what is right and wrong (Beckwith, Burnett, & Cook 2002). Even in programming environments designed for beginners such as Alice (Kelleher, Pausch, & Kiesler 2007) and Scratch (Maloney et al. 2010), where syntax errors are impossible and most runtime errors are avoided by having the runtime do something sensible rather than fail, the communication between the learner and the computer is framed as one-way: the computer does not express its interpretation of the code, it simply acts upon it without explanation. These relationships between learners and programming tools are more command-and-control than collaboration.

And yet, how people perceive their relationship to a computer is a critical determinant of not only their attitudes towards computers, but also their performance in using them to accomplish tasks (Klein, Moon, & Picard 1999). Moreover, studies have shown that people expect computers to behave with the same social responses that people do (Nass 2000); for example, automated

[5] This chapter has been adapted from my ICER 2011 publication (Lee & Ko 2011).

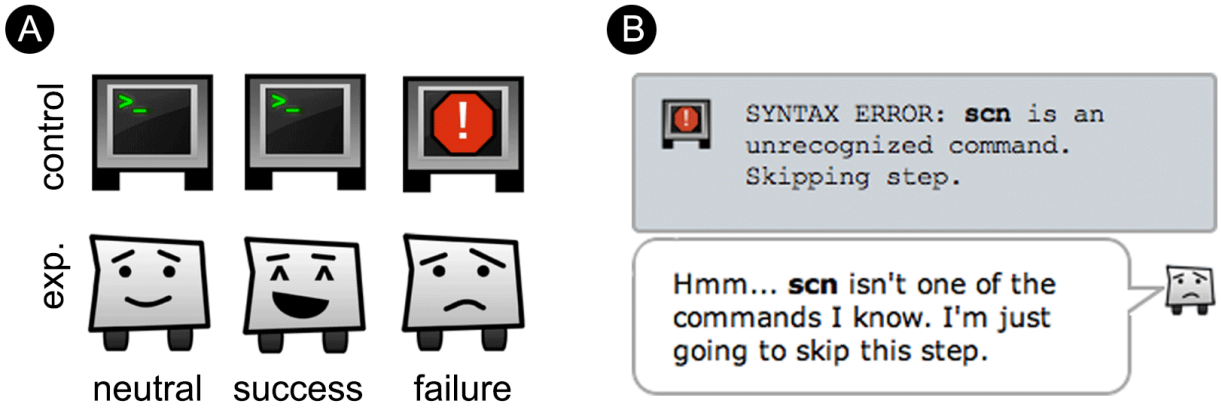


Figure 4.1. Representations and error messages of Gidget based on its game condition.

systems that blame users for errors negatively affect users' performance and their attitudes toward computers (Fogg & Nass 1997).

Since many people view computers as authoritative figures (Beckwith, Burnett, & Cook 2002), and negative feedback from computers affects people's performance on conventional computer tasks (Klein, Moon, & Picard 1999), we were curious how programming tool feedback might affect novice programmers engagement with a learning activity. To investigate this, we designed two versions of the Gidget game (Figure 3.9), changing the way feedback was presented by manipulating the robot protagonist's level of personification, changing communication style, sound effects, and appearance (Figure 4.1).

4.2. METHODOLOGY

The goal of this study was to investigate the role of programming tool feedback on learners' engagement. To do this, we designed a study using the Gidget game, shown in Figure 3.9, with two conditions manipulating the personification of the robot protagonist, Gidget. By personifying Gidget, we aimed to increase the agency of the character, adding human-like qualities to an otherwise cold and emotionless entity. In the control condition, Gidget was represented as a faceless terminal screen that provided terse, impersonal feedback in response to commands and error messages (Figure 4.1-A and Figure 4.1-B). In contrast, the experimental condition represented Gidget as an emotive robot that included the use of personal pronouns such as "I" in the feedback, coupled with facial expressions corresponding to the runtime error state of

the program (Figure 4.1-A and 4.1-B). Participants were recruited on Amazon Mechanical Turk and offered $0.40 for completing the first level and $0.10 for each additional level completed. The total bonus and the levels completed were displayed in the upper right corner of the interface, along with a button giving the participants the option to quit at any time (Figure 3.9). The key dependent variable in this study was *levels completed* as a measure of learners' engagement with the game.

Our null hypothesis was:

> *H_0: There is no difference in levels completed between the control condition, using conventional, emotionless feedback and the experimental condition, using personified feedback.*

In the rest of this section, we discuss the experiment designed to test this hypothesis.

4.2.1. Control vs. Experimental Condition

Personification of the robot's appearance was a key manipulation in our experiment. In the control condition, Gidget was designed to be a cold, emotionless computer terminal – something that the player would feel minimal emotional attachment towards. In contrast, in the experimental condition, Gidget was designed to be more humanlike – a cute, unconfident robot with changing facial expressions based on the success of its execution. In the control condition, Gidget had two distinct states: an error/fail state that was shown during any syntax or runtime error, and a neutral state that was shown otherwise (Figure 4.1-A). The error state, with its large, jarring stop icon, attempts to capture the style common to compiler error messages. In contrast, the experimental condition had three distinct states for Gidget: an error/fail state that was shown during any kind of error, a success state that was displayed when a goal was completed, and a neutral state that was shown otherwise (Figure 4.1-A). These facial expressions were specifically designed to make Gidget more human-like and add affect to its messages throughout the game.

In both conditions, Gidget was designed to be verbose to help players know what was going on with the code during execution. The messages in the control condition were terse, actionable facts about the program state, presented in conventional fixed-width Courier New font (see

Table 4.1. Examples of the variations error messages and font styling.

control	experimental
"Unknown command, so skipping to next step."	"I don't know what this is, so I'll just go on to the next step."
"Dropped cat. Removing from memory banks."	"I dropped the cat. I'll remove it from my memory."
"ERROR: Nothing to *ask* by that name."	"Hmm… I couldn't find anything to *ask* by that name."

Figure 4.1-B and Table 4.1). The text in the experimental condition contained the exact same information, using the softer, sans-serif Verdana font (see Figure 4.1-B & Table 4.1), but was personified in three specific ways. We started with the control text, then followed one or more of these rules: use a personal pronoun (e.g. "I," "you"), admit failure (e.g. "I don't know this command"), and express affect (via exclamation points and emoticons). More examples are shown in Table 4.1.

The dialogue pane between Gidget and the player exhibit another major difference between the two conditions. In the control condition, the player is portrayed as a satellite dish (Figure 4.2) to signify that there is a large physical distance between the learner and robot, requiring radio communication. In the experimental condition, players are given the choice between three avatars (Figure 4.2) to represent themselves when they first start the game. This image is used in place of the satellite dish from the control condition, signifying that there is closeness and teamwork between Gidget and the player.

Next, the shape of the communication text boxes are different between the two conditions (as seen in Figure 4.2). The control condition was designed to look visually cold and direct. In contrast, the experimental condition used comic speech-bubbles for both Gidget and the player with the intention of having the exchange look like a conversation (Figure 4.2). These themes

Figure 4.2. Layout of execution button used to express different styles of communication.

were extended to other parts of the interface, where the control condition's interface boxes have sharper curves than their experimental condition counterparts, which have larger, rounded corners.

Furthermore, there were labeling differences between conditions. First, level titles in the experimental condition were composed of the control conditions' level name with the addition of "Gidget" to add agency. For example, level 1 was titled "Testing Scanner" or "Testing Gidget's Scanner," and level 5 was titled "Utilizing Special Items" or "Using Special Items with Gidget." In the same manner, the memory pane was labeled "Memory banks" in the control condition, and "Gidget's memory" in the experimental condition.

Finally, sound effects were played in both conditions when Gidget performed an action or when a major event, such as Gidget running out of energy or Gidget not completing his goals, occurred. They were designed to supplement the text and provide additional depth to the world as Gidget moved through it. All sound effects were identical between conditions, except the general error and parser error sounds, which were manipulated to evoke different feelings. Errors in the control condition used sounds similar to those heard in operating systems when a critical error occurs. In contrast, errors in the experimental condition used sounds to attract players' attention without making it seem like the computer was "yelling." These sounds were deliberately chosen to add or subtract from the personification between the two conditions.

4.2.2. Recruitment

Previous studies have shown effects due to giving computers personality traits in adult populations of varying ages (Fogg & Nass 1997,; Nass, Fogg, & Moon 1996; Nass 2000). We focused on replicating these studies in programming tools for adults of a similar age range. To recruit these individuals, we used Amazon.com's Mechanical Turk (n.d.), an online marketplace where individuals can receive micro-payments for doing small tasks called Human Intelligence Tests (HITs). It is an attractive platform for researchers because it provides quick, easy access to a large workforce willing to receive a small monetary compensation for their time (Ross et al. 2010). Since workers are sampled from all over the globe, Mechanical Turk studies have the benefit of generalizing to varied populations more than samples from limited geographic

diversity that are more common in traditional recruiting methods (Kittur, Chi, & Suh 2008). However, due to the nature of the low monetary compensation and anonymity of the workers, careful consideration has to be taken to ensure that participants are not "gaming the system" (Downs et al. 2010, Kittur, Chi, & Suh 2008). To address this, we required that participants complete at least one level to receive credit for the HIT, ensuring that they actually had to interact with Gidget and the code before being allowed to quit.

4.2.3. Pricing and Validation

Since our game had a total of 18 levels (see Chapter 3.4), we decided that we would compensate our participants with a base rate and a nominal bonus payment for each level they completed. Previous studies have found that higher payment does not necessarily equate to better results (Hsieh, Kraut, & Hudson 2010), so we wanted to calibrate our payments to established market prices. To do this, we observed Mechanical Turk HITs tagged "game" for a period of 14 days. These HITs were further filtered to include only those that had an actual gameplay element as the main component as opposed to tasks such as writing reviews for third party games. From these HIT descriptions, we constructed a list of 'reward' and 'time allotted' values, along with any explicit bonus payments mentioned. Our goal was to set a base reward that was high enough to attract participants, but also as low as possible to minimize participants' sense of obligation to spend time on our HIT. Likewise, we wanted our bonus payment per stage to have a minimal factor on participants' decision to continue the game.

Based on our data, we determined our optimal base reward as $0.30 for starting the HIT, and an additional $0.10 for each level completed. To ensure participants actually tried the game, we required that they complete at least one level to get paid. This meant the range of compensation participants received was between $0.40 and $2.00. Participants were not informed of the total number of levels, eliminating this factor from their decisions to continue playing the game. Finally, we deliberately avoided mentioning anything about programming in the HIT description and tags to prevent people from self-selecting out of the HIT because of its association with programming. However, since the HIT description included the words "game" and "robot," we may have introduced some gender-biased self-selection effects.

To further validate our pricing model and detect defects and usability problems in the game, we conducted a pilot test on Mechanical Turk with 12 paid participants. In addition, an informal, 4-participant, lab study was conducted to gather information that we could not capture from Mechanical Turk. In this lab study, participants were asked to think-aloud while playing the game to test the clarity of the instructions and observe any problems they had with the interface. Observational study participants were volunteers and were not compensated for their time.

The pilot study results verified that participants were willing to complete levels and that the system functioned as-intended overall. Based on the data we received, we clarified some of the post-game survey questions and fixed several minor defects. We also set the ceiling for submission time to 3 hours to make the HIT less intimidating, as setting it too high could be misinterpreted by potential participants as the task being overly difficult. The observational study surfaced unclear instructions, confusing interface elements, defects, and usability problems in the game. Based on this information, we improved the text and interface elements, running another pilot to ensure that the usability and clarity of the game had improved.

4.2.4. The Participants

On game load, each participant was randomly assigned one of two conditions: control or experimental. This information, along with their current state in the game were logged on the client-side to ensure participants would not be exposed to the other condition, even if they refreshed their browser. Once a participant chose to quit, they were given a post-survey and a unique code to receive payment for their submission. The survey was designed to get demographic information (e.g. gender, age, education, country), identify prior programming experience, and solicit feedback and attitudes about the game. In addition to the survey responses, we automatically collected the following information from each participant upon quitting: the number of levels completed; time stamps for level start, level complete, quit, and any execution button invocations; all character-level edits to each level's program, execution button presses, game condition, choice of user avatar (if in the experimental group), IP address (to verify location), and payment code.

We defined "novice programmers" as participants who reported in the survey that they have never had: 1) "taken a programming course," 2) "written a computer program," or 3) "contributed code towards the development of a computer program." This information was cross-validated with an additional question later in the survey that asked them to rate their agreement with the statement, "I identify myself as a beginner/novice programmer."

Because we deliberately chose not to mention anything about programming in our HIT description, we were not able to control for a specific target audience. Therefore, we recruited a sample of 250 participants from Mechanical Turk, with 116 meeting our criteria as being novice programmers. Since the scope of this study is how personification of the computer and its feedback affects novice programmers, these 116 participants are the primary focus of the analyses discussed in this chapter (a short discussion on the differences found with experienced programmers can be found in my publication: Lee & Ko 2011).

The study used a balanced, between subjects design with 58 participants in each condition. Demographic data revealed that there that participants from the control and experimental conditions were well proportioned, with no significant differences between groups by gender, age, or education (see Table 4.2). There were a total of 50 females and 66 males across the two conditions, ranging from 18 to 59 years old. As shown in Table 4.2, participants were spread across 24 countries, with most participants coming from the USA (27.6%) followed closely by India (22.4%). About 13.8% of participants were the lone representatives of their respective

Table 4.2. The Personification Study's participant demographics.

	control (n=58)	experimental (n=58)
gender	35 males, 23 females	31 males, 27 females
age	18-55 years median = 25	19-59 years median = 27
some college or greater	49/58 = 84.4%	51/58 = 87.9%
Location: USA	15	17
Location: India	13	13
Location: Other	16	14
Location: No data	14	14

countries. Many did not provide geographical data (24.1%). Consistent with other Mechanical Turk study demographics, our sample of novice programmers were well educated (Downs et al. 2010, Kittur, Chi, & Suh 2008), answering that their highest level of education achieved was: less than high school (<1%), high school (13%), some college (23%), an associates degree (3%), a bachelor's degree (38%), a masters degree (14%), or a doctoral degree (6%).

4.3. Study Results

This section reports the quantitative results comparing players' *engagement* from our two groups. Throughout this analysis, we use the non-parametric Wilcoxon rank sums test with $\alpha=0.05$ confidence, as our data were not normally distributed.

4.3.1. Experimental Condition Players Complete More Levels

The minimum and maximum number of levels completed for both conditions were the same, at 1 and 15, respectively (see Figure 4.3 for a distribution box plot). The median number of levels completed for the control and experimental conditions were 2 and 5, respectively. There was a significant difference in the number of levels participants completed between the two conditions ($W=3803$, $Z=2$, $N=116$, $p<.05$), with the experimental condition players completing significantly more levels – meaning that we reject our null hypothesis.

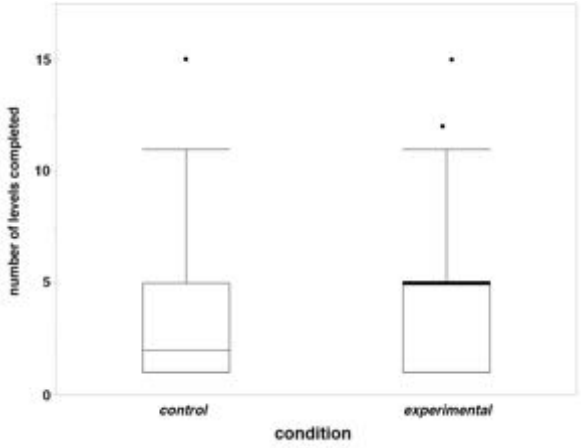


Figure 4.3. Comparison of the Personification Study's levels completed by condition.

The distribution of 'levels completed' (Figure 4.4) shows that a large number of participants from both groups quit the game after completing the first level. This was particularly true for those in the control group, who lost 41.3% of their members in contrast to the 29.3% lost by the experimental group. The large drop off in the sixth level for both conditions will be addressed in the discussion section, below. Since all participants were classified as novice programmers and

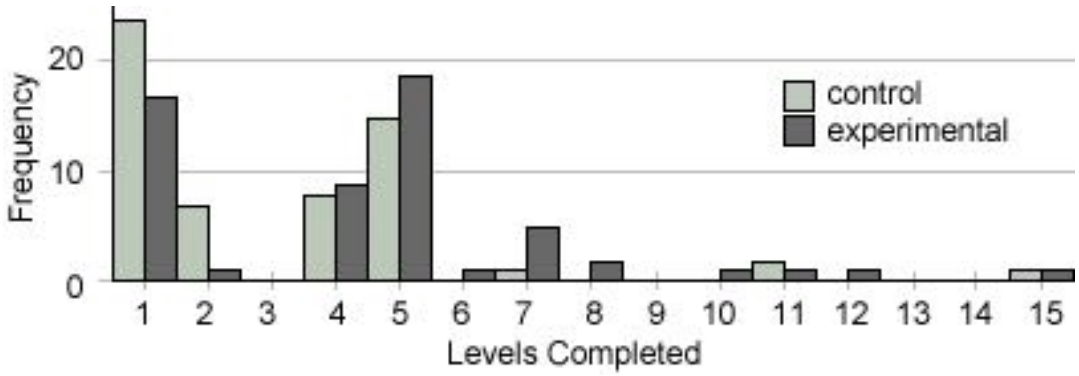


Figure 4.4. Histogram of levels completed for each condition.

there was no statistical difference in demographics, this suggests that our personification of Gidget in the experimental condition had a positive effect on participants' motivation to play.

4.3.2. No Significant Differences in Play Time

The minimum time spent playing the game for the control and experimental condition was 5.4 minutes and 8.4 minutes, respectively (Figure 4.5 plots the full distribution). The maximum time spent playing the game was 2.81 hours and 2.97 hours respectively. The median overall play time for the control and experimental conditions were 27.1 minutes and 35 minutes, respectively. There was no significant difference in the length of time participants in either condition played the game overall (W=3689.5, Z=1.6, N=116, n.s.).

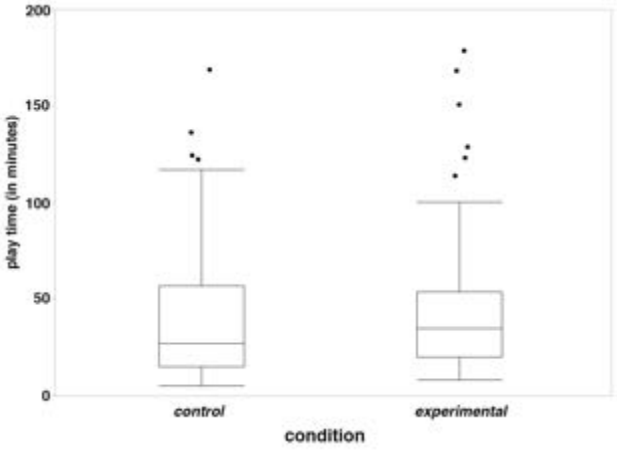


Figure 4.5. Comparison of the Personification Study's play time by condition.

Since the previous result showed that the experimental group completed more levels than the control group, we checked to see if participants in either group were spending more time per individual level. To do this, we calculated the median time each participant took to complete the levels they attempted, and then compared the two resulting distributions of medians. We found that there was no significant difference in the median time to successfully complete levels

between conditions (W=3407.5, Z=0.08, N=116, n.s.). Likewise, there was no significant difference in the time participants spent on the level they attempted, but did not complete (W=3387.5, Z=-0.03, N=116, n.s.).

The difference in levels completed, but the lack of significant difference in playing time suggests that those in the experimental condition learned commands (i.e., by completing more levels) more efficiently. This suggests that something in our manipulation caused the experimental condition participants to better understand and use the commands to fix Gidget's problematic code. We address possible explanations for this in our discussion.

4.3.3. No Significant Differences in Execution

There were no significant differences in how frequently the participants used the four execution control buttons overall (one step: W=3693.5, Z=1.7, n.s., one line: W=3532, Z=0.8, n.s., all steps: W=3488, Z=0.5, n.s., to end: W=3740, Z=1.9, n.s.; N=116) (see Figure 4.6 for the distribution of clicks for each button).

Since we found previously that the experimental group completed more levels than the control group, we checked to see if there was a difference between the conditions for the number of code executions used per individual level. To do this, we calculated the median number of code executions each participant used to complete the levels they attempted, and then compared the two resulting distributions of medians. This was repeated for each execution button. We found that there were no significant differences in the median number of code executions for completed levels by condition (one step: W=3293, Z=-0.5, n.s., all steps: W=3061.5, Z=-1.9, n.s., to end: W=3305.5, Z=-0.5, n.s.; N=116). However, we found that the use of *one line* was significantly different: W=2987.5, Z=-2.3, N=116, $p<.05$. On closer inspection of the data, we found that this difference was due to participants in the control condition using a higher median number of *one line* code executions. This means that participants in the control condition were running their code line-by-line, but skipping some of the finer details provided by the one step execution.

Finally, we checked both conditions to see if there was a difference in the raw number of code executions for levels the participants attempted but did not complete. We found that there

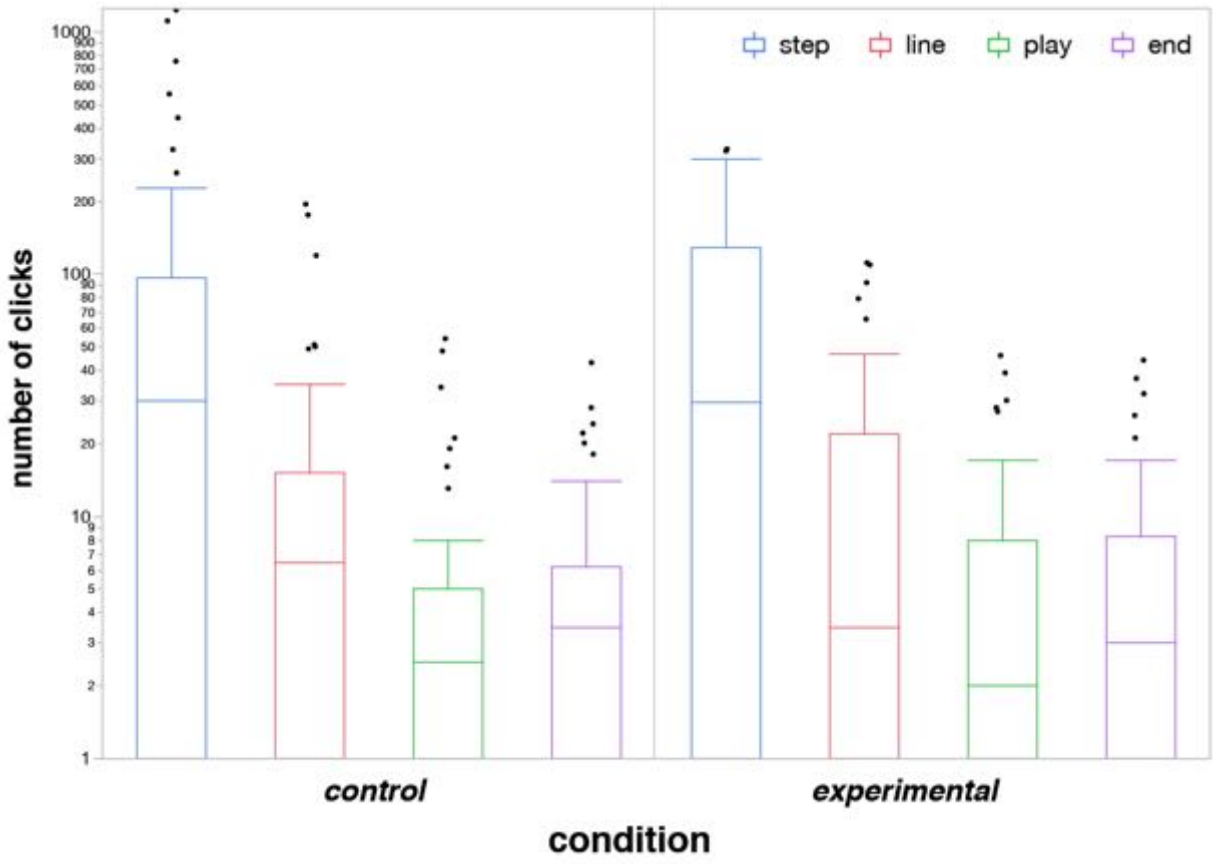


Figure 4.6. Comparison of the Personification Study's button presses by condition.

were no significant differences between conditions in the number of code executions for levels that participants attempted but did not complete (one step: W=3339.5, Z=-0.3, n.s., one line: W=3310, Z=-0.5, n.s., all steps: W=3303.5, Z=-0.5, n.s., to end: W=3483, Z=0.5, n.s.; N=116). Since participants quit on different levels of varying difficulty, this suggests that those from both conditions put approximately the same amount of effort into testing and executing their code before deciding to give up, independent of the level they were playing.

4.3.4. Experimental Condition Players Want to Help the Game Character

There was no significant difference in participants' self-reported level of enjoyment playing the game between the two conditions (W=3117, Z=-1.1, N=116, n.s.). Likewise, there was no significant difference in participants' reporting whether they would recommend the game to a friend wanting to learn programming (W=3629.5, Z=1.4, N=116, n.s.). These results are

consistent with reports by Nass et al. (1996), who found that participants did not attribute success or enjoyment of an activity to changes in their performance.

There was, however, a significant difference in participants' reporting that they wanted to help Gidget succeed ($W=3901$, $Z=3.1$, $N=116$, $p<.01$). Participants in the experimental condition were significantly more likely than those in the control condition to agree to the statement, "I wanted to help Gidget succeed."

4.4. DISCUSSION

Our findings from this study demonstrate that more personified programming tool feedback can have a positive effect on novice programmers' engagement with a debugging game. More specifically, we have shown that casting the computer as a verbose but naïve and unconfident teammate that blames itself for errors has demonstrated to have a positive effect on novice learners' engagement using a simple textual programming language. We also found that novice programmers exposed to this unconfident teammate were more likely to report that they wanted to help it.

These results, combined with the lack of a significant difference in median time spent on levels or execution of the program, suggests that the experimental group was likely making better use of the information provided by the robot than the control group. One possible explanation for this is that by personifying the feedback provided by the programming environment, experimental group participants were more likely to *attend* to the information content in the messages, and thus more likely to understand the program semantics. This is supported by our finding that the control group participants were significantly more likely to use the *one line* execution control, skipping over many (but not all) of the robot's messages. Another interpretation is that both groups attended to the messages similarly, but the phrasing led the experimental group participants to somehow process the information more deeply, by framing it as human rather than computer.

Although our results suggest that our manipulation increased success on completing levels, we did not find that participants were willing to spend more time playing the game. This may be due to the unconstrained nature of Mechanical Turk tasks, which provide no additional extrinsic

incentives to continue; it may also be due to difficulties that learners encountered in particular levels of the game. This was particularly true of level 6, where there was a major drop off of participants in both conditions (Figure 4.4). This level introduced conditional statements, suggesting that it is an inherently difficult concept for novice programmers to comprehend. More work needs to be done to uncover how feedback tool personification affects other aspects of motivation such as wanting to continue to work on a problem after multiple failures on a single level.

4.5. LIMITATIONS

This study has a number of limitations that limit its generalizability. First, Mechanical Turk allows participants to self-select into HITs given that they meet certain qualifications. The HIT did not require any special qualifications and used the default setting from Amazon. Although we tried to account for factors that would affect the HITs listing on Amazon's HIT page, those who filtered for higher-paying HITs would be less likely to find our HIT, whereas those filtering for a tag labeled "game" would be more likely to find our HIT.

Also, the game was accessible by computer, connected to the Internet, listed on a website requiring login. Although not directly translatable to programming ability, gaining access to the game requires a fair amount of computer knowledge. As our demographic data indicated, our participants were well-educated, with 86% of them reporting that they had some college education or beyond.

Finally, though small, there was an economic incentive for participants to participate in the study. Moreover, they would receive a bonus payment for levels they completed. Since these economic incentives would not exist in a place like a classroom, it is unclear how or findings would generalize to other extrinsically motivated learning contexts. For instance, Mechanical Turk turk users have a choice of which tasks to engage in; students in a classroom often do not.

4.6. SUMMARY

This chapter presented a controlled experiment testing players' engagement using the Gidget game. By personifying the robot protagonist – characterizing it as fallible, having it convey

information about coding errors conversationally, and having it take the blame for mistakes – we found that novice programmers complete significantly more game levels than learners who received more conventional feedback, in a comparable amount of time. Given our results, we conclude that personifying the computer and making it less authoritative has immediate benefits for engaging novices wanting to learn how to program.

5. EFFECT OF PURPOSEFUL GOALS ON ENGAGEMENT[6]

This chapter describes the second of three studies that addresses RQ1 – do players of an educational debugging game show measurable signs of engagement playing the game? The study in this chapter explores how the goals in the game affect players' engagement.

5.1. Background and Motivation

Engagement is a necessary condition for learning (Garris, Ahlers, & Driskell 2002), especially for challenging topics such as computer programming (Carter 2006). Such engagement may only occur, however, when objectives are meaningful (i.e. have a purpose) to the learner. Although these effects have been examined in formal educational settings (Layman, Williams, and Slaten 2007, Margolis & Fisher 2002), much less is known about their effects in informal contexts, especially in the space of educational games. For example, dropouts in CS1 courses are often attributed to students feeling that their programs did not solve meaningful problems (Margolis & Fisher 2002) or were lacking any practical context (e.g. sorting a list of meaningless numbers) (Layman, Williams, and Slaten 2007). Additionally, the previous chapter described a study where players who worked with a robot that used personal pronouns and had a face were significantly more likely to report wanting to help it and completed approximately twice as many levels in a similar amount of time as the other group (Lee & Ko 2011).

Whereas the study from the previous chapter investigated the effect of the visual presentation of the program interpreter, in this study we investigate the effect of game goals, manipulated by the presentation of data elements. Recent work has demonstrated that humans have evolved to empathize with animals (Bradshaw & Paul 2010), suggesting that players may attribute more purpose in the goals working with animate data objects, particularly vertebrates (Batt 2009). In Gidget programs, data are the objects that the robot scans, analyzes, and moves, which are directly tied to the goals that the player is trying to accomplish. Goals in the game include transferring spilled chemicals into containers, checking attributes of objects, and moving animals to safety. We hypothesized that changing the presentation of the data referred to in these

[6] This chapter has been adapted from my VL/HCC 2012 publication (Lee & Ko 2012).

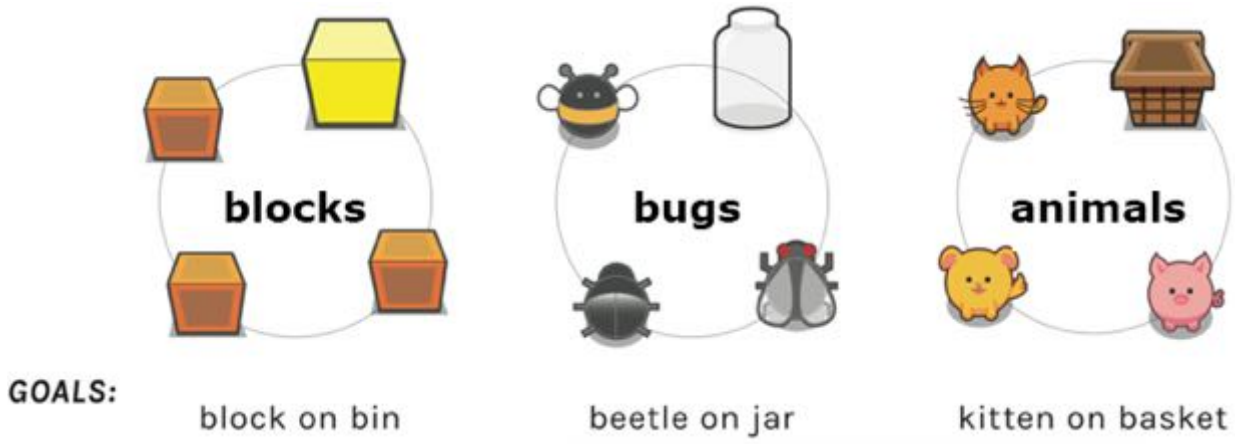


Figure 5.1. Visual representations, names, and goals for the three conditions.

goals would influence the purposefulness of goals, thereby affecting players' motivation to achieve them, especially as goals become increasingly difficult to accomplish.

5.2. METHODOLOGY

We aimed to investigate how the purposefulness of goals, manipulated by the visual representation of data elements and their labeling, affects learners' voluntary engagement. Our study had three conditions involving *block*, *bug*, and *animal* data elements, within the Gidget game (see Figure 5.1). We used a between-subjects design with 41 participants in the animal condition, and 40 each in the block and bug conditions. The key dependent variable in our study was engagement, which we operationalized as the number of levels completed, the time spent on each level, and the use of different UI elements.

5.2.1. The Three Level Conditions

The independent variables we manipulated in our experiment were the labels and visual appearance of the objects referred to in the level goals (e.g. Figure 5.1). The data elements in the *block* condition were inanimate colored blocks, and were intended to diminish the purposefulness of the goals, separating them from the context of the story. In contrast, the other two conditions' data elements were designed to be specific, animate objects. In the *bug* condition, the data elements included beetles, flies, ladybugs, bees, termites, butterflies, and

spiders. In the *animal* condition, the data elements included cats, birds, dogs, kittens, puppies, piglets, and rats. These conditions were intended to increase the purposefulness of the goals, tying them to the context of the story. Supporting objects across the conditions did not follow these categories, but were modified to be consistent with the game's story (i.e. cleaning up an oil spill) and type of object (e.g. block::bin, beetle::jar, cat::basket, respectively). Our hypothesis, based on prior work showing that humans empathize and attribute more positive attitudes towards vertebrates (Batt 2009, Bradshaw & Paul 2010), was that players would ascribe more purpose in saving animals than inanimate objects (i.e. blocks), and therefore complete more levels.

Since some levels included defects in the references to object names, we were careful to inject defects consistently in all conditions. For example, level 1 included the following misspelling of an object that Gidget needed to scan:

- **block condition:** `scan bluck` // should be *block*
- **animal condition:** `scan ketten` // should be *kitten*
- **bug condition:** `scan baetle` // should be *beetle*

In this case, we replaced the data elements' first occurring vowel with an alternate vowel. We were careful to control for these inconsistencies across all levels and conditions that included misspelled names.

5.2.2. Participant Recruitment, Compensation, and Demographics

We recruited participants using Mechanical Turk and our pricing model and validation method was carried over from the study described in the Chapter 4.2.3, which was designed to minimize the effect of monetary compensation on players' motivation to start and continue playing the game. Participants were given $0.40 for completing the mandatory first level, and an additional $0.10 per level completed thereafter. These decisions were validated to attract participation in a new pilot test using the conditions developed for this study, consisting of 29 players from Mechanical Turk, and 6 people in-person.

We focused on self-reported, rank novice programmers. Our HIT description deliberately did not mention programming to prevent people from self-selecting out of the task. A total of 121 participants met our criteria for being novice programmers (see Chapter 1.4 and Chapter 4.2.4). These were those who responded "never" to all of the following statements: 1) "taken a programming course," 2) "written a computer program," and 3) "contributed code towards the development of a computer program." This information was cross-validated with an additional question later in the survey that asked them to rate their agreement with the statement, "I identify myself as a beginner/novice programmer."

Participants were distributed proportionally among our three conditions by demographics (see Table 5.1), with no statistically significant differences in age ($F(2,117)=1.46, MSE=111.3$, n.s.), gender ($\chi^2(2,N=121)=1.1$, n.s.), level of education ($\chi2(14,N=121)=4.0$, n.s.), or country of residence ($\chi^2(32,N=121)=30.7$, n.s.). The median age was 26, ranging from 18 to 66 years old. Our sample included a total of 63 females and 58 males. Fifteen countries were represented in our study, with participants primarily from the US (61.6%) and India (14%). Our sample was well-educated, with 80.9% reporting that their highest level of education was some college or beyond.

5.2.3. Procedure and Dependent Measures

On game load, each participant was randomly assigned one of the three conditions. Once a participant chose to quit, they were given a post-survey asking about gender, age, country,

Table 5.1. The Purposeful Goals Study's participant demographics.

	block (n=40)	animal (n=41)	bug (n=40)
gender	20 males, 20 females	20 males, 21 females	21 males, 19 females
age	18-60 years median = 26.5	18-56 years median = 25	19-62 years median = 27
some college or greater	32/40 = 80%	32/40 = 80%	34/40 = 85%
Location: USA	25	27	22
Location: India	5	5	7
Location: Other	10	8	11

education, programming experience, and asked to select their agreement to the following attitude-measurement statements on a 5-level Likert scale: 1) “I enjoyed playing the game,” 2) “I would recommend this game to a friend wanting to learn programing,” 3) “I wanted to help Gidget succeed,” and 4) “I enjoyed interacting with the objects in Gidget’s world.”

In addition to the survey responses, we collected a timestamped activity log of all participants’ attempted levels including: (1) Each *press* of the execution buttons and a copy of the code at the time of execution; (2) *Level start & level end*: events marking when a player started, completed, or quit a level; (3) *Idle start & idle stop*: events marking mouse or keyboard inactivity (of 30 seconds or more), and where in the UI the idle time occurred. Events were also recorded marking resumption in activity; (4) *Edit time* (edit in & edit out): events marking when the player clicked inside the code pane to edit code or clicked elsewhere to leave the editing pane; (5) *Pane time* (time in & time out): timestamps of mouse cursor movement over or out of the major UI panes.

From these, we calculated the following dependent measures for each participant: (1) *Time on level*: how long individual participant was actively engaged with the code and interface of each level overall, adjusted by subtracting idle time. This was calculated for each level by first taking the difference of level end and level start, then subtracting idle time for that level; (2) *Time overall*: how long each participant played the game overall, adjusted by subtracting idle time. This was calculated by summing up the all of the time on level data per participant and subtracting the sum of their idle time.

Finally, each participants’ number of levels completed, time to complete or quit a level, and logs of execution buttons and UI pane activity, were used to compute dependent measures of activity proportional to overall time spent on levels.

5.3. Study Results

This section reports the quantitative results comparing players’ *engagement* from our three groups. Throughout this analysis, we use non-parametric Chi-Squared and Wilcoxon rank sums tests with $\alpha=0.05$ confidence, as our data were not normally distributed. For post-hoc analyses, we use the Bonferroni correction for three comparisons ($\alpha / 3 = 0.0167$).

5.3.1. Animal Condition Players Complete More Levels

All participants completed at least one level. The maximum number of levels completed in the block, animal, and bug conditions were 9, 18, and 16, respectively (see Figure 5.2). There was a significant difference in the number of levels participants completed between the three conditions ($\chi^2(2,N=121)=7.3,p<.05$). Further post-hoc analysis with a Bonferroni correction shows that the significantly different pair was the block vs. animal conditions (W=1380.5,Z=-2.5,p<.01), with the animal group completing more levels. Comparison of the block vs. bug (W=1669.5,Z=0.5,n.s.) and the animal vs. bug (W=1427.5,Z=-2.0,n.s.) conditions showed no significant differences.

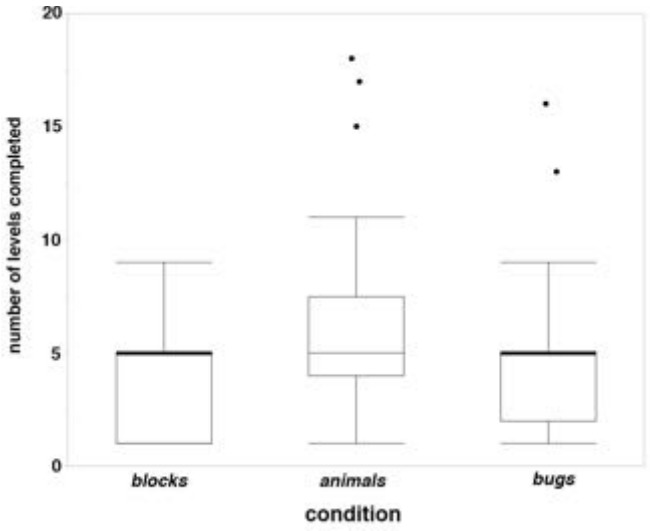


Figure 5.2. Comparison of the Purposeful Goals Study's levels completed by condition.

Investigating further, Figure 5.3 shows that approximately 25% of the participants from each group quit the game after completing only the first level. Next, many participants quit on level 4, which required them to use the command learned in the previous level with a new command. Finally, participants quit again in large numbers on level 6, which introduced conditional statements. This is consistent with others' findings that novice programmers have difficulty with conditional logic (Dahorte, Zhang, & Scaffidi 2010). Here, the block condition had the most drastic drop with 90% of its participants quitting, followed by a drop of 77.5% and 67.5% of participants in the bug and animal conditions, respectively. All of the block condition's participants quit by level 10, whereas the other two conditions had a few participants complete or nearly complete all the levels.

Since all participants were novice programmers with no statistical difference in demographics, these results suggest that interacting with goals that use animal data elements had

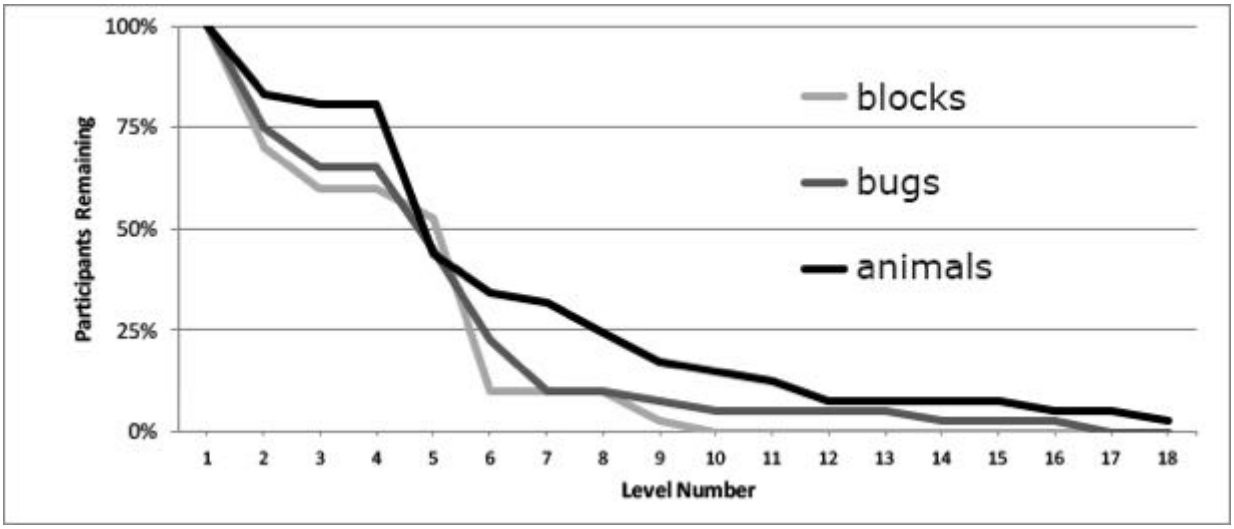


Figure 5.3. Histogram of levels completed for the Purposeful Goals Study's conditions.

a significant positive effect on participants' engagement with the game, particularly on levels introducing difficult concepts.

5.3.2. Animal and Bug Condition Players Play Longer

There was a wide range of overall play times for the block, animal, and bug conditions (4.9 min to 1.3 hrs, 6.9 min to 2.8 hrs, and 8.3 min to 1.9 hrs, respectively; see Figure 5.4 for a distribution box plot). There was a significant difference in the length of time participants played the game overall by condition ($\chi^2(2,N=121)=10.2,p<0.01$). A post-hoc analysis with Bonferonni correction reveals that two conditional pairs were significantly different: block vs. animal ($W=1330,Z=-2.9,p<.016$) and block vs. bug ($W=1889,Z=2.6,p<.016$). In both cases, the block condition players spent significantly less time playing the game than the other two conditions. Play time between the animal and bug conditions did not differ ($W=1620,Z=-0.2$,n.s.).

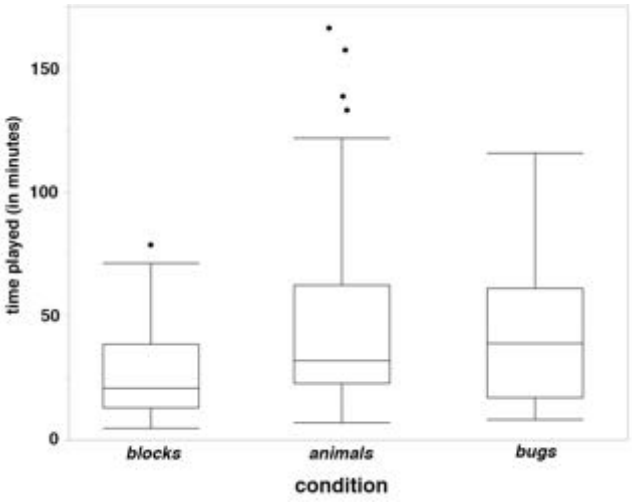


Figure 5.4. Comparison of the Purposeful Goals Study's play time by condition.

Next, we investigated how quickly players completed levels by comparing participants' ratio of total play time to number of levels played, finding no significant difference ($\chi^2(2,N=121)=3.7$,n.s.). In particular, the median times to complete the first 5 levels were very close across conditions.

5.3.3. No Significant Differences in Code Execution Strategies

One possible explanation for the differences in levels completed was a different use of the game UI. Therefore, we investigated the proportion of execution button presses per unit time on completed levels (see Figure 5.5 for a distribution box plot), for each of the four execution buttons, finding no significant differences in usage (one step: ($\chi^2(2, N=121)=2.2$,n.s.), one line: ($\chi^2(2,N=121)=0.5$,n.s.), all steps: ($\chi^2(2,N=121)=1.6$,n.s.), to end: ($\chi^2(2,N=121)=0.1$,n.s.)). These

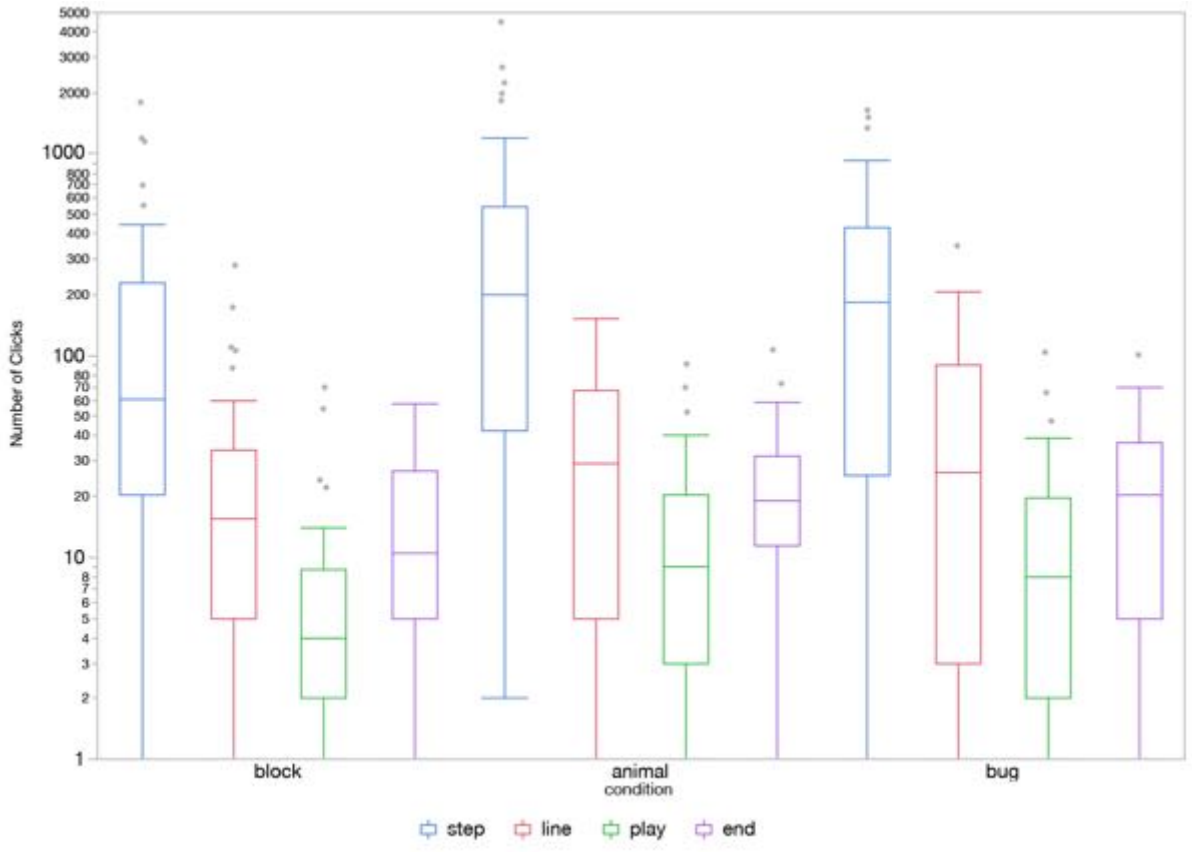


Figure 5.5. Comparison of the Purposeful Goals Study's button presses by condition

results show that the differences in success were likely not due to one condition executing the program more frequently or stepping through it differently.

5.3.4. No Significant Differences in User Interface Usage

Another possible explanation for the disparity in levels completed was differences in how participants used the various panels in the UI. We examined the proportion of interface pane usage to overall time on levels played (see Figure 5.6 for a distribution box plot), again finding no significant differences among conditions (Code: (χ^2(2,N=121)=2.5,n.s.), Goals: (χ^2(2,N=121)=4.0,n.s.), Execution: (χ^2(2,N=121)=3.7, n.s.), Feedback: (χ^2(2,N=121)=0.3,n.s.), World: (χ^2(2,N=121)=4.3,n.s.), Memory: (χ^2(2,N=121)=1.0,n.s.), CheatSheet: (χ^2(2,N=121)=5.8,n.s.)). We also tested the proportion of time spent editing code (computed from

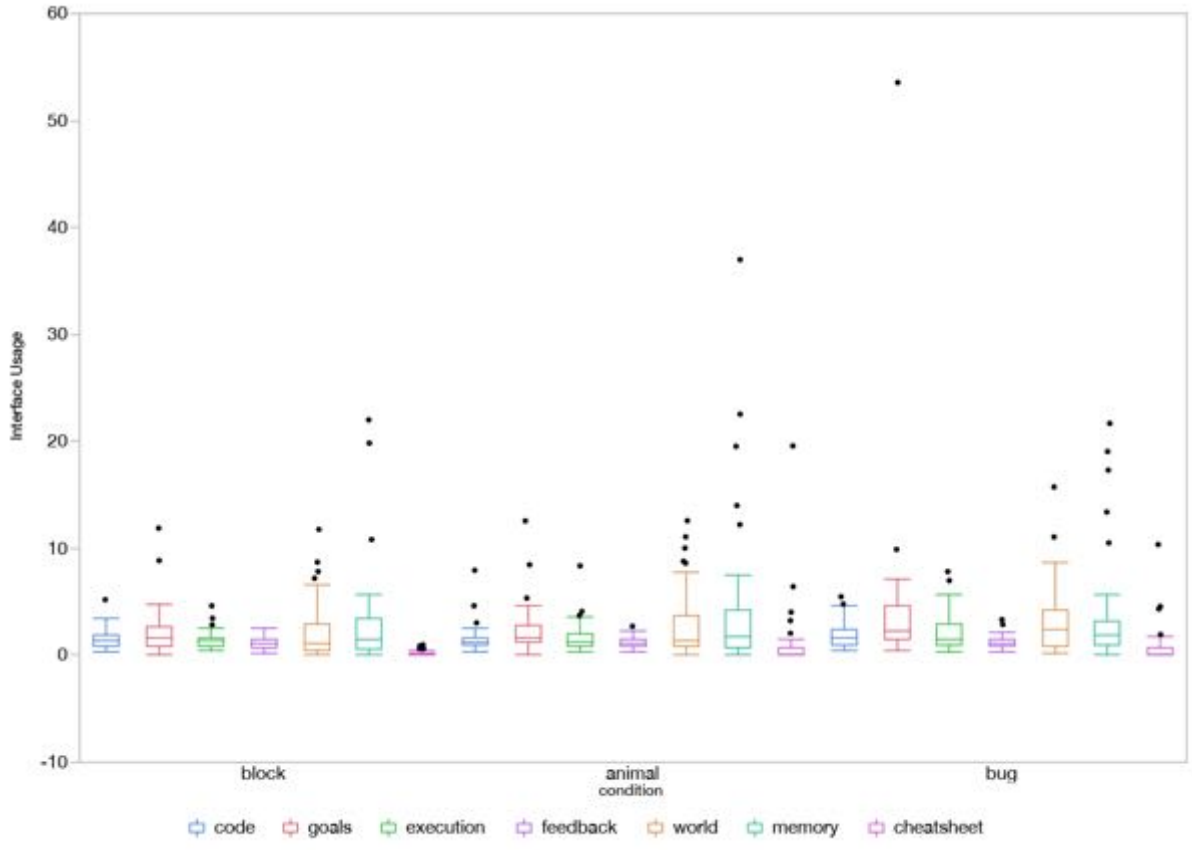


Figure 5.6. Proportion of the Purposeful Goals Study's interface usage to overall time on levels played.

the logs) to overall time on levels and found no significant differences among conditions ($\chi^2(2,N=121)=0.9$,n.s.). All of these results suggest that the differences in success and play time were not due to players' variations of user interface usage in the game.

5.3.5. No Significant Differences in Attitudes

Although there was a trend in survey responses indicating a positive experience playing the game, there was no significant difference in participants' self-reported level of enjoyment comparing the three conditions ($\chi^2(8,N=121)=5.8$,n.s.) or whether they would recommend the game to a friend wanting to learn programming ($\chi^2(8,N=121)=4.1$,n.s.). Similarly, there was a positive trend in responses across conditions, but no significant difference in participants' self-reported desire to help Gidget succeed ($\chi^2(8,N=121)=11.6$,n.s.) or whether they enjoyed working with their data elements ($\chi^2(8, N=121)=5.5$,n.s.).

5.4. DISCUSSION

Our findings show that goals involving animal objects, rather than block objects, significantly increase learners' engagement in a programming game, leading rank novices to play significantly longer and complete significantly more levels. Moreover, we showed that these effects were not due to differences in how players executed the programs, how they used the game UI, how long they attempted each level, or how much time they spent editing their code.

There are several possible interpretations of these results. For example, how did participants complete more levels even though they spent comparable amounts of time attempted them? One possibility is that when reaching a difficult level, players were more motivated to help animals and bugs, rather than blocks, and therefore kept playing. Prior research on computer science recruitment shows that there are gender specific effects in motivation to enroll, specifically related to the reasons for computing (females are enticed when they see how computing can be used for a purpose) (Margolis & Fisher 2002). Players may have seen a greater purpose in saving an animal or bug than in moving a block.

Another potential explanation is that participants paid closer attention to code involving animal and bug objects and therefore understood the semantics of the programming language

better, making the difficult levels easier. The variation in object names between levels may also have had an effect.

Because there were no distinguishable differences in play time between levels in each condition, it is also possible that participants had comparable learning, but different amounts of retention. The amygdala is also known to play a role in learning related retention (Chavez et al. 2009) and our study was motivated by research showing that the amygdala had a preference for images of animals over other objects; therefore, players may simply have remembered more about the meaning of commands from previous levels when the levels involved animals or bugs.

Our results have many potential implications for our understanding of online learning, the role of game elements in engagement, and computing education pedagogy. Our results show that purposeful goals may play a significant role in engagement in the context of self-guided, discretionary, educational games. These findings support prior works done in classroom settings (Kelleher & Pausch 2005, Margolis & Fisher 2002), and broadens them to informal learning settings. Future work should investigate the effects of these factors on learning, both in formal and informal contexts In addition, this study demonstrated that small changes to the game elements can have a significant effect on engagement in educational games. Here, we had large effect sizes, with double the overall play time and level completion, as was the case in our prior study described in Chapter 4. This suggests that in the growing amount of work in educational games research, game designers should be doing more on low-level factors that are predicted to be influential by research in learning, memory, and attention.

5.5. LIMITATIONS

Our results have a number of limitations that may restrict their generalizability and validity. These are the same limitations outlined in Chapter 4.5, including self-selection into a Mechanical Turk HIT, prior computing ability, education (80.1% of the participants in this study had some college education or better), and economic incentive.

In addition, capturing a timestamped activity log of participants' interactions with the game interface is an coarse instrument for measuring attention, particularly when it is done remotely, tracking only mouse and keyboard activity. Since we defined interaction with an interface

element as having the mouse cursor over it, it is plausible that users may have been concentrating on other parts of the interface without necessarily having the mouse cursor over it. This would be acceptable if all the measurements were randomly distributed in the same way across conditions, but could be problematic if they were systematically different.

5.6. SUMMARY

This chapter presented a controlled experiment testing players' engagement using the Gidget game. By manipulating the visual representation of data elements players interacted with in the game, we have found that novice programmers complete more levels and play longer when presented with images of animals instead of block objects. Given our results, we conclude that purposeful goals, manipulated by the objects players interact with in a game, has immediate benefits for engaging novices wanting to learn how to program. These findings, coupled with those found in Chapter 4, support the notion from RQ1 that players of Gidget show measurable signs of engagement playing the game.

6. EFFECT OF IN-GAME ASSESSMENTS ENGAGEMENT[7]

This chapter describes the last of three studies that addresses RQ1 – do players of an educational debugging game show measurable signs of engagement playing the game? More specifically, this chapter explores how explicit testing within Gidget affects players' engagement playing the game.

6.1. Background and Motivation

As more people go online and new content becomes available, many are turning to online educational resources to learn new skills such as programming. Sites such as code.org (n.d.) attract millions of users annually and provides potential learners with links to many different educational resources (Beres 2014). Unfortunately however, many of these resources struggle to keep learners engaged (Daniel 2012) and few of them involve explicit evaluations of learning, making it unclear how much learners actually learn or retain. Therefore, as these resources increase in popularity, a significant design challenge will be improving engagement, while also demonstrably improving understanding.

One way to potentially improve both engagement and understanding is to use assessments (Poehner 2007). Assessments, which directly tests learners' knowledge by asking them to explicitly answer questions about the material, are widely used in compulsory settings not only to measure learners' progress and what they know (Butler & Roediger 2008), but also to improve students' learning itself (Black & William 1998). Assessments improve learning and understanding partly by helping students practice course material and by identifying and correcting misconceptions (Carpenter, Pashler & Vul 2006; Karpicke & Roediger 2007).

Unfortunately, there is a lack of research about how including assessments might affect learners' use of discretionary learning resources (Boustedt et al. 2011). Moreover, there is reason to believe that assessments could actually harm engagement, even if they improve learning. For example, assessments can lead to test-anxiety, negatively affecting engagement (Shute 2011), especially if they get the wrong answer or feedback is lacking (Butler & Roediger 2008).

[7] This chapter has been adapted from my ICER 2013 publication (Lee, Ko, & Kwan 2013).

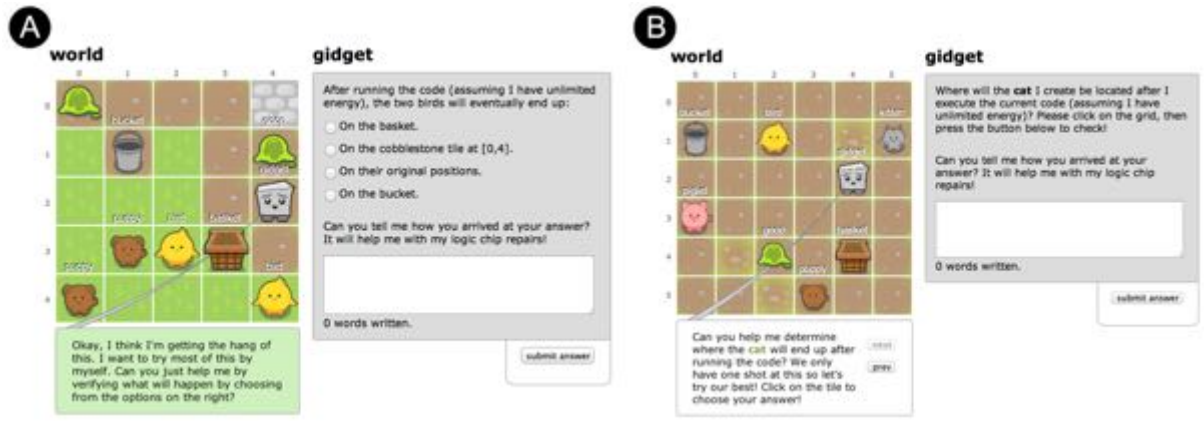


Figure 6.1. Example of a multiple choice question (left) and a click-grid question (right).

Including assessments in educational games or resources that use game mechanics may be even more harmful, as they may interfere with a player's enjoyment of the game, creating a "testing" mode that is poorly integrated with the rest of the game, leading the learner to disengage or even quit the activity.

To begin exploring the role of assessments in discretionary computing education games, we investigated the effect of integrated learning assessments on both engagement and speed across two controlled experiments using Gidget. In our two experiments, we manipulated the inclusion of explicit assessments like the ones shown in Figure 6.1-A, which asks learners to indicate the final position of an object by mentally simulating the given program's execution.

6.2. METHODOLOGY

The aim of our study was to determine how integrated, explicit assessments in an educational computing game affects engagement and task completion speed in self-directed learners, and to identify the extent of these effects. To do this, we conducted two separate controlled experiments using Gidget, with the first measuring engagement and the second measuring task completion speed (see Table 6.1). Each of our experiments had two conditions: the *control* condition's curriculum consisted of a series of levels without assessments, whereas the experimental condition (which we will call the *assessment* condition), was identical, but also included two assessment levels at the end of each set (i.e., unit) of levels.

Table 6.1. Experimental design for the two Assessment Studies

	Study 1. Engagement (quit any time)	Study 2. Speed (complete half the game)
independent variables	Gidget game *with* or *without* assessments	
dependent variables	*# of regular levels completed* and *total time playing the game*	*time required to complete a set of levels* (half of the total game levels)

In our *engagement* study, learners could quit any time, as with any discretionary learning material. We hypothesized that the learners in the assessment condition would play the game for longer and complete more levels because the assessments would offer additional opportunities to practice each units' concepts, leading to better understanding of the material and reducing the likelihood of encountering difficulties and discouragement in subsequent levels. We measured both the *total number of levels completed* and the *total time playing the game*.

Our *speed* study followed the same structure as the engagement study but was designed to enable a direct comparison of how quickly participants completed Gidget game levels. To enable this comparison, we operationalized *speed* as the total time required to complete the first three sets of levels in the game. We hypothesized that even though players in the assessment condition would have to spend more time on the assessment levels, the extra practice and feedback they received through the assessments would result in them being more successful in subsequent levels, completing individual levels faster than those in the control condition.

Both the *engagement* and *speed* studies were between-subjects designs with 200 and 30 participants respectively, split evenly across conditions (see Table 6.2). Participants were recruited on Amazon.com's Mechanical Turk. The speed study was launched after the *engagement* study, without overlap, to prevent people from playing the game simultaneously; we also prevented players from participating more than once. Though we attempted to recruit the same number of participants for both treatments, the *speed* study attracted fewer participants because it required a larger up-front time commitment in its task description.

Table 6.2. The Assessment Study's participant demographics.

	Study 1. Engagement		Study 2. Speed	
	control (n=100)	assessment (n=100)	control (n=15)	assessment (n=15)
gender	55 males, 45 females	58 males, 42 females	9 males, 6 females	8 males, 7 females
age	18-57 years median = 27.5	18-64 years median = 26	21-40 years median = 29	19-36 years median = 26
some college or greater	86%	87%	93%	100%

6.2.1. Assessment Levels

The primary manipulation in our two studies was the inclusion or exclusion of assessment levels in the game. Control condition learners played the game *without* assessment levels for 7 units, spanning a total of 25 levels (Figure 6.2). In contrast, assessment condition learners played the game *with* two assessment levels at the end of each unit (except the final unit) for a total of 37 levels (Figure 6.2). Other than the inclusion of these assessment levels, the sequence and content of the levels were identical in both conditions.

The assessment levels were framed in a way to flow with the story and encourage learners to help the robot with repairs to its logic chip. We took extra care to ensure that the assessments were as close as possible to other game levels, using the same interface, but disabling the code editor, code execution buttons, tooltips, and reference guide, requiring learners to recall their

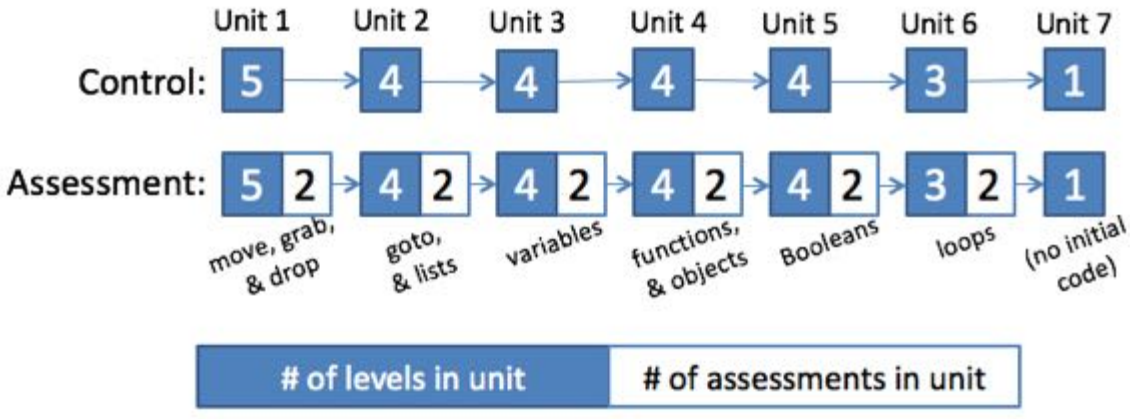


Figure 6.2. The different level sequence for the control and assessment conditions.

knowledge from the previous levels, much like an exam would in a classroom setting. Related studies have found that "closed book" exams demand more difficult and intricate retrieval mental processes, but also amplify testing effects (Bjork 1999; Karpicke & Roediger 2007). Gidget explained these constraints by stating the desire to complete the assessment levels using minimal help from other resources.

Assessment levels came in two varieties: *multiple choice* (Figure 6.1-A), and *click-on-the-grid* (Figure 6.1-B). Multiple choice assessments required the player to select from one of the provided options, which were randomized to minimize ordering effects (Kehoe 1995). All multiple choice questions had one correct key, and three or four incorrect distractors. Click-on-the-grid assessments required the player to select a grid location as their answer to level question which asked where either Gidget or another object in the world would be located after the given code was run. The number of possible choices were equal to the number of grid tiles for the level.

In addition to requiring the selection of a multiple choice option or grid location, learners also had to write an explanation of 8 words or more explaining their reasoning before submitting their answer. This self-explanation approach has been shown to minimize guessing and contribute to students' learning and understanding (Chi et al. 1994;, Smith 2007).

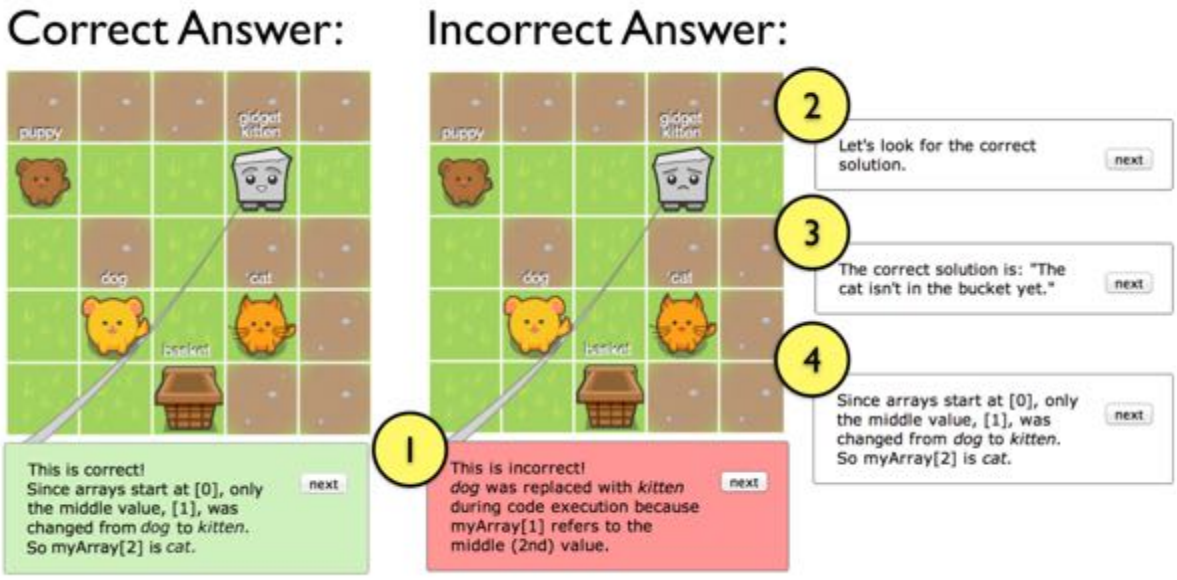


Figure 6.3. The sequence of messages for correct and incorrect answers.

Both types of assessments required learners to inspect the grid, program, and goals, and then mentally simulate the execution of the program to determine the intermediate or final state of some object in the game world. These were therefore direct assessments of players ability to precisely and accurately reason about the language semantics. Clicking the "submit answer" button ran the code, step by-step, visually showing the player how the code was being processed, and the final state of the program. Gidget would then check the learners' answer choice. If the choice was incorrect, Gidget would show a sad face, give an explanation about why it was wrong, then, show a happy face and proceed to explain what the correct answer was, and why (Figure 6.3, right). If the answer choice was correct, Gidget would show a happy face and explained why the learners' answer choice was correct (Figure 6.3, left). These design decisions were based on our prior study detailed in Chapter 4, which found personified feedback affected learners' engagement in a game, and studies in classroom settings that show immediate feedback for exam questions enhances retention of the tested materials and reduces negative effects by incorrect answer choices or distractors (Butler & Roediger 2008).

The content of the assessments were designed to test the specific ideas, concepts, and syntax rules covered in each unit. Distractors were designed deliberately to test for common programming misconceptions. Unit 1's assessments were designed to be straightforward so that learners could get familiar with how the assessment levels worked. Unit 2's assessments identified if learners could follow the control flow and use the correct syntax for list queries. Unit 3's assessments tested variable assignment and accessing array values correctly by index. Unit 4 tested variable passing to functions and objects. Finally, Unit 5 tested whether the learners could correctly trace control flow through conditional statements. Like our curriculum, all assessment levels were validated with participants in-person and online by observing that they were sufficiently challenging, that they covered the concepts from our list of learning objectives, and that they were not a barrier in progressing through the game.

6.2.2. Participants and Procedure

We targeted non-programmers, defined as individuals who self reported that they had never written computer code and had never taken a course related to computer programming (see

Chapter 4.2.4). We used Amazon.com's Mechanical Turk to recruit participants, and our pricing model and validation method was primarily carried over from the two prior studies described in Chapters 4 and 5. Our goal was to set a base reward that was high enough to attract participants, but also as low as possible to minimize participants' sense of obligation to spend time on our HIT. Likewise, we wanted to have any bonus payments to have a minimal effect on a worker's decision to continue playing.

For our *engagement* study, where learners could quit at any time after the first level, we set our base reward as $0.30 for starting the HIT, and an additional $0.10 for each level completed. We set the ceiling for submission time to 5 hours so that participants could gauge the difficulty of the HIT compared to other HITs.

For our *speed* study, players were required to complete the first three units to receive payment (totaling 13 or 19 levels, depending on the condition). There were no similar HITs to base our payment on, so we ran several pilot tests to determine an optimal payment rate. The HIT description was identical to that of the *engagement* study, but also included text explaining that players were required to complete "half the game" before being allowed to quit, and that it could take several hours based on our past observations. We found that nobody accepted/completed our HIT until we started paying $7.00 to complete the first half of the game and $0.10 for each additional level completed (interestingly, *engagement* study participants would have only been paid $2.20 to complete the same number of 19 levels, or a maximum of $4.00 for completing the entire game).

On game load, each participant was randomly assigned to the control or assessment conditions. This information, along with their current state in the game were logged on the client-side to ensure participants would not be exposed to the other condition, even if they refreshed their browser. Once a participant chose to quit, they were given a survey to collect demographic data (e.g. gender, age, education) and a unique code to receive payment for their submission. In addition to the survey responses, we automatically collected the number of levels completed, timestamps for level start, level completion, quit, all character-level edits to each level's program, and execution button presses.

Each of our studies were between-subjects, with an even split between the two conditions. Demographic data revealed that participants in both studies and conditions were well proportioned, with no significant differences between groups by gender, age, or education (see Table 6.2). Consistent with other studies about the demographics of Mechanical Turk workers (Ross et al. 2010), we found that our participants were well-educated, with the majority (86% or more in all conditions) reporting that they had at least some college education or beyond (see Table 6.2).

6.3. Study Results

We provide quantitative evidence for our hypotheses about engagement and speed. Throughout this analysis, we use the nonparametric Wilcoxon rank-sum test with $\alpha=0.05$ confidence, as our data were not normally distributed.

6.3.1. Engagement Study: Assessment Condition Players Complete More Levels

One of our measures of engagement was the number of levels completed (see Table 6.3). To enable comparison of how far learners had progressed through the game's instructional content, we subtracted the number of assessment levels completed from the total number of levels completed (see Table 6.3 – labeled as "adjusted").

Table 6.3. Summary statistics for the two Assessment Studies and conditions.

	Study 1. Engagement		Study 2. Speed	
	control (n=100)	assessment (n=100)	control (n=15)	assessment (n=15)
Min. levels completed	2	2 (2 adjusted)	13	19 (13 adjusted)
Median levels completed	6	10 (8 adjusted)	14	19.5 (14 adjusted)
Max.levels completed	25	37 (25 adjusted)	25	37 (25 adjusted)
Min. time played	6.9 minutes	6.8 minutes	57.1 minutes	63.4 minutes
Median time played	26.3 minutes	41.9 minutes	121 minutes	102.2 minutes
Max. time played	142 minutes	296 minutes	186.6 minutes	198.3 minutes

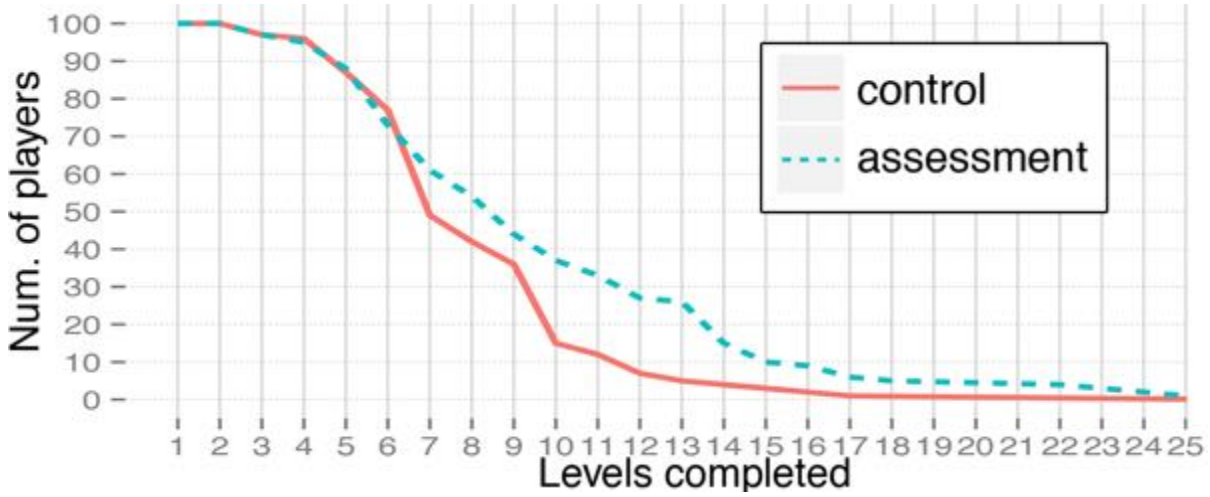


Figure 6.4. Number of players remaining after each level in the engagement study.

There was a significant difference in the number of non-assessment levels completed between the control and the assessment conditions (W=10851,Z=1.97,N=200,p<.05). Participants in the assessment condition voluntarily completed a median of 8 levels, whereas the control condition completed a median of 6 levels (see Figure 6.5). As additional confirmation, we identified that within the speed study, assessment condition participants were more likely to continue playing the game past the minimum required 3 units, voluntarily completing significantly more levels than the control condition participants (W=277.5, Z=2, p<.05).

We examine more closely what may have influenced a participant's decision to stop playing in Figure 6.4. Everyone completed at least 2 levels and 1 control condition participant and 4 assessment condition participants completed the entire game. Many participants from both groups quit the game after completing level 5 (10 players in the control, 15 players in the assessment) and level 6 (28 players in the control, 12 players in the assessment). Level 6 corresponds to the beginning of a new unit (in this case, starting the *goto & lists* unit), and Level 7 required learners to combine the use of keywords from the previous unit and the new unit. Next, 21 of the control group players quit after level 9 (level 10 started the *variables* unit), and 11 assessment group players quit after level 13 (level 14 began the *functions & objects* unit). Since participants had little programming knowledge and there was no difference in demographics, the assessments likely affected motivation when new concepts were being

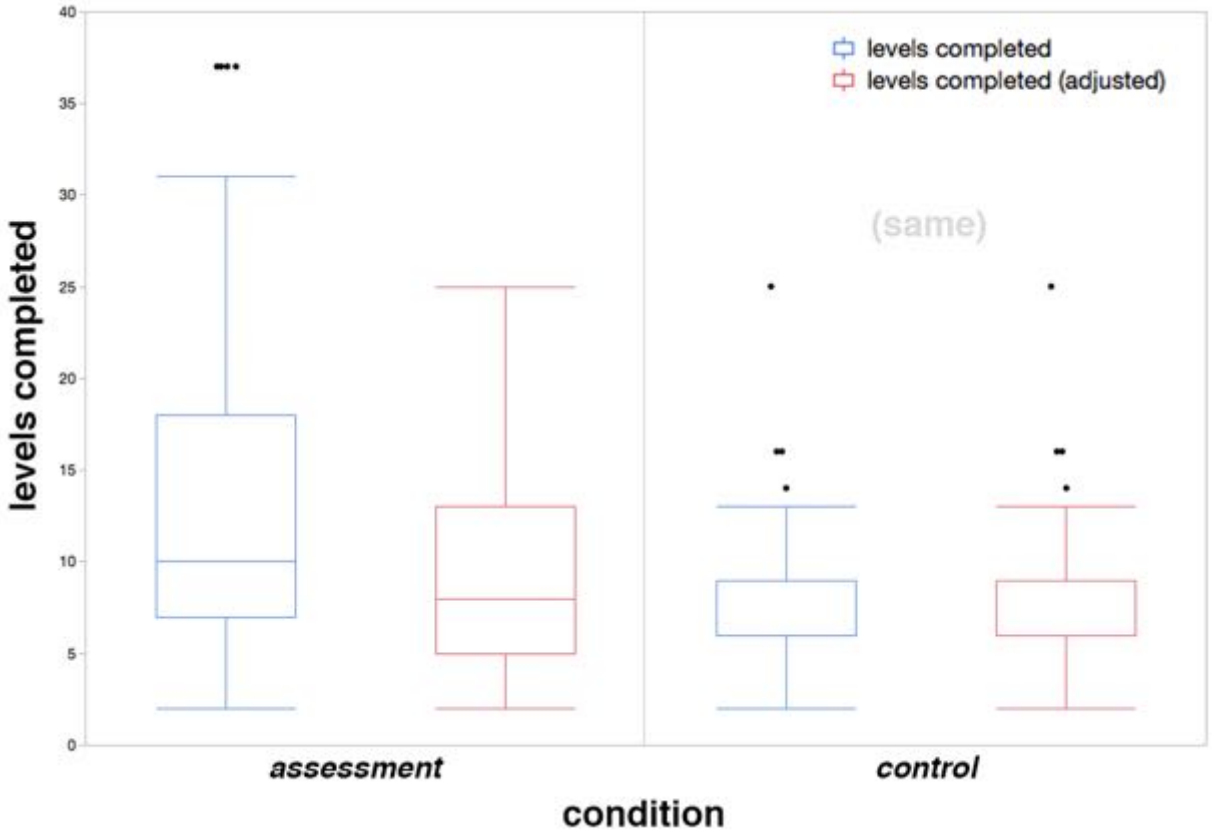


Figure 6.5. Comparison of Assessment Study's (Engagement) levels completed by condition.

introduced (i.e., starting a new unit) and when they had to be combined with previously learned concepts.

6.3.2. Engagement Study: Assessment Condition Players Play the Game Longer

Our other measure of engagement was *total time played*. After subtracting the time played in assessment levels, we found that participants in the engagement study's assessment group voluntarily played the game for significantly more time than participants in the control group ($W=8434.5$, $Z=-3.9$, $N=200$, $p<.01$). As shown Table 6.3 and Figure 6.6, the assessment condition learners voluntarily played twice as long as those in the control condition, with a median overall play time of 41.9 minutes and 26.3 minutes, respectively.

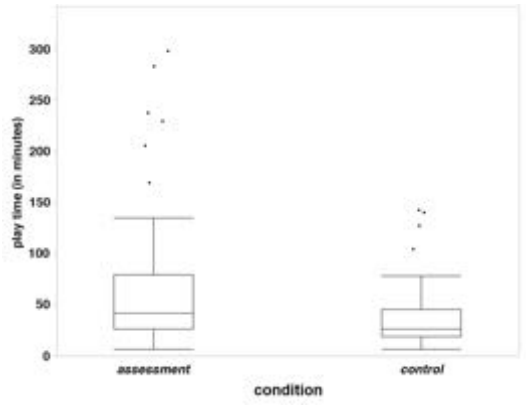


Figure 6.6. Comparison of Assessment Study's (Engagement) play time by condition.

Combined with the large difference in levels completed described in the previous section, the significant differences in play time suggests that assessments caused learners to continue playing even when reaching unit boundaries or difficult levels.

6.3.3.Speed Study: Assessment Condition Player Complete the Same Levels Faster

While the engagement study results show that participants stayed engaged longer when given assessments, this effect could be due to either improved motivation, improved understanding, or a combination of the two. To separate these effects, our speed study held the incentives constant, requiring every participant to complete a minimum number of levels for compensation.

Table 6.3 shows the descriptive statistics for the speed study results. We found no significant difference in the total time participants played the first three units of the game (W=222, Z=-0.4, N=30, n.s.), even though learners in the assessment condition were required to play an additional six levels. However, if we adjust the times by excluding the time spent on assessment levels, we find the difference in completion time was significant (W=171, Z=-2.5, N=30, $p<.05$), with participants in the assessment condition completing the three modules *twice* as fast overall, compared to control condition learners (see Figure 6.7). This shows suggests that assessments helped learners master the game's concepts faster and that adding a small number of assessment levels essentially costs no extra time for participants, but leads to better performance and engagement.

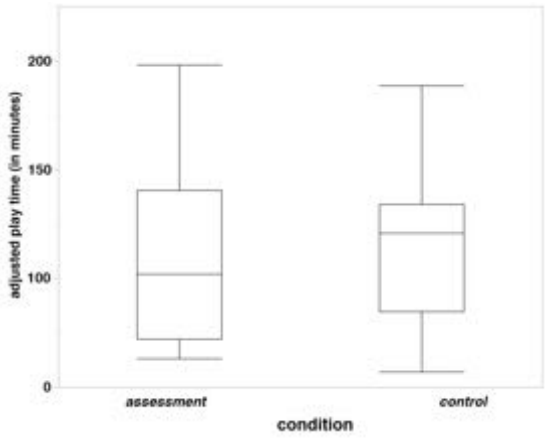


Figure 6.7. Comparison of Assessment Study's (Speed) adjusted play time on same levels by condition.

Table 6.4. Summary statistics for the Speed Study's learners' play styles.

	control (n=15)	assessment (n=15)	significance test (n=30)	
code versions	58 - 223 (med=75)	50 - 124 (med=71)	W=204.5, Z=-1.1	not significant
"one step" clicks	0 - 2901 (med=268)	1 - 1883 (med=256)	W=243, Z=0.4	not significant
"one line" clicks	8 - 426 (med=90)	0 - 357 (med=40)	W=243, Z=0.5	not significant
"to end" clicks	**11 - 73 (med=31)**	**4 - 59 (med=19)**	**W=166.5, Z=-2.7**	***p < .05***
"stop" clicks	0 - 112 (med=13)	1 - 51 (med=21)	W=241.5, Z=0.35	not significant
focus time	**29.1 - 177.2 sec (med=61.7)**	**22.7 - 105.2 sec (med=39.6)**	**W=185, Z=-1.9**	***p = .051***
unit 1 completion	10.8 - 57.2 min (med=22)	7.6 - 46.9 min (med=24.2)	W=227, Z=-0.02	not significant
unit 2 completion	12.1 - 70.2 min (med-37.3)	20.1 - 73.9 min (med=33.5)	W=210, Z=-0.9	not significant
unit 3 completion	**8 - 108 min (med=40.8)**	**6.4 - 110 min (med=19.8)**	**W=176, Z=-2.3**	***p < .05***

6.3.4. Speed Study: Effects on Play Time and Style

To better understand how participants used their time playing the game in the speed study, we examined participants' code versions, code executions, and code edit time. Descriptive statistics for all the data reported in this section can be seen in Table 6.4.

There was no significant difference in the overall number of code versions participants ran for the first three units between conditions. In addition, there was no significant differences in how frequently the participants used the incremental execution control buttons (o*ne step, one line,* and *stop*) overall for the first three units of the game. However, participants in the control condition used the *to end* execution button significantly more than their counterparts (see Figure 6.8), suggesting that they consumed significantly less instructional content, as the *to end* execution prevented players from reading Gidget's explanations of program execution. Control condition participants also spent (nearly significant) more time editing their code (as indicated by having their mouse cursor or text caret in the coding pane) than those in the assessment condition. These results suggest that learners in the assessment condition may have spent more time understanding program semantics by executing the program stepwise instead of reading it or executing it at full speed.

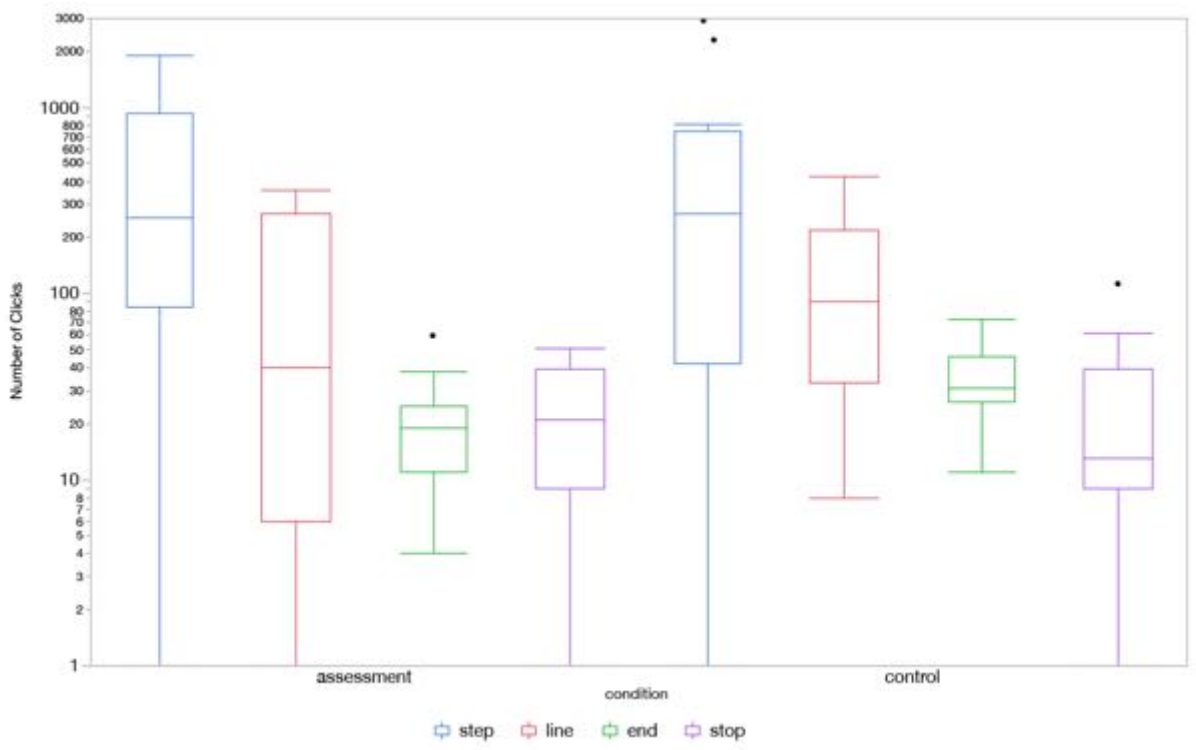


Figure 6.8. Comparison of Assessment Study's (Speed) execution button presses by condition.

Inspecting the overall time each participant spent on each unit, we found a general trend of assessment condition participants completing levels faster than the control group (see Table 6.4, median times in the last 3 rows). This is especially true of the third unit, which shows that the assessment condition participants completed the unit significantly faster than their control counterparts. Closer examination shows that participants in the assessment condition finished significantly faster in the first level of the third module ($W=164$, $Z=-2.8$, $N=30$, $p<.01$), which introduced variables, and the fourth level of the third module ($W=172$, $Z=-2.5$, $N=30$, $p<.05$), which required the use of all the keywords and concepts used throughout the unit.

Finally, we calculated how much time assessment condition participants spent on assessment levels in relation to regular levels. Overall, they played a median of 22 minutes across 6 assessment levels. Checking the ratio of assessment play time to overall play time, we found that participants spent a median of 23.8% of their total time playing assessment levels.

We also examined how well assessment condition learners performed on the assessments and found that they averaged 4 out of 6 correct. We read through participants' incorrect responses

to identify their misconceptions. We found that the majority of misconceptions were the ones we expected and for which we created appropriate distractors (as detailed in Chapter 6.2.1). The one misconception that learners encountered that we had not expected was in the first unit. In the first assessment level, 6 learners were unsure whether a number was required after a move command (e.g., “up” vs. “up 1”) and got the answer incorrect. However, 5 of these 6 participants were able to correctly answer the next assessment, which asked a similar question. We suspect that misconceptions here and in later assessment levels were addressed and clarified since each assessment showed the code execution, explained why the participants’ chosen answer was false, and why the correct answer was true even if the participants’ answer choice was correct.

6.4. Discussion

These findings demonstrate that including explicit multiple choice assessments with self-explanations in a discretionary programming game can significantly increase learner’s engagement and speed. In the case of Gidget, these effects were strong, with the learners given assessments completing 30% more non-assessment levels (Table 6.3), playing twice as long (Table 6.3), and completing levels about 20% faster (Figure 6.6), than those not given assessments.

We also found that speed study learners in the assessment condition were only getting the correct answer 66.67% of the time. However, they were spending an average of 24.9% of their total play time on the assessments, and in the free-form answer section, many participants gave reasonable explanations for why they thought their particular answer was correct (see Chapter 6.3.4). This indicated that learners were trying on the assessment levels, even though they could proceed regardless of the outcome of their answer.

There are several possible interpretations of our results. The difference in performance that we saw in the speed study might be because the assessment levels corrected misconceptions by providing the correct answers (whether or not the learner submitted the correct answer), which has been shown to improve performance in compulsory settings (Carpenter, Pashler, & Vul 2006; Karpicke & Roediger 2007).

It is also possible that since the code in assessment levels were not executable, learners had to mentally simulate and trace the program's execution, giving them practice understanding the program semantics unaided. Since control condition leaners' were never presented levels with these constraints and were always able to execute their code to see what happens, they were likely more inclined to do that; this is consistent with the finding that control condition learners used the *to end* execution button significantly more than the assessment condition learners (see Chapter 6.3.4), rather than taking the time to understand how the code was running.

A third explanation is that since access to the reference guide and tooltips were disabled in assessment levels, learners in the assessment conditions were required to recall how the keywords and syntax worked using their memory. This may have allowed them to understand both the syntax and semantics of the keywords better than those in the control condition, who were always presented with levels that allowed quick access to definitions and examples through the tooltips and reference guide.

Finally, since assessment levels required an explanation of the answer choice, assessment condition learners had extra opportunity to reflect on their selections and translate that into text. This may have allowed them to identify misconceptions on their own, and may have also provided a means of direct comparison to the answer explanations Gidget gave about the incorrect and correct answer choices. Allowing learners to explain their answer choices may also increase the positive effect of assessments (Aleven & Koedinger 2002), including improved understanding of the material being assessed (Chi et al. 1994).

These observations may also explain the completion of more levels and the longer play time by the assessment group in our engagement study. Those in the control condition may have found later levels too difficult, causing frustration and ultimately making them quit. In contrast, those in the assessment condition may have found these levels less difficult due to their experience from assessment levels, but sufficiently challenging to keep them engaged with the game.

6.5. LIMITATIONS

This study had the same limitations as those mention in the prior studies (see Chapter 4.5 and Chapter 5.5), including self-selection into a Mechanical Turk HIT, prior computing ability, education, and economic incentive. There may also be limitations to the generalizability of the game itself. Gidget uses a specific type of programming language and a specific framing of programming. These may have interacted with assessments in a way that may not occur in other settings.

6.6. SUMMARY

This chapter investigated how providing assessments to self-directed, independent learners playing a game designed to teach programming would affect engagement and play time. By manipulating the inclusion or exclusion of assessments at the end of each game unit—a collection of levels designed to teach a set of programming keywords or concepts—we found that learners who were given assessment levels complete more game levels and play the game longer and that they completed non-assessment levels faster. Given our results, we conclude that well-integrated, in-game assessments, have immediate benefits for engaging novices wanting to learn how to program. Combined with the findings from Chapters 4 and 5, these results support the notion from RQ1 that players of Gidget show measurable signs of engagement playing the game.

7. EFFECT OF THE GIDGET GAME ON LEARNING[8]

This chapter describes a study that addresses RQ2 – do players of an educational debugging game show measurable learning of programming concepts covered in a typical CS1 course? More specifically, this chapter explores to what extent players are able to transfer their understanding of fundamental CS1 concepts from the Gidget language to pseudo-code.

7.1. Background and Motivation

Chapters 4 through 6 have demonstrated that novice programmers can be engaged playing an online educational debugging game. The results from Chapter 6 also showed some evidence of learning: adding in-game assessments seamlessly into the storyline and gameplay of Gidget improved the speed in which players were able to complete levels. Players were also able to answer 66.7% of their assessment questions correctly, and were able to provide reasonable explanations for their answer choices. However, even though these findings are promising in regards to learning, it is still not well understood how effective Gidget and other online resources are *actually* at teaching programming concepts in a measurable way.

Therefore, to investigate the learning outcomes of these various online resources, we conducted a pretest-posttest experiment using three types of online educational technologies, comparing the learning gains of each. We specifically compared the Python course on Codecademy (see Figure 7.1), the Gidget game (see Figures 1.1 and 3.1), and the open-ended creative environment found in Gidget called the Puzzle Designer (see Figure 3.8), which is analogous to other creative development environments such as Scratch (Maloney et al. 2010) and Alice (Cooper, Dann, & Pausch 2000; Dann, Cooper, & Pausch 2011).

7.2. Methodology

The goal of this study was to examine the extent to which novices showed measurable learning gains after using one of three online learning technologies. To do this, we first selected three learning activities that are representative of the types of discretionary, online resources that

[8] This chapter has been adapted from my ICER 2015 publication (Lee & Ko 2015).

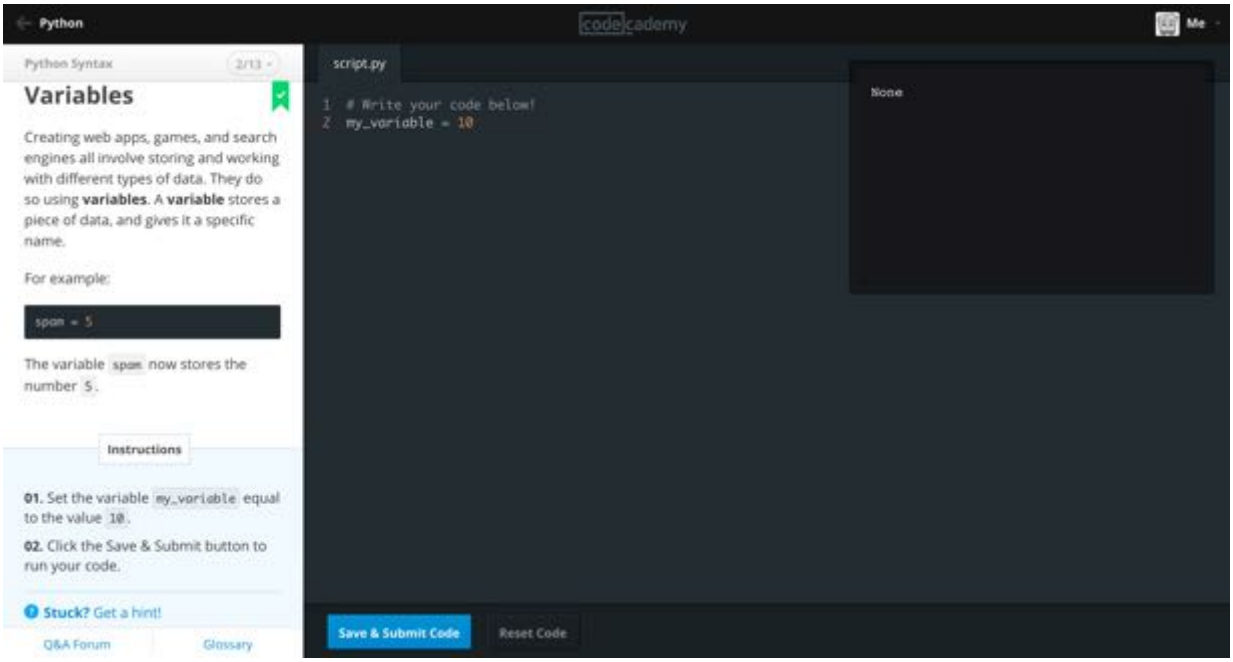


Figure 7.1. Screenshot of a Codecademy beginner's Python tutorial.

people currently use to learn programming: 1) an online tutorial system using a web-based IDE, where learners go through a didactic, structured curriculum, 2) an educational game using an IDE, where learners go through a problem-based, structured curriculum, and 3) an open-ended creation IDE, where there is no planned curriculum, where learners acquire skills by creating with code. Next, we created a test designed to measure one's knowledge of different introductory programming concepts before and after completing one of the learning activities.

Our null hypothesis was:

> *H_0: There is no difference in learners' post-test performance among the conditions after completing their assigned learning activity.*

In the rest of this section, we describe our three learning activities in more detail, explain the design of the pre-post test, and discuss the experiment designed to test the hypothesis.

7.2.1. Learning Activity 1: Codecademy Course

Codecademy (n.d.) is a popular online interactive tutorial website that offers free courses in multiple programming languages (see Figure 7.1). It has had over 24 million users who have completed over 100 million exercises (Summers n.d.). For our study, learners participated in the introductory "Python Language Skills" course. According to the Codecademy website, over 2.5 million users are enrolled in this course designed for beginners. The website also states that the Python course takes an estimated 13 hours to complete.

Codecademy's course interface consists of a two-pane window split vertically on the screen (see Figure 7.1). The left pane lists consists of instructions, examples and hints for the user to follow. For each activity, it contains a numbered list of explicit instructions for the user to follow (e.g., "01. Set the variable *my_varaiable* equal to the value *10*" and "02. Click the Save & Submit button to run your code."). The right pane is an IDE for users to type in and execute their code, with an overlay on the upper-right corner that shows console output on code execution. In case learners get stuck, there is a "Stuck? Get a hint!" button below the list of instructions on the left pane that users can click to open up more help text. The hints are typically explicit instructions (e.g., "All you need to do is type 3 after the equals sign on line 8.") or closely related examples (e.g., "Make sure you're setting your variable like this: *the_machine_goes = 'Ping!'* ".). Finally, the bottom-most area of the left pane includes two buttons that opens up a new browser tab: one labeled "Q&A forum" where fellow Codecademy users can post and answer questions, and another one labeled "glossary," that goes to a dictionary of Python commands and concepts.

The introductory Python course has a total of 12 modules covering the following topics: syntax, variables, mathematical and logical operator, strings, conditionals, control flow, functions, lists and dictionaries, and advanced concepts (e.g., classes and file input/output). Each module is split into two parts. The first part is designed to teach a specific concept or set of concepts and consists of several activities that subsequently build on the previous activity. The second part of the section is an exercise to practice combining the first part to build something interesting. For example, in the case of the syntax module (where learners are introduced to

variable assignment and the use of mathematical operators), the second part of the module tasks users to fill in variables with values to calculate gratuity for a meal.

To ensure that the concepts covered by Codecademy and the Gidget game were as close as possible, we asked learners to complete only the first 8 of 12 modules before taking our post-test. Although learners would not be tested on these extra advanced concepts on the post-test exam, finishing them would have given them additional practice with many of the previously learned concepts. We asked learners to keep track of the time they spent using Codecademy so that they could report their total time after taking the post-test exam. Since the Codecademy website states it takes around 13 hours to complete the 12 modules in the Python course, we told our tutorial condition participants it would take approximately 10 hours to complete their assigned 8 modules before they started their activity.

7.2.2. Learning Activity 2: Gidget Game

Our second learning activity was the Gidget game and curriculum as described in Chapter 3 (also see Figures 1.1 and 3.1). Based on our experience with observing novices playing Gidget (Lee et al. 2014), we told our game condition participants that Gidget would take about 5 hours to complete before they started the activity. We required learners to complete all the levels before taking the post-test. For this specific condition, we automatically logged the time learners took to complete the game.

7.2.3. Learning Activity 3: Gidget Puzzle Designer

The Gidget Puzzle Designer (GPD) is an integrated development environment used to create and edit Gidget levels (see Figure 3.8). It is normally unlocked after finishing the Gidget game (as described in Chapter 3.2). However, for this study, participants were given access to the GPD without any prior experience playing the Gidget game. This was to mirror other open-ended, creation-oriented learning environments Scratch (Maloney et al. 2010), Alice (Cooper, Dann, & Pausch 2000; Dann, Cooper, & Pausch 2011), and others (Kelleher & Pausch 2007), where users are free to explore and tinker to make their own projects.

The interface for the GPD is a modified version of the regular Gidget game interface, allowing modification of previously un-editable code such as the starting world code, the level goals, the dimension of the world grid, and Gidget's introductory dialogue and emotional state at the beginning of the level. In addition, the status pane on the rightmost section is replaced by a tabbed inventory of available characters and objects, ground tiles, and sounds that the learner can use to populate and enrich their programs.

All of the same help tools available in the Gidget game are also available in the GPD. This includes the syntax highlighting, tooltips, dictionary, and Idea Garden suggestions. In addition to the help systems, the learners also had access to view and edit all the regular game levels, giving learners puzzles to modify for creative purposes. These examples excluded the assessment levels at the end of each module, and listed the levels in sequential order without indicating which module they belonged to. Similar types of help and examples are available in both Scratch and Alice to help bootstrap learner engagement.

Unlike Codecademy and the Gidget game, the GPD did not have a clear sequence of steps or storyline for its users to follow. Therefore, to help orient our GPD users, we showed them a list of directions before they started with their activity. First, we told them their task was to "Use a creative canvas tool to create multiple stories for a robotic character named Gidget." This is based on several works, primarily by Kelleher et al. (2007a, 2000b), which shows that adding storytelling elements to open-ended creative environments can significantly increase users' engagement (Ivala et al. 2013; Ryokai, Lee, & Brietbart 2009; Umaschi 1997). Second, we told them about the various help features available, and how to access them. Third, we asked them to "create, explore, and play with the website for at least several hours to get the full learning experience" with the activity. For this specific condition, we automatically logged the time learners spent in the GPD, and collected records of all their levels.

7.2.4. Knowledge Test for CS1 Concepts

In order to measure how much participants learned and what they learned, we created and validated a test designed to be taken before and after the learning activities. We adopted this pre-test/post-test design as it widely used in both educational and non-educational contexts to

measure change resulting from experimental treatments (Bonate 2000; Chumley-Jones, Dobbie, & Alford 2002; Dimitrov & Rumrill 2003).

First, we determined which concepts to test by comparing the topics that are taught commonly in introductory programming courses (Deitel & Deitel 2005; Felleisen et al 2001; Lewis & Loftus 2005; Tew 2010; Zelle 2004) to the set of concepts that were covered in our Codecademy and Gidget game activities. We chose a total of eight concepts: basics (i.e., variables, mathematical operators, relational operators, Booleans), logical operators, selection statements (i.e., conditionals), arrays, indefinite loops (i.e., while), definite loops (i.e., for), function parameters, and function returns.

We modeled our test questions after Allison Tew's dissertation work (2010) on the FCS1, a programming language-independent test using pseudo-code. In her studies, Tew showed that testing introductory programming students in the classroom with their native course language and in pseudo-code were strongly correlated (Tew & Guzdial 2011) and has the extra benefit of demonstrating transfer of learning (Bransford, Brown, & Cocking 1999). We generated pseudo-code questions using the examples, descriptions, and two-page pseudo-code guide Tew provided (Tew 2010). Questions used a verbose style adapted from guides for programmers published by Whitford (n.d.) and Shackelford (1997). To minimize confounding factors in syntax design, we followed the latest evidence on syntax learnability, excluding semi-colons and curly braces, indenting code blocks, upper-casing reserved words, and closing program blocks with explicit keywords (Sime, Green, & Guest 1976) (see Figure 7.2 for examples).

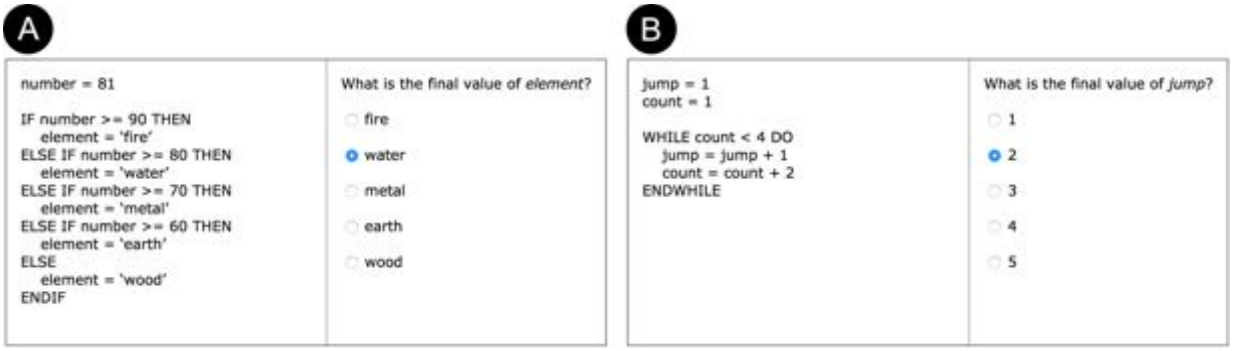


Figure 7.2. Screenshot of two different pseudo-code questions from the pre- & post-tests.

Based on guidelines and examples from Tew's dissertation, we designed 5 multiple choice questions for each of the concepts covered in our learning activities, for a total of 40 questions. All questions had one correct response and four incorrect distractors. We designed distractors to deliberately test for common programming misconceptions (Bonar & Soloway 1985; Thompson 2006).

To validate our 40 questions, we recruited people on Mechanical Turk. Our participants were paid $0.02 to answer one pseudo-code question, indicate their experience with programming, and optionally provide their email address. No additional demographic information was collected. Each participant could answer up to an additional 39 questions for $0.02 each. In these cases, each additional question would be new, and the participant did not have to re-enter their answers for programming experience or their e-mail address (if provided previously). To mitigate ordering effects, questions were randomly sequenced each time a participant took the survey. Answer choices for questions that did not require a specific order were randomly arranged as well.

To identify problems with our questions and answer choices, we ran two rounds of pilot tests, with each question getting at least 3 responses for each iteration of testing. We corrected issues dealing with ambiguous/confusing wording, inappropriate distractors, syntax errors, and typos. To achieve this, we looked for data anomalies (e.g., nobody getting the answer correct, or everyone choosing the same answer) and requested open-ended feedback from our respondents. We then ran a full test with 1,494 participants on Mechanical Turk and had a total of 8,011 responses to our questions (approximately 200 responses per question). The majority of our participants only answered one question, with 11% completing 3 or more questions.

To avoid ceiling and floor effects and to maximize discriminability of the assessment, we categorized our data by splitting responses by the Mechanical Turk participants' self-reported programming experience. As in our past studies, we categorized novices as those who responded "never" to all of the following statements: 1) "taken a programming course," 2) "written a computer program," and 3) "contributed code towards the development of a computer program." All other respondents were considered experienced programmers. For our finalized list of exam questions, we selected the top 3 questions for each concept (for a total of 24) with highest

variance between novice and experienced programmers (that is, those that novices tended to get incorrect and those with experience tended to get correct).

7.2.5. Participants and Procedure

The independent variable in our experiment was the *instructional approach*, which had three levels: 1) tutorial (complete the introductory Python programming tutorial on Codecademy), 2) game (play through the Gidget game), or 3) canvas (use the GPD to create Gidget levels). We told participants that they were allowed seven days to complete their assigned task, and provided an estimate of the number of hours their task would take (10 hours for the tutorial condition, 5 hours for the game condition, and open-ended for the canvas condition).

We recruited our participants from Mechanical Turk, specifically those who self-reported that they had no experience with programming. We also required participants to be U.S. residents to minimize English language barriers with the instructions and activities. Participants were compensated $10.00 for completing their assigned task. This amount was carried over from the study described in Chapter 6 and adjusted to account for the extra time required for the pre-test and post-test.

We sent participants an e-mail with a link that randomly assigned them to a condition and redirected them to the web-based pre-test. Each link was uniquely associated with a specific e-mail address, so that we could identify the owner of each test. Like our pilot study described in the previous section, we randomly ordered our finalized collection of 24 questions to minimize ordering effects, also randomizing the order of the answers, where appropriate. The test only showed one question at a time (see Figures 7.2-A and 7.2-B) and it was not possible to go back to a previous question. Each question required a response before being able to move onto the next question. There was a progress indicator on the top of the page showing participants how many questions remained. The system automatically logged each answer choice and the total time to complete the exams.

The pre-tests and post-tests were identical across all conditions. The only exceptions to this were as follows: The post-test for those in the tutorial condition had two additional questions for the participants to report how many modules they completed, and the time they spent to complete

Table 7.1. The Learning Study's participant demographics.

	tutorial (n=20)	game (n=20)	canvas (n=20)
gender	10 males, 10 females	11 males, 9 females	11 males, 9 females
age	18-35 years median = 23	18-41 years median = 25	19-29 years median = 23
some college or greater	19/20 = 95%	20/20 = 100%	20/20 = 100%
Location: USA	20/20 = 100%	20/20 = 100%	20/20 = 100%

their Codecademy activity. The introductory text for the pre-tests briefly explained that participants would be answering coding questions and that they should try their best even though they might not be familiar with the content. The introductory text for the post-tests briefly explained that the questions were written in another, related programming language that covered the same concepts available in the learning activity they had completed.

Our study was a between-subjects design, with an even split of 20 people each among the three conditions. Our participants did not differ significantly by gender, age, or education (see Table 7.1). Consistent with other studies about the demographics of Mechanical Turk participants (Ross et al. 2010), we found that our participants were well-educated, with the majority reporting that they had at least a bachelor's degree (see Table 7.1).

7.3. Study Results

This section reports the quantitative results comparing the learning outcomes from our three groups. Throughout this analysis, we use non-parametric Chi-Squared and Wilcoxon rank sums tests with α=0.01 confidence, as our data were not normally distributed. For post-hoc analyses, we use the Bonferroni correction for three comparisons (α / 3 = 0.0033).

7.3.1. Better Post-Scores with Tutorial & Game Condition Players

Overall, participants did poorly on the pre-test exams, with a median score of 5 out of 24 questions correct (20.8%) across all three conditions (see Table 7.2 and Figure 7.3). This was expected, as we had selected the questions most difficult for novices from our original set. We compared the pre-test scores across the conditions and found no significant difference

Table 7.2. The Learning Study's summary statistics of pre-test and post-test scores.

	tutorial (n=20)	game (n=20)	canvas (n=20)
Minimum score on pre-test	2 of 24 = 8.3%	0 of 24 = 0%	3 of 24 = 12.5%
Median score on pre-test	5 of 24 = 20.8%	5 of 24 = 20.8%	5.5 of 24 = 22.9%
Maximum score on pre-test	8 of 24 = 33.3%	6 of 24 = 25%	9 of 24 = 37.5%
Minimum score on post-test	6 of 24 = 25%	4 of 24 = 16.7%	3 of 24 = 12.5%
Median score on post-test	12 of 24 = 50%	10 of 24 = 41.7%	5 of 24 = 20.8%
Maximum score on post-test	18 of 24 = 75%	16 of 24 = 66.7%	9 of 24 = 37.5%
Percent increase between median pre-test and post-test scores	140%	100%	-9.1%

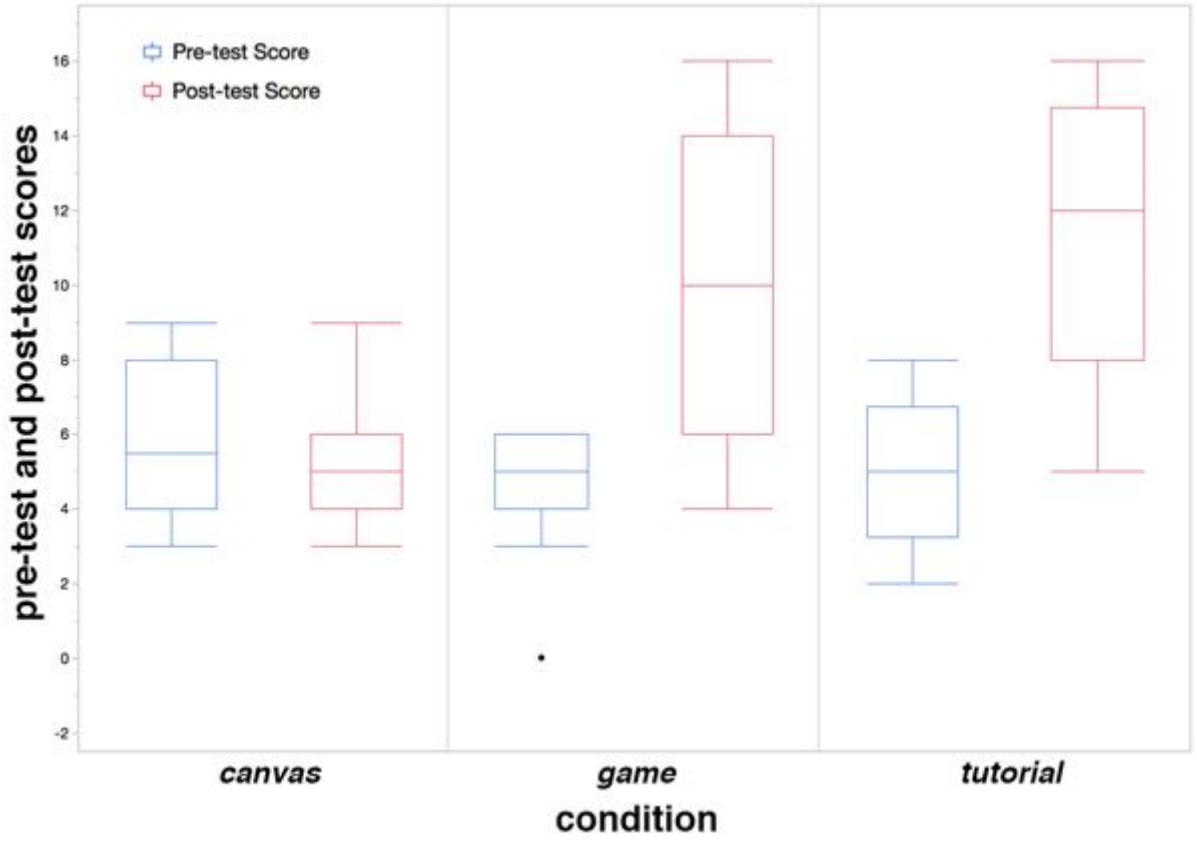


Figure 7.3. Comparison of Learning Study's pre-test and post-test scores by condition.

($\chi^2(2,N=60)=4.30$, n.s.), confirming that all of our participants' programming knowledge was roughly equivalent prior to the learning activities.

Participants also did poorly on the post-tests, with the highest median score among the conditions being 12 out of 24 questions correct (50%). However, comparing the post-test scores across the conditions reveal that there is a significant difference in learning gains between conditions ($\chi^2(2,N=60)=27.03,p<.01$). Post-hoc analysis with Bonferroni correction revealed that two conditional pairs were significantly different: the tutorial vs. canvas conditions ($W=226,Z=-5.00$, $p<.01/3$) and the game vs. canvas conditions ($W=272.5,Z=-3.72$, $p<.01/3$). The scores on the post-test between the tutorial and game conditions did not show a significant difference ($W=365.5,Z=-1.20$, n.s.). Based on these findings, we reject our null hypothesis.

These results indicate that though all the participants had approximately the same programming knowledge during the pre-test, participants from the tutorial and game condition performed significantly better on their post-test, and that their degree of improvement was also significantly greater than that of the canvas condition. As seen in Table 7.2, the effect sizes of learning gains were 140% and 100% increase in scores for the tutorial and game conditions, respectively, whereas the median score from the canvas condition did not change significantly (and were actually 9.1% worse). Since participants had little programming knowledge to start with and there was no difference in demographics, the learning activities are likely the primary cause of the increase in scores for the tutorial and game condition participants.

7.3.2. Differences in Percent Increase of Scores

Although we had a relatively small sample size of 20 participants per condition, we found consistent patterns, particularly in the tutorial and game conditions, where participants made large percent gains answering questions correctly in their post-tests compared to their pre-tests (see Tables 7.3 and 7.4). As we saw in Chapter 7.3.1, the tutorial and game condition participants performed much better than their canvas condition counterparts. This was particularly true for the basic concepts (i.e., variables, mathematical and relational operators, and Booleans), logical operators, while loops, for loops, function parameters, and function returns, where they increased their rate of correct answers by at least 100% in their post-test compared to their pre-test.

Table 7.3. The Learning Study's percent increase between pre-test and post-test scores.

Question + Concept (actual questions ordered randomly)	(posttest - pretest) / pretest		
	tutorial	game	canvas
Q1 basics	**175%***	60%	-40%
Q2 basics	**120%**	60%	0%
Q3 basics	**100%**	50%	0%
Q4 logical operators	***175%****	**133.3%**	-66.7%
Q5 logical operators	***125%***	**120%**	-40%
Q6 logical operators	***150%***	**166.7%**	-20%
Q7 if / else	100%	100%	20%
Q8 if / else	100%	80%	0%
Q9 if / else	80%	100%	0%
Q10 arrays	75%	100%	-33.3%
Q11 arrays	60%	50%	-40%
Q12 arrays	100%	50%	-33.3%
Q13 while	***225%***	***166.7%***	-60%
Q14 while	***333.3%***	***266.7%***	0%
Q15 while	***266.7%***	***125%***	0%
Q16 for	**100%**	**66.7%**	-50%
Q17 for	**200%**	**133.3%**	0%
Q18 for	**80%**	**100%**	-40%
Q19 function parameters	***140%***	***166.7%***	25%
Q20 function parameters	***233.3%***	***200%***	0%
Q21 function parameters	***233.3%***	***200%***	25%
Q22 function return	**166.7%**	***200%***	-50%
Q23 function return	**80%**	***166.7%***	-33%
Q24 function return	**125%**	***160%***	0%

*Groupings with a mean greater-than-or-equal-to 150% are in **bold**.
Groupings with a mean greater-than-or-equal-to 150% are also ***italicized in red*.

Tutorial and game condition participants made the largest improvements (greater than or equal to 150% increase) with *while loop* and *function parameters* concepts. Tutorial condition participants also made these large improvements answering questions about *logical operators*, while the game condition participants also made similarly large improvements answering questions about *function returns*. These results indicate that the tutorial and game conditions' learning activities were successful in helping their participants learn about all the concepts we tested for.

Canvas condition participants did not do well compared to their counterparts. Although we know from Chapter 7.3.1 that the canvas condition participants did not do significantly worse on their post-tests compared to their pre-tests overall, Table 7.3 shows that they struggled answering many of the post-test questions, actually performing worse on many concepts in the post-test, despite encountering the identical questions. This suggests that in some cases, learning activities promoting exploration and creation without guidance might actually lead to confusion.

These results indicate that online, educational tutorial and game resources can be successful at teaching users about programming concepts, but that open-ended creative resources, at least in solitary, discretionary settings are likely not. Tutorial and game condition participants' scores indicate that there are large, measurable learning outcomes (see last row of Table 7.2), and that these learning activities might teach certain concepts better than others (see above and italicized, red text in Table 7.3).

7.3.3. More Time on Exams for Tutorial & Game Condition Players

During the pre-test, participants from all conditions spent roughly the same amount of time on their exams ($\chi^2(2,N=60)=5.39$, n.s.) (see Figure 7.4 and Table 7.4). However, when we examine the time they spent on their post-test, there is a significant difference in time spent by condition ($\chi^2(2,N=60)=17.87,p<.01$). Doing post-hoc analysis with Bonferroni correction, we found that the tutorial participants spent significantly more time on the post-test than the canvas condition ($W=288.5$, $Z=-3.29$, $p<.01/3$); the same was true of the game vs. canvas conditions ($W=263.5,Z=-3.96$, $p<.01/3$). The time spent on the post-test between the tutorial and game conditions did not show a significant difference ($W=417$, $Z=0.18$, n.s.).

Table 7.4. The Learning Study's summary statistics for activity times.

	tutorial (n=20)	game (n=20)	canvas (n=20)
Minimum time on pre-test	20 minutes	22 minutes	20 minutes
Median time on pre-test	25.5 minutes	28 minutes	26 minutes
Maximum time on pre-test	33 minutes	31 minutes	41 minutes
Minimum time on activity	7.0 hours	3.61 hours	1.25 hours
Median time on activity	9.25 hours	4.76 hours	1.94 hours
Maximum time on activity	14.0 hours	7.22 hours	2.98 hours
Minimum time on post-test	23 minutes	29 minutes	19 minutes
Median time on post-test	35 minutes	34 minutes	24 minutes
Maximum time on post-test	55 minutes	42 minutes	35 minutes

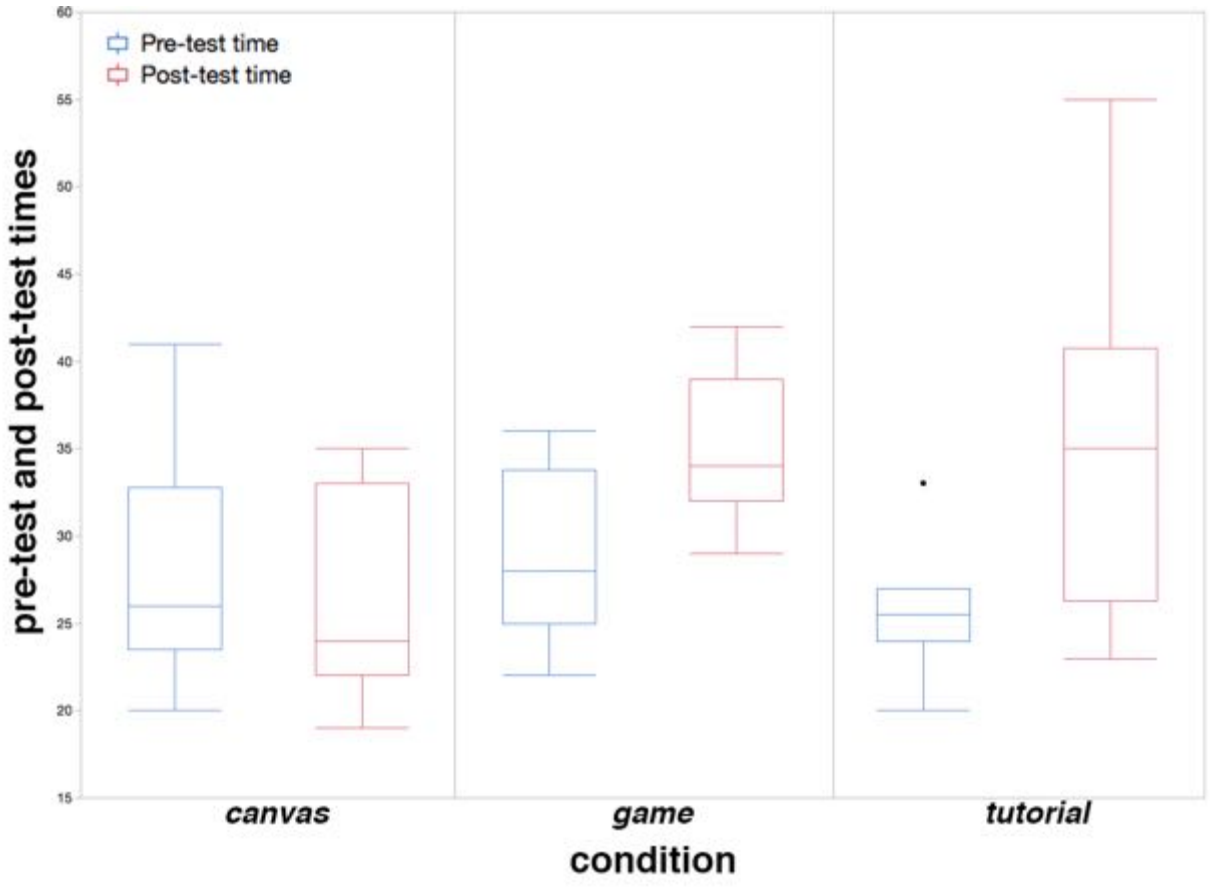


Figure 7.4. Comparison of Learning Study's pre-test and post-test play time by condition.

7.3.4. Differences on Learning Activity Time

Each of the three learning activities had largely different estimated times for completion (10 hours for Codecademy, 5 hours for Gidget, and open-ended for the GPD). Examining the time spent on each task (see Table 7.4), we see that there was indeed a significant difference in time participants spent on their respective activities ($\chi^2(2,N=60)=52.34$, $p<.01$). With a post-hoc analysis with Bonferroni correction, we found that all pairwise comparisons were significantly different: tutorial vs. canvas (W=210, Z=-5.40, p<. 01/3), tutorial vs. game (W=211, Z=-5.37, p<.01/3), and game vs. canvas (W=210, Z=-5.40, p<.01/3).

Combined with the large difference on exam performance in the post-test and pre-test (from Chapter 7.3.1), this suggests that the game condition was the most efficient of the three conditions at improving participants' post-test scores. Examining Table 7.4 reveals that the tutorial condition participants took nearly twice as long as the game condition participants to complete their assigned learning activity that covered the same materials (see Chapters 7.2.1 and 7.2.2), but performed similarly in their post-test (from Chapter 7.3.2), further demonstrating that the game condition participants were most efficient at improving their post-test scores.

7.3.5. No Significant Demographic Differences in Test Scores

We found that there were no significant differences in learning gains within the groups by gender. Respectively, the pre-test and post-test scores for each condition were: tutorial (W=118, Z=0.96, n.s.) & (W=115, Z=0.72, n.s.); game (W=102.5, Z=0.59, n.s.) & (W=108, Z=1.00, n.s.); and canvas (W=106, Z=0.85, n.s.) & (W=112, Z=1.33, n.s.). This indicates that males and females all performed similarly within their respective conditions.

Next, we used a simple linear regression for each condition's pre-test and post-test to predict test scores based on age. No significant correlation was found between test scores or age for any of the conditions in either of the tests. Respectively, the pre-test and post-test scores for each condition were: tutorial ($F(1,18)=0.10$,n.s.;$R^2=0.01$) & ($F(1,18)=0.27$,n.s.;$R^2=0.01$); game ($F(1,18)=0.15$,n.s.;$R^2=0.01$) & ($F(1,18)=0.46$,n.s.;$R^2=0.03$); and canvas

($F(1,18)=2.32$,n.s.;$R^2=0.11$) & ($F(1,18)=0.39$,n.s.;$R^2=0.02$). This indicates everyone performed similarly within their respective conditions, regardless of their age.

For completeness, we also examined if prior education (as measured in Table 7.1) had an effect on pre-test and post-test scores by condition. We found no significant differences within groups by education. Respectively, the pre-test and post-test scores each condition were: tutorial ($\chi^2(2,N=20)=1.49$,n.s.) ($\chi^2(2,N=20)=4.30$,n.s.); game ($\chi^2(2,N=20)=0.51$,n.s.) ($\chi^2(2,N=20)=3.95$,n.s.); and canvas ($\chi^2(2,N=20)=0.93$,n.s.) ($\chi^2(2,N=20)=1.66$,n.s.). This indicates everyone, regardless education, performed similarly within their respective conditions.

7.3.6. No Significant Demographic Differences in Test Time

We examined if gender had any effect on the time participants spent on the pre-tests and post-tests by condition. We found that there were no significant differences within the groups by gender. Respectively, the pre-test and post-test times for each condition were: tutorial (W=113.5, Z=0.62, n.s.) & (W=103, Z=-0.11, n.s.); game (W=85, Z=-0.69, n.s.) & (W=108.5, Z=1.03, n.s.); and canvas (W=86.5, Z=-0.57, n.s.) & (W=108.5, Z=1.04, n.s.). This indicates that males and females all spent a similar amount of time on their tests within their respective conditions.

Next, we used a simple linear regression for each condition's pre-test and post-test to predict the time spent on tests based on age. No significant correlation was found between the time participants spent on the tests and their age for any of the conditions in either of the tests. Respectively, the pre-test and post-test scores for each condition were: tutorial ($F(1,18)=0.28$,n.s.;$R^2=0.016$) & ($F(1,18)=0.09$,n.s.;$R^2=0.005$); game ($F(1,18)=1.20$,n.s.; $R^2=0.06$) & ($F(1,18)=0.14$,n.s.;$R^2=0.008$); and canvas ($F(1,18)=2.60$,n.s.;$R^2=0.13$) & ($F(1,18)=0.30$,n.s.;$R^2=0.59$). This indicates everyone spent a comparable amount of time on their tests within their respective conditions regardless of their age.

Finally, we examined if education had any effect on the time spent on the pre-tests and post-tests by condition. We found no significant differences within groups by participants' level of education. Respectively, the pre-test and post-test scores for each condition were: tutorial ($\chi^2(2,N=20)=2.02$,n.s.) & ($\chi^2(2,N=20)=0.34$,n.s.); game ($\chi^2(2,N=20)=0.04$,n.s.) & ($\chi^2(2,N=20)=3.15$,n.s.); and canvas ($\chi^2(2,N=20)=0.94$,n.s.) & ($\chi^2(2,N=20)=3.49$,n.s.). This

indicates everyone, regardless of their level of education, spent a similar amount of time on their tests within their respective conditions.

7.4. DISCUSSION

Our findings show that online discretionary resources for computing education such as tutorial websites and games can be successful in teaching novices programming concepts without the need for additional external help. Even with relatively small sample sizes, we were able to see large differences in the time players spent on the learning activities, the exams, and their exam scores. All participants performed consistently within their own groups, without any significant differences in the time they spent on either the pre-tests or post-tests, the time on their learning activities, or on their exam scores. This consistency is also reflected in participants' demographics, which showed no differences between males or females, people of different ages, or level of education, within all conditions. This is particularly important for online discretionary learning, because our results indicate that all of our learning activities were gender-neutral, with everyone performing at equal levels within their respective conditions, which does not typically happen in programming-related classroom settings (Rubio et al. 2015; Werner, Hanks, & McDowell 2004).

We found that participants in the tutorial and game conditions significantly increased their overall post-test scores by over 100% in comparison to their pre-test scores (see Table 7.2). These participants showed considerable gains for similar questions (see Table 7.3), suggesting that the learning activities from both the conditions taught similar concepts and also taught them equally well. Although these participants showed improvements across all the concepts we tested (see Table 7.3), the highest increases were in: basics, logical operators, while loops, for loops, function parameters, and function returns. Moreover, participants from the tutorial condition appeared to do slightly better on logical operator questions while participants from the game condition did slightly better on the function return questions. We examined the instruction of these two concepts in both Codecademy and the Gidget game, but did not find anything obviously different from the modules teaching those specific concepts from the other concepts within the same learning activity. Like the rest of the interactions within those groups,

Codecademy had its users follow step-by-step instructions entering code into its IDE, and the Gidget game required participants to look through, diagnose, and fix broken code like every other level. None of these findings were true for the canvas participants, indicating this condition's learning activity failed to teach the same concepts even though all the necessary help resources were available to users.

We did not find any significant difference in the time participants spent on their pre-test exams. However, participants in the tutorial and game conditions spent significantly more time on their post-tests compared to their canvas condition counterparts. This suggests that those in the tutorial and game condition found more reason to concentrate and take their time on their respective post-test exams, possibly because they were better equipped to answer the questions correctly. Conversely, without clear goals or instruction in the Gidget Puzzle Designer, participants in the canvas condition likely did not learn the concepts necessary for them to engage successfully with the post-test.

Although it is understandable that novices scored poorly on the pre-test since it was their first time seeing programming code, examining the overall scores for both the pre-test and post-test exams may give the impression that the exams were too difficult (i.e., the highest median score was 12 out of 24 questions correct). However, novices performing badly on programming-related concepts they recently learned is not uncommon (Bonar & Soloway 1985; Lister et al. 2004; McCracken et al. 2001; Soloway 1986). Comparable scores were also reported by Tew (2011) who administered a similar pseudo-code test to students in various introductory programming courses. Since these test questions were generated by combining scores from a crowdsourced group of novices and experienced programmers, our results suggest that there is a major gap in programming knowledge between beginners and those with more experience, and that one short exposure with code, whether it be an online tutorial, online game, or even a formal high school or university class (Tew & Guzdial 2011), may be enough to show some learning outcomes, but is still far from mastery of the subject.

There was a large difference in the time people spent on their respective learning activities. Learners spent the most time on the Codecademy course and spent the least amount of time using the Gidget Puzzle Designer. The time spent on the Codecademy and Gidget game tasks are not

surprising, given that they are close to the developers' estimated time to complete the activity. It is also not too surprising that participants spent the least amount of time on the Gidget Puzzle Designer task. Without any clear goals or instructions, the participants likely lacked the motivation required to go beyond tinkering with the interface a bit. This means that goals are important for engagement with an activity, and that without proper motivation, people are likely to disengage with the activity. This is particularly worrisome for discretionary learning resources, because this one negative/boring experience might cause a lasting impression of programming being just that and the user deciding that computer science is not for them based on this one experience. More generally, it may be that open-ended, but solitary creative learning tasks such as these fail to engage online with more substantial extrinsic motivators, such as teachers, online community, and more directed creative tasks.

7.5. LIMITATIONS

This study had the same limitations as those mention in the prior studies (see Chapters 4.5, 5.5, and 6.5), including self-selection into a Mechanical Turk HIT, prior computing ability, and level of education. There was an economic incentive for participants to participate in the study. We tried to minimize this effect as much as possible. We believe that the economic incentive in our study was minimal, as usage data for both Gidget and Codecademy show that thousands and millions of people have used these systems without being paid to play.

We gave participants up to 7 days to complete their assigned activity. Although we asked them to refrain from using any other resources to learn or practice programming, participants could have potentially learned coding concepts from other places, even if it was unintentional. Learning or practicing programming concepts outside of the assigned task could have potentially affected exam outcomes, but could have happened in all conditions. However, unlike the numerous resources to get guidance for Python, there are no external resources to get explicit help for Gidget.

Finally, part of our Codecademy data was reliant on self-reported data that participants provided, including the time they spent on the task and how far they got in the course. Although we asked participants to stop at the "advanced concept" modules so the learning interventions

were as similar as possible, we had no way of enforcing that since Codecademy is a third-party website.

7.6. SUMMARY

This chapter investigated the learning outcomes of three different types of programming resources designed for beginners. By comparing the test scores of learners before and after their respective learning activities, we found that the learners who took a Codecademy course and the learners who played through the Gidget game showed considerable improvement in their test scores. Though this was true of both cases, learners who played the Gidget game were able to match the post-test performance of learners who completed the Codecademy tutorial, in approximately half the time. Furthermore, we found that these participants also spent more time on the post-test exam, suggesting that they found reason to try harder the second time taking the exam. In contrast, those who were assigned to create programs from scratch using the Gidget Puzzle Designer spent approximately the same amount of time on their pre-test and post-test exams, and did not show significant improvements in their post-test exam scores. In addition to these differences, we found that performance by demographics was consistent within all the conditions, meaning that all three of our learning activities worked equally well (or equally worse in the case of the canvas condition) regardless of gender, age, or level of education. Given these results, we conclude that players of an educational debugging game do show measurable learning outcomes of programming concepts covered in a typical CS1 course and they are able to transfer their understanding of these programming concepts from their language of instruction to pseudo-code.

8. OUTREACH ACTIVITIES AND PUBLIC RELEASE[9]

This chapter addresses RQ3 – who is playing the educational debugging game? More specifically, I ran four summer camps to see if the game appeal to underrepresented groups in computing. I also released the game to the public to learn more about who is attracted to this type of game, and how these users do playing the game.

8.1. MOTIVATION

Since I view research being as much about impact as it is discovery, I wanted to confirm that Gidget is appealing to underrepresented groups in computing, and that could reach a wide and diverse audience. Currently, little is known about who is actually using the many available online learning resources. As Chapter 1.1 mentioned, few (if any) of these resources report anything beyond the number of users that have signed up for their services and how many activities their users have completed. This lack of evaluation makes it unclear how useful these tools are beyond merely engaging learners for a brief period of time and how they might work with underrepresented groups. Without this knowledge, we risk designing instructional tools that are not actually helpful for learners (Garris, Ahlers, & Driskell 2002).

8.2. OUTREACH ACTIVITIES FOR UNDERREPRESENTED GROUPS

I ran a total of four summer camps over two consecutive years on college campuses in Corvallis, Oregon, and in Seattle, Washington. Each summer consisted of two camps – one at each location, focusing on two different kinds of underrepresented groups in computing. The Oregon camps served male and female teenagers from rural communities (students from Corvallis and its surrounding towns) while the Washington camps served only female teenagers.

The summer camps were intended to expose teenagers to programming, and to provide enrichment activities related to computing. Each camp ran 3 hours per day for 5 days, for 15 hours total. About 5 hours were devoted to the Gidget puzzle curriculum; 5 hours to other

[9] Parts of this chapter describing summer camps have been adapted from my VL/HCC 2014 publication (Lee et al. 2014) and an unpublished manuscript (Jernigan et al. 2015).

activities such as icebreakers, guest speakers, and breaks; and 5 hours to creating new levels with the puzzle designer and sharing them. Campers' parents were invited to attend the last 1.5 hours of the camp to learn about the camp activities and play through their teenagers' levels. The camps used identical staff members for each pair of summer camps. The staff provided no formal instruction about Gidget or programming and redirected participants' requests for assistance to the game's help system whenever possible.

8.2.1. Camp Participants

We had a total of 68 participants in our four summer camps over two years who ranged from 13 to 19 years old (see Table 8.1 for a detailed breakdown). For both years, we had 18 and 16 campers in each of the Oregon and Washington camps, respectively. Participants were divided into same-gender pairs of similar age and were instructed to follow pair programming practices, which are known to benefit both males and females (Werner, Hanks, & McDowell 2004). Three participants had some prior experience with programming (one camper from each of the Washington camps, and one from the first year's Oregon camp). All other participants had no prior programming experience. Staff members took extra care to ensure that teammates paired with more experienced partners got their fair share of time working on the puzzles and creating their own levels.

A total of three campers (one from the first Washington camp, and two from the second Washington camp) were withdrawn from the camp by their parents after completing the first day. The summer camp coordinator informed us that these parents did so because they "did not want their child[ren] spending [summer camp] time playing a game." Although the camp coordinator

Table 8.1. The campers' demographics.

	Summer 1		Summer 2	
	Oregon (n=18)	Washington (n=16)	Oregon (n=18)	Washington (n=16)
gender	10 males 8 females	0 males 16 females	7 males 11 females	0 males 16 females
age	13-19 years median = 13.5	13-19 years median = 14	13-17 years median = 15	13-17 years median = 15

Table 8.2. Summary of the levels created by the campers.

	Summer 1		Summer 2	
	Oregon (n=18)	Washington (n=18)	Oregon (n=18)	Washington (n=16)
levels created	2-10 levels median = 5		1-12 levels median = 6.5	
total levels created	101 levels		109 levels	

reassured the parents of these three teenagers that Gidget was a legitimate educational tool designed to engage novices and teach then programming concepts, none of them re-enrolled their children in the camp. These withdrawn campers are not included in our statistics in Table 8.1.

8.2.2. Results & Discussion

After only about 5 hours of self-directed instruction with our debugging game, participant teams from all four of our camps created a total of 210 Gidget levels, with every team applying programming concepts they encountered playing the game in this creation process. We examined these participant-created levels, focusing particularly on the programming concepts used in the

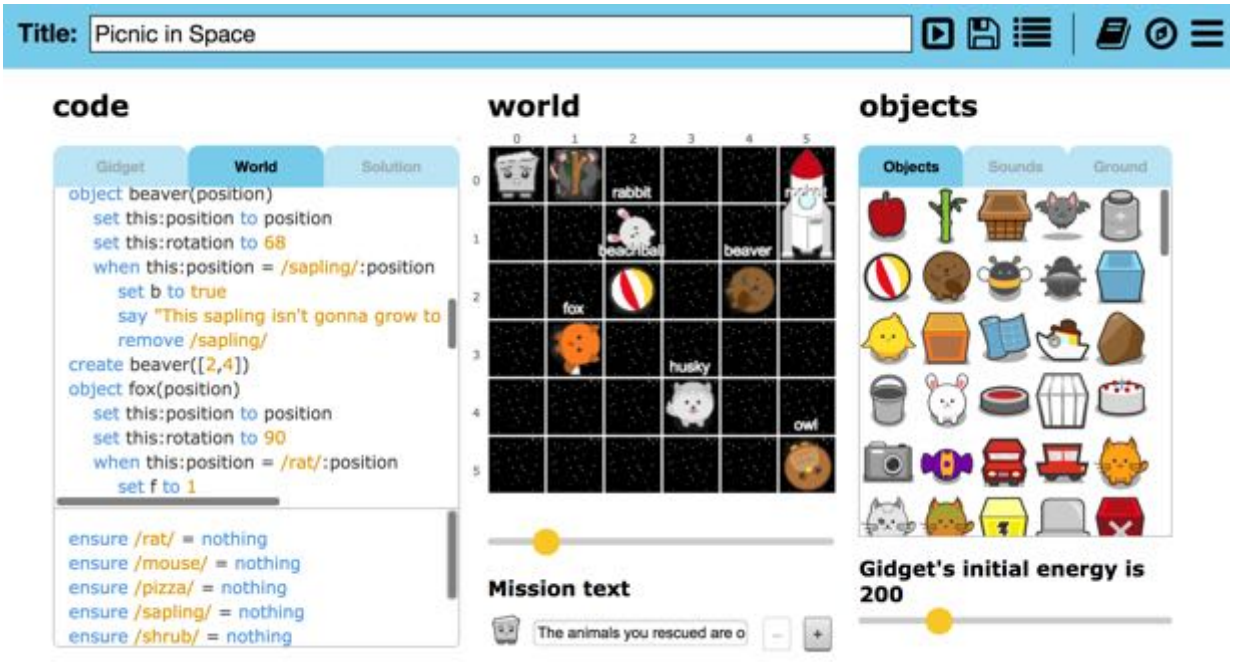


Figure 8.1. Most campers' levels included several programming concepts and a storyline.

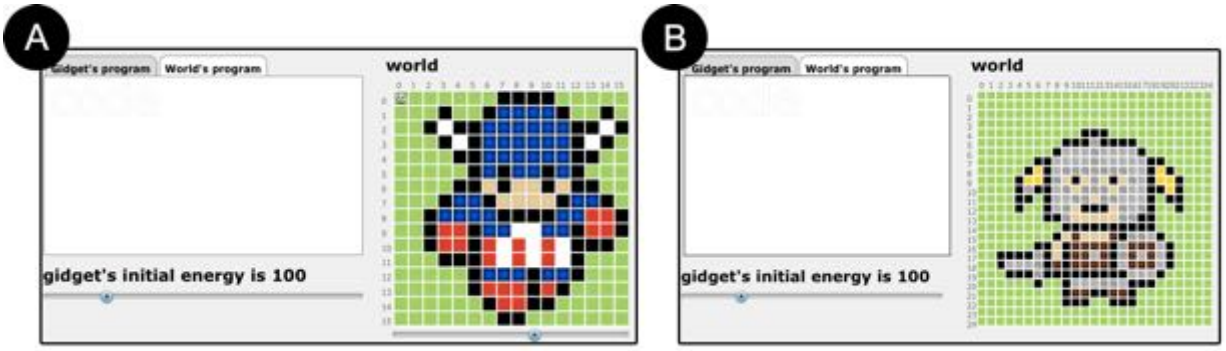


Figure 8.2 Some teams from the first Oregon camp used Gidget to create pixel art.

levels and the level's story, as storytelling elements in these environments are known to affect engagement (Kelleher, Pausch, & Kiesler 2007; Kerr et al. 2013).

Each team created between 1 and 12 levels (see Table 8.2). Most campers created Gidget puzzles (see Figure 8.1) or mazes meant to challenge other players, but some participants also had partially-completed or proof-of-concept levels. Some participants repurposed the level designer for unintended functionality. For example, some teams built levels to hold solutions to their other levels, some made story-related levels without any puzzle-solving elements, and a few teams used the level designer to draw pixel art (see Figure 8.2).

Every team designed two or more complete levels that used at least one of the programming constructs they encountered while playing through the came curriculum. The minimum knowledge to create a Gidget level is a Boolean expression to indicate a goal. Non-trivial Gidget levels (such as Figure 8.1) require knowledge of variables, Booleans, objects, and events. All teams used at least one Boolean expression in their levels since it was mandatory to have a goal

Figure 8.3 Campers taught their parents how to play Gidget using their levels.

(written as an assertion). Some teams demonstrated their knowledge by writing their own incomplete puzzle code containing functions and loops for other players to debug.

Most teams motivated their levels using stories in Gidget's mission text: over 82% of the teams motivated at least one level with story text. Approximately 20% of all the teams created multiple levels with a continuous story thread. The Gidget character was popular as a domestic figure (having a house or partner) or as an altruistic hero (often rescuing animals in outer space). None of our participants developed stories focused on popular culture as observed in other camp studies (Maloney et al. 2008); this may have been due to participants treating Gidget as a character upon which they could build their own ideas.

Parents were invited to come during the last 1.5 hours of the camp to see what their children had been working on over the week. Campers used this opportunity to teach their parents how to program using Gidget, and challenged them (and their peers) to complete their levels (see Figure 8.3). Throughout the four camps, we received overwhelmingly positive feedback from the parents about the camp activities. Many parents asked if there were follow-up courses available immediately or the following summer. Parents' enthusiasm for Gidget was mirrored by the campers themselves, many asking if they could continue to play Gidget from their homes. This sentiment is best illustrated by Figure 8.4, which shows an unprompted doodle by one of the participants on her exit survey stating that she loves Gidget.

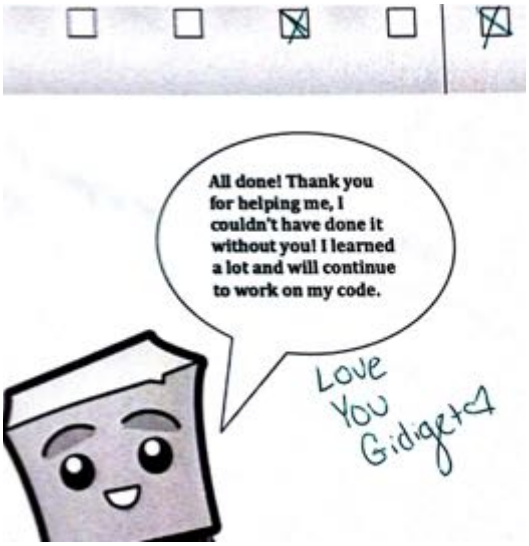


Figure 8.4. A camper's unprompted doodle expressing her affinity towards Gidget.

8.2.3. Summary

All the teams completed the Gidget game in 5 hours or less with no instruction outside of the game, and minimal help from staff members. In this relatively short time allocated to level design, campers were able to try out many ideas and share results with their peers at every stage

of their progress. Despite the fact that the level designer had the constraints of a 2D world and Gidget rules, our campers used it to not only program challenging puzzles, but to also tell imaginative stories.

Overall, the camps were a great success with both parents and campers expressing their affinity towards the game. We were able to successfully reach and two underrepresented groups in computing and provide them with a positive experience with coding. However, our experience with three camp withdrawals suggests that some people (especially adults/parents) may have negative preconceptions of games used as educational tools. We hope with the ever-increasing number of discretionary learning resource online, and positive experiences from games like Gidget, these biases against educational games will decrease with time.

8.3. PUBLIC RELEASE

To give a wider audience access to the game, I released Gidget to the public, announcing it via social media (Facebook and Twitter). This version of Gidget was a culmination of all the studies described in the previous chapters with a few updates to fix minor bugs, address usability issues, add new graphics/objects, and improve the account creation processes (see Figure 8.5-A). It also gave visitors the option to play the game immediately and create an account later to save their progress (see Figure 8.5-B). This was done to allow instant access to the game for those who wanted to quickly see and interact with the game without the need for any kind of commitment.

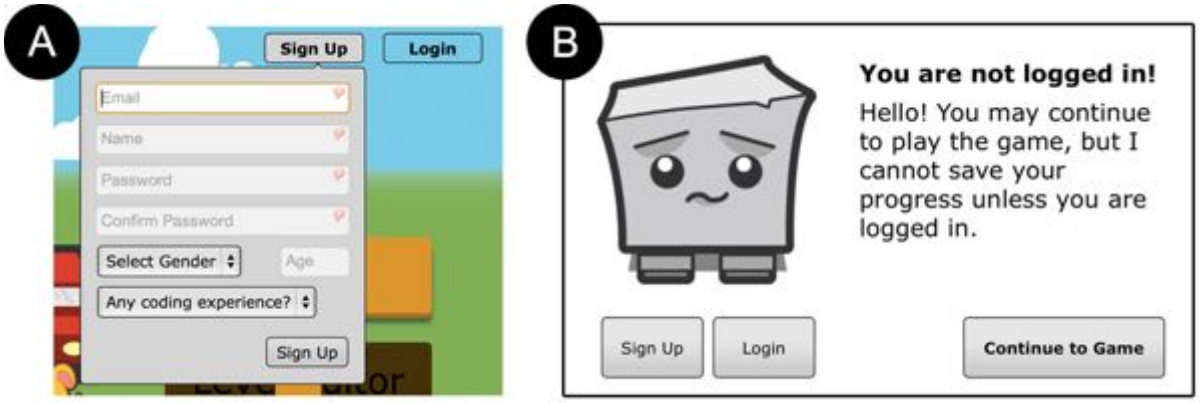


Figure 8.5. Screenshots of the account creation prompt and instant access warning.

Players wanting to save their progress had to create an account. The account creation prompt asked for: name, e-mail address, password, and age (see Figure 8.5-A). It also had two dropdown menus that required the player to select their gender, and one to select either "yes" or "no" to the following prompt, "Any coding experience?". Once the player validated their e-mail address (by clicking on a link provided by an automated e-mail message), they were able to login to the game using their e-mail address and password.

8.3.1. Online Players

A total of 3,023 people have played the game since its deployment (see Table 8.3). Among these players, 844 individuals (27.9%) made accounts to save their progress (called the *saved-account group*). The logs indicates that account holders are comprised of 54.8% males and 45.2% females, with an overall median age of 19 and a range from 6 to 65 years old (see Table 8.3). The game also collected data for those who played the game without making accounts (called the *no-account* group), but did not include the player-reported information such age and gender.

Table 8.3. Player information from the public release.

total number of players	3023
total accounts created	844 of 3023 = 27.9%
gender distribution	male: 54.8% female: 45.2%
age (minimum, median, maximum)	**overall: 6, 19, 65 years** male: 6, 20, 62 years female: 7, 19, 65 years

Visitors came from all over the world, with the most coming from the USA, Brazil, the United Kingdom, Russia, and Canada (see Table 8.7 and Figure 8.6). The majority of players came from the USA, making up 80.31% of the total visitors to the Gidget website. The states of Illinois (26.76%), Washington (22.71%), and California (11.19%) represented 60.46% of the total number of visitors from the USA (see Figure 8.7). Inspecting Figure 8.7 also reveals that Gidget's visitors are largely from urban/metropolitan areas.

8.3.2. Results & Discussion

Here, I compare the differences between the no-account and saved-account groups, and also look for gender and age difference within the latter. For analysis, I use non-parametric Chi-Squared

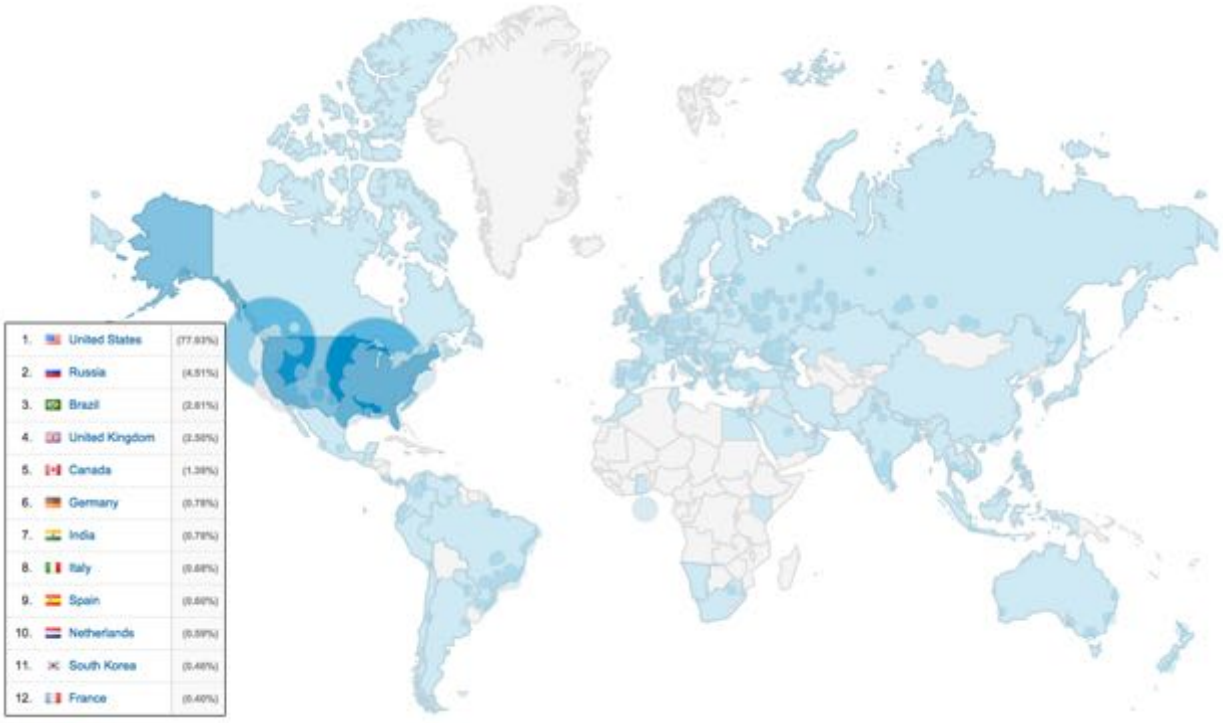


Figure 8.6. People from all over the word have played Gidget.

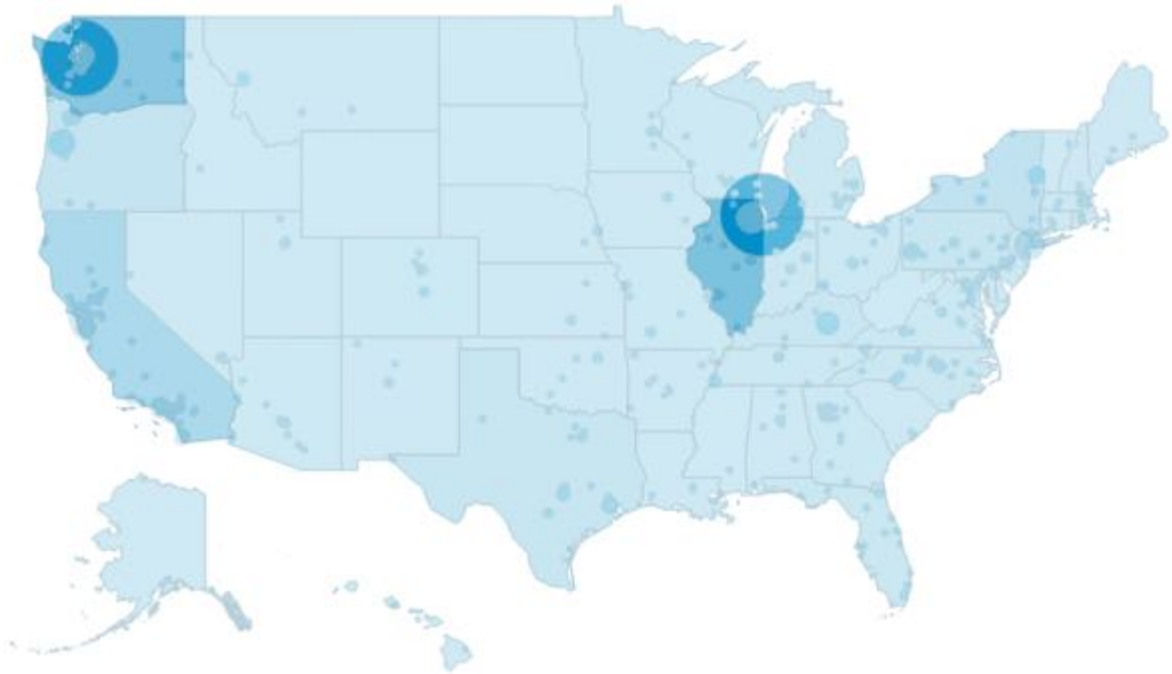


Figure 8.7. Players mostly come from urban areas within the USA.

and Wilcoxon rank sums tests with α=0.01 confidence, as the data were not normally distributed.

8.3.2.1. Saved-account players complete more levels and play longer than no-account players

Table 8.4 describes the descriptive statistic for the no-account and saved-account groups. Players in the saved-accounts group completed a median of 6 levels, ranging between 0 and 37 levels (i.e., completing the game). They played for a median of 32.6 minutes, ranging between 1 minute and 5.22 hours (see Table 8.4). In contrast, players in the no-account group completed a median of 1 level, ranging between 0 and 30 levels. Their median time playing the game was 12.2 minutes, ranging from 1 minute to 4.94 hours (see Table 8.4). Nobody from the accounts group completed the game. As seen in Figure 8.8, many people did not progress after completing the first level. This was especially true for the no-account group players, where 57.3% of players completed a maximum of 1 level (29.4% completed 0 levels; 27.9% completed 1 level). Whereas only 23.2% of saved-account group players completed 0 (3.1%) or 1 (20.1%) levels.

There was a significant difference in the number of levels completed by account status ($W=1150975.5$, $Z=21.75$, $p<.01$), where the saved-account group players completed significantly more levels compared to those in the no-account group. Similarly, there was a significant difference in the time spent playing the game ($W=1027244$, $Z=15.41$,$p<.01$), where saved-account players played much longer than those in the no-account condition.

The results indicate that the saved-account group completed more levels and spent more time on the game than the no-account group. The no-account group's results may be an indication that there are many (even experienced programmers) who are interested in puzzles or

Table 8.4. Summary statistics from the public release.

	no-account (n=2179)	saved accounts (n=844)
minimum levels completed	0	0
median levels completed	1	6
maximum levels completed	30	37
minimum play time	1 minute	1 minute
median play time	12.2 minutes	32.6 minutes
maximum play time	4.94 hours	5.22 hours

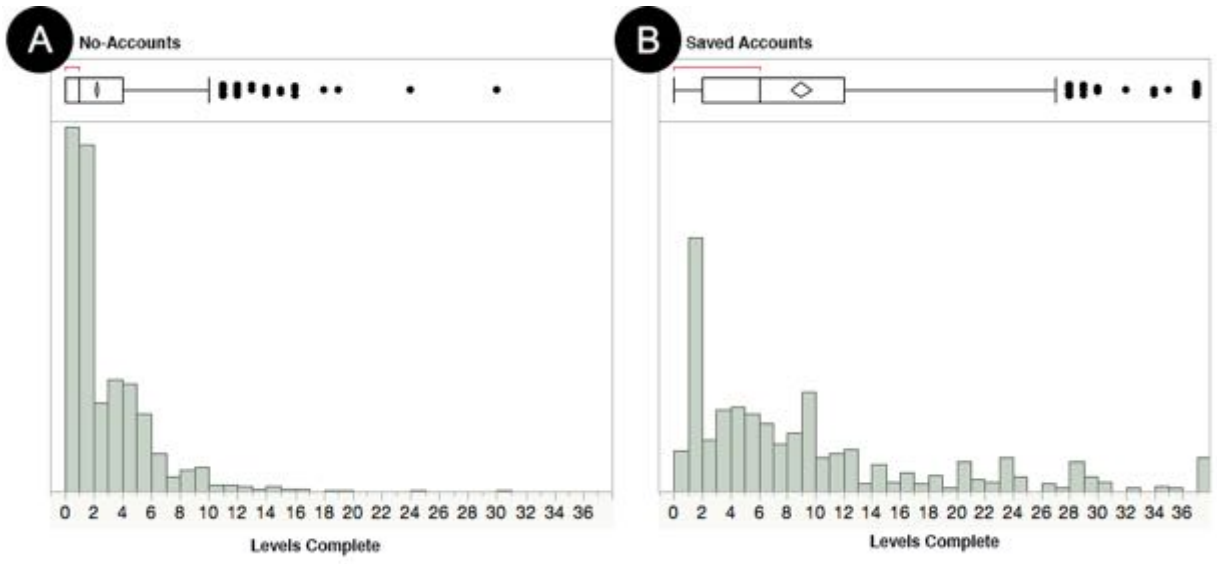


Figure 8.8. Most players quit after the first level, but account holders (B) finish more levels.

even coding, but ultimately decided that the game was not a fit for them. However, it is also probable that these results are skewed, as someone could have started the game immediately, completed several levels, played for some stretch of time, and then decided to make an account to save their progress (the game did not keep track of who converted their no-account to a saved-account). Cases like this may explain the no-account group's low play time and levels completed, especially compared to the saved-accounts group (see Table 8.4 and Figure 8.8). In either case, it appears that allowing players to play the game without making an account immediately was a good design choice since it allowed more people to try out the game and get some exposure to a coding activity. However, we should update the game to collect more information about these types of players so we can better understand how we might be able to have more them play longer and have a positive experience with the game.

8.3.2.2. Saved-accounts: No difference by gender, but difference in levels completed by age

Next, I looked exclusively at the saved-account players, testing to see if there were any differences in the number of levels completed or time played by either age or gender. There was no significant difference in the number of levels completed by gender (W=68783, Z=-1.65,n.s.). Similarly, there was no significant difference in the game play time by gender (W=73757.5,

Z=0.23,n.s.). However, a simple linear regression to predict the number of levels completed based on age was significant ($F(1,818)=35.68, p<.01.; R^2=0.059$), with a younger age correlating with a higher number of levels completed (see Figure 8.9). Another simple linear regression showed no significant correlation between the time players spent playing the game and their age ($F(1,818)=0.64$,n.s..;$R^2=0.001$). These results indicate that younger players completed more levels than older players in comparable amounts of time.

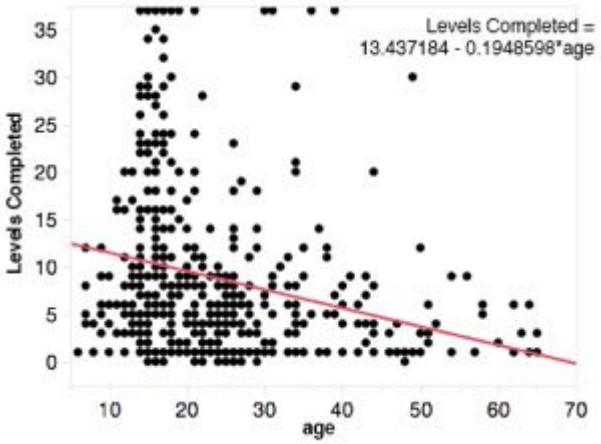


Figure 8.9. Relationship between saved-account holders' age and levels completed.

It was unsurprising that males and females performed similarly in level completion and play time in the saved-account group. My observations from past studies (described in Chapters 4-7) and summer camps also did not find any difference in male and female performance or behavior with the game. This might be attributed to our principle of gender-inclusiveness (described in Chaoter 3 and Table 3.1), where we designed the game to appeal to all genders.

Conversely, it was surprising that age was inversely related to the number of levels completed, as none of my past studies (described in Chapters 4-7) found differences in Gidget players by age. This may be partly explained by the large number of younger players compared to older players (the median age of saved-account players was 19, with a range between 6-65 years; also see Figure 8.9), especially since the statistical coefficient is low. Although we designed Gidget to be appealing to a wide audience (Table 3.1 lists related principles: Principle 2 – keeping the game engaging and entertaining, and Principle 7 – keeping the game gender-inclusive), younger people may be more familiar with games than older individuals, and have completed more levels because they found the game particularly entertaining. Also, since the game was advertised through social media and word-of-mouth, certain age groups may have been more likely to share the link to the game with their peer groups.

8.3.3. Summary

Over 3,000 people from all over the world have played Gidget since its release, with nearly 30% making accounts to save their progress and play again later. Among those who indicated their gender, 54.8% reported being male and 45.2% 54.8%. These numbers are slightly better than recent figures reporting that 42% of online gamers are women (newzoo.com 2011), and a good indication of the successful integration of our principle for gender-inclusiveness (see Table 3.1) into the game.

We found that most people (72.1%) played without creating accounts, and 57.3% of these players finished either 0 or 1 levels (see table 8.4). Overall, this group played for a median of 12.2 minutes and completed a median of 1 level, with nobody finishing the game. In contrast, players with accounts (27.9% of all players) played for a median of 32.6 minutes, completed a median of 6 levels, and had several people (2.6%) complete the game. The players' age or gender did not have any significant relationship to the amount of time they played. Similarly, the players' gender did not have a significant relationship to the number of levels they completed. However, we found a inverse relationship with age and the number of levels a player completed, where a younger age was correlated to more levels completed.

Overall, Gidget's public launch was a success with thousands of people coming to play it. We now know that a debugging game can be attractive to a wide audience from all over the world, and that it can successfully engage a near even number of males and females. Unfortunately, the overall number of levels completed is relatively low. This was different from our experience in the summer camps where everyone completed the game playing it a few hours per day. However, participants in our our past controlled experiments (described in chapter 4-7) also performed similarly with level completion, with only a few finishing the game for each study. Taken together, these observations indicate that the game itself is engaging, but needs to do more to motivate people to come back and continue playing the game.

8.4. LIMITATIONS

The activities described in this chapter were primarily concerned with outreach and impact, so the reported results are not necessarily meant to generalize. In particular, our summer camps included a relatively small number of participants from a very specific group of underrepresented individuals in technology (females and those living in rural areas), within a narrow age range (13-19 years old). In addition, we did not actively advertise Gidget, only announcing it on social media and relying word-of-mouth from our players. This may have attracted certain types of people who are attracted to games, puzzles, or programming. In the future, I plan to work with entities such as code.org to have a version of Gidget that will be approved to be listed on their website. This will help attract millions of more people to play Gidget, especially during annual events such as code.org's annual Hour of Code event during CS Education Week (Beres 2014).

9. CONCLUSION AND FUTURE WORK

The main goal of this dissertation has been to investigate if and how an online debugging game can engage a wide audience of novices while showing measurable learning outcomes. The dissertation described a series of controlled experiments, summer camp outreach activities, and a public launch of the Gidget game, to demonstrate that a debugging game *can* be effective in engaging, teaching, and attracting a broad range of people.

More specifically on engagement, Chapter 4 showed that compiler/interpreter feedback given by a personified, fallibly-cast character increases players' engagement with the game. Similarly, Chapter 5 demonstrated that the purposefulness of goals is directly related to the objects players' interact with in the game, and that players are more engaged when the goals relate to animal characters. Finally, Chapter 6 established that assessments that are well-integrated into the flow and storyline of a game can engage players and make them more effective with their time on the game tasks.

In regards to learning, Chapter 7 showed that completing the Gidget game leads to similar learning gains as finishing a Codecademy tutorial course, in approximately half the time. Participants in these studies were able to demonstrate transfer from their learned language (i.e., the Gidget language or Python) to pseudo-code. They were also more engaged with the testing material, spending more time on their post-exams compared to their pre-exams.

Finally, this dissertation demonstrated that a debugging game like Gidget can be appealing to a wide range of people. Chapter 8 showed that teenagers from rural communities and females participating in a summer camp using Gidget found the camp informative and entertaining. Chapter 8 also described the success of Gidget attracting thousands of people from all over the world to play the game since its public launch.

I have discussed the key findings, insights, and limitations of my studies within each chapter of this dissertation. Next, I discuss some ideas for future research directions that can build on the work described in this dissertation and address some of its limitations. Lastly, I restate the major contributions of this dissertation, and close with some final remarks.

9.1. Future Direction

While Gidget is a great example of one approach to teaching a broad audience to code, the research possibilities in this space are deep and broad. For example, I imagine that there is great potential investigating how to 1) use *social features* to support learning and engagement, 2) create and evaluate *new pedagogies for teaching* coding using player-generated data, and 3) *broaden participation* in computer science, particularly for underrepresented groups.

I believe that *social features* will add a new and exciting dimension to online pedagogy that is currently not available. The learning technologies to teach coding I have encountered so far are largely solitary experiences. However, supporting social features related to collaboration, instruction, and play can be very powerful additions. For example, collocated pair programming and master-apprentice teaching have both been shown to greatly improve learning and productivity. In Chapter 8, we saw evidence of this woking successfully in my summer camps, where 68 participants, working in pairs, beat all of the Gidget levels and created 210 of their own levels. We also saw in Chapter 8's description of Gidget's public launch that a relatively small number of players completed the game. Adding social features to the game might be one way to incentivize more players to come and continue playing the game to completion and creating their own levels to share. New research should examine how we can we bring the positive benefits of pair (or group) programming and peer instruction into a game, what would it look like, and how would pairs interact, all while maintaining flow with the gameplay and continuing to keep players engaged and learning.

New *pedagogies for teaching* are needed to attract and inspire the new generation of people wanting to learn programming online. I am most interested in developing these new techniques for teaching programming through games. Unlike other disciplines that use multiple ways of teaching students the material, computer science largely relies solely on feedback of written code to teach (Guzdial 2014). As seen in Chapter 6, I have had great success adding assessments into Gidget – something that works well in formal classroom settings. New research can focus on adapting, developing, and evaluating other pedagogical tools or technologies that can be used for teaching computer science. These techniques might include different kinds of interactive

exercises, code reflection, code peer-review, and simulations. Along this thread, I can also leverage the hundreds of thousands of code versions and user interaction data collected during gameplay to provide new players with information or guidance that will help them learn new concepts and succeed at the game.

Broadening participation in computer science is important because it can open up many new opportunities for underrepresented groups. My work with Gidget is specifically designed to be inclusive for both males and females, and my four summer camps described in Chapter 8.2 recruited only females (since males are already well represented in computer science) or those living in rural areas. I have had thousands of people from all over the world play Gidget and succeed through many or all of the levels. My analytics data from Chapter 8.3 shows that 45% of the players (who provide gender information) are female, and that most people are coming from urban areas. I want to do better. I want to reach more females, more people from rural areas, and more people who have low literacy skills or are non-native English speakers. Although the analytics I have for Gidget is a good start to understand who is playing the game and where they are coming from, the research community still knows very little about the larger group of people using these sites. I plan to leverage my professional relationships with the co-founders and creators of other popular online computing education technologies including Codecademy (n.d.), Lightbot (n.d.), and CodeSpells (Esper 2014), to provide the research community with more insight into who are using these resources, so that we can develop new and better tools to broaden participation.

While each of these three research strands can be worked on separately, combining the results from all of them will help me reach my vision of teaching the world to code. Because my research ultimately involves teaching, working on these will be synergistic with my teaching goals and vice-versa. In addition, due to the interdisciplinary nature of my research and training, following this research trajectory will create opportunities to collaborate with researchers in other areas, such as computing education, learning sciences, computer-mediated communication, computer supported cooperative work, new media, and education research.

9.2. Summary of Contributions

The contributions I have made through this dissertation are broken down thematically as follows:

9.2.1. Guidelines & Technology

- A description of seven design principles that define the components needed to make an educational game that effectively engages and teaches introductory programming concepts to novices. (Chapter 3)
- An interactive system that embodies the seven design principles that constitute a debugging game. (Chapter 3)

9.2.2. Study Results

- Empirical evidence that novice programmers are more engaged with an educational game when compiler/interpreter feedback is personified. (Chapter 4)
- Empirical evidence that novice programmers are more engaged with an educational game when the game goals are made more purposeful using specific types of data elements. (Chapter 5)
- Empirical evidence that novice programmers are more engaged with an educational game when assessments are added at the end of each subject module. (Chapter 6)
- Empirical evidence that novice programmers can effectively and measurably learn introductory programming concepts using an educational game. (Chapter 7)
- Evidence that an educational programming game can attract a wide range of players. (Chapter 8)
- Knowledge about who is attracted to play the game. (Chapter 8)
    - Teenagers from underrepresented groups (females and those living in rural communities) were deeply engaged through summer camps. They were able to complete the game in approximately 5 hours and create their own expressive levels with minimal outside help.

- People from all over the world are playing the game. In descending order, the highest concentration of people come from urban/metropolitan areas of the USA, Russia, Brazil, the United Kingdom, and Canada. In total, players who made accounts consisted of 45.2% females (7-65 years old, median 19 years old) and 54.8% males (6-62 years old, median 20 years old).

9.3. Final Remarks

This dissertation has addressed and demonstrated the following thesis:

> *An online game can engage and measurably teach programming concepts covered in a typical introductory computer science (CS1) course to a wide range of learners.*

I have provided a definition of an educational debugging game using seven design principles and evidence that this kind of game can be a viable educational tool that can engage novices and produce measurable learning outcomes in its players. My studies related to engagement have shown that small changes can lead to major effects in players' interactions with the game. My studies about learning have shown that an educational game can be just as effective (and even more efficient) at teaching novices introductory programming concepts as popular online tutorial sites, and that players can demonstrate learning in a transfer task to a language agnostic, pseudo-code exam. My work has also shown that programming can be appealing to a wide audience when packaged appropriately, and that people from all over the world are interested in learning how to code using an educational game.

There appears to be a countless number of educational tools, including games, coming online every day. However, without proper research, data collection, analyses, and sharing of information with other developers and researchers, we run the risk of creating and re-creating tools that ultimately do not benefit intended audience. It is my hope that my research demonstrates the benefit of designing technology from a human-centric process using controlled experiments to gather data from real users and iteratively improving the technology for their use.

I also hope that the work and ideas presented in this dissertation is one small step in the right direction to improve computing education for the masses, and that it shows that educational games are a viable way to do so.

BIBLIOGRAPHY

1. Aleven, V., & Koedinger, K.R. (2002). An effective metacognitive strategy: Learning by doing and explaining with a computer-based cognitive tutor. *Cognitive Science*, 26(2), 147-179.
2. Aleven, V., Myers, E., Easterday, M., & Ogan, A. (2010). Toward a framework for the analysis and design of educational games. *IEEE DIGITEL*, 69-76.
3. Amazon Mechanical Turk. http://www.mturk.com
4. Andersen, E., Liu, Y. E., Snider, R., Szeto, R., Cooper, S., & Popović, Z. (2011). On the harmfulness of secondary game objectives. *ACM FDG*, 30-37.
5. Answerdash. http://www.answerdash.com. Accessed: 2015-03-26.
6. Armoni, M., Meerbaum-Salant, O., & Ben-Ari, M. (2015). From scratch to "real" programming. *ACM TOCE*, 14(4), 25.
7. Armstrong, S., Brown, S., & Thompson, G. (1998.) *Motivating Students*. Routledge.
8. Atlas, G.D., Taggart, T., & Goodell D.J. (2004). The effects of sensitivity to criticism on motivation and performance in music students. *British Journal of Music Education*, 21(1), 81-87.
9. Barnes, D.J. (2002). Teaching introductory Java through LEGO MINDSTORMS models. *ACM SIGCSE Bulletin 34,* 147-151.
10. Barnes, T., Richter, H., Powell, E., Chaffin, A., & Godwin, A. (2007). Game2Learn: building CS1 learning games for retention. *ACM SIGCSE Bulletin*, 121–125.
11. Batt, S. (2009). Human attitudes towards animals in relation to species similarity to humans. *Bioscience Horizons*, 2(2), 180–190.
12. Beckwith, L., Burnett, M., & Cook, C., (2002). Reasoning about Many-to-Many Requirement Relationships in Spreadsheet Grids. *IEEE VL/HCC*, 149-157.
13. Begel, A. (1996). LogoBlocks: A Graphical Programming Language for Interacting with the World. *Electrical Engineering and Computer Science Department*, MIT, Boston, MA.
14. Beres, D. (2014). Obama Writes His First Line Of Code. Retrieved 2015-02-08, from http://www.huffingtonpost.com/ 2014/12/09/obama-code_n_6294036.html
15. Bjork, R.A. (1999). Assessing our own competence: heuristics and illusions. In Gopher, D., & Koriat, A., *Attention and performance XVII: Cognitive regulation of performance*, 435-459. Cambridge, MA: MIT Press.
16. Black, P., & Wiliam, D. (1998). Assessment and classroom learning. *Assessment in education*, 5(1), 7-74.

17. Blackwell, A.F. (2002). First steps in programming: A rationale for attention investment models. *IEEE HCC*, 2-10.

18. Bonar, J., & Soloway, E. (1985). Preprogramming knowledge: A major source of misconceptions in novice programmers. *Human–Computer Interaction*, 1(2), 133-161.

19. Bonate, P.L. (2000). *Analysis of pretest-posttest designs*. CRC Press.

20. Boustedt, J., Eckerdal, A., McCartney, R., Sanders, K., Thomas, L., & Zander C. (2011). Students' perceptions of the differences between formal and informal learning. *ACM ICER*, 61–68.

21. Bowman, R.F. (1982). A Pac-Man theory of motivation: tactical implications for classroom instruction. *Educational Technology*, 22(9), 14-16.

22. Boyce, A., & Barnes, T. (2010). BeadLoom Game: Using Game Elements to Increase Motivation and Learning. *ACM FDG*, 25-31.

23. Bradshaw, J.W.S., Paul, E.S. (2010). Could empathy for animals have been an adaptation in the evolution of Homo sapiens? *Animal Welfare*, 19(1), 107-112.

24. Brandt, J., Guo, P.J., Lewenstein, J., Dontcheva, M., & Klemmer, S.R. (2009). Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. *ACM CHI*, 1589-1598.

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

26. Braught, G., Eby, L.M., & Wahls, T. (2008). The effects of pair-programming on individual programming skill. *ACM SIGCSE*, 200-204.

27. Breslow, L., Pritchard, D.E., DeBoer, J., Stump, G.S., Ho, A. D., & Seaton, D.T. (2013). Studying learning in the worldwide classroom: Research into edX's first MOOC. *Research & Practice in Assessment*, 8(1), 13-25.

28. Bruckman, A. (1997). MOOSE Crossing: Construction, Community, and Learning in a Networked Virtual World for Kids. *MIT Media Lab*. Boston, MA.

29. Bruckman, A., Biggers, M., Ericson, B., McKlin, T., Dimond, J., DiSalvo, B., Hewner, M., Ni, L., & Yardi, S. (2009). Georgia Computes!: Improving the Computing Education Pipeline. *ACM SIGCSE*, 86-89.

30. Bureau of Labor Statistics (2012). 2010-2012 Report on Labor Across Industries. http://www.bls.gov, Accessed: 2015-01-12.

31. Burnett, M., Beckwith, L., Wiedenbeck, S., Fleming, S.D., Cao, J., Park, T.H., Grigoreanu, V., Rector, K. (2011). Gender pluralism in problem solving software. *Interacting with Computers*, 23, 450-460.

32. Burnett, M., Churchill, E., Lee, M.J. (2015). SIG: Gender-Inclusive Software: What We Know About Building It. *ACM CHI*, 857-860.

33. Butler, A.C., & Roediger, H.L. (2008). Feedback enhances the positive effects and reduces the negative effects of multiplechoice testing. *Memory & Cognition*, 36(3), 604-616.

34. Campbell, J., & Mayer, R.E. (2009). Questioning as an instructional method: Does it affect learning from lectures. *Applied Cognitive Psychology*, 23, 747-759.

35. Cao, J., Kwan, I., White, R., Fleming, S.D., Burnett, M., & Scaffidi, C. (2012). From barriers to learning in the idea garden: An empirical study. *IEEE VL/HCC*, 59-66.

36. Cao, J., Kwan, I., Bahmani, F., Burnett, M., Fleming, S.D., Jordahl, J., Horvath, A., & Yang, S. (2013). End-user programmers in trouble: Can the Idea Garden help them to help themselves? IEEE VL/HCC, 151-158.

37. Carpenter, S.K., Pashler, H., & Vul, E. (2006). What types of learning are enhanced by a cued recall test? *Psychonomic Bulletin & Review*, 13, 826-830.

38. Carter, L. (2006). Why students with an apparent aptitude for computer science don't choose to major in computer science. *ACM SIGCSE Bulletin*, 27–31.

39. Chaffin, A., & Barnes. T (2010). Lessons from a course on serious games research and prototyping. *ACM FDG*, 32-39.

40. Charters, P., Lee, M.J., Ko, A.J., & Loksa, D. (2013). Challenging Stereotypes and Changing Attitudes: The Effect of a Brief Programming Encounter on Adults' Attitudes Toward Programming. *ACM SIGCSE*, 653-658.

41. Chavez, C.M., McGaugh, J.L., Weinberger, N.M. (2009). The basolateral amygdala modulates specific sensory memory representations in the cerebral cortex. *Neurobiology of Learn Memory*, 91, 382–392.

42. Chi, M.T., De Leeuw, N., Chiu, M.H., & LaVancher, C. (1994). Eliciting self-explanations improves understanding. *Cognitive Science*, 18(3), 439-477.

43. Chumley-Jones, H.S., Dobbie, A., & Alford, C.L. (2002). Web-based learning: Sound educational method or hype? A review of the evaluation literature. *Academic medicine*, 77(10), S86-S93.

44. Cliburn, D.C. (2006). The effectiveness of games as assignments in an introductory programming course. *Frontiers in Education Conference*, 6–10.

45. Code.org. http://www.code.org. Accessed: 2013-02-02.

46. Codecademy. http://www.codecademy.com. Accessed: 2015-03-26.

47. Code Combat. http://www.codecombat.com. Accessed: 2015-03-26.

48. Code Hunt. http://www.codehunt.com. Accessed: 2015-03-26.

49. Code School. http://www.codeschool.com. Accessed: 2015-03-26.

50. Connolly, T.M., & Stansfield, M.H. (2006). *Enhancing eLearning: Using Computer Games to Teach Requirements Collection and Analysis*. WG HCI & UE of the Austrian Computer Society.

51. Cooper, S., Dann, W., & Pausch, R. (2000). Alice: a 3-D tool for introductory programming concepts. *Journal of Computing Sciences in Colleges,* 15(5), 107-116.

52. Cordova, D.I., & Lepper, M.R. (1996). Intrinsic motivation and the process of learning: Beneficial effects of contextualization, personalization, and choice. *Journal of Educational Psychology* 88, 4, 715.

53. Corno, L., Mandinach, E.B. (2004). What we have learned about student engagement in the past twenty years. *Big Theories*, 297–326.

54. Cross, J. (2006). *Informal learning: rediscovering the natural pathways that inspire innovation and performance*. San Francisco, CA: Pfeiffer.

55. Csikszentmihalyi, M. (1990). *Flow: The psychology of optical experience*. New York, NY: Harper Perrennial.

56. Dahotre, A., Zhang, Y., Scaffidi, C. (2010). A qualitative study of animation programming in the wild. *ACM-IEEE ESEM*, 1-10.

57. Daniel, J. (2012). Making sense of MOOCs: Musings in a maze of myth, paradox and possibility. *Journal of Interactive Media in Education*, 3.

58. Dann, W.P., Cooper, S., & Pausch, R. (2011). *Learning to Program with Alice*. Prentice Hall Press.

59. Deitel, H., & Deitel, P. (2005). *C++: How to program (5th ed.).* Upper Saddle River, NJ: Prentice Hall.

60. Dimitrov, D.M., & Rumrill, Jr, P.D. (2003). Pretest-posttest designs and measurement of change. *Work: A Journal of Prevention, Assessment and Rehabilitation*, 20(2), 159-165.

61. Downs, JS., Holbrook, MB, Sheng, S., & Cranor, L.F. (2010). Are your participants gaming the system?: screening mechanical turk workers. *ACM CHI*, 2399-2402.

62. Eagle, M., & Barnes, T. (2008). Wu's castle: teaching arrays and loops in a game. *ACM SIGCSE Bulletin,* 40(3), 245-249.

63. Eagle, M., & Barnes, T. (2009). Experimental evaluation of an educational game for improved learning in introductory computing. *ACM SIGCSE Bulletin*, 41(1), 321-325.

64. edX. https://www.edx.org. Accessed: 2015-03-26.

65. Ellis, A. (2005). Research On Educational Innovations. *Eye On Education, Inc.*, Larchmont, NY.

66. ESA (2011). Essential facts about the computer and video game industry. Entertainment Software Association. Web. 21 Feb. 2012. <http://www.theesa.com/facts/pdfs/ESA_EF_2011.pdf>

67. Esper, S., Foster, S. R., Griswold, W. G., Herrera, C., & Snyder, W. (2014). CodeSpells: bridging educational language features with industry-standard languages. *ACM Koli Calling International Conference on Computing Education Research*, 5-14.

68. Farthing, D.W. (1997). The three Ms of open learning: Medium, Material and Motivation. *Conference on the Teaching of Computing*.

69. Feldgen, M., & Clua, O. (2004). Games as a motivation for freshman students learn programming. *Frontiers in Education,* S1H–11.

70. Felleisen, M., Findler, R. B., Flatt, M., & Krishnamurthi, S. (2001). *How to design programs: An introduction to programming and computing.* Cambridge, MA: MIT Press.

71. Fenton, J. and Beck, K. (1989). Playground: An Object Oriented Simulation System with Agent Rules for Children of All Ages. *ACM OOPSLA*, 123-137.

72. Fitzgerald, S., Lewandowski, G., McCauley, R., Murphy, L., Simon, B., Thomas, L. and Zander, C., (2008) Debugging: finding, fixing, and flailing, a multi-institutional study of novice debuggers. *Computer Science Education.* v18. 93-116.

73. Fogg, B. J., & Nass, C. (1997). How users reciprocate to computers: an experiment that demonstrates behavior change. *ACM CHI*, 331-332.

74. Franklin, D., Conrad, P., Boe, B., et al. (2013). Assessment of computer science learning in a scratch-based outreach program. *ACM SIGCSE*, 371-376.

75. Frayling, C. (1993). *Research in art and design*. Royal College of Art London.

76. Freeman, P., Aspray, W. (1999). The supply of information technology workers in the United States. *Computing Research Association*.

77. Garris, R., Ahleer, R., & Driskell, J.E. (2002). Games, motivation, and learning: A research and practice model. *Simulation & Gaming* 33, 4, 441–467.

78. Gee, J.P. (2003). What video games have to teach us about learning and literacy. *Computers in Entertainment* 1, 20–20.

79. Gee, J.P. (2014). *What video games have to teach us about learning and literacy*. Macmillan.

80. Gentile, D.A. (2009). Video Games Affect the Brain—for Better and Worse. *The DANA Foundation, Cerebrum*. July 23, 2009.

81. Gidget. http://www.helpgidget.org. Accessed: 2014-09-12.

82. Goode, J., Estrella, R., & Margolis, J. (2006). Lost in translation: Gender and high school computer science, In *Women and Information Technology: Research on Underrepresentation*, MIT Press, 89-114.

83. Griffiths, M.D. (1997). Video games: the good news. *Education and Health*, 15:10–12.

84. Gross, P. and Kelleher, C., (2010). Non-programmers identifying functionality in unfamiliar code: strategies and barriers. *Journal of Visual Languages & Computing,* 21, 5, 263-276.

85. Guzdial, M. (2003). A media computation course for non-majors. *ACM SIGCSE Bulletin*, 104–108.

86. Guzdial, M. (2014, 10/15). Teaching Computer Science Better to get Better Results [Web log post]. Retrieved from http://computinged.wordpress.com/2014/10/15/we-need-to-fix-the-computer-science-teaching-problem/

87. Hamari, J., Koivisto, J., & Sarsa, H. (2014). Does gamification work? – a literature review of empirical studies on gamification. *HICSS*, 3025-3034.

88. Harel, I. (1991). *Children Designers*. Ablex Publishing, N.J.

89. Hays, R.T. (2005). The effectiveness of instructional games: A literature review and discussion (Technical Report 2005-004). *Naval air warfare center training systems division*. Orlando, FL.

90. Hsieh, G., Kraut, RE, & Hudson, SE. (2010). Why pay?: exploring how financial incentives are used for question & answer. *ACM CHI*, 305-314.

91. Huotari, K., & Hamari, J. (2012). Defining Gamification – A Service Marketing Perspective. *ACM International Academic MindTrek Conference*, 17-22.

92. Itin, C. M. (1999). Reasserting the Philosophy of Experiential Education as a Vehicle for Change in the 21st Century. *The Journal of Physical Education,* 22(2), 91-98.

93. Ito, M., Baumer, S., Bittanti, M., boyd, d., Cody, R., Herr B., Horst, H.A., Lange, P.G., Mahendran, D., Martinez, K., Pascoe, C.J., Perkel, D., Robinson, L., Sims, C., and Tripp, L. (2009). *Hanging Out, Messing Around, Geeking Out: Living and Learning with New Media*. Cambridge: MIT Press.

94. Ivala, E., Gachago, D., Condy, J., & Chigona, A. (2013). Enhancing student engagement with their studies: a digital storytelling approach. *Creative Education*, 4(10), 82.

95. Jeffries, R. (1982). A comparison of the debugging behavior of expert and novice programmers. *AERA Annual Meeting*.

96. Jernigan, W., Horvath, A., Lee, M.J., Burnett, M., Cuilty, T., Kuttal, S.K., Peters, A., Kwan, I., Bahmani, F., and Ko, A.J. (2015). It's the Principle(s) of the Thing! A Principled Evaluation for a Principled Idea Garden. *Unpublished manuscript*.

97. Johnson, C.I., & Mayer, R.E. (2009). A testing effect with multimedia learning. *Journal of Educational Psychology*, 101(3), 621-629.

98. Kapp, K.M. (2012). *The gamification of learning and instruction: game-based methods and strategies for training and education*. San Francisco, CA: Pfeiffer.

99. Karpicke, J.D., & Roediger, H.L. (2007). Expanding retrieval promotes short-term retention, but equally spaced retrieval enhances long-term retention. *Journal of Experimental Psychology: Learning, Memory, and Cognition*, 33, 704-719.

100. Karpicke, J.D., & Roediger, H.L. (2007). Repeated retrieval during learning is the key to long-term retention. *Journal of Memory and Language*, 57, 151-162.

101. Kay, A. (1997). Etoys and Simstories. http://www.squeakland.org

102. Kearsley, G., & Shneiderman, B. (1998). Engagement theory: A framework for technology-based teaching and learning. *Educational technology*, 38(5), 20-23.

103. Kehoe, J. (1995). Writing multiple-choice test items. Practical Assessment, Research & Evaluation, 4(9), retrieved April 2013 from http://pareonline.net/getvn.asp?v=4&n=9

104. Kelleher, C. and Pausch, R. (2005). Lowering the barriers to programming: A taxonomy of programming environments and languages for novice programmers. *ACM CSUR*, 37(2), 83-13.

105. Kelleher, C., & Pausch, R. (2007). Using storytelling to motivate programming. *Communications of the ACM*, 50(7), 58-64.

106. Kelleher, C., Pausch, R., & Kiesler, S. (2007). Storytelling Alice Motivates Middle School Girls to Learn Computer Programming. *ACM CHI*, 1455-1464.

107. Keller, J.M. & Suzuki, K. (1988). *Use of the ARCS Motivation Model in courseware design. In: Instructional designs for microcomputer courseware*. Lawrence Erlbaum, Hillsdale, NJ.

108. Kerr, J., Kelleher, C., Ellis, R. & Chou, M (2013). Setting the scene: scaffolding stories to benefit middle school students learning to program. *IEEE VL/HCC*, 95-98.

109. Khan Academy. http://www.kahnacademy.com. Accessed: 2015-03-26.`

110. Kinnunen, P. & Simon, B. (2010). Experiencing Programming Assignments in CS1: The Emotional Toll. ACM *ICER*, 77-86.

111. Kittur, A., Chi, E.H., & Suh, BW. (2008). Crowdsourcing user studies with Mechanical Turk. *ACM CHI*, 453-456.

112. Kirriemuir, J., & McFarlane, A. (2004). Literature Review in Games and Learning. *Report 8*, *NESTA*, Futurelab, Bristol.

113. Klein, J., Moon, Y., Picard, R.W. (1999). This computer responds to user frustration. *ACM CHI*, 242–243.

114. Ko, A. J., Myers, B.A., & Aung, H.H. (2004). Six learning barriers in end-user programming systems. *IEEE VL/HCC*, 199-206.

115. Ko, A. J. (2009). Attitudes and self-efficacy in young adults' computing autobiographies. *IEEE VL/HCC*, 67-74.

116. Kulesza, A. (2009). *Approximate learning for structured prediction problems*. UPenn WPE-II Report.

117. Layman, L., Williams, L., Slaten, K. (2007). Note to self: make assignments meaningful, *ACM SIGCSE*, 459-463.

118. Lee, M.J., Bahmani, F., Kwan, I., Laferte, J., Charters, P., Horvath, A., Luor, F., Cao, J., Law, C., Beswetherick, M., Long, S., Burnett, M., & Ko, A.J. (2014). Principles of a Debugging-First Puzzle Game for Computing Education. *IEEE VL/HCC*, 57-64.

119. Lee, M.J. & Ko, A.J. (2011). Personifying Programming Tool Feedback Improves Novice Programmers' Learning. *ACM ICER*, 109-116.

120. Lee, M.J., & Ko, A.J. (2012). Investigating the Role of Purposeful Goals on Novices' Engagement in a Programming Game. *IEEE VL/HCC,* 163-166.

121. Lee, M.J., & Ko, A.J. (2015). Comparing the Effectiveness of Online Learning Approaches on CS1 Learning Outcomes. *ACM ICER*, to appear.

122. Lee, M.J., Ko, A.J., & Kwan, I. (2013). In-Game Assessments Increase Novice Programmers' Engagement and Level Completion Speed. *ACM ICER,* 153-160.

123. Lewis, J., & Loftus, W. (2005). *Java software solutions (Java 5.0 version): Foundations of program design (4th ed.)*. Boston, MA: Addison Wesley.

124. Light-Bot. http://armorgames.com/play/2205/light-bot. Accessed: 2015-03-26.

125. Linderbaum, B. (2006) The Development and Validation of the Feedback Orientation Scale. *Journal of Management*, 1372-1405.

126. Lionet, F., & Lamoureux, Y., *Klik and Play*, Maxis, 1994.

127. Lister, R., Adams, E.S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, et al. (2004). A multi-national study of reading and tracing skills in novice programmers. *ACM SIGCSE Bulletin*, 36(4), 119-150.

128. Logo Computer Systems, Inc., *My Make Believe Castle*, 1995.

129. Malone, T.W. (1981). *What Makes Things Fun to Learn? A Study of Intrinsically Motivating Computer Games*. Pipeline.

130. Maloney, J., Resnick, M., Rusk, N., Silverman, B., & Eastmond, E. (2010). The Scratch Programming Language and Environment. *ACM TOCE*.

131. Maloney, J.H., Peppler, K., Kafai, Y., Resnick, M., & Rusk, N. (2008). Programming by choice: Urban youth learning programming with scratch. *ACM SIGCSE Bulletin*, 40(1), 367-371.

132. Margolis, J. & Fisher, A. (2001). *Unlocking the Clubhouse: Women in Computing.* The MIT Press.

133. McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y.B.D., Laxer, C., Thomas, L., Utting, I. & Wilusz, T. (2001). A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. *ACM SIGCSE Bulletin*, 33(4), 125-180.

134. McDaniel, M.A., Anderson, J.L., Derbish, M.H., & Morrisette, N. (2007). Testing the testing effect in the classroom. *European Journal of Cognitive Psychology*, 19(4-5), 494-513.

135. McDowell, C., Werner, L., Bullock, H., & Fernald, J. (2002). The effects of pair-programming on performance in an introductory programming course. *ACM SIGCSE*, 38–42.

136. McNamara, D., Jackson, G., Graesser, A. (2009) Intelligent tutoring and games. *Artificial Intelligence in Education*, 1–10.

137. Mechanical Turk. http://www.mturk.com. Accessed: 2010-07-26.

138. Meyers-Levy, J. (1989). Gender differences in information processing: A selectivity interpretation, In *Cognitive and Affective Responses to Advertising*, Lexington Books, 219-260.

139. MindStorms. http://www.mindstorms.lego.com

140. Minecraft. http://www.minecraft.net

141. Misa, T. (2010). *Gender codes: Defining the problem, in Gender Codes: Why Women are Leaving Computing*, Wiley, 3-24.

142. Mislevy, R.J., Behrens, J.T., Dicerbo, K.E., Frezzo, D.C., & West, P. (2012). Three things game designers need to know about assessment. *Assessment in game-based learning*, 59-81.

143. Murphy, L., Fitzgerald, S., Hanks, B., & McCauley, R. (2010) Pair debugging: a transactive discourse analysis. *ACM ICER*, 51-58.

144. Murphy, L. and Thomas, L. (2008). Dangers of a fixed mindset: Implications of self-theories research for computer science education. *ITiCSE*, 271-275.

145. Moghaddam, R.S., Ko, A.J., Loksa, D., & Lee, M.J. (2015). Effects of Social Information on Engagement and Attitudes in Online Learning Environments. *Unpublished manuscript*.

146. Monroy-Hernández, A., & Resnick, M. (2008). Empowering kids to create and share programmable media. *Interactions*, 15(2), 50-53.

147. Nass, C. (2000). Machines and Mindlessness: Social Responses to Computers. *Journal of Social Issues*, 56, 81-103.

148. Nass, C., Fogg, B.J., & Moon, Y. (1996). Can computers be teammates? *International Journal of Human-Computer Studies*, 45, 669-678.

149. NCWIT (2010). NCWIT Scorecard: A report on the status of women in information technology. Nat'l Ctr. for Women & IT. Web. 30 Mar. 2013. <http://www.ncwit.org/pdf/Scorecard2010.pdf>

150. NetMarket Analytics. Retrieved September 12, 2011, from http://www.netmarketshare.com

151. newzoo.com (2011). "High-level Game Facts from the US National Gamers Survey."

152. NPD (2011). *Kids & gaming 2011 report*. NPD Group.

153. Oblinger, D., Oblinger, J. (2005). *Educating the net generation*. Educause.

154. Pane, J., & Myers, B. (2006). More natural programming languages and environments, In *End User Development*, Springer, 31-50.

155. Pane, J. Myers, B.A., & Miller, L.B. (2002). Using HCI Techniques to Design a More Usable Programming System. *IEEE VL/HCC*, 198-206.

156. Papastergiou, M. (2009). Digital Game-Based Learning in high school Computer Science education: Impact on educational effectiveness and student motivation. *Computers & Education* 52, 1, 1–12.

157. Papert, S. (1980). *Mindstorms: Children, Computers, and Powerful Ideas*. Basic Books New York, NY.

158. Papert, S., & Harel, I. (1991). Situating constructionism. *Constructionism*, 36, 1-11.

159. Pear, J.J. (2004). Enhanced feedback using computer-aided personalized system of instruction. In W. Buskist, V. W. Hevern, B.K. Saville, & T. Zinn, (Eds.), *Essays from excellence in teaching* (Chapter 11).

160. Poehner, M.E. (2007). Beyond the test: L2 dynamic assessment and the transcendence of mediated learning. *The Modern Language Journal*, 91(3), 323-340.

161. Polya, G. (1971). *How to Solve It: A New Aspect of Mathematical Method*, Princeton Univ. Press.

162. Prensky, M. (2003). Digital game-based learning. *Computers in Entertainment,* 1(1), 21-21.

163. Prensky, M. (2005). Computer games and learning: Digital game-based learning. *Handbook of computer game studies*, 18, 97-122.

164. Prensky, M. (2006). *Don't Bother Me, Mom, I'm Learning!: How Computer and Video Games Are Preparing Your Kids for 21st Century Success and How You Can Help*. Saint Paul, Paragon House.

165. Ram, A., & Leake, D.B. (1995). *Goal-Driven Learning*. MIT Press, Boston, MA.

166. Randel, J.M., Morris, B.A., Wetzel, C.D., & Whitehill, B.V. (1992). The effectiveness of games for educational purposes: A review of recent research. *Simulation & Gaming*, 23(3), 261-276.

167. Reinecke, L., Trepte, S., & Behr, K.M. (2008). *Why Girls Play. Results of a Qualitative Interview Study with Female Video Game Players*. Universitäts-und Landesbibliothek.

168. Resnick, M., Martin, F., Sargent, R., & Silverman, B. (1996). Programmable Bricks: Toys to Think With. *IBM Systems Journal*, 35(3-4), 443-452.

169. Ricci, K.E., Salas, E., & Cannon-Bowers, J.A. (1996). Do computer-based games facilitate knowledge acquisition and retention? *Military Psychology,* 8(4), 295–307.

170. Riggio, R. E. (2007). Reciprocal peer tutoring: Learning through dyadic teaching. In B. K. Saville, T. E. Zinn, S. A. Meyers, & J. R. Stowell (Eds.), *Essays from excellence in teaching*, (Chapter 10).

171. Rising, L. (1999). Patterns: A way to reuse expertise. *IEEE Communications*, 37(4), 34-36.

172. Roberts, T.A. (1991). Gender and the influence of evaluations on self-assessments in achievement settings. *Psychological Bulletin,* 109(2), 297-308.

173. Ross, J., Irani, I., Silberman, M. Six, Zaldivar, A., & Tomlinson, B. (2010). Who are the Crowdworkers?: Shifting Demographics in Amazon Mechanical Turk. *ACM CHI*, 2863-2872.

174. Rubio, M.A., Romero-Zaliz, R., Mañoso, C., & Angel, P. (2015). Closing the gender gap in an introductory programming course. *Computers & Education*, 82, 409-420.

175. Ruf, A., Mühling, A., & Hubwieser, P. (2014). Scratch vs. Karel: impact on learning outcomes and motivation. *ACM WiPCSE*, 50-59.

176. Ryokai, K., Lee, M.J., & Breitbart, J.M. (2009). Children's storytelling and programming with robotic characters. *ACM Creativity & Cognition*, 19-28.

177. Sadler, D. R. (1989). Formative assessment and the design of instructional systems. *Instructional Science*, 18(2), 119-144.

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

179. Scaffidi, C., Shaw, M., and Myers, B.A. (2005). Estimating the Numbers of End Users and End User Programmers. *IEEE VL/HCC*, 207-214.

180. Schifter, C., & Cipollone, M. (2013). Minecraft as a teaching tool: One case study. *In Society for Information Technology & Teacher Education International Conference*, (1), 2951-2955.

181. Schön, D.A. (1983). *The Reflective Practitioner: How Professionals Think in Action*. Basic Books, NY.

182. Scratch Online. http://scratch.mit.edu

183. Shute, V.J. (1993). A macroadaptive approach to tutoring. *Journal of AI in Education*, 4(1), 61-93.

184. Schute, V.J. (2011). Stealth Assessments in Computer-Based Games to Support Learning. *Computer games and instruction*, 55(2), 503-524.

185. Shute, V.J. (2011). Stealth assessment in computer-based games to support learning. *Computer games and instruction*, 55(2), 503-524.

186. Sheard, J., Carbone, A., Chinn, D., Laakso, M.J., Clear, T., et al. (2011). Exploring programming assessment instruments: a classification scheme for examination questions. *ACM ICER*, 33-38. ACM.

187. Short, D. (2012). Teaching scientific concepts using a virtual world—Minecraft. *Teaching Science-the Journal of the Australian Science Teachers Association*, 58(3), 55.

188. Shute, V.J., Ventura, M., Bauer, M., & Zapata-Rivera, D. (2009). Melding the power of serious games and embedded assessment to monitor and foster learning. *Serious games: Mechanisms and Effects*, 295-321.

189. Shackelford, R. L. (1997). *Introduction to computing and algorithms.* Boston, MA: Addison Wesley.

190. Sime, M., Green, T., & Guest, D. (1976). Scope marking in computer conditionals: A psychological evaluation. *International Journal of Man-Machine Studies*, 9, 107–118.

191. Smith, D., Cypher, A., & Tesler, L. (2002). Programming by example: novice programming comes of age. *Communications of the ACM*, 75-81.

192. Smith, T. (2007). Exams as learning experiences: One nutty idea after another. *Beyond Tests and Quizzes: Creative Assessments in the College Classroom*, 115, 71.

193. Soflano, M., Connolly, T.M., & Hainey, T. (2015). An Application of Adaptive Games-Based Learning based on Learning Style to Teach SQL. *Computers & Education*.

194. Soloway, E. (1986). Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29(9), 850-858.

195. Steffe, L.P., & Gale, J. E. (Eds.). (1995). *Constructivism in education*. Hillsdale, NJ: Lawrence Erlbaum, 159.

196. Subrahmaniyan, N., Kissinger, C., Rector, K., Inman, D., Kaplan, J., Beckwith, L., & Burnett, M. (2007). Explaining debugging strategies to end-user programmers. *IEEE VL/HCC*, 127-136.

197. Summers, N. (n.d.) Codecademy surpasses 24 million unique users for its free online coding courses. The Next Web. Retrieved 23 April 2014.

198. Sykes, E.R. (2007). Determining the effectiveness of the 3D Alice programming environment at the computer science I level. *Journal of Educational Computing Research*, 36(2), 223-244.

199. Sweller, J. (2006). The worked example effect and human cognition. *Learning and Instruction*, 16(2) 165–169.

200. Tanimoto, S., & Runyan, M. (1986). Play: an iconic programming system for children. *Visual Programming Environments*, 367-377.

201. Tarkan, S., Sazawal, V., Druin, A., Golub, E., Bonsignore, E.M., Walsh, G., & Atrash, Z. (2010). Toque: designing a cooking-based programming language for and with children. *ACM CHI*, 2417-2426.

202. Tew, A.E. (2010). *Assessing fundamental introductory computing concept knowledge in a language independent manner.* Georgia Institute of Technology PhD Dissertation.

203. Tew, A.E., & Guzdial, M. (2011). The FCS1: a language independent assessment of CS1 knowledge. *ACM SIGCSE*, 111-116.

204. Thompson, S.M. (2006). *An Exploratory Study of Novice Programming Experiences and Errors*. Thesis., University of Victoria, Victoria.

205. Tsukamoto, H., Nagumo, H., Takemura, Y., & Matsumoto, K. (2008). Analyzing the transition of learners' motivation to learn programming. *Frontiers in Education Conference*, S4B-6-S4B-11.

206. Umaschi, M. (1997). Soft toys with computer hearts: Building personal storytelling environments. *ACM CHI*, 20-21.

207. UK DFE (2013). *National Curriculum in England: Computing Programmes of Study*. (Dept. Education No. DFE-00171-2013). UK.

208. Vihavainen, A., Paksula, M., & Luukkainen, M. (2011). Extreme apprenticeship method in teaching programming for beginners. *ACM SIGCSE*, 93-98.

209. Vrugt, A.J., Langereis, M.P., & Hoogstraten, J. (1997). Academic self-efficacy and malleability of relevant capabilities as predictors of exam performance. *Journal of Experimental Education*, 66(1), 61-72.

210. Webb, H.C., & Rosson, M. B. (2011). Exploring careers while learning Alice 3D: a summer camp for middle school girls. *ACM SIGCSE*, 377-382.

211. Werner, L.L., Hanks, B., & McDowell, C. (2004). Pair-programming helps female computer science students. *ACM JERIC*, 4(1).

212. Whitfort, T. (n.d.). Pseudo code guide. http://ironbark.bendigo.latrobe.edu.au/subjects/PE/ 2005s1/ other_resources/pseudocode_ guide.html. Accessed: 2015-03-26.

213. Williamson, B. (2009). *Computer games, schools, and young people: A report for educators on using games for learning.* Bristol: Futurelab.

214. Wilson, B.C., & Shrock, S. (2001). Contributing to success in an introductory computer science course: a study of twelve factors. *ACM SIGCSE Bulletin* 33, 1, 184–188.

215. Yee, N. (2006). Motivations for play in online games. *Cyber Psychology & Behavior,* 9(6), 772-775.

216. Young, J. (2008). "Badges" earned online pose challenge to traditional college diplomas. *Chronicle of Higher Education.*

217. Zelle, J. M. (2004). *Python programming: An introduction to computer science.* Wilsonville, OR: Franklin Beedle.

218. Zimmerman, J., Forlizzi, J., & Evenson, S. (2007). Research through design as a method for interaction design research in HCI. *ACM CHI*, 493–502.

219. Zorn, C., Wingrave, C.A., Charbonneau, E., & LaViola Jr, J.J. (2013). Exploring Minecraft as a conduit for increasing interest in programming. *FDG*, 352-359.

APPENDIX

The appendix contains the following sections:

- Gidget Language Grammar
- Pseudo-code Tests
- Game Assets
- Gidget Game Screenshots
- Detailed Level Breakdown

- GIDGET LANGUAGE GRAMMAR

Gidget Language Grammar

 # _ indicates token

BLOCK	:: _newline _in STATEMENT* _out	
STATEMENT	:: GOTO | GO | GRAB | DROP | IF | WHILE | FOREACH | ASSIGNMENT | FUNCTION | CALL | OBJECT | CREATE | REMOVE | ADD | EXPR	
ERROR	:: _token* _eol	# For lines that don't parse. These have no effect, except for displaying an error message
OBJECT	:: object _id (_id*) [BLOCK]	
FUNCTION	:: function _id (_id*) BLOCK	
WHEN	:: when EXPR BLOCK	# Executed after each step of the world
# Control		
FOREACH	:: for _id in EXPR BLOCK	
WHILE	:: while EXPR BLOCK	
IF	:: if EXPR BLOCK else (IF | BLOCK)	
DONE	:: done	# Done looping
RETURN	:: return [EXPR]	# Return a value to the caller
SAY	:: say EXPR	# Say something to a character (i.e., print output to screen)
SOUND	:: sound EXPR	# Play a sound (expects a string)
ENSURE	:: ensure EXPR	# Used in the goals to determine whether a program passes
# Objects in the world		
QUERY	:: /_string/[s]	# e.g., 'rock'. Finds all objects matching the string in the world and pushes a list of them onto the value stack. We use the slash character (/) as the opening and closing delimiter based on novice programmers' participatory design feedback saying separate opening and closing characters (e.g., { and }, or [and]) is confusing, and that they would rather use the same character for both
CREATE	:: create _id LIST	# "Executes" the object's assignments and functions, creating an instance, giving it name and location
REMOVE	:: remove EXPR	# Removes the object referred to from the world
GOTO	:: goto EXPR	# Moves Gidget to the location of a specific object, optionally avoiding an object
GO	:: (up|down|left|right) [EXPR]	# Moves Gidget a specific direction by an optional amount
GRAB	:: grab EXPR	# Constrains an object's location to Gidget's location
DROP	:: drop EXPR	# Unconstrains an object's location from Gidget's location
# Modifications		
ASSIGNMENT	:: set REF to EXPR	
# Expressions		
EXPR	:: OR	
OR	:: AND [or OR]	
AND	:: EQUAL [and AND]	
EQUAL	:: INEQUALITY [= EQUAL]	
COMPARISON	:: ON [_inequality COMPARISON]	# e.g., 5 > 5, 5 = 5
ON	:: ADDITION [on ON]	# /rock/ on the /pit/, returns list of objects at the location of the object
ADDITION	:: MULT [_add ADD]	# length + width

Gidget Language Grammar

```
ADDITION    :: MULT [_add ADD]                  # length + width

MULT        :: PRIMARY [_multiply MULTIPLY]     # length * width

PRIMARY     :: # ON | (EXPR) | ITEM | NOT | QUERY | [-+] EXPR
               true | false | _number |
               _string | nothing | REF | LIS

LIST        :: [ EXPR? [, EXPR*] ]              # Comma separated list

ITEM        :: (first|rest|last) ON             # Retrieve the first or last item of the list

NOT         :: not EXPR                         # Negate a boolean value

REF         :: (_id | _id LIST) (: REF |        # rock, gidget:arm(), gidget:goto(1, 2, 3), battery:component():energy, battery[2]
               [ EXPR ])*
```

NOTES:

- The “world code,” “gidget code,” and “goals” all use the same language.
  - The world code sets up each level’s starting conditions. It is not visible to the player; the only exception to this is in the Gidget Puzzle Designer.
  - Gidget code is what players interact with in the game. This typically contains deliberate errors/bugs for players to fix. Gidget code controls the Gidget object.
  - Goal code consists of one or more ensure statements that must *all* evaluate to true to pass the level.
- All objects are instances with properties.
- Properties can point to functions and values.
- All objects have the following reserved properties (default values are indicated within parentheses:
  - energy (100), grabbed ([]), image (“default”), labeled (true), layer (1), name ({name of the object}), position ({position of the object that invoked creation})
- Gidget can't access certain pre-defined members (i.e. reserved properties) of other objects, but all other objects can.
  - This is to limit cheating (e.g., Gidget being able to change its own energy or being able to change objects’ locations)
- Types are dynamically assigned and can be objects, booleans, numbers, strings, and lists.
- Functions declared in a Gidget program are properties of the "gidget" object.
- All variables are global except for those referred to in functions. Those use function-level scoping.
- Create and remove can be invoked.
- All loops are explicit. None of the constructs operate on multiple values.
- Gidget can only remove objects that occupy the same space. Other objects do not have this constraint.
- Structure is indicated by indentation.
- Function call semantics require Gidget to be at the location of the object.

- PSEUDO-CODE TESTS

The following multiple-choice questions (written in pseudo-code) were used in the pre-tests and post-tests that were described in Chapter 7. Questions (and answer choices) are grouped topically here for readability, but were administered to participants in randomized order to minimize ordering effects. The solution to each question is highlighted in green.

Pseudo-code questions; SET 1 – Basics

BASICS #1 - variables	

`x = 8`
`y = 5`
`z = 10`
`x = 2`
	What is the final value of *x*?

	2
	5
	8
	10
	16

BASICS #2 - mathematical operators (+, -, *, /)	

`x = 3 * 2`
`y = 3 - 1`
`z = x + y + y`
	What is the final value of *z*?

	2
	6
	8
	10
	14

BASICS #3 - relational operators (==, >, >=, <, <=, !=)	

`x = 3`
`y = 5`
`z = 7`
	Which of the following expressions is *True*?

	x < y
	x > y
	x == y
	z < y
	x > z

Pseudo-code questions; SET 2 – Logical Operators (NOT, AND, OR)

LOGICAL OPERATORS #1 - NOT	

`x = True`
`y = False`
`z = NOT y`
	Which of the following expressions is *True*?

	x == NOT True
	x == y
	y == z
	y == NOT True
	z == NOT True

LOGICAL OPERATORS #2 - AND	

`a = True`
`b = False`
 	
`c = a AND b`
	What is the final value of *c*?

	c == a
	c == b
	a == b
	b == True
	a == False

LOGICAL OPERATORS #2 - OR	

`x = True`
`y = False`
 	
`z = x OR y`
	What is the final value of *z*?

	z == x
	z == y
	x == y
	y == True
	x == False

Pseudo-code questions; SET 3 – Selection Statements (IF, ELSE)

SELECTION STATEMENTS #1

```
number = 81

IF number >= 90 THEN
    element = 'fire'
ELSE IF number >= 80 THEN
    element = 'water'
ELSE IF number >= 70 THEN
    element = 'metal'
ELSE IF number >= 60 THEN
    element = 'earth'
ELSE
    element = 'wood'
ENDIF
```

What is the final value of *element*?

- fire
- **water**
- metal
- earth
- wood

SELECTION STATEMENTS #2

```
sales = 0
apples = 4

IF apples > 1 THEN
    sales = 2
ELSE
    sales = 3
ENDIF
```

What is the final value of *sales*?

- 0
- 1
- **2**
- 3
- 4

SELECTION STATEMENTS #3

```
x = 3
y = 5
z = 7

IF (x + x) > 5 THEN
    answer = 'apple'
ELSE IF (x + y) > 9 THEN
    answer = 'orange'
ELSE
    answer = 'banana'
ENDIF

IF (y + z) > 13 THEN
    answer = 'watermelon'
ELSE
    answer = 'strawberry'
ENDIF
```

What is the final value of *answer*?

- apple
- orange
- watermelon
- banana
- **strawberry**

Pseudo-code questions; SET 4 – Arrays

ARRAY #1	

`sports = ['basketball', 'baseball'], 'volleyball', 'hockey', 'football']`
 	
`playingToday = sports[2]`
	What is the final value of *playingToday*?

	basketball
	baseball
	volleyball
	hockey
	football

ARRAY #2	
`roster=['Angela','Amy','Alice']` `roster[1] = 'Anne'`	What is the final value of *roster*?
	['Angela','Anne','Alice']
	['Angela','Amy','Anne']
	['Anne','Amy','Alice']
	['Angela','Anne','Amy','Alice']
	['Anne','Angela','Amy','Alice']

ARRAY #3	

`list = [0,1,2,3,1]`
 	
`myNumber = list[1] + list[3]`
	What is the final value of *myNumber*?

	0
	1
	2
	3
	4

Pseudo-code questions; SET 5 – Indefinite Loops (WHILE)

INDEFINITE LOOP #1

```
jump = 1
count = 1

WHILE count < 4 DO
    jump = jump + 1
    count = count + 2
ENDWHILE
```

What is the final value of *jump*?
1
2
3
4
5

INDEFINITE LOOP #2

```
x = 3
y = 6
counter = 0

WHILE x < y DO
    counter = counter + 1
    x = x + 1
ENDWHILE
```

What is the final value of *counter*?
0
1
2
3
4

INDEFINITE LOOP #3

```
total = 0
x = 0
y = 8

WHILE y > 0 DO
    IF y > x THEN
        y = y - 2
        total = total + 1
    ENDIF
ENDWHILE
```

What is the final value of *total*?
0
1
2
3
4

Pseudo-code questions; SET 6 – Definite Loops (FOR)

DEFINITE LOOP #1

```
myCoins = 8

FOR counter = 0 to 2 BY 1 DO
    myCoins = myCoins - 2
ENDFOR
```

What is the final value of *myCoins*?
0
2
4
6
8

DEFINITE LOOP #2

```
x = 4

FOR y = 1 to 4 BY 2 DO
    IF y > 2 THEN
        x = x + 2
    ELSE
        x = x - 2
    ENDIF
ENDFOR
```

What is the final value of *x*?
0
2
4
6
8

DEFINITE LOOP #3

```
FOR x = 1 to 5 BY 1 DO
    xSquared = x * x
ENDFOR
```

What is the final value of *xSquared*?
1
4
9
16
25

Pseudo-code questions; SET 7 – Function Parameters

FUNCTION PARAMETERS #1

```
DEFINE getSize(num)
    IF num >= 70 AND num < 100 THEN
        size = 'large'
    ELSE IF num >= 30 AND num < 70 THEN
        size = 'medium'
    ELSE IF num >= 0 AND num < 30 THEN
        size = 'small'
    ELSE IF num >= 100 THEN
        size = 'x-large'
    ELSE
        size = 'x-small'
    ENDIF

    PRINT size
ENDDEFINE
```

What is the output of *getSize(100)*?

- x-small
- small
- medium
- large
- **x-large**

FUNCTION PARAMETERS #2

```
DEFINE doSomething(x, y)
    IF x > y THEN
        z = x
    ELSE IF x == y THEN
        z = x - y
    ELSE
        z = y
    ENDIF

    PRINT z
ENDDEFINE
```

What is the output of *doSomething(5,2)*?

- 2
- 3
- **5**
- 25
- 52

FUNCTION PARAMETERS #3

```
DEFINE getColor(a, b, c)
    IF (a + 1) > 1 THEN
        color = 'red'
    ELSE IF (a + b) > 1 THEN
        color = 'orange'
    ELSE IF (b + c) > 1 THEN
        color = 'yellow'
    ELSE IF (a + c) > 1 THEN
        color = 'green'
    ELSE
        color = 'blue'
    ENDIF

    PRINT color
ENDDEFINE
```

What is the output of *getColor(0,1,2)*?

- red
- orange
- **yellow**
- green
- blue

Pseudo-code questions; SET 8 – Function Returns

FUNCTION RETURNS #1

```
DEFINE func1()
    RETURN 'Mercury'
ENDDEFINE
DEFINE func2()
    RETURN 'Venus'
ENDDEFINE
DEFINE func3()
    RETURN 'Earth'
ENDDEFINE
DEFINE func4()
    RETURN 'Mars'
ENDDEFINE
DEFINE func5()
    RETURN 'Jupiter'
ENDDEFINE

planet = func1()
planet = func3()
planet = func5()
planet = func2()
planet = func4()
```

What is the final value of *planet*?

- Mercury
- Venus
- Earth
- **Mars**
- Jupiter

FUNCTION RETURNS #2

```
DEFINE number()
    x = 4
    y = 3
    z = 2
    RETURN x
ENDDEFINE

a = 1
b = number() + 0
c = a + b - number()
```

What is the final value of *c*?

- 0
- **1**
- 2
- 3
- 4

FUNCTION RETURN #3

```
DEFINE numbers()
    x = 2
    y = 1
    z = 0
    RETURN x + y - z
ENDDEFINE

x = 4
y = 3
z = numbers() + x + y
```

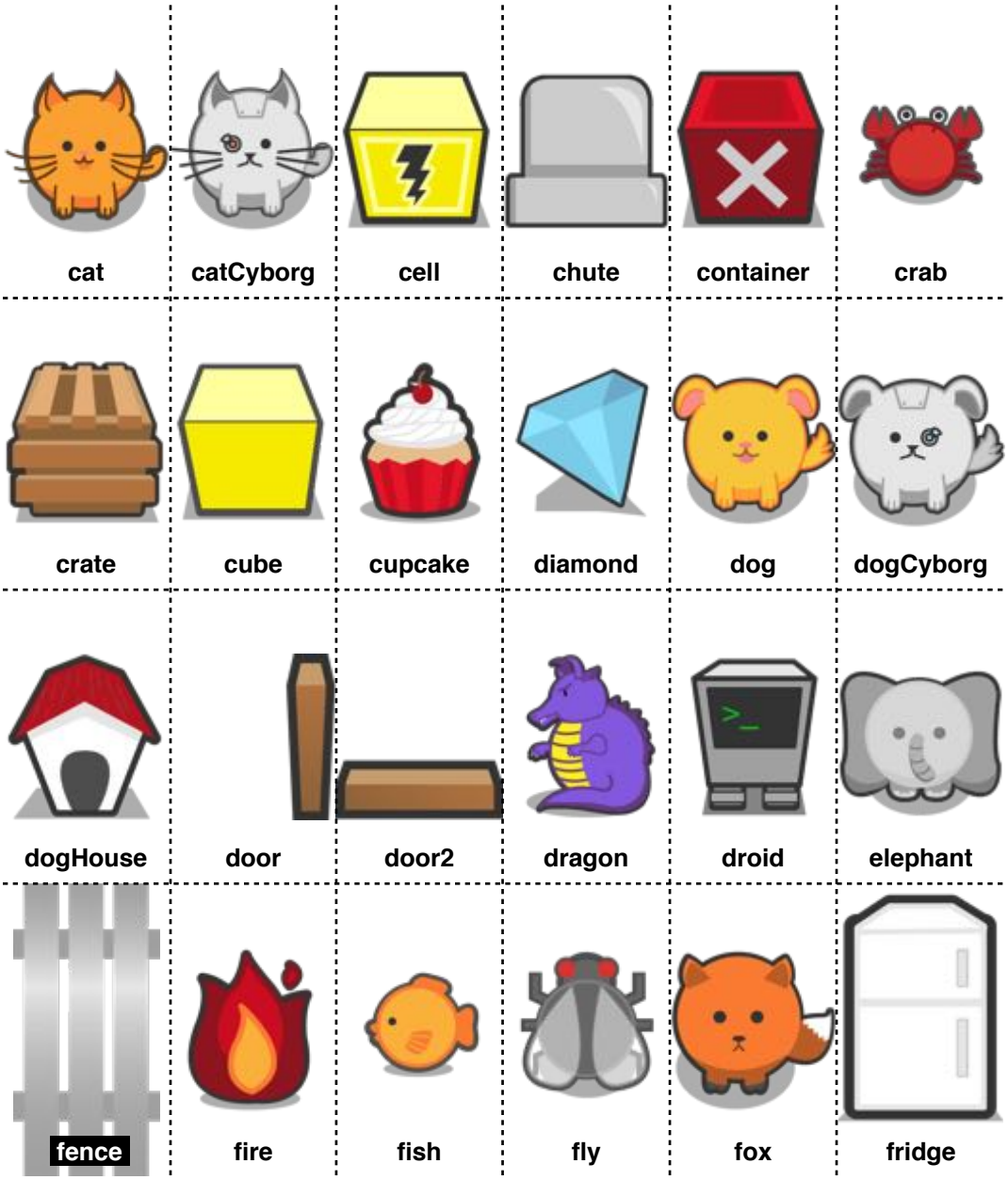
What is the final value of *z*?

- 0
- 6
- 7
- 10
- **14**

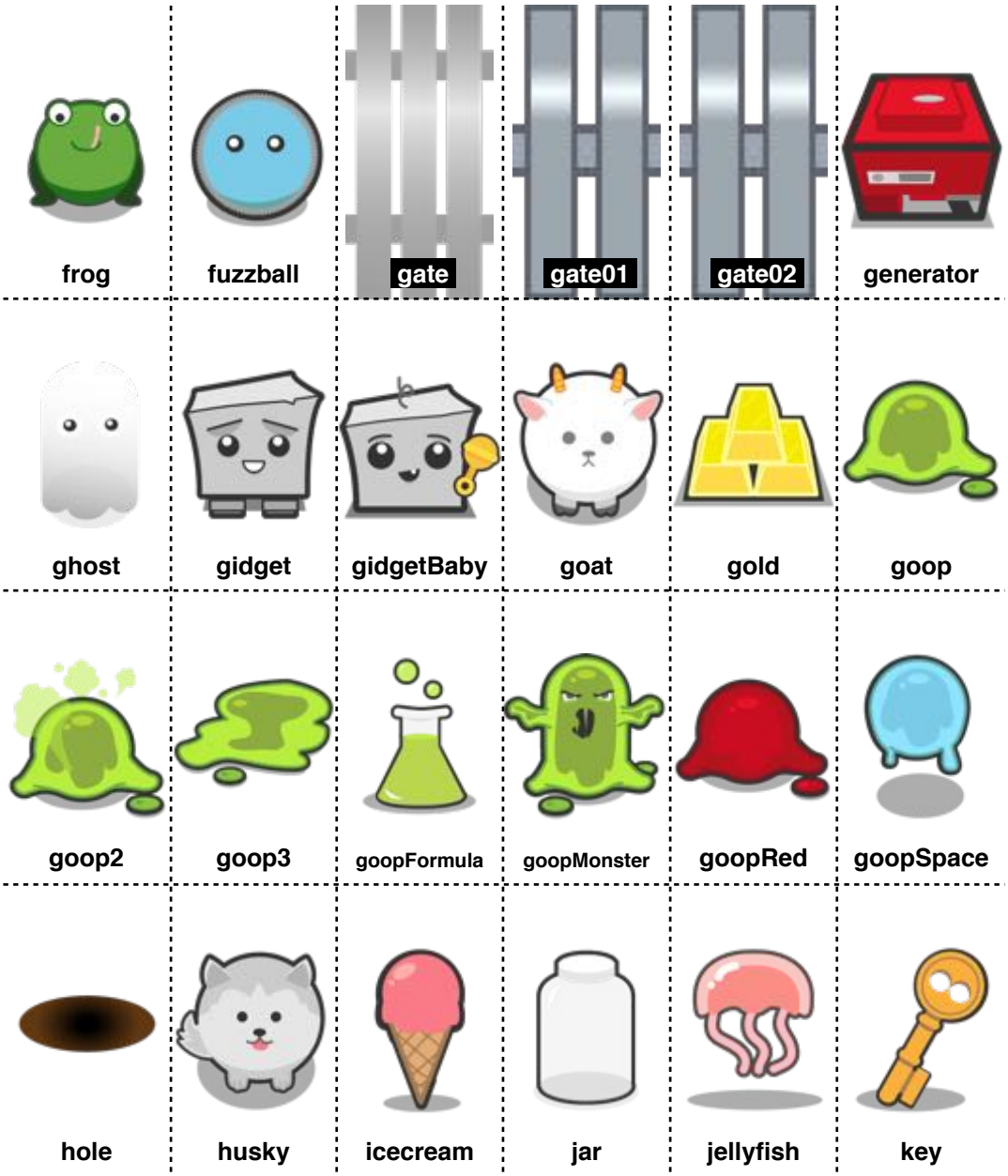
- GIDGET GAME ASSETS (IMAGES + SOUND EFFECTS)

Objects & Characters (Page 1 of 6)

Objects & Characters (Page 2 of 6)



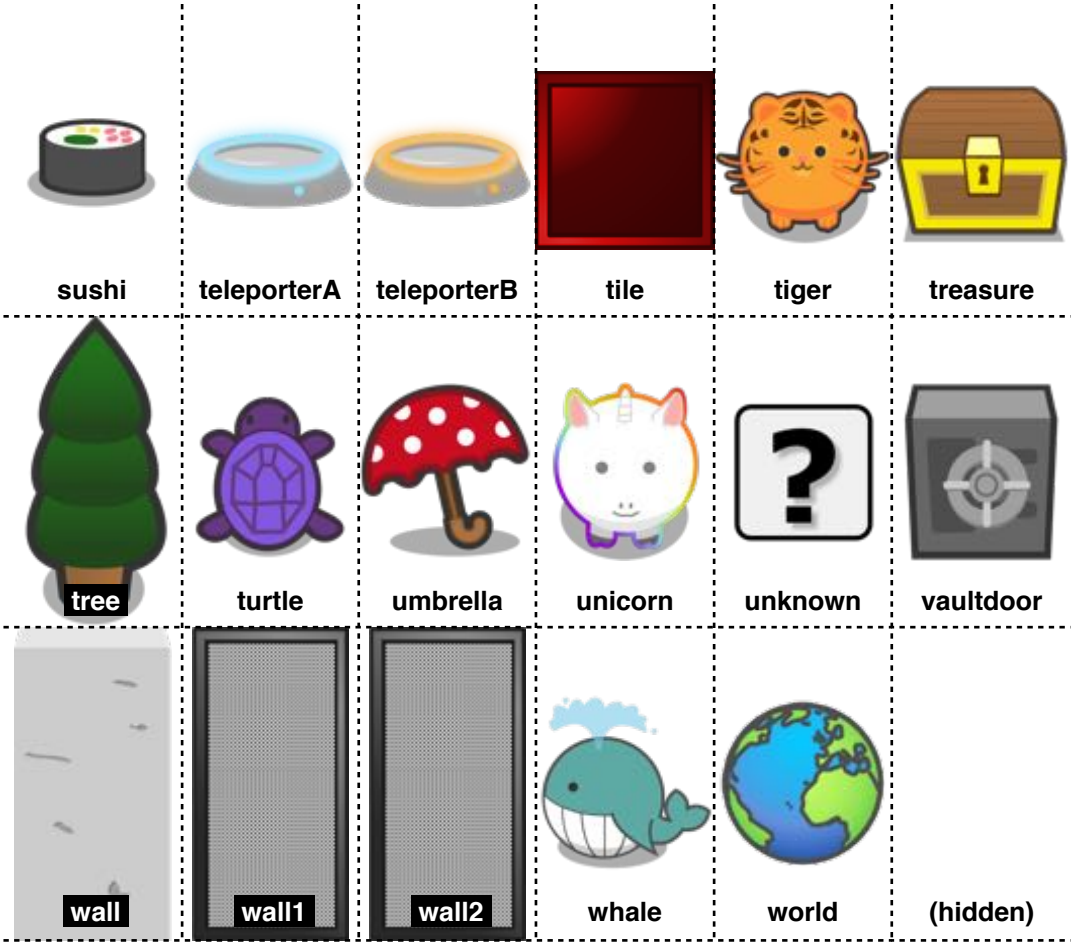
Objects & Characters (Page 3 of 6)



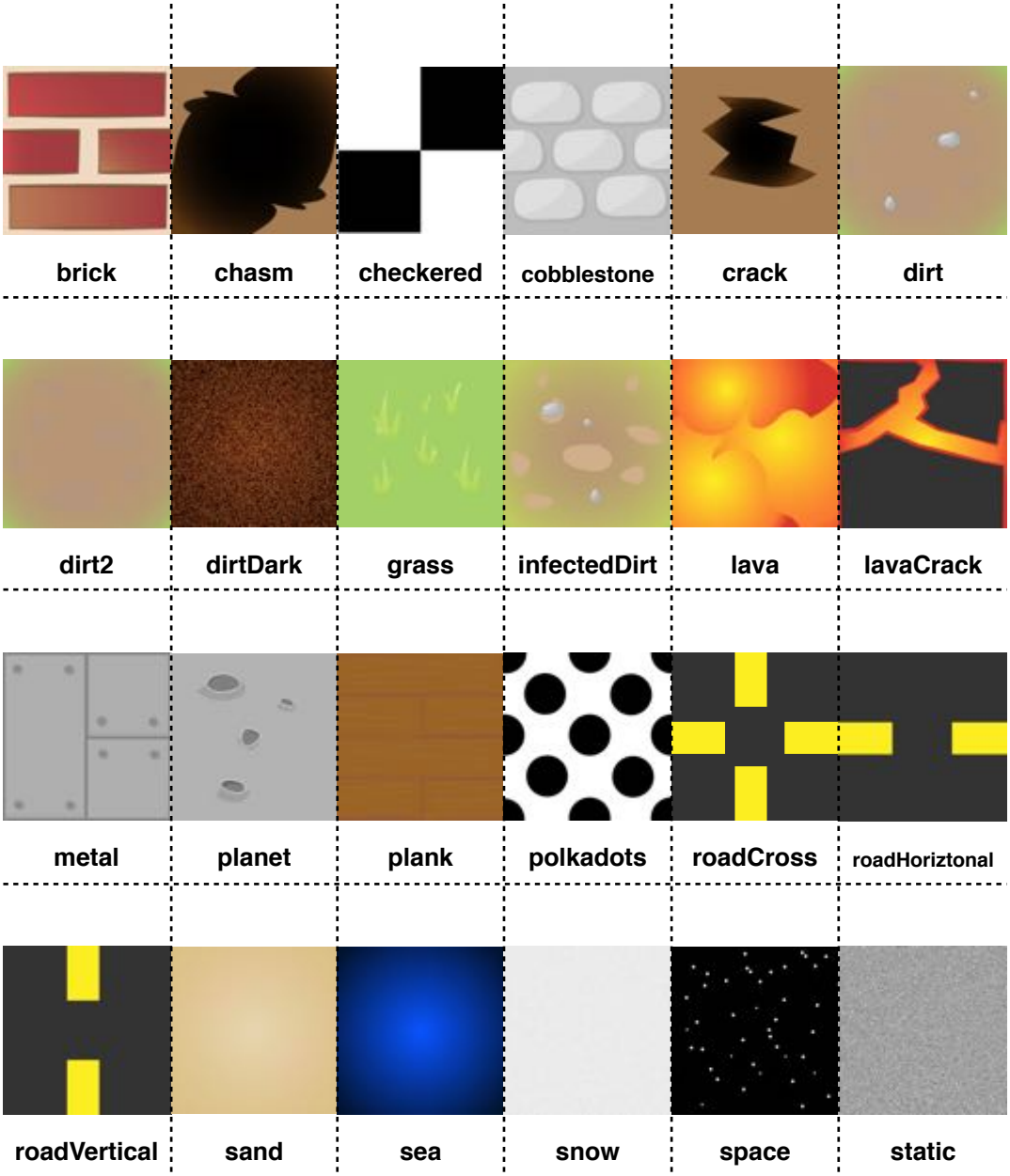
Objects & Characters (Page 4 of 6)

Objects & Characters (Page 5 of 6)

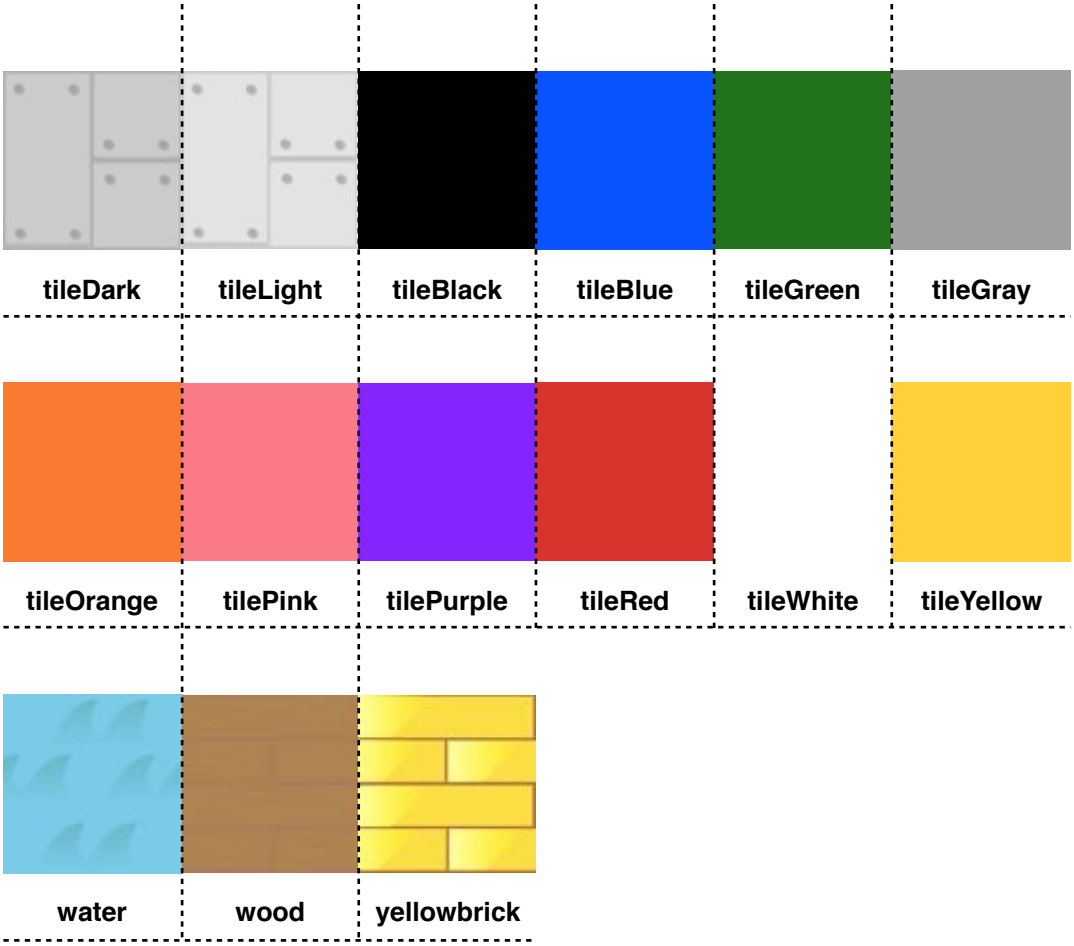
Objects & Characters (Page 6 of 6)



Ground Tiles (Page 1 of 2)

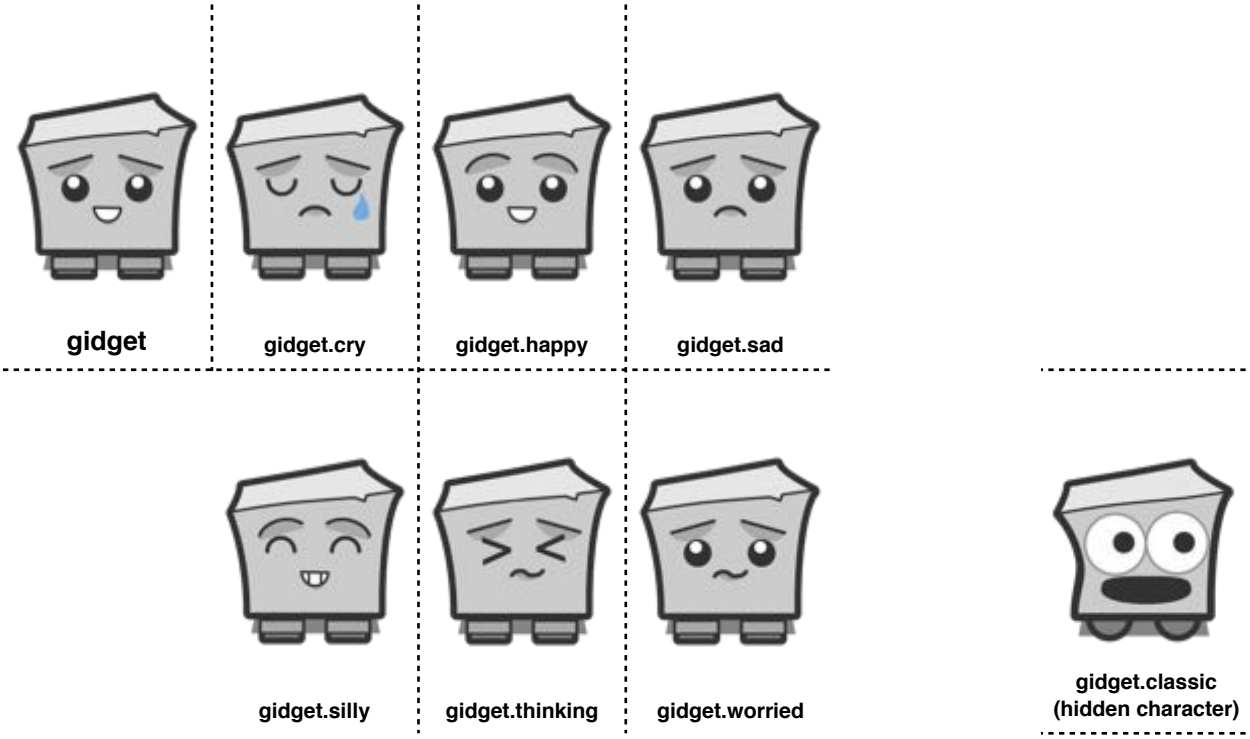


Ground Tiles (Page 2 of 2)

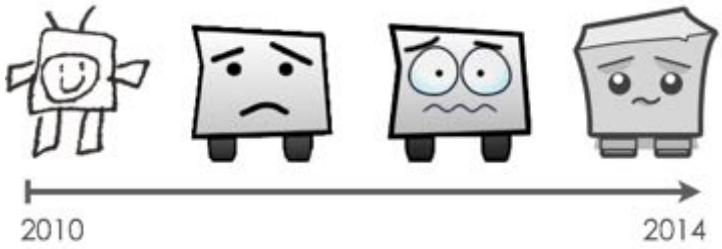


Extra Object Attributes (Page 1 of 1)

Extra Gidget Attributes (Page 1 of 1)



Gidget's Evolution



*Original sketch by Ellen Ko; Prototype Gidgets by Andrew Ko & Michael Lee; Finalized design by Fanny Luor.

Sounds Effects (Page 1 of 1)

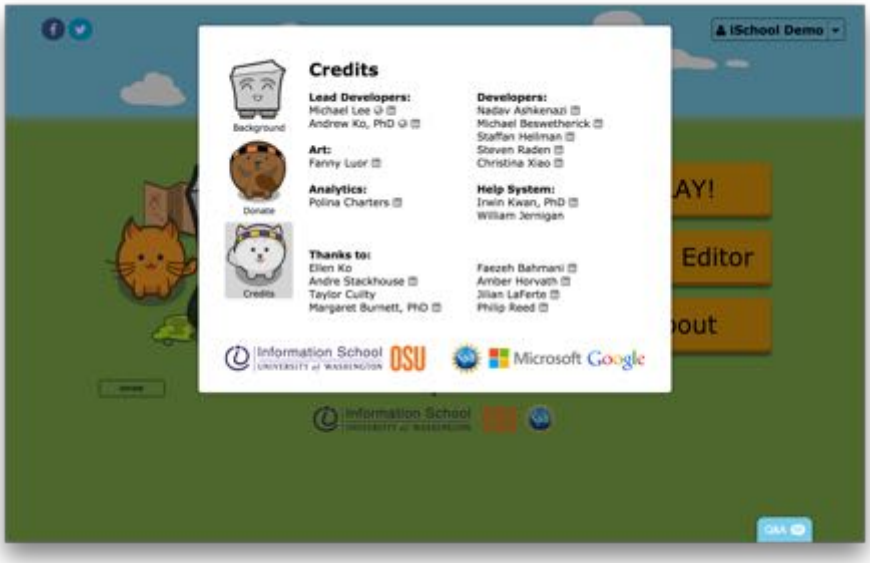
analyze*	block-F3	dog bark 3	kids cheering
bark	block-G3	dramatic accent	kitten
beep-A3	booming rumble	drop*	kitten meow
beep-B3	bounce	electricity surge	meow
beep-C3	brontosaurus wail	electro beep accent 01	mystery accents 01
beep-C4	cartoon boing	electro beep accent 02	mystery accents 02
beep-D3	cartoon chipmunk	electro beep accent 03	pig
beep-E3	cartoon cymbal hit	electro static accent 01	piglet oh dear
beep-F3	cartoon party horn	electro static accent 02	robot beeps
beep-G3	cartoon timpani	electro static accent 03	rooster call
bell-A3	cat	energy down*	scan*
bell-B3	chicken	energy up*	sheep
bell-C3	chimpanzee calls	error*	sitcom laughter 01
bell-C4	comedy whistle	focus in*	sitcom laughter 02
bell-D3	communication engaged	gliss apreggios 01	slamming metal lid
bell-E3	communication static	gliss apreggios 02	suspense accents 01
bell-F3	computer data 01	goal check failure*	suspense accents 02
bell-G3	computer data 02	goal check success*	telephone busy signal
bird chirp	computer data 03	goal final failure*	telephone dial done
bird chirp 2	computer data 04	goal final success*	telephone dial done
block-A3	computer data 05	goat	telephone no connection
block-B3	computer data 06	goto*	telephone no service
block-C3	cow moo	grab*	telephone ringing 01
block-C4	dinosaur growl	growling animal	telephone ringing 02
block-D3	dog bark	kids booing	tuning laser beam
block-E3	dog bark 2		

*these sounds effects were also used by the game for its default sounds.
all sound effects were provided by freesound.org.

- GIDGET GAME SCREENSHOTS

Landing Page & "About" Screen


iSchool Demo
Donate
Research and development have been generously sponsored by the National Science Foundation (NSF), Microsoft's Software Engineering Innovation Foundation (SEIF), and Google.
If you enjoy playing Gidget and would like to help keep it running, please consider donating to help pay for server and bandwidth costs.
donate
Background
Donate
Credits
Information School
OSU
Microsoft
Google
Editor
Q&A


iSchool Demo
Credits
Lead Developers:
Michael Lee
Andrew Ko, PhD
Art:
Fanny Luor
Analytics:
Polina Charters
Thanks to:
Ellen Ko
Andre Stackhouse
Taylor Cuilty
Margaret Burnett, PhD
Developers:
Nadav Ashkenazi
Michael Beswetherick
Staffan Heilman
Steven Raden
Christina Xiao
Help System:
Irwin Kwan, PhD
William Jernigan
Faezeh Bahmani
Amber Horvath
Jilian LaFerte
Philip Reed
Background
Donate
Credits
Information School
OSU
Microsoft
Google
Editor
Q&A

Gidget Game Levels


Level 3. Let's go grab the piglet!
code
Original Code
Clear Code
left 2
up
grab /piglet/
left 2
up 2
right 6
ensure /piglet/:position = [3,1]
One step
One line
To end
world
piglet
bucket
Objects in the world like the /piglet/ and myself need to be enclosed in slashes, / /, for me to understand what they are!
Prev
gidget
energy 18
grabbed []
image "default"
resized true
layer 1
name "gidget"
position [4, 4]
rotation 0
scale 1
transparency 1
Q&A

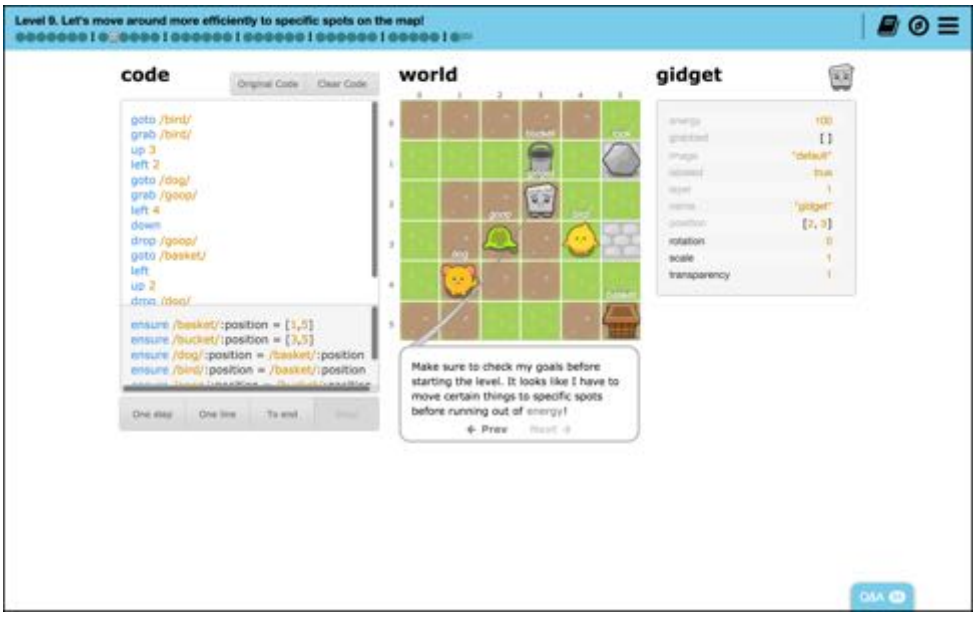

Level 4. Let's drop things into the correct containers!
code
Original Code
Clear Code
left
up
drop /goop/
down 3
left 1
grab /puppy/
left 2
up 3
grab /goop/
right 4
drop /puppy/
ensure /puppy/:position = /basket/:position
ensure /goop/:position = /bucket/:position
One step
One line
To end
world
bucket
goop
piglet
puppy
basket
The goal of this level is to drop the /puppy/ into the /basket/ and the /goop/ into the /bucket/. It might help to try running my code before editing it to see what happens!
Prev
gidget
energy 50
grabbed []
image "default"
resized true
layer 1
name "gidget"
position [2, 4]
rotation 0
scale 1
transparency 1
Q&A


Level 5. Let's organize everything correctly!
code
world
gidget
right 2
up
grab /cat/
down
grab /goop/
down 2
drop /goop/
left 2
up
grab /piglet/
down 2
right 3
grab /dog/
up
ensure /goop/:position = /bucket/:position
ensure /puppy/:position = /basket/:position
ensure /kitten/:position = /basket/:position
ensure /piglet/:position = /basket/:position
The goal of this level is to drop the /goop/ into the /bucket/ and the animals into the /basket/.
Prev
Q&A


Level 6. Where Will Gidget End Up?
code
world
gidget
down 2
grab /puppy/
up
left 3
drop /puppy/
up 2
left
grab /goop/
right 3
drop /goop/
down 2
left 2
Okay, I think I'm getting the hang of this. I want to try most of this by myself. Can you just help me by verifying what will happen by choosing from the options on the right?
After running the code (assuming I have unlimited energy), I will eventually end up:
On the final position of the the /puppy/.
On the final position of the /bird/.
On the final position of the /goop/.
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Q&A


Level 7. Where Will the Goop End Up?
code
world
gidget
up 1
right
grab /kitten/
left 3
grab /bird/
left 2
down 2
grab /piglet/
right 2
down 1
grab /goop/
right
down
grab /puppy/
Where will the /goop/ be after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Can you help me determine where the /goop/ will end up after running the code? We only have one shot at this so let's try our best! Click on the tile to choose your answer!
Prev
Next
Q&A


Level 8. Let's move more efficiently to each object!
code
world
gidget
Original Code
Clear Code
goto /goop/
grab /goop/
left 4
down
drop /kitten/
goto /basket/
left
up 2
drop /rock/
drop /goop/
ensure /kitten/:position = /basket/:position
ensure /piglet/:position = /basket/:position
ensure /goop/:position = /bucket/:position
ensure /rock/:position = /bucket/:position
One step
One line
To end
Try running my starting code to see how goto works! My goal is to get everything into the correct containers, but remember that I use more energy as I carry more things.
Prev
Next
Q&A


Level 9. Let's move around more efficiently to specific spots on the map!
code
Original Code
Clear Code
goto /bird/
grab /bird/
up 3
left 2
goto /dog/
grab /goop/
left 4
down
drop /goop/
goto /basket/
left
up 2
drop /dog/
ensure /basket/:position = [1,5]
ensure /bucket/:position = [3,5]
ensure /dog/:position = /basket/:position
ensure /bird/:position = /basket/:position
One step
One line
To end
world
Make sure to check my goals before starting the level. It looks like I have to move certain things to specific spots before running out of energy!
Prev
gidget
energy 100
grabbed []
image "default"
isGoal true
layer 1
name "gidget"
position [2, 3]
rotation 0
scale 1
transparency 1
Q&A


Level 10. Let's save the kitten twins!
code
Original Code
Clear Code
goto first /kitten/s
grab first /kitten/
goto last /basket/s
grab last /kitten/s
goto first /basket/s
ensure # /kitten/s on /basket/ = 2
One step
One line
To end
world
To solve this problem, I'll need to add an "s" to the end of the objects' name to make a list. Then I can use the same commands, with first and last.
Prev
gidget
energy 100
grabbed []
image "default"
isGoal true
layer 1
name "gidget"
position [0, 0]
rotation 0
scale 1
transparency 1
Q&A


Level 11. Let's organize two of a kind!
code
world
gidget
goto first /kitten/s
grab /kitten/
goto basket
drop /cat/
goto first /cat/s
grab first /cats/
goto last /cat/s
goto first /goop/s
goto last /goop/s
grab last /goop/s
goto /bucket/
drop last /goop/s
right 2
ensure gidget::position = [4,4]
ensure # /cat/s on /basket/ = 2
ensure # /kitten/s on /basket/ = 1
ensure # /goop/s on /bucket/ = 2
Remember, to use lists, I have to add an "s" to an object's name, and use first and last to access specific list items. After I'm done, move me to the corner /cobblestone/ tile!
Prev
Next
Q&A


Level 12. Where Will the Kittens End Up?
code
world
gidget
goto /bird/
grab /bird/
goto first /kitten/s
grab first /kitten/s
goto last /kitten/s
grab last /kitten/s
goto /puppy/
grab /puppy/
goto /basket/
drop /bird/
drop /puppy/
goto first /goop/s
grab first /goop/s
goto /bucket/
Where will the /kitten/s be after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Can you help me determine where the kittens will end up after running the code? We only have one shot at this so let's try our best! Click on the tile to choose your answer!
Prev
Next
Q&A


Level 13. Where Will the Birds End Up?
code
world
gidget
goto first /puppy/s
grab first /puppy/s
goto last /puppy/s
grab last /puppy/s
goto first /goop/s
grab first /goop/s
goto last /goop/s
grab last /goop/s
goto first /bird/s
grab first /birds/
goto last /bird/s
grab last /birds/
goto /basket/
drop first /puppy/s
Okay, I think I'm getting the hang of this. I want to try most of this by myself. Can you just help me by verifying what will happen by choosing from the options on the right?
After running the code (assuming I have unlimited energy), the two /bird/s will eventually end up:
On their original positions.
On the /basket/.
On the /bucket/.
On the /cobblestone/ tile at [0,4].
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Q&A


Level 14. The dog will help us!
code
world
gidget
Original Code
Clear Code
goto /dog/
set numBoulders to 6
say numBoulders
set sayThis to "Please Haelp me dogg"
say sayThis
ensure sayThis = "Please help me Dog!"
ensure /goopy/:position = /bucket/:position
ensure /dog/:position = /basket/:position
One step
One line
To end
I can use say followed by my variable name to display what I have stored in it. Let's try my code and ask the /dog/ for help! Check my goals first because what I store in the variable has to be exactly the same!
Prev
Next
Q&A


Level 15. The piglet will help us!
code
Original Code
Clear Code
set /piglet/:scale to 0.6
goto /goop/
grab /goop/
goto /bucket/
drop /goop/
ensure /piglet/:rotation = 0
ensure # /goop/s on /bucket/ = 1
One step
One line
To end
world
Let's help the /piglet/ get back up! Modify its rotation so that it's back to normal (check my rotation value or the goals for an example).
← Prev
gidget
energy 100
grabbed []
image "default"
selected true
layer 1
name "gidget"
position [3, 2]
rotation 0
scale 1
transparency 1
Q&A


Level 16. Read the note, pass the rat!
code
Original Code
Clear Code
set password to ["first", "second", "third"]
say password[0]
goto first password[0]
grab first password[0]
say password[1]
goto last password[0]
grab last password[0]
say password[2]
goto password[2]
drop
ensure password = [/goop/s, 2, /bucket/]
ensure # /goop/s on /bucket/ = 2
One step
One line
To end
world
I'm pretty sure that my starting code is correct, except for the password! Maybe going to that /note/ over there might give us a clue!
← Prev
gidget
energy 100
grabbed []
image "default"
selected true
layer 1
name "gidget"
position [1, 2]
rotation 0
scale 1
transparency 1
Q&A


Level 17. The dog will help us here!
code
world
gidget
goto /rock/
say "I'm going to ask this helpful dog to mo
set moveThis to [5,0]
say "Thanks for moving the thing at " + mo
set moveThis to [1,2]
say "Thanks for moving the thing at " + mo
up 2
ensure moveThis = [2,6] or moveThis = [3,
ensure /bird/:position = /basket/:position
ensure /piglet/:position = /basket/:position
One step
One line
To end
Once we assign moveThis with the correct coordinates, the /dog/ will move any object there for us! We can keep reassigning moveThis with new values until we're done!
Prev
Next
Q&A


Level 18. What's the array's value?
code
world
gidget
set a to /bird/:position
set b to /bucket/:position
set c to /goop/:position
set myArray to [a, b, c]
goto /goop/
grab /goop/
goto /bucket/
drop /goop/
goto /bird/
grab /bird/
up
I want to try most of this by myself. Can you just help me by verifying what will happen after we run the program by choosing from the options on the right?
Prev
Next
I'm going to try remembering the objects' positions. After running the code (assuming I have unlimited energy), the variable "myArray" will be equal to:
[[3,2], [1,3], [1,0]]
[[2,2], [1,3], [1,3]]
[[3,2], [1,3], [1,3]]
[[1,0], [1,3], [3,2]]
nothing, because there is an error in the code.
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Q&A


Level 19. What Will Gidget Say?
code
world
gidget
set myArray to [/puppy/, /dog/, /cat/]
goto myArray[1]
set myArray[3] to /kitten/
goto myArray[1]
grab myArray[1]
goto /basket/
drop myArray[1]
say "The " + myArray[2] + " isn't in the bas
Alright, I'm getting the hang of this! I want to try most of this by myself. Can you just help me by verifying what will happen by choosing from the options on the right?
I'm going to try remembering the objects' positions. After running the code (assuming I have unlimited energy), what will my final "say" be at the end of the code output?:
"The /cat/ isn't in the /basket/ yet."
"The /puppy/ isn't in the /basket/ yet."
"The /dog/ isn't in the /basket/ yet."
"The /kitten/ isn't in the /basket/ yet."
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Q&A


Level 20. Press the button, open the gate!
code
world
gidget
Original Code
Clear Code
goto /button/
say "Let's click the button to see its function"
/button/:openFence()
function getPiglet()
goto /piglet/
set /piglet/:nickname to "wilbur"
set /piglet/:age to 3
grab /piglet/
getBird()
getThePiggy()
goto /basket/
ensure /piglet/:nickname = "babe"
ensure /piglet/:age = 3
ensure # /piglet/ on /basket/ = 1
One step
One line
To end
Don't forget you can click on objects to see their properties, and you should try running my code first to see what happens!
Prev
Q&A


Level 21. Flip the animals right-side-up!
code
world
gidget
function transferThing(whichThing, toWher
goto whichThing
grab whichThing
set whichThing:scale to 1.4
say whatIDid(whichThing)
goto toWhere
drop
function whatIDid(item)
set sentence to "I grabbed the " + item
return sentence
transferThing(/goop/, /bucket/)
transferThing()
transferObject(/bird/, /bucket/)
You can use keep using the same function over-and-over! Try running my code to see what happens!
Prev


Level 22. Let's plug the pipe with rocks!
code
world
gidget
object rock(saySomething)
say saySomething
goto /goop/
remove /goop/
up 3
create rock("hello! i'm a rock!")
grab /rock/
I can also create /rock/s on each of the /cobblestone/ spots to plug the pipes! I can create new objects by declaring them like functions (but using the object command).
Prev


Level 23. Clean up the goop, plug the pipe, and plant the sapling!
code
world
gidget
object puppy(sayThis)
say sayThis
set this:age to 5
object kitten()
say "meow"
set this:fluffiness to 50
function removeGoop()
goto /goop/
remove /goop/
create kitten()
removeGoop()
up
create puppy("woof woof")
ensure /goop/ = nothing
The goals for this level are to put a /rock/ in the pipe again, remove all the /goop/s, and create a /sapling/ to take each of their places on any of the ground spots.
Prev
Next
rotation
scale
transparency

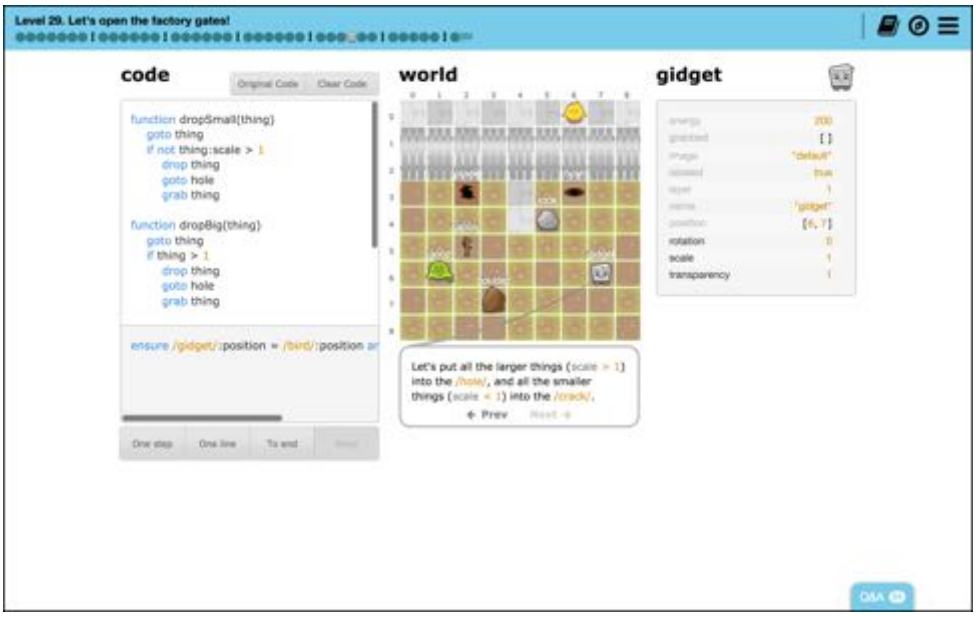

Level 24. Where Will The Cat End Up?
code
world
gidget
object cat()
say "meow"
function makeCat(num)
goto /goop/
up num
right num
down num - 1
create cat()
left num * num
up num
makeCat(2)
Can you help me determine where the /cat/ will end up after running the code? We only have one shot at this so let's try our best! Click on the tile to choose your answer!
Prev
Next
Where will the /cat/ I create be located after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer


Level 25. Where Will Gidget End Up?
code
world
gidget
set x to [30, 10, 0]
object piglet(number)
set this:weight to number
function shuffle(newNumber, myArray)
set myArray[0] to myArray[0] + newNu
set myArray[1] to myArray[1] + myArr
set myArray[2] to myArray[2] * newNu
return myArray
set x to shuffle(x[1], x)
create piglet(x[1]+x[2])
Okay, I should get a little more practice using functions and objects and I want to try most of this by myself. Can you just help me by verifying what will happen by choosing from the options on the right?
After running the code (assuming I have unlimited energy), the /piglet/'s weight will be:
The /piglet/'s weight is 10.
The /piglet/'s weight is 30.
The /piglet/'s weight is 20.
There will be no /piglet/.
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Q&A


Level 26. True or False: Are the animals infected?
code
world
gidget
Original Code
Clear Code
function checkForInfection(objectName)
goto objectName
if objectName:infected = true
grab objectName
checkForInfection(/shrub/)
checkTheDog(/dog/)
checkForInfection(/sapling/)
checkForInfection(/rock/)
checkForInfection(/bird/)
goto /basket/
We can check the property of each object for boolean values (which means they are always either true or false) and an if statement, which let's us do something based on whether the boolean is true or false!
Prev
Q&A

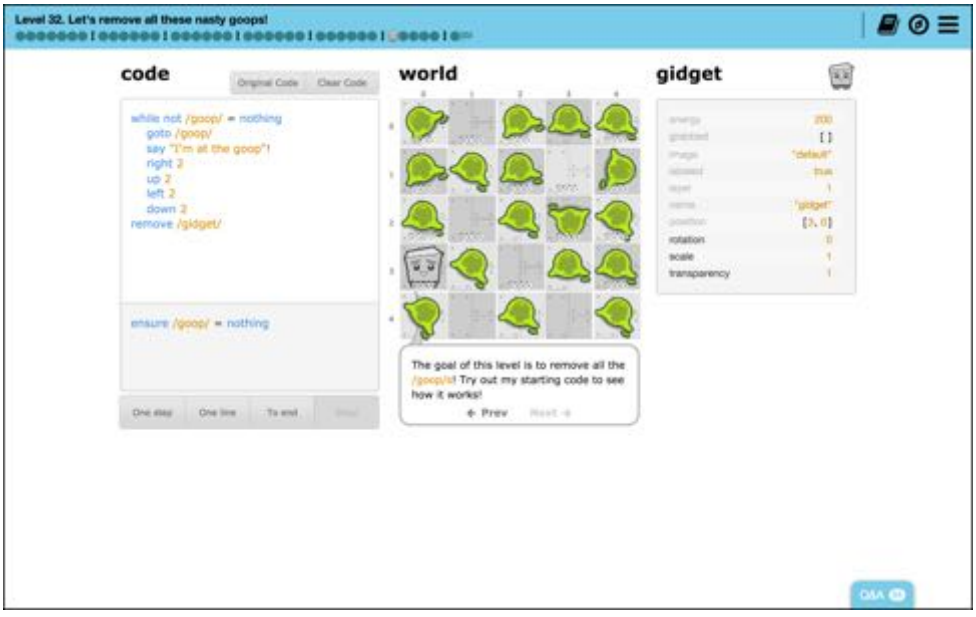

Level 27. Let's clean up some more!
code
world
gidget
function organize(objectName)
goto objectName
grab objectName
if objectName:infected = true
goto /container/
else
goto /container/
drop
organize(/sapling/)
organize(/puppy/)
organize()
organize("kitten")
organize(/shrub/)
The goal of this level is to put the uninfected things into the /basket/, and the infected things into the /container/ (so we can disinfect them later)!
Prev


Level 28. Are the animals infected or sick?
code
world
gidget
function organize(objectName)
goto thingName
grab thingName
goto /shrub/
drop
else
grab thingName
goto /container/
drop
organize(/shrub/)
Help me organize these animals into the correct bins. Animals that are infected and sick should go in the /container/. Animals that are infected or sick should go into the /basket/.
Prev


Level 29. Let's open the factory gates!
code
world
gidget
function dropSmall(thing)
goto thing
if not thing:scale > 1
drop thing
goto hole
grab thing
function dropBig(thing)
goto thing
if thing > 1
drop thing
goto hole
grab thing
Let's put all the larger things (scale > 1) into the /hole/, and all the smaller things (scale < 1) into the /crack/.
Prev


Level 30. Where Will Gidget End Up?
code
world
gidget
goto /cat/
if /cat/:position = [0,5]
goto /bird/
else
goto /piglet/
if not /dog/:position[1] = 3
down 2
else
right 3
Where will I be after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Can you help me determine where I will end up after running the code? Since we only have one shot at this, I hope you get it right! Click on the tile to choose your answer!
Prev


Level 31. Where Will Gidget End Up?
code
world
gidget
goto /piglet/
if /gidget/:position = /piglet/:position and /
goto /bird/
else
goto /button/
if /goop/:position = /bucket/:position or /kit
up 3
else
left 2
Can you just help me by verifying what will happen by choosing from the options on the right?
Prev
After running the code (assuming I have unlimited energy), I will eventually end up on the:
/piglet/
/puppy/
/goop/ & /bucket/
/kitten/ & /basket/
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Q&A


Level 32. Let's remove all these nasty goops!
code
Original Code
Clear Code
world
gidget
while not /goop/ = nothing
goto /goop/
say "I'm at the goop"!
right 2
up 2
left 2
down 2
remove /gidget/
ensure /goop/ = nothing
One step
One line
To end
The goal of this level is to remove all the /goop/s! Try out my starting code to see how it works!
Prev
Q&A


Level 33. Let's gather materials and clean up the goops!
code
world
gidget
Original Code
Clear Code
set materialList to /material/s
for m in materialList
goto m
grab m
goto /bucket/
set myGoops to /goop/s
for g in goopList
goto g
set g:scale to 1.5
grab g
goto /bucket/
ensure # /material/s on /container/ = 4
ensure AllMaterialScaleGreaterThanOne()
ensure # /goop/s on /bucket/ = 11
One step
One line
To end
The goals of this level are to get the /goop/s into the /bucket/ and enlarge the /material/ before putting them into the /container/. Try running my starting code first to get an idea about how for works!
Prev
Q&A


Level 34. Let's shut down the goop factory!
code
world
gidget
Original Code
Clear Code
function workGoops(test)
if test = "small"
for g in /goop/s
goto g
if g:scale = 1
grab g
else
for g in /goop/s
goto g
if g:scale > 1
remove g
workGoops("small")
goto /bucket/
drop
doGoop("large")
ensure # /goop/s on /bucket/ = 2
ensure # /goop/s on /container/ = 6
ensure # /kitten/s on /basket/ = 2
ensure /masterswitch/:activated = true
One step
One line
To end
Once we're in the vault control room, we need to use the /masterswitch/ button's built-in function to shut down the factory!
Prev
Q&A

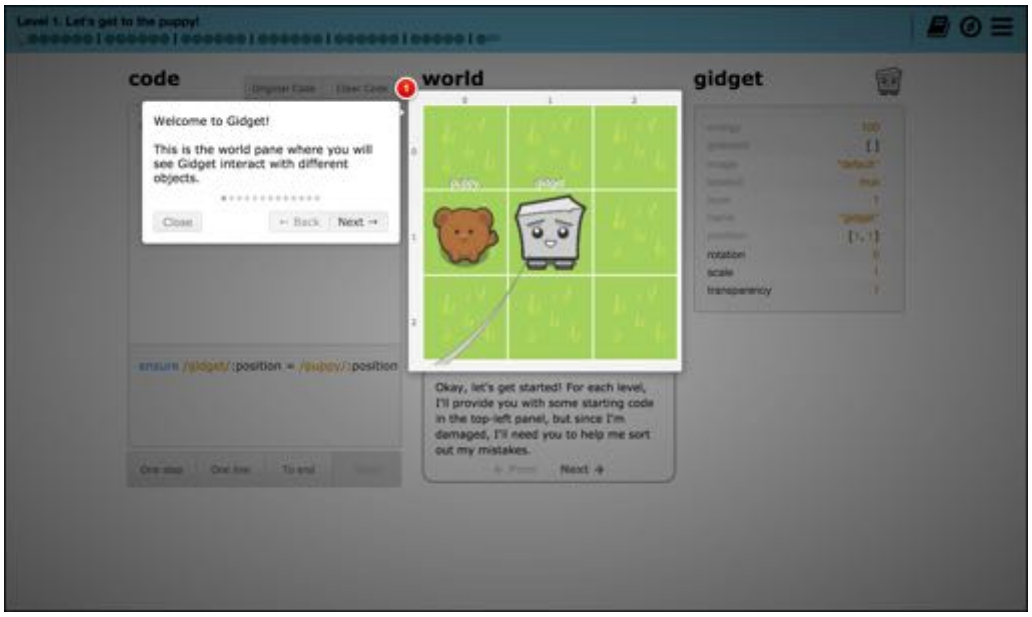

Level 35. Where Will Gidget End Up After a Victory Dance?
code
world
gidget
set loop1 to 2
set loop2 to 3
say "Yippee!"
while not loop1 = 0
left 2
down 2
right 1
up 2
set loop1 to loop1 - 1
while loop2 > 1
up 1
down 2
Can you help me determine where I will end up after running the code? We only have one shot at this so let's try our best! Click on the tile to choose your answer!
Prev
Where will I be after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Q&A

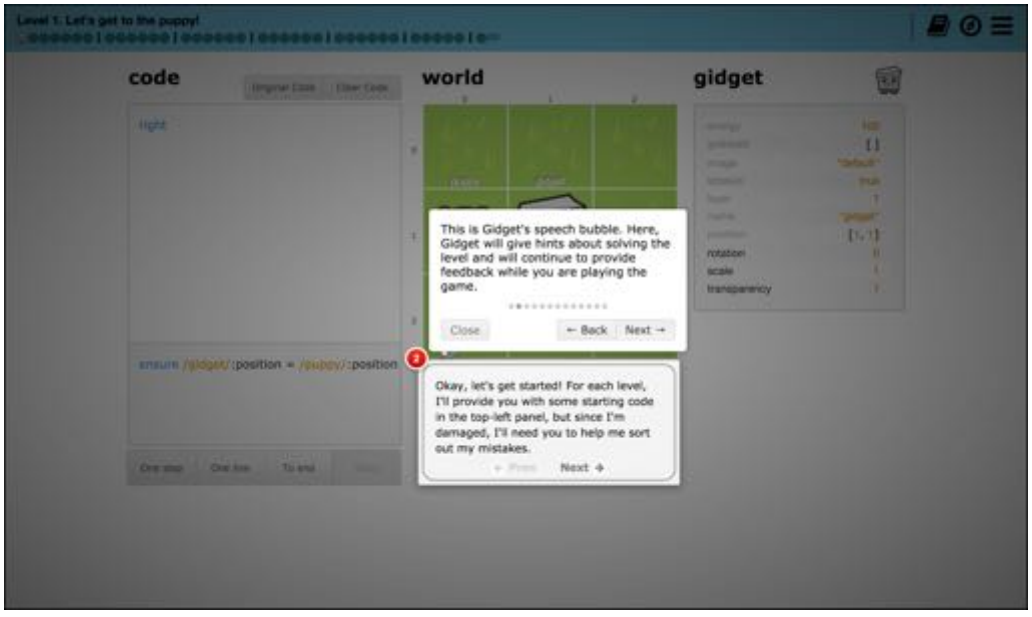

Level 36. Where Will Gidget End Up After a Victory Dance?
code
world
gidget
set leaves to 0
set plants to /sapling/s
for s in plants
if /bird/:position = [1,3]
set leaves to leaves + 3
else
set leaves to leaves + 2
goto /bird/
grab /bird/
goto /basket/
I want to try counting the leaves mostly by myself. Can you just help me by verifying what will happen by choosing from the options on the right?
Prev
After running the code (assuming I have unlimited energy), the variable leaves will be equal to:
9
0
8
7
6
Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!
0 words written.
submit answer
Q&A

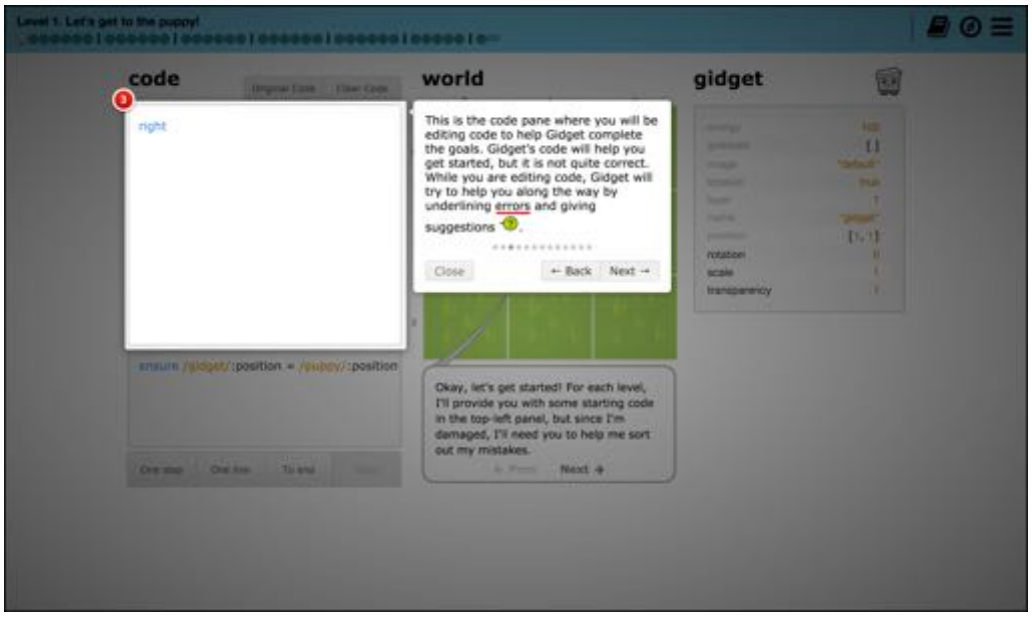

Level 37. Let's help Gidget run out of energy!
code
world
gidget
ensure /gidget/:energy < 15
ensure /kitten/:position = /basket/:position
ensure /dog/:position = /basket/:position
I should reduce my energy to less than 15 (but not all the way down to 0!), and you can use any of the commands you have mastered to get me there!
Prev
Q&A

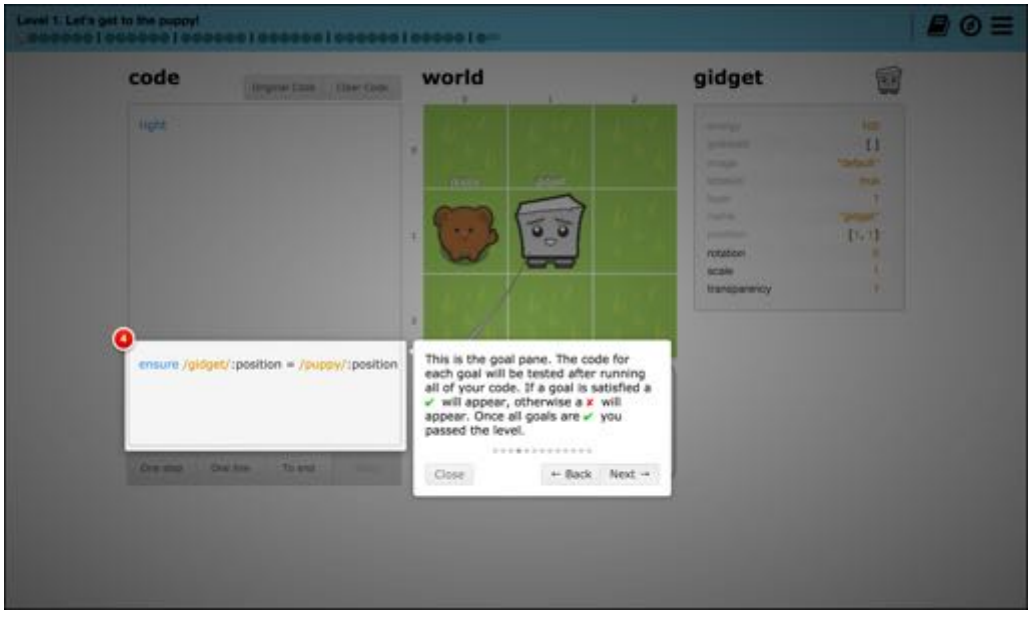

Level 37. Let's help Gidget run out of energy!
code
Success!
Congratulations, you have helped Gidget complete all the missions!
Thanks to your efforts, Gidget's logic chip is almost repaired, the goop factory is shut down, and all the animals are safe!
Now you can create your own adventures with Gidget using the level editor! (You can get to the Editor from the game menu or start page).
Enable the Level Editor!
Thank you for helping me complete my mission! Now you can use the level editor to make new adventures for me!
Q&A

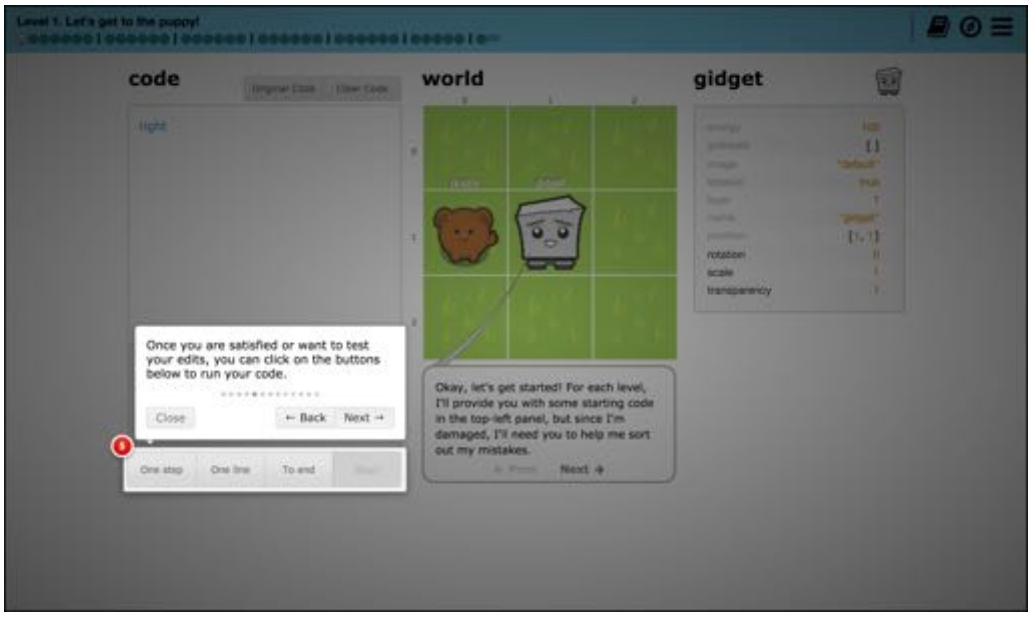
Interactive Tutorial





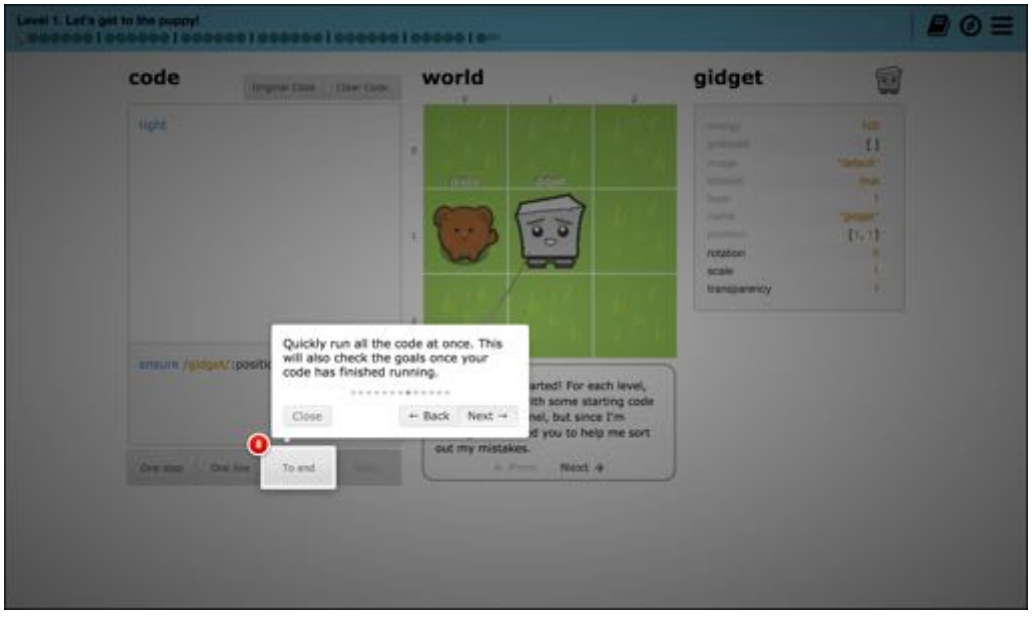

Level 1: Let's get to the puppy!
code
world
gidget
right
ensure /gidget/:position = /puppy/:position
This is the code pane where you will be editing code to help Gidget complete the goals. Gidget's code will help you get started, but it is not quite correct. While you are editing code, Gidget will try to help you along the way by underlining errors and giving suggestions
Close
← Back
Next →
Okay, let's get started! For each level, I'll provide you with some starting code in the top-left panel, but since I'm damaged, I'll need you to help me sort out my mistakes.
Next →

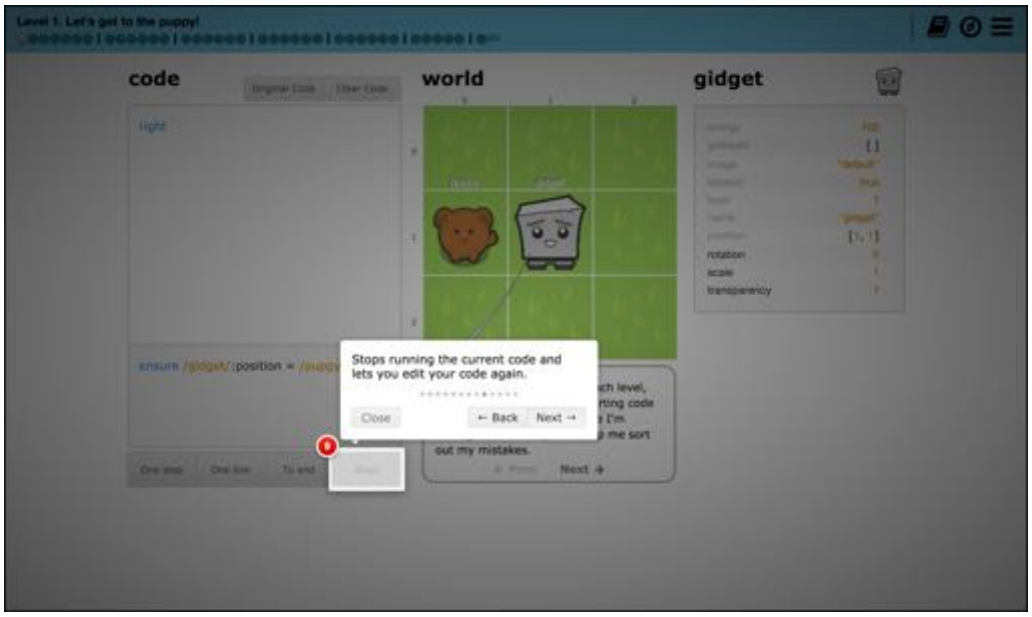

Level 1: Let's get to the puppy!
code
world
gidget
right
ensure /gidget/:position = /puppy/:position
This is the goal pane. The code for each goal will be tested after running all of your code. If a goal is satisfied a ✔ will appear, otherwise a ✘ will appear. Once all goals are ✔ you passed the level.
Close
← Back
Next →

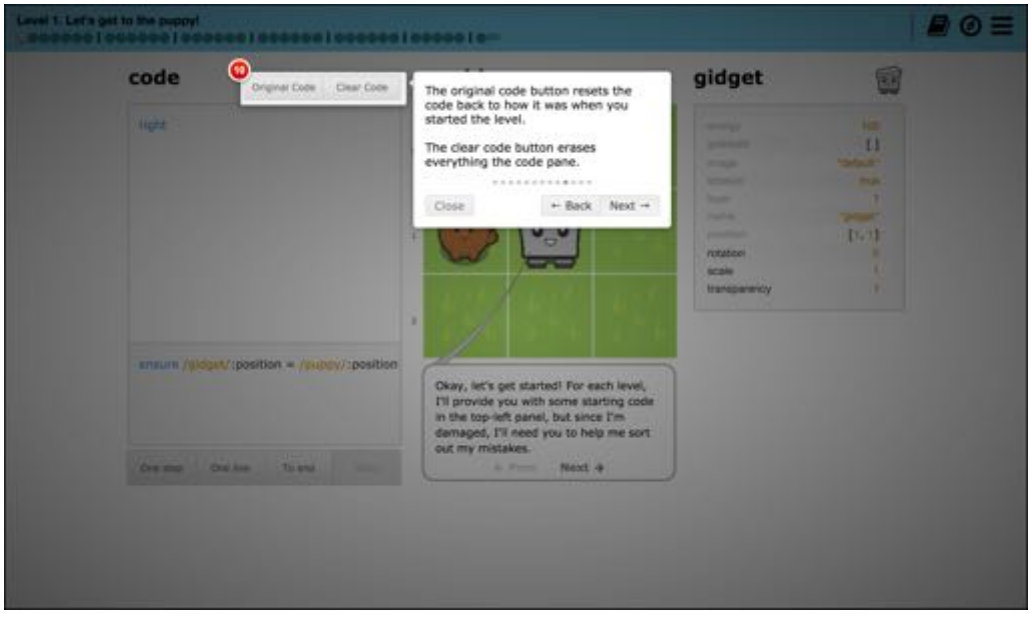

code
world
gidget
Once you are satisfied or want to test your edits, you can click on the buttons below to run your code.
Close
← Back
Next →
One step
One line
To end
Okay, let's get started! For each level, I'll provide you with some starting code in the top-left panel, but since I'm damaged, I'll need you to help me sort out my mistakes.
Next →

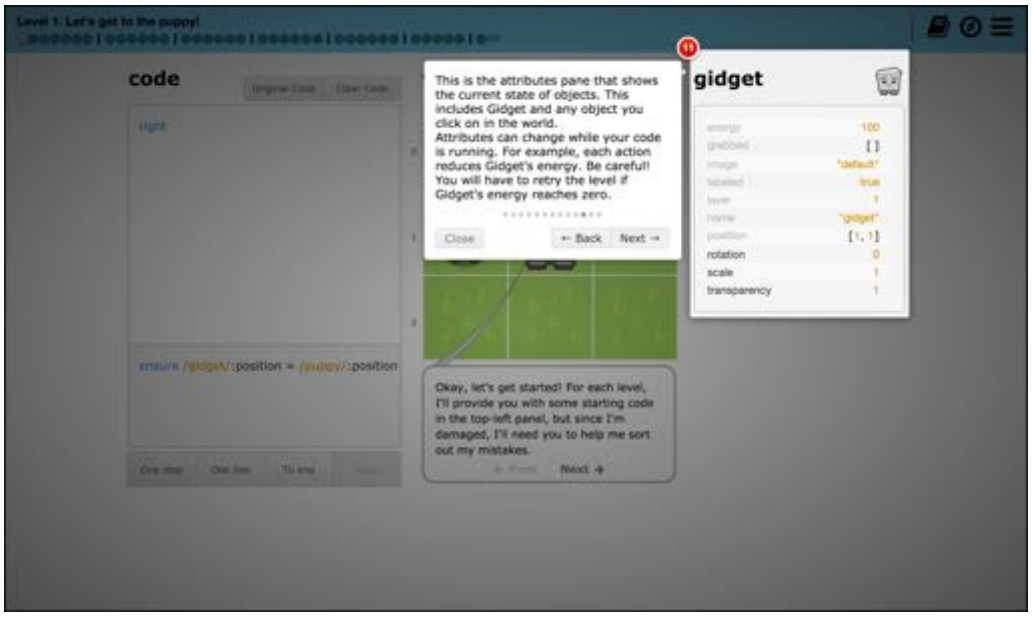

code
world
gidget
Run the code with explanations between each step. Make sure you click this button multiple times to see multiple steps.
Close
← Back
Next →
One step
Okay, let's get started! For each level, I'll provide you with some starting code in the top-left panel, but since I'm damaged, I'll need you to help me sort out my mistakes.
Next →

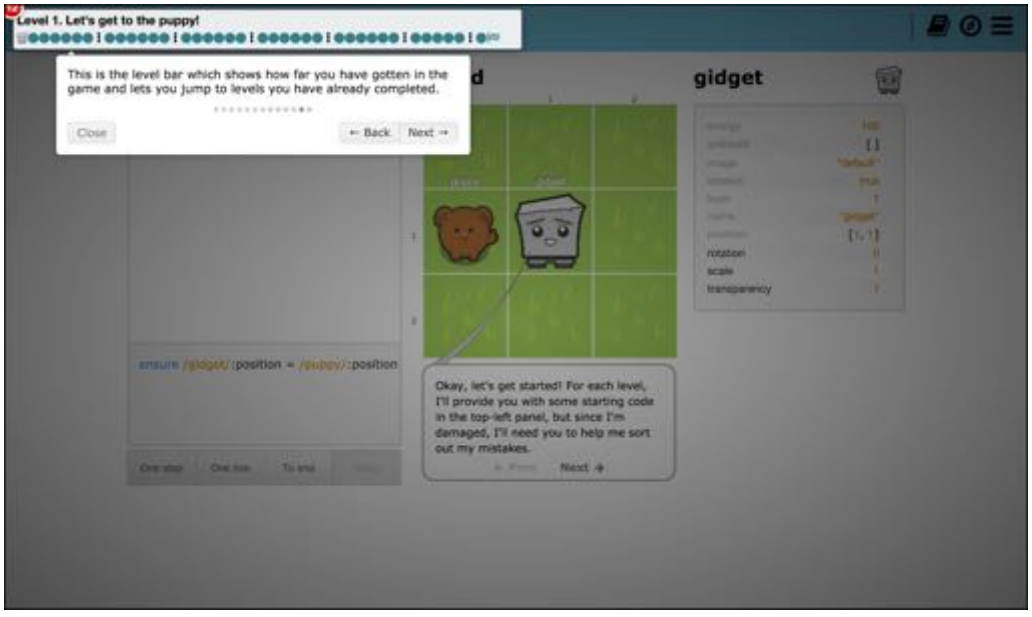

Level 1: Let's get to the puppy!
code
world
gidget
right
Run the code an entire line at a time. Make sure you click this button multiple times to see multiple lines.
Close
← Back
Next →
One line
Next →

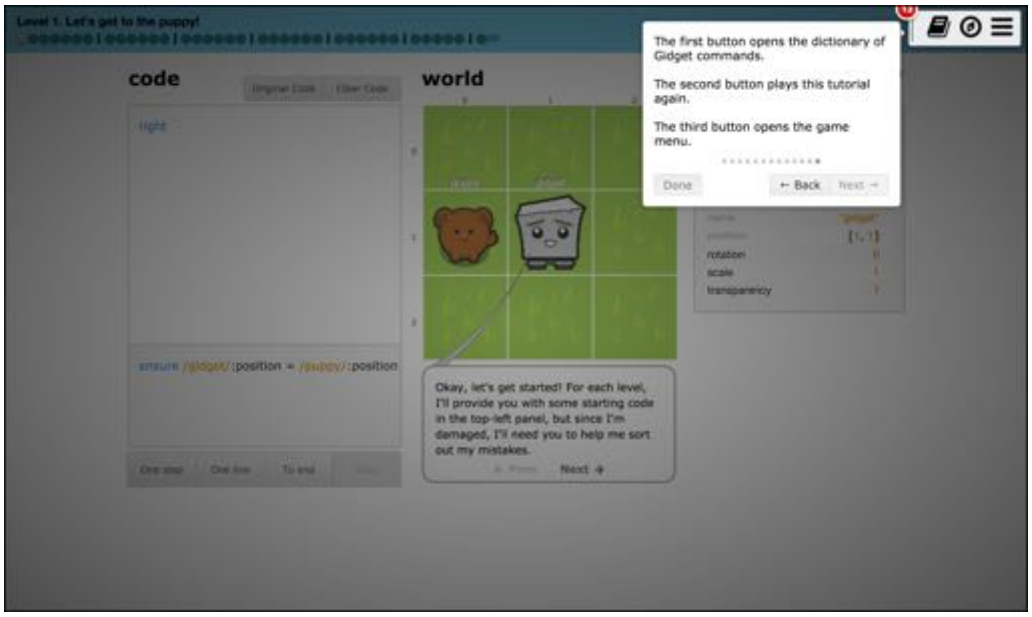

Level 1: Let's get to the puppy!
code
world
gidget
right
Quickly run all the code at once. This will also check the goals once your code has finished running.
Close
← Back
Next →
To end
Next →


code
world
gidget
Stops running the current code and lets you edit your code again.
Close
← Back
Next →
out my mistakes.
Next →


code
Original Code
Clear Code
The original code button resets the code back to how it was when you started the level.
The clear code button erases everything the code pane.
Close
← Back
Next →
gidget
Okay, let's get started! For each level, I'll provide you with some starting code in the top-left panel, but since I'm damaged, I'll need you to help me sort out my mistakes.
Next →

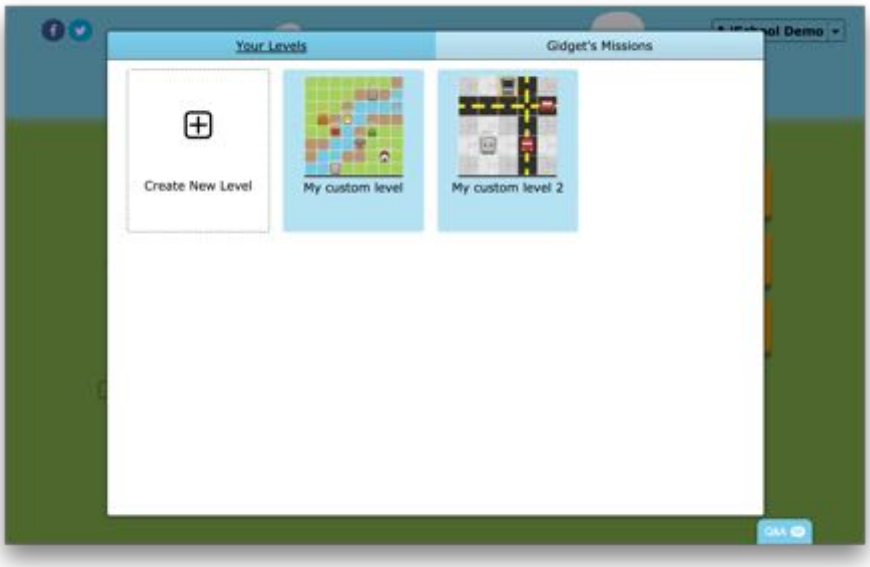

Level 1. Let's get to the puppy!
code
This is the attributes pane that shows the current state of objects. This includes Gidget and any object you click on in the world.
Attributes can change while your code is running. For example, each action reduces Gidget's energy. Be careful! You will have to retry the level if Gidget's energy reaches zero.
Close
← Back
Next →
gidget
rotation
scale
transparency
Okay, let's get started! For each level, I'll provide you with some starting code in the top-left panel, but since I'm damaged, I'll need you to help me sort out my mistakes.
Next →


Level 1. Let's get to the puppy!
This is the level bar which shows how far you have gotten in the game and lets you jump to levels you have already completed.
Close
← Back
Next →
gidget
rotation
scale
transparency
Okay, let's get started! For each level, I'll provide you with some starting code in the top-left panel, but since I'm damaged, I'll need you to help me sort out my mistakes.
Next →


Level 1. Let's get to the puppy!
code
world
right
ensure /gidget/:position = /puppy/:position
The first button opens the dictionary of Gidget commands.
The second button plays this tutorial again.
The third button opens the game menu.
Done
← Back
Next →
rotation
scale
transparency
Okay, let's get started! For each level, I'll provide you with some starting code in the top-left panel, but since I'm damaged, I'll need you to help me sort out my mistakes.
Next →

Options Menu & Dictionary Interface

Gidget Puzzle Designer (Level Editor) Menus

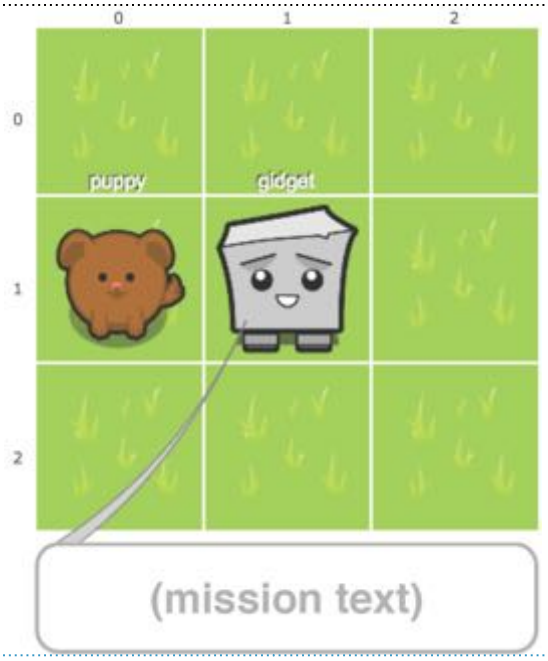


- GIDGET CURRICULUM – DETAILED LEVEL BREAKDOWN
    - This section contains a detailed breakdown of each level.
    - Each level is explained in two pages.
        - The first page contains:
            - The World Code – this is the code that the game initializes when the level is loaded. It sets up the position of all the elements in the level. It is inaccessible to the player during gameplay, but editable using the level editor.
            - Mission Text – this is the dialogue that Gidget says at the beginning of each level. Each bullet point represents one speech bubble's message, requiring the player to click on the "next" button to proceed to the next message.
        - The second page (for regular levels) contains:
            - Gidget Code – this is the intentionally broken code that the player must fix to pass the level.
            - Gidget Code (solution) – this is a solution to the level. There may be other solutions to pass the level.
            - Gidget Goals – these are the goals that the player must satisfy with the Gidget Code to pass the level.
        - The second page (for assessment levels) contains:
            - Gidget Code – this is the code that the player must read to select the right solution to the assessment.
            - Assessment Question – this is the question that Gidget asks, and the possible answer choices that the player can choose (bulleted).
            - Solution Code & Responses – these are the various responses that Gidget will use to respond to correct and/or incorrect answers.

Level 1. Let's get to the puppy!

World Code:

```
object puppy(row, column)
 set this:position to [row, column]
create puppy(1, 0)
```



Mission Text:

- Okay, let's get started! For each level, I'll provide you with some starting code in the top-left panel, but since I'm damaged, I'll need you to help me sort out my mistakes.
- Feel free to modify, delete, or reuse any of the code I give you! There should be clues inside to teach you how to use my commands effectively!
- Make sure you always read the goals of the level on the bottom-left panel first, and then try running the code at least once using the buttons below the goals to see how the starting code works.
- It looks like the goal of this level is to move myself to the /puppy/! Use the buttons on the bottom-left to see what my code does, and click on the top-left white panel to start editing!

Level 1. Let's get to the puppy!

Gidget Code (broken):

```
right
```

Gidget Code (solution):

```
left
```

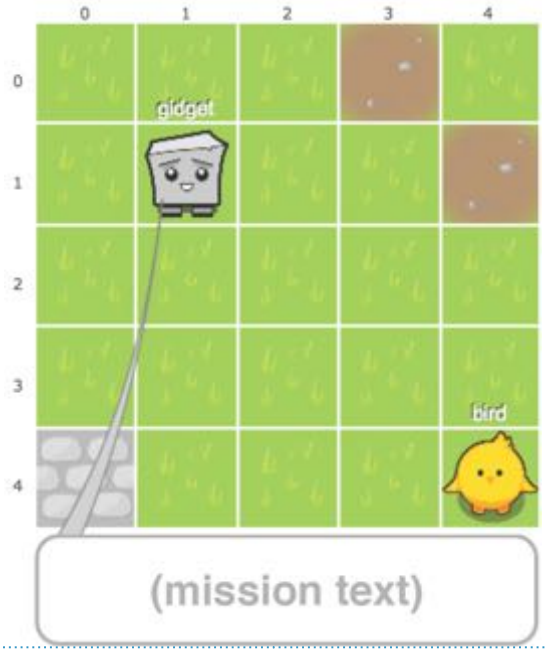
Gidget Goals:

```
ensure /gidget/:position = /puppy/:position
```

Level 2. Let's get to the bird!

World Code:

```
object bird(row, column)
 set this:position to [row, column]
create bird(4, 4)
```



Mission Text:

- I'm getting better at this thanks to you! My next goal is to get to that /bird/ over there.
- I should be able to move more easily by including an optional number literal immediately after my movement command.

Level 2. Let's get to the bird!

Gidget Code (broken):

```
up
right 5
down 4
left 3
```

Gidget Code (solution):

```
right 3
down 3
```

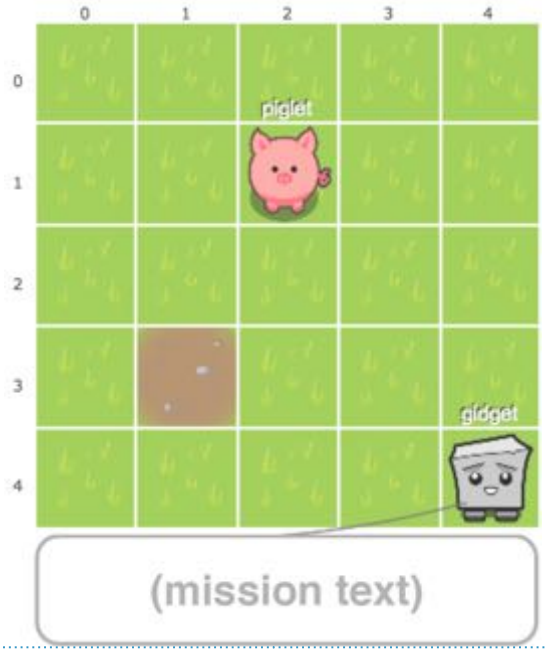
Gidget Goals:

```
ensure /gidget/:position = /bird/:position
```

Level 3. Let's go grab the piglet!

World Code:

```
object piglet(row, column)
 set this:position to [row, column]
create piglet(1, 2)
```



Mission Text:

- I should be able to grab things and transport them to other spaces, but it will take more energy to move around while carrying something!
- The goal for this level is to move the /piglet/ to that patch of /dirt/ at [3,1]! Remember that in my world, the grid system uses the row, then the column like in a spreadsheet.
- I should use up/down/left/right to move around, and the grab command to pick up the /piglet/!
- Objects in the world like the /piglet/ and myself need to be enclosed in slashes, / /, for me to understand what they are!

Level 3. Let's go grab the piglet!

Gidget Code (broken):

```
left 2
up
grab /piglet/
left 2
up 2
right 6
```

Gidget Code (solution):

```
left 2
up 3
grab /piglet/
down 2
left
```

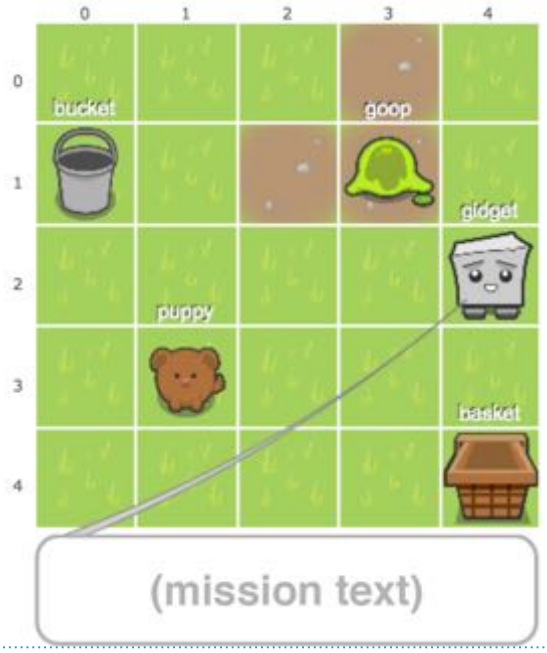
Gidget Goals:

```
ensure /piglet/:position = [3,1]
```

Level 4. Let's drop things into the correct containers!

World Code:

```
object puppy(row, column)
 set this:position to [row, column]
object goop(row, column)
 set this:position to [row, column]
object bucket(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]
create puppy(3, 1)
create goop(1, 3)
create bucket(1, 0)
create basket(4, 4)
```



Mission Text:

- Alright, I'll start cleaning up this mess by putting things in the correct containers!
- Remember, it takes more energy to move around when I'm grabbing something, so I might need to take an efficient path before I run out!
- The goal of this level is to drop the /puppy/ into the /basket/ and the /goop/ into the /bucket/. It might help to try running my code before editing it to see what happens!

Level 4. Let's drop things into the correct containers!

Gidget Code (broken):

```
left
up
drop /goop/
down 3
left 1
grab /puppy/
left 2
up 3
grab /goop/
right 4
drop /puppy/
```

Gidget Code (solution):

```
left
up
grab /goop/
left 3
drop /goop/
down 2
right
grab /puppy/
right 3
down
drop /puppy/
```

Gidget Goals:

```
ensure /puppy/:position = /basket/:position
ensure /goop/:position = /bucket/:position
```

Level 5. Let's organize everything correctly!

World Code:

```
object puppy(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]
object goop(row, column)
 set this:position to [row, column]
object bucket(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]
create puppy(5, 3)
create goop(4, 2)
create kitten(1, 5)
create piglet(3, 0)
create bucket(1, 0)
create basket(4, 4)
```

Mission Text:

- Wow, there are a lot of animals here! The /goop/ is toxic, so I should clean this place up quickly!
- Remember, it takes more energy to move around when I'm grabbing something, so I might need to take an efficient path before I run out!
- The goal of this level is to drop the /goop/ into the /bucket/ and the animals into the /basket/.

Level 5. Let's organize everything correctly!

Gidget Code (broken):

```
right 2
up
grab /cat/
down
grab /goop/
down 2
drop /goop/
left 2
up
grab /piglet/
down 2
right 3
grab /dog/
up
right
drop /puppy/
```

Gidget Code (solution):

```
right 2
up
grab /kitten/
down 2
left 5
grab /piglet/
down 2
right 3
grab /puppy/
up
right
drop
left 2
grab /goop/
up 3
left 2
drop /goop/
```

Gidget Goals:

```
ensure /goop/:position = /bucket/:position
ensure /puppy/:position = /basket/:position
ensure /kitten/:position = /basket/:position
ensure /piglet/:position = /basket/:position
```

Level 6. Where Will Gidget End Up?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]

object goop(row, column)
 set this:position to [row, column]

object bucket(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

object bird(row, column)
 set this:position to [row, column]

create basket(3, 1)
create bucket(1, 3)
create goop(1, 0)
create puppy(4, 4)
create bird(3, 2)
```

Mission Text:

- Okay, I think I'm getting the hang of this. I want to try most of this by myself. Can you just help me by verifying what will happen by choosing from the options on the right?

Level 6. Where Will Gidget End Up?

Gidget Code (assessment):

```
down 2
grab /puppy/
up
left 3
drop /puppy/
up 2
left
grab /goop/
right 3
drop /goop/
down 2
left 2
```

Assessment Question:

After running the code (assuming I have unlimited energy), I will eventually end up:

- On the final position of the the /puppy/.
- On the final position of the /bird/.
- On the final position of the /goop/.

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

```
correct:On the final position of the the <span class='object'>/puppy/</span>.:After
dropping off the <span class='object'>/goop/</span> on the <span class='object'>/
bucket/</span>, I will move back on the <span class='object'>/basket/</span> and
the <span class='object'>/puppy/</span>. Remember that number <span
class='dictionaryTerm'>literal</span>s after the <span class='keyword'>up</span>/
<span class='keyword'>down</span>/<span class='keyword'>left</span>/<span
class='keyword'>right</span> commands are optional.

wrong:On the final position of the <span class='object'>/bird/</span>.:Actually, if
you look at my code carefully, I never interact with the <span class='object'>/
bird/</span> at all!.
wrong:On the final position of the <span class='object'>/goop/</span>.:I'll
actually drop the <span class='object'>/goop/</span> off the <span class='object'>/
bucket/</span>, and move several more steps after that.
```

Level 7. Where Will the Goop End Up?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]
object bird(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]

object goop(row, column)
 set this:position to [row, column]

object bucket(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

create puppy(5, 3)
create goop(4, 2)
create kitten(1, 5)
create piglet(3, 0)
create bird(1, 2)
create bucket(1, 0)
create basket(4, 4)
```

Mission Text:

- Hmm… I'm thinking I'm getting the hang of this because of your help!
- I made some temporary adjustments to my logic chip and I want to try this level out myself in one shot!
- Can you help me determine where the /goop/ will end up after running the code? We only have one shot at this so let's try our best! Click on the tile to choose your answer!

Level 7. Where Will the Goop End Up?

Gidget Code (assessment):

```
up 1
right
grab /kitten/
left 3
grab /bird/
left 2
down 2
grab /piglet/
right 2
down 1
grab /goop/
right
down
grab /puppy/
up 4
left 3
down 3
right 4
```

Assessment Question:

Where will the /goop/ be after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

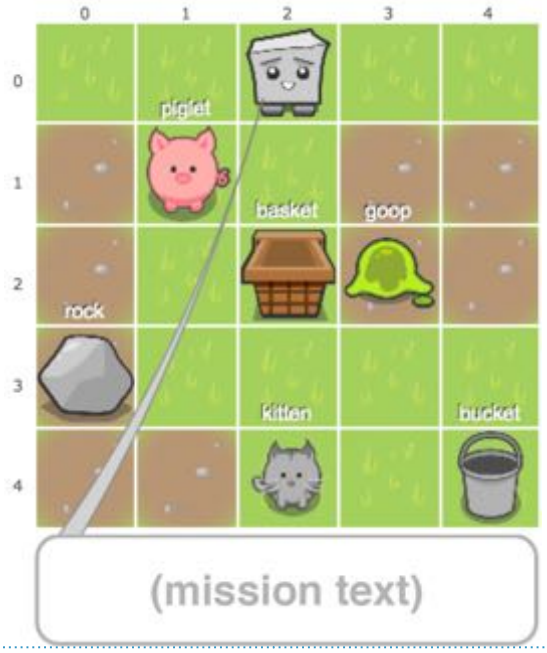
```
wrong:: I won't end up on this space.
wrong:1,0:I stop on the <span class='object'>/bucket/</span>, but I never dropped anything
on it.
wrong:1,2:This is the original position of the <span class='object'>/bird/</span>, but I
picked it up and moved it!
wrong:1,5:This is the original position of the <span class='object'>/kitten/</span>, but I
picked it up and moved it!
wrong:3,0:This is the original position of the <span class='object'>/piglet/</span>, but I
picked it up and moved it!
wrong:4,2:This is the original position of the <span class='object'>/goop/</span>, but I
picked it up and moved it!
wrong:5,3:This is the original position of the <span class='object'>/puppy/</span>, but I
picked it up and moved it!

correct:4,4:I picked up all the animals and the <span class='object'>/goop/</span>, but
never dropped anything. So I'll finally end up at the <span class='object'>/basket/</span>
with everything, including the <span class='object'>/goop/</span>!
```

Level 8. Let's move more efficiently to each object!

World Code:

```
object kitten(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]
object goop(row, column)
 set this:position to [row, column]
object rock(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]
object bucket(row, column)
 set this:position to [row, column]
create basket(2,2)
create bucket(4,4)
create kitten(4, 2)
create piglet(1, 1)
create goop(2, 3)
create rock(3, 0)
```



Mission Text:

- Oh no! The /goop/s are starting to make the /grass/ sick. I should hurry to the /goop/ factory before it spreads even more!
- I just remembered that using the goto command should be very helpful moving to objects.
- Try running my starting code to see how goto works! My goal is to get everything into the correct containers, but remember that I use more energy as I carry more things.

Level 8. Let's move more efficiently to each object!

Gidget Code (broken):

```
goto /goop/
grab /goop/
left 4
down
drop /kitten/
goto /basket/
left
up 2
drop /rock/
drop /goop/
```

Gidget Code (solution):

```
goto /goop/
grab /goop/
goto /rock/
grab /rock/
goto /bucket/
drop
goto /kitten/
grab /kitten/
goto /piglet/
grab /piglet/
goto /basket/
drop
```

Gidget Goals:

```
ensure /kitten/:position = /basket/:position
ensure /piglet/:position = /basket/:position
ensure /goop/:position = /bucket/:position
ensure /rock/:position = /bucket/:position
```

Level 9. Let's move around more efficiently to specific spots on the map!

World Code:

```
object bird(row, column)
 set this:position to [row, column]
object dog(row, column)
 set this:position to [row, column]
object goop(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]
object bucket(row, column)
 set this:position to [row, column]
object rock(row, column)
 set this:position to [row, column]
create basket(5,5)
create bucket(1,3)
create bird(3, 4)
create dog(4, 1)
create goop(3, 2)
create rock(1, 5)
```

Mission Text:

- Great! Using goto is helpful, but I'll still need to use the up/down/left/right commands to move to specific spaces!
- Make sure to check my goals before starting the level. It looks like I have to move certain things to specific spots before running out of energy!

Level 9. Let's move around more efficiently to specific spots on the map!

Gidget Code (broken):

```
goto /bird/
grab /bird/
up 3
left 2
goto /dog/
grab /goop/
left 4
down
drop /goop/
goto /basket/
left
up 2
drop /dog/
drop /rock/
```

Gidget Code (solution):

```
goto /basket/
grab /basket/
up 4
drop /basket/
goto /bucket/
grab /bucket/
right 2
down 2
drop /bucket/
goto /bird/
grab /bird/
goto /dog/
grab /dog/
goto /basket/
drop
goto /goop/
grab /goop/
goto /rock/
grab /rock/
goto /bucket/
drop
```

Gidget Goals:

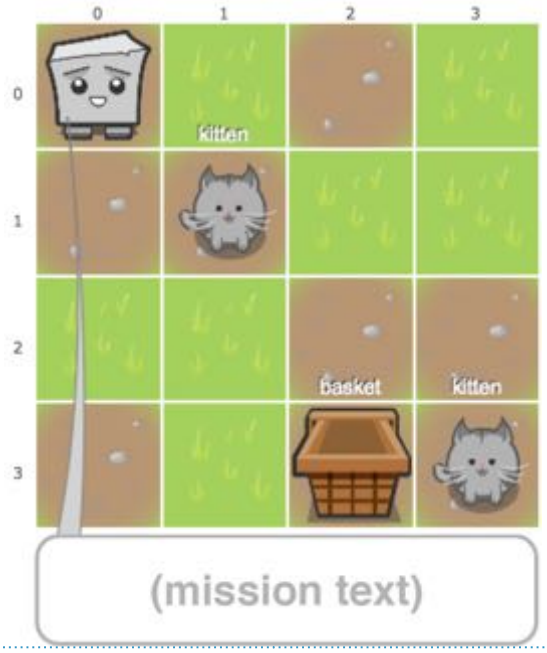
```
ensure /basket/:position = [1,5]
ensure /bucket/:position = [3,5]
ensure /dog/:position = /basket/:position
ensure /bird/:position = /basket/:position
ensure /goop/:position = /bucket/:position
ensure /rock/:position = /bucket/:position
```

Level 10. Let's save the kitten twins!

World Code:

```
object basket(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]

create kitten(1, 1)
create kitten(3, 3)
create basket(3, 2)
```



Mission Text:

- Oh, there are two /kitten/s in this level! I get really confused when I have to interact with things with the same name.
- To solve this problem, I'll need to add an "s" to the end of the objects' name to make a list. Then I can use the same commands, with first and last.

Level 10. Let's save the kitten twins!

Gidget Code (broken):

```
goto first /kitten/s
grab first /kitten/
goto last /basket/s
grab last /kitten/s
goto first /basket/s
```

Gidget Code (solution):

```
goto first /kitten/s
grab first /kitten/s
goto last /kitten/s
grab last /kitten/s
goto /basket/
drop
```

Gidget Goals:

```
ensure # /kitten/s on /basket/ = 2
```

Level 11. Let's organize two of a kind!

World Code:

```
object basket(row, column)
 set this:position to [row, column]
object cat(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]
object goop(row, column)
 set this:position to [row, column]
object bucket(row, column)
 set this:position to [row, column]

create cat(1, 4)
create cat(3, 3)
create basket(4, 2)
create goop(2, 1)
create goop(3, 2)
create bucket(1, 0)
create kitten(1, 2)
```

Mission Text:

- Ok, it looks like I'll have to move many things to their proper places for this level.
- Remember, to use lists, I have to add an "s" to an object's name, and use first and last to access specific list items. After I'm done, move me to the corner /cobblestone/ tile!

Level 11. Let's organize two of a kind!

Gidget Code (broken):

```
goto first /kitten/s
grab /kitten/
goto basket
drop /cat/
goto first /cat/s
grab first /cats/
goto last /cat/s
goto first /goop/s
goto last /goop/s
grab last /goop/s
goto /bucket/
drop last /goop/s
right 2
down 2
```

Gidget Code (solution):

```
goto /kitten/
grab /kitten/
goto first /cat/s
grab first /cat/s
goto last /cat/s
grab last /cat/s
goto /basket/
drop
goto first /goop/s
grab first /goop/s
goto last /goop/s
grab last /goop/s
goto /bucket/
drop
right 4
down 3
```

Gidget Goals:

```
ensure gidget:position = [4,4]
ensure # /cat/s on /basket/ = 2
ensure # /kitten/s on /basket/ = 1
ensure # /goop/s on /bucket/ = 2
```

Level 12. Where Will the Kittens End Up?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]
object bird(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]

object goop(row, column)
 set this:position to [row, column]

object bucket(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

create puppy(5, 2)
create goop(4, 3)
create goop(5, 0)
create kitten(1, 0)
create kitten(5, 3)
create piglet(3, 5)
create piglet(3, 1)
create bird(1, 3)
create bucket(1, 5)
create basket(4, 1)
```

Mission Text:

- I made some temporary adjustments to my logic chip and I want to try this level out myself in one shot!
- Can you help me determine where the /kitten/s will end up after running the code? We only have one shot at this so let's try our best! Click on the tile to choose your answer!

Level 12. Where Will the Kittens End Up?

Gidget Code (assessment):

```
goto /bird/
grab /bird/
goto first /kitten/s
grab first /kitten/s
goto last /kitten/s
grab last /kitten/s
goto /puppy/
grab /puppy/
goto /basket/
drop /bird/
drop /puppy/
goto first /goop/s
grab first /goop/s
goto /bucket/
drop first /goop/s
```

Assessment Question:

Where will the /kitten/s be after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

```
wrong:: I won't end up on this space.
wrong:3,1:I never interact with the <span class='object'>/piglet/</span> on this
space.
wrong:3,5:I never interact with the <span class='object'>/piglet/</span> on this
space.
wrong:4,3:I never interact with the <span class='object'>/goop/</span> on this
space.
wrong:4,1:I only moved the <span class='object'>/bird/</span> and <span
class='object'>/puppy/</span> here.

correct:1,5:I picked up all the animals except the <span class='object'>/piglet/s</
span>, but only dropped off the single animals at the <span class='object'>/basket/
</span>. I brought the <span class='object'>/kitten/s</span> and the single <span
class='object'>/goop/</span> to the <span class='object'>/bucket/</span>!
```

Level 13. Where Will the Birds End Up?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]
object bird(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]

object goop(row, column)
 set this:position to [row, column]

object bucket(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

create puppy(5, 2)
create goop(4, 3)
create goop(5, 0)
create kitten(1, 0)
create kitten(5, 3)
create piglet(3, 5)
create piglet(3, 1)
create bird(1, 3)
create bucket(1, 5)
create basket(4, 1)
```

Mission Text:

- Okay, I think I'm getting the hang of this. I want to try most of this by myself. Can you just help me by verifying what will happen by choosing from the options on the right?

Level 13. Where Will the Birds End Up?

Gidget Code (assessment):

```
goto first /puppy/s
grab first /puppy/s
goto last /puppy/s
grab last /puppy/s
goto first /goop/s
grab first /goop/s
goto last /goop/s
grab last /goop/s
goto first /bird/s
grab first /birds/
goto last /bird/s
grab last /birds/
goto /basket/
drop first /puppy/s
drop last /puppy/s
drop first /bird/s
drop last /bird/s
goto /bucket/
drop first /goop/s
drop last /goop/s
```

Assessment Question:

After running the code (assuming I have unlimited energy), the two /bird/s will eventually end up:

- On their original positions.
- On the /basket/.
- On the /bucket/.
- On the /cobblestone/ tile at [0,4].

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

```
wrong:: I won't end up on this space.
wrong:3,1:I never interact with the <span class='object'>/piglet/</span> on this
space.
wrong:3,5:I never interact with the <span class='object'>/piglet/</span> on this
space.
wrong:4,3:I never interact with the <span class='object'>/goop/</span> on this
space.
wrong:4,1:I only moved the <span class='object'>/bird/</span> and <span
class='object'>/puppy/</span> here.

correct:1,5:I picked up all the animals except the <span class='object'>/piglet/s</
span>, but only dropped off the single animals at the <span class='object'>/basket/
</span>. I brought the <span class='object'>/kitten/s</span> and the single <span
class='object'>/goop/</span> to the <span class='object'>/bucket/</span>!
```

Level 14. The dog will help us!

World Code:

```
object boulder(row, column)
 set this:position to [row, column]
 set this:layer to 2
 set this:labeled to false
object goop(row, column)
 set this:position to [row, column]
object dog(row, column)
 say "Ask me for help!"
 set this:saidHello to false
 set this:position to [row, column]
 when (sayThis = "Please help me
Dog!") and (this:saidHello = false)
  say "I'm here to help!"
  set this:saidHello to true
  if not (/goop/:position = /
bucket/:position)
   goto /goop/
   grab /goop/
   goto /bucket/
   drop /goop/
   goto /basket/

object bucket(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]

create boulder(4,0)
create boulder(4,1)
create boulder(4,2)
create boulder(4,3)
create boulder(4,4)
create boulder(4,5)
create basket(5,1)
create dog(5,0)
create goop(5,2)
create bucket(5,5)
```

Mission Text:

- Hmm, the /goop/s are even in this mountainous area! I'm getting closer to the factory, but I can't move /boulder/s, so it'll be difficult to get to the /goop/s now.
- It looks like one of my goals is to ask the /dog/ for help. Maybe I can ask for help by using a variable, which is used to store data for use later. I can set variables using the set command followed by any name I want.
- I can use say followed by my variable name to display what I have stored in it. Let's try my code and ask the /dog/ for help! Check my goals first because what I store in the variable has to be exactly the same!

Level 14. The dog will help us!

Gidget Code (broken):

```
goto /dog/
set numBoulders to 6
say numBoulders
set sayThis to "Please Haelp me dogg"
say sayThis
```

Gidget Code (solution):

```
set sayThis to "Please help me Dog!"
```

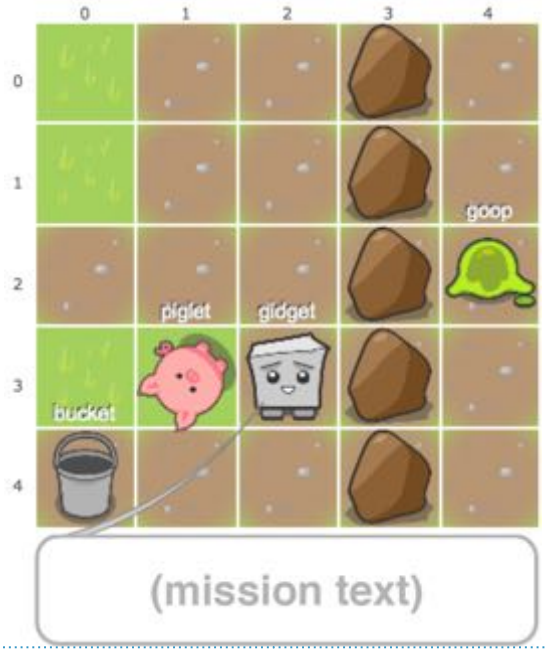
Gidget Goals:

```
ensure sayThis = "Please help me Dog!"
ensure /goop/:position = /bucket/:position
ensure /dog/:position = /basket/:position
```

Level 15. The piglet will help us!

World Code:

```
object bucket(row, column)
 set this:position to [row, column]
object boulder(row, column)
 set this:position to [row, column]
 set this:layer to 2
 set this:labeled to false
object goop(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]
 set this:rotation to -135
 set this:saidThanks to false
 say "Please help me! I can't get up!"
 when this:rotation = 0 and
this:saidThanks = false
  set this:saidThanks to true
  say "Thanks for helping me! I'll remove
a boulder for you!"
  set boulder to /boulder/s:first()
  if not (boulder = nothing)
   set boulder:layer to 1
   goto boulder
   remove boulder
   set /piglet/:scale to 1.5
   left 2
   up 2
   say "Yum! You can go through now!"
create goop(2, 4)
create piglet(3, 1)
create bucket(4, 0)
create boulder(0,3)
create boulder(1,3)
create boulder(2,3)
create boulder(3,3)
create boulder(4,3)
```



Mission Text:

- Other objects (and even me!) have variables too! Their names and values are displayed to the right for me, and for any other object when you click on it.
- Some variables are reserved, which means we can't modify them. But we can change 'rotation,' 'scale,' and 'transparency', or even add our own!
- Since we can't get past these /boulder/s, maybe this /piglet/ can help us after we turn it around! After it removes the /boulder/s, we can clean up the /goop/!
- Let's help the /piglet/ get back up! Modify its rotation so that it's back to normal (check my rotation value or the goals for an example).

Level 15. The piglet will help us!

Gidget Code (broken):

```
set /piglet/:scale to 0.6
goto /goop/
grab /goop/
goto /bucket/
drop /goop/
```

Gidget Code (solution):

```
set /piglet/:rotation to 0
goto /goop/
grab /goop/
goto /bucket/
drop /goop/
```

Gidget Goals:

```
ensure /piglet/:rotation = 0
ensure # /goop/s on /bucket/ = 1
```

Level 16. Read the note, pass the rat!

World Code:

```
object boulder(row, column)
 set this:position to [row, column]
 set this:layer to 2
 set this:labeled to false
object goop(row, column)
 set this:position to [row, column]
object bucket(row, column)
 set this:position to [row, column]
object note(row, column)
 set this:position to [row, column]
 set this:gaveInfo to false
 when /gidget/:position = this:position and
this:gaveInfo = false
  say "Use an array!
Arrays are lists of values, separated by a
comma. The rat wants you to set the password
variable to: [/goop/s, 2, /bucket/]"
  set this:gaveInfo to true
object rat(row, column)
 set this:position to [row, column]
 set this:gaveInfo to false
 when /gidget/:position = this:position and
this:gaveInfo = false
  set this:gaveInfo to true
  if not password = nothing and password =
[/goop/s, 2, /bucket/]
   say "Grr.. how did you you figure out how
to use arrays!?

I'll let you pass this time!"
   up 3
   left 2
  else
   say "You cannot pass without the
password! I'm going to take all your
energy!"
   set /gidget/:energy to 0
create goop(4,5)
create goop(1,6)
create rat(3,2)
create bucket(6,1)
create note(1,4)
create boulder(3,0)
create boulder(3,1)
create boulder(3,3)
create boulder(2,4)
create boulder(1,5)
create boulder(0,6)
```

Mission Text:

- Oh no, that mean looking /rat/ won't let me pass without a password with multiple values!
- I remember that there is a special kind of variable called an array, which can hold multiple values. For example, my position (which you can see on the right), is an array of two number literals.
- For arrays, we start counting from "0" and put square brackets around it. For example, my position is [1,2]. So, /gidget/:position[0] would be 1.
- I'm pretty sure that my starting code is correct, except for the password! Maybe going to that /note/ over there might give us a clue!

Level 16. Read the note, pass the rat!

Gidget Code (broken):

```
set password to ["first", "second", "third"]
say password[0]
goto first password[0]
grab first password[0]
say password[1]
goto last password[0]
grab last password[0]
say password[2]
goto password[2]
drop
```

Gidget Code (solution):

```
goto /note/
set password to [/goop/s, 2, /bucket/]
goto first /goop/s
grab first /goop/s
goto last /goop/s
grab last /goop/s
goto /bucket/
drop
```

Gidget Goals:

```
ensure password = [/goop/s, 2, /bucket/]
ensure # /goop/s on /bucket/ = 2
```

Level 17. The dog will help us here!

World Code:

```
object chasm(row, col)
 set this:position to [row, col]
 set this:labeled to false
 when /gidget/:position = this:position
  set /gidget/:scale to 0
  set /gidget/:energy to 0
  say "Oh no Gidget, you fell into me and
lost all your energy!"
object rock(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]
object bird(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]
object dog(row, column)
 set this:position to [row, column]
 set this:movedRock to false
 set this:movedBird to false
 set this:movedPiglet to false
 set this:movedChasm to false
 when moveThis = [5,0] and this:movedRock =
false
  say "Sorry Gidget, I can't reach the
rock!"
  set this:movedRock to true
 when moveThis = [3,4] and this:movedPiglet
= false
  say "Okay Gidget, I'll move the piglet to
the basket for you!"
  set this:movedPiglet to true
  goto /piglet/
  grab /piglet/
  goto /basket/
  drop
 when moveThis = [2,6] and this:movedBird =
false
  say "Okay Gidget, I'll move the bird to
the basket for you!"
  set this:movedBird to true
  goto /bird/
  grab /bird/
  goto /basket/
  drop
 when moveThis[1] = 2 and this:movedChasm =
false
  set this:movedChasm to true
  say "Don't be silly Gidget, I can't move
the chasm!"
create rock(5,0)
create basket(6,6)
create piglet(3,4)
create bird(2,6)
create chasm(0,2)
create chasm(1,2)
create chasm(2,2)
create chasm(3,2)
create chasm(4,2)
create chasm(5,2)
create chasm(6,2)
create dog(1,3)
```

Mission Text:

- Oh no, there appears to be a /chasm/ here. I can't cross it and will have to find another way around, but I should help these animals now!
- I'm detecting that the /dog/ over there can help us! The /dog/ will listen for changes in the moveThis array.
- Coordinates are actually just an array of two number literal values. The first number is the row and the second number is the column.
- Once we assign moveThis with the correct coordinates, the /dog/ will move any object there for us! We can keep reassigning moveThis with new values until we're done!

Level 17. The dog will help us here!

Gidget Code (broken):

```
goto /rock/
say "I'm going to ask this helpful dog to
move things to the bucket by saying a
positions array."
set moveThis to [5,0]
say "Thanks for moving the thing at " +
moveThis + " to the basket!"
set moveThis to [1,2]
say "Thanks for moving the thing at " +
moveThis + " to the basket!"
up 2
```

Gidget Code (solution):

```
set moveThis to [2,6]
set moveThis to [3,4]
```

Gidget Goals:

```
ensure moveThis = [2,6] or moveThis = [3,4]
ensure /bird/:position = /basket/:position
ensure /piglet/:position = /basket/:position
```

Level 18. What's the array's value?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]

object goop(row, column)
 set this:position to [row, column]

object bucket(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

object bird(row, column)
 set this:position to [row, column]

create basket(3,1)
create bucket(1,3)
create goop(1,0)
create puppy(4,4)
create bird(3,2)
```

Mission Text:

- Alright, I still get confused with my code sometimes, but I'm getting much better thanks to you!
- I want to try most of this by myself. Can you just help me by verifying what will happen after we run the program by choosing from the options on the right?

Level 18. What's the array's value?

Gidget Code (assessment):

```
set a to /bird/:position
set b to /bucket/:position
set c to /goop/:position
set myArray to [a, b, c]
goto /goop/
grab /goop/
goto /bucket/
drop /goop/
goto /bird/
grab /bird/
up
```

Assessment Question:

I'm going to try remembering the objects' positions. After running the code (assuming I have unlimited energy), the variable "myArray" will be equal to:

- [[3,2], [1,3], [1,0]]
- [[2,2], [1,3], [1,3]]
- [[3,2], [1,3], [1,3]]
- [[1,0], [1,3], [3,2]]
- nothing, because there is an error in the code.

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

```
wrong:[[2,2], [1,3], [1,3]]:A <span class='dictionaryTerm'>variable</span>'s value won't
change unless it's directly modified. So moving the <span class='object'>/goop/</span> to
the <span class='object'>/bucket/</span> and the <span class='object'>/bird/</span> up
won't change the values you stored earlier.
wrong:[[3,2], [1,3], [1,3]]:A <span class='dictionaryTerm'>variable</span>'s value won't
change unless it's directly modified. So moving the <span class='object'>/goop/</span> to
the <span class='object'>/bucket/</span> won't change the value you stored earlier.
wrong:[[1,0], [1,3], [3,2]]:When you assign an <span class='dictionaryTerm'>array</span>
with values, it puts them in the same order you put them in. We put the <span
class='dictionaryTerm'>variable</span> values in order, [a, b, c].
wrong:nothing, because there is an error in the code.:There were no errors. Remember that
we can set <span class='dictionaryTerm'>variable</span>s to other <span
class='dictionaryTerm'>variable</span>'s values at a given time, and that <span
class='keyword'>up</span> can be used without a number.

correct:[[3,2], [1,3], [1,0]]:<span class='dictionaryTerm'>Variable</span>'s values won't
change unless they're directly modified. Moving objects won't affect the values you stored
earlier. So, myArray's value is the same as when we started, [a, b, c].
```

Level 19. What Will Gidget Say?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]
object dog(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]
object cat(row, column)
 set this:position to [row, column]

create basket(4,2)

create dog(3,1)
create puppy(1,0)
create kitten(1,3)
create cat(3,3)
```

Mission Text:

- Alright, I'm getting the hang of this! I want to try most of this by myself. Can you just help me by verifying what will happen by choosing from the options on the right?

Level 19. What Will Gidget Say?

Gidget Code (assessment):

```
set myArray to [/puppy/, /dog/, /cat/]
goto myArray[1]
set myArray[1] to /kitten/
goto myArray[1]
grab myArray[1]
goto /basket/
drop myArray[1]
say "The " + myArray[2] + " isn't in the
basket yet."
```

Assessment Question:

I'm going to try remembering the objects' positions. After running the code (assuming I have unlimited energy), what will my final "say" be at the end of the code output?:

- "The /cat/ isn't in the /basket/ yet."
- "The /puppy/ isn't in the /basket/ yet."
- "The /dog/ isn't in the /basket/ yet."
- "The /kitten/ isn't in the /basket/ yet."

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

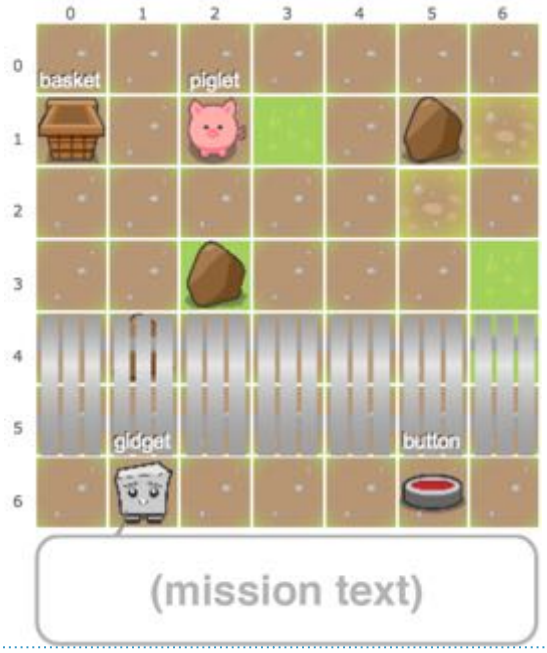
```
wrong:"The <span class='object'>/puppy/</span> isn't in the <span class='object'>/basket/</
span> yet.":The value for myArray[0], which was <em><span class='object'>/puppy/</span></
em> never changed.
wrong:"The <span class='object'>/dog/</span> isn't in the <span class='object'>/basket/</
span> yet.":<em><span class='object'>/dog/</span></em> was replaced with <em><span
class='object'>/kitten/</span></em> while running the code because myArray[1] refers to the
middle (2nd) value.
wrong:"The <span class='object'>/kitten/</span> isn't in the <span class='object'>/basket/
</span> yet.":Since <span class='dictionaryTerm'>array</span>s start at [0], <em><span
class='object'>/kitten/</span></em> is stored in the second value of the <span
class='dictionaryTerm'>array</span>, which is myArray[1].

correct:"The <span class='object'>/cat/</span> isn't in the <span class='object'>/basket/</
span> yet.":Since <span class='dictionaryTerm'>array</span>s start at [0], only the middle
value, [1], was changed from <em><span class='object'>/dog/</span></em> to <em><span
class='object'>/kitten/</span></em>. So myArray[2] is <em><span class='object'>/cat/</
span></em>.
```

Level 20. Press the button, open the gate!

World Code:

```
object boulder(row, column)
 set this:position to [row, column]
 set this:layer to 2
 set this:labeled to false
object piglet(row, column)
 set this:position to [row, column]
 function oink()
  say "Oink oink!"
object basket(row, column)
 set this:position to [row, column]
object fence(row, column)
 set this:position to [row, column]
 set this:layer to 2
 set this:labeled to false
object gate(row, column)
 set this:position to [row, column]
 set this:layer to 2
 set this:labeled to false
object button(row, column)
 set this:position to [row, column]
 function openGate()
  set /gate/:layer to 1
  set /gate/:transparency to 0
create boulder(4,1)
create boulder(3,2)
create boulder(1,5)
create piglet(1,2)
create basket(1,0)
create button(6,5)
create fence(5,0)
create fence(5,1)
create fence(5,2)
create gate(5,3)
create fence(5,4)
create fence(5,5)
create fence(5,6)
```



Mission Text:

- Oh no, there's almost no /grass/ here. We must be getting closer to the leaking /goop/ factory!
- All that grabbing & dropping made me remember a way to save time writing my programs though... functions!
- Objects can have built-in functions that you can see labeled when you click on them, or you can write your own! Objects' functions will have () at the end of their names.
- You can call these functions by stating the object's name, using a colon, then writing the function name with parentheses like this: /button/:openGate()
- Check out the example I gave you and you'll notice code belonging to a function is grouped together with indents.
- Let's figure out how to open the /gate/ with the /button/, and give that /piglet/ some new properties before we put it in the /basket/!
- Don't forget you can click on objects to see their properties, and you should try running my code first to see what happens!

Level 20. Press the button, open the gate!

Gidget Code (broken):

```
goto /button/
say "Let's click the button to see its
function name. It has to be exact!"
/button/:openFence()
function getPiglet()
 goto /piglet/
 set /piglet/:nickname to "wilbur"
 set /piglet/:age to 3
 grab /piglet/
getBird()
getThePiggy()
goto /basket/
```

Gidget Code (solution):

```
goto /button/
/button/:openGate()
function getPiglet()
 goto /piglet/
 set /piglet/:nickname to "babe"
 set /piglet/:age to 3
 grab /piglet/
 goto /basket/
 drop /piglet/
getPiglet()
```

Gidget Goals:

```
ensure /piglet/:nickname = "babe"
ensure /piglet/:age = 3
ensure # /piglet/ on /basket/ = 1
```

Level 21. Flip the animals right-side-up!

World Code:

```
object kitten(row, column, angle)
 set this:saidThanks to false
 set this:position to [row, column]
 set this:rotation to angle
 when this:rotation = 0 and
this:saidThanks = false
  say "Thanks, mewwww!"
  set this:saidThanks to true
object bird(row, column, angle)
 set this:saidThanks to false
 set this:position to [row, column]
 set this:rotation to angle
 when this:rotation = 0 and
this:saidThanks = false
  say "Thanks, chirp chirp!"
  set this:saidThanks to true
object piglet(row, column, angle)
 set this:saidThanks to false
 set this:position to [row, column]
 set this:rotation to angle
 when this:rotation = 0 and
this:saidThanks = false
  say "Thanks, oink oink!"
  set this:saidThanks to true
object dog(row, column, angle)
 set this:saidThanks to false
 set this:position to [row, column]
 set this:rotation to angle
 when this:rotation = 0 and
this:saidThanks = false
  say "Thanks, bow wow!"
  set this:saidThanks to true
object goop(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]
object bucket(row, column)
 set this:position to [row, column]
create basket(5,0)
create bucket(5,5)
create goop(3,3)
create kitten(4,2, -55)
create bird(3,5, 75)
create piglet(1,4, -120)
create dog(1,1, -85)
```

Mission Text:

- Oh, what's happening here?! It looks like the animals are dizzy from the /goop/! We need to help them!
- We can use functions here to take care of the tedious task more efficiently.
- Functions can be more useful by passing values inside their parentheses! Functions can also return values back to whatever called it.
- The goal for this level is to get all the animals right-side-up, and move everything to their respective containers.
- You can use keep using the same function over-and-over! Try running my code to see what happens!

Level 21. Flip the animals right-side-up!

Gidget Code (broken):

```
function transferThing(whichThing,
toWhere)
 goto whichThing
 grab whichThing
 set whichThing:scale to 1.4
 say whatIDid(whichThing)
 goto toWhere
 drop
function whatIDid(item)
 set sentence to "I grabbed the " + item
+  "!"
 return sentence
transferThing(/goop/, /bucket/)
transferThing()
transferObject(/bird/, /bucket/)
transferThing(/kitten/)
```

Gidget Code (solution):

```
function transferThing(whichThing,
toWhere)
 goto whichThing
 grab whichThing
 set whichThing:rotation to 0
 say whatIDid(whichThing)
 goto toWhere
 drop
function whatIDid(item)
 set sentence to "I grabbed the " + item
+  "!"
 return sentence
transferThing(/goop/, /bucket/)
transferThing(/bird/, /basket/)
transferThing(/dog/, /basket/)
transferThing(/piglet/, /basket/)
transferThing(/kitten/, /basket/)
```

Gidget Goals:

```
ensure /goop/:position = /bucket/:position
ensure /dog/:rotation = 0 and /dog/:position = /basket/:position
ensure /bird/:rotation = 0 and /bird/:position = /basket/:position
ensure /kitten/:rotation = 0 and /kitten/:position = /basket/:position
ensure /piglet/:rotation = 0 and /piglet/:position = /basket/:position
```

Level 22. Let's plug the pipe with rocks!

World Code:

```
object kitten(row, column)
 set this:position to [row, column]
object bird(row, column)
 set this:position to [row, column]
object puppy(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]
object goop(row, column)
 set this:position to [row, column]
object pipe(row,col)
 set this:position to [row,col]
 set this:labeled to false
 set this:layer to 2
create goop(4,3)
create goop(5,3)
create pipe(4,2)
create pipe(5,2)
create bird(2, 5)
create puppy(2, 5)
create kitten(2, 5)
create basket(2, 5)
```

Mission Text:

- Oh no! Those pipes are oozing /goop/s, but I don't have a /bucket/ to store them, or a way to plug the pipes!
- Since I don't have a /bucket/, let's use the remove command to get rid of all the /goop/s from this level!
- I can also create /rock/s on each of the /cobblestone/ spots to plug the pipes! I can create new objects by declaring them like functions (but using the object command).

Level 22. Let's plug the pipe with rocks!

Gidget Code (broken):

```
object rock(saySomething)
 say saySomething
goto /goop/
remove /goop/
up 3
create rock("hello! i'm a rock!")
grab /rock/
```

Gidget Code (solution):

```
object rock(saySomething)
 say saySomething
goto first /goop/s
remove first /goop/s
create rock("My first rock!")
goto last /goop/s
remove last /goop/s
create rock("My second rock!")
```

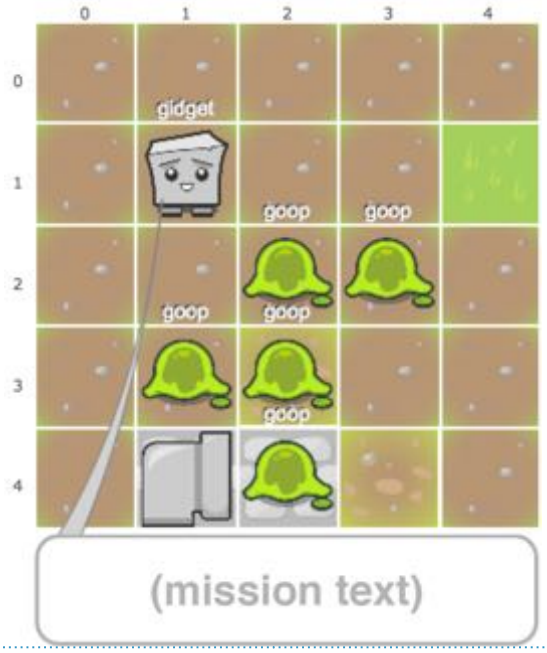
Gidget Goals:

```
ensure /rock/s:first():position = [4,3] or /rock/s:first():position = [5,3]
ensure /rock/s:last():position = [4,3] or /rock/s:last():position = [5,3]
ensure /goop/ = nothing
```

Level 23. Clean up the goop, plug the pipe, and plant the sapling!

World Code:

```
object cat(row, column)
 set this:position to [row, column]
object goop(row, column)
 set this:position to [row, column]
object pipe(row,col)
 set this:position to [row,col]
 set this:labeled to false
 set this:layer to 2
create pipe(4,1)
create goop(4,2)
create goop(3,1)
create goop(3,2)
create goop(2,3)
create goop(2,2)
```



Mission Text:

- Oh no! It looks like my /rock/s couldn't hold the /goop/ back and one of the pipes spilled again.
- We should set the variable "mass" of the /rock/ to be "heavy" so that it doesn't move again. We can do this by using the this command to the object declaration like I have in my starting example code.
- The goals for this level are to put a /rock/ in the pipe again, remove all the /goop/s, and create a /sapling/ to take each of their places on any of the ground spots.

Level 23. Clean up the goop, plug the pipe, and plant the sapling!

Gidget Code (broken):

```
object puppy(sayThis)
 say sayThis
 set this:age to 5
object kitten()
 say "meow"
 set this:fluffiness to 50
function removeGoop()
 goto /goop/
 remove /goop/
 create kitten()
removeGoop()
up
create puppy("woof woof")
removeGoop()
right
```

Gidget Code (solution):

```
object rock()
 set this:mass to "heavy"
object sapling()
function removeGoop()
 goto /goop/
 remove /goop/
 create sapling()
removeGoop()
removeGoop()
removeGoop()
removeGoop()
removeGoop()
create rock()
```

Gidget Goals:

```
ensure /goop/ = nothing
ensure # /sapling/s = 5
ensure /rock/:position = [4, 2] and /rock/:mass = "heavy"
```

Level 24. Where Will The Cat End Up?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]
object bird(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]

object goop(row, column)
 set this:position to [row, column]

object bucket(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

create puppy(5, 3)
create goop(4, 2)
create kitten(1, 5)
create piglet(3, 0)
create bird(1, 2)
create bucket(1, 0)
create basket(4, 4)
```

Mission Text:

- Great, functions and objects are important, so I want to try using them on my own!
- I made some temporary adjustments to my logic chip and I want to try this level out myself in one shot!
- Can you help me determine where the /cat/ will end up after running the code? We only have one shot at this so let's try our best! Click on the tile to choose your answer!

Level 24. Where Will The Cat End Up?

Gidget Code (assessment):

```
object cat()
 say "meow"

function makeCat(num)
 goto /goop/
 up num
 right num
 down num - 1
 create cat()
 left num * num
 up num
makeCat(2)
```

Assessment Question:

Where will the /cat/ I create be located after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

```
wrong::That <span class='object'>/cat/</span> won't end up on
this space.
wrong:1,0:I will end up on this space, but that <span
class='object'>/cat/</span> won't.
wrong:3,0:I already created the <span class='object'>/cat/</
span> earlier and moved away from this spot without it.

correct:3,4:I circle around, create the <span class='object'>/
cat/</span>, and continue on by myself without it. So, the
<span class='object'>/cat/</span> remains in the same spot I
made it!
```

Level 25. Where Will Gidget End Up?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]

object goop(row, column)
 set this:position to [row, column]

object bucket(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

object bird(row, column)
 set this:position to [row, column]

create basket(3, 1)
create bucket(1, 3)
create goop(1, 0)
create puppy(4, 4)
create bird(3, 2)
```

Mission Text:

- Okay, I should get a little more practice using functions and objects and I want to try most of this by myself. Can you just help me by verifying what will happen by choosing from the options on the right?

Level 25. Where Will Gidget End Up?

Gidget Code (assessment):

```
set x to [30, 10, 0]

object piglet(number)
 set this:weight to number

function shuffle(newNumber, myArray)
 set myArray[0] to myArray[0] + newNumber
 set myArray[1] to myArray[1] + myArray[2]
 set myArray[2] to myArray[2] * newNumber
 return myArray

set x to shuffle(x[1], x)
create piglet(x[1]+x[2])
```

Assessment Question:

After running the code (assuming I have unlimited energy), the /piglet/'s weight will be:

- The /piglet/'s weight is 10.
- The /piglet/'s weight is 30.
- The /piglet/'s weight is 20.
- There will be no /piglet/.

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

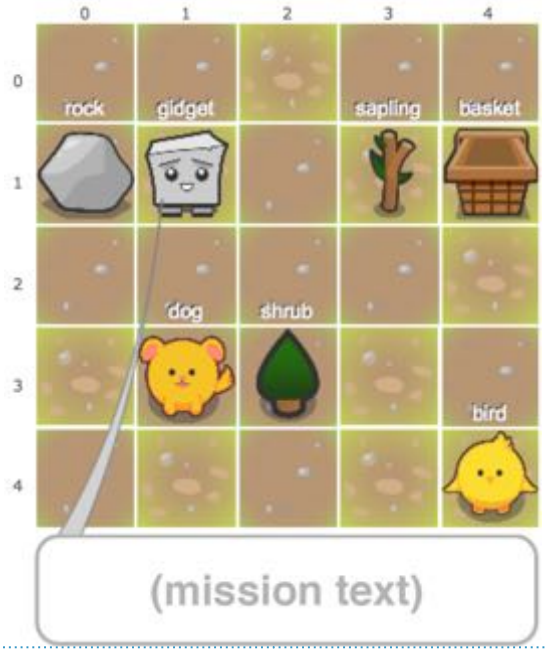
```
wrong:The <span class='object'>/piglet/</span>'s weight is <span
class='object'>30</span>.:
wrong:The <span class='object'>/piglet/</span>'s weight is <span
class='object'>20</span>.:
wrong:There will be no <span class='object'>/piglet/</span>.:I actually declared an
<span class='keyword'>object</span> named <span class='object'>/piglet/</span>, and
used <span class='keyword'>create</span> to make one, so a <span class='object'>/
piglet/</span> <em>will</em> exist.

correct:The <span class='object'>/piglet/</span>'s weight is <span
class='object'>10</span>.:The <span class='object'>/piglet/</span>'s weight depends
on <span class='object'>x[1]</span> and <span class='object'>x[2]</span>. If you
look carefully, the values of <span class='object'>x[1]</span> and <span
class='object'>x[2]</span> remain the same, so <span class='object'>/piglet/</
span>'s weight is <span class='object'>10</span><span class='object'>+</span><span
class='object'>0</span>=<span class='object'>10</span>.
```

Level 26. True or False: Are the animals infected?

World Code:

```
object basket(row, column)
 set this:position to [row, column]
object sapling(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object shrub(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object rock(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object bird(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object dog(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object dog(row, column, status)
 set this:position to [row, column]
 set this:infected to status
create basket(1,4)
create sapling(1,3,false)
create rock(1,0,true)
create bird(4,4,false)
create dog(3,1,false)
create shrub(3,2,true)
```



Mission Text:

- Alright, I'm at the inner ground of the factory now, so we're almost there! But yuck! The ground here has been infected with the /goop/s!
- Oh no, other things are getting infected by the /goop/ too! Let's put the non-infected in the /basket/ to get them out of here first!
- We can check the property of each object for boolean values (which means they are always either true or false) and an if statement, which let's us do something based on whether the boolean is true or false!

Level 26. True or False: Are the animals infected?

Gidget Code (broken):

```
function checkForInfection(objectName)
 goto objectName
 if objectName:infected = true
  grab objectName
checkForInfection(/shrub/)
checkTheDog(/dog/)
checkForInfection(/sapling/)
checkForInfection(/rock/)
checkForInfection(/bird/)
goto /basket/
```

Gidget Code (solution):

```
function checkForInfection(objectName)
 goto objectName
 if objectName:infected = false
  grab objectName
  goto /basket/
  drop objectName
checkForInfection(/sapling/)
checkForInfection(/bird/)
checkForInfection(/dog/)
checkForInfection(/shrub/)
```

Gidget Goals:

```
ensure /sapling/:infected = false and /sapling/:position = /basket/:position
ensure /bird/:infected = false and /bird/:position = /basket/:position
ensure /dog/:infected = false and /dog/:position = /basket/:position
ensure not /rock/:position = /basket/:position
ensure not /shrub/:position = /basket/:position
```

Level 27. Let's clean up some more!

World Code:

```
object basket(row, column)
 set this:position to [row, column]
object container(row, column)
 set this:position to [row, column]
object sapling(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object shrub(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object puppy(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object piglet(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object kitten(row, column, status)
 set this:position to [row, column]
 set this:infected to status
object dog(row, column, status)
 set this:position to [row, column]
 set this:infected to status
create basket(5,0)
create container(1,4)
create sapling(2,3,true)
create puppy(1,2,true)
create piglet(5,3,false)
create kitten(3,1,false)
create shrub(4,5,true)
```

Mission Text:

- Okay, now that we know we can control what I do using if statements, let's add an else statement, which is used when the if is not true.
- This will make it much easier for me to organize things!
- The goal of this level is to put the uninfected things into the /basket/, and the infected things into the /container/ (so we can disinfect them later)!

Level 27. Let's clean up some more!

Gidget Code (broken):

```
function organize(objectName)
 goto objectName
 grab objectName
 if objectName:infected = true
  goto /container/
 else
  goto /container/
 drop
organize(/sapling/)
organize(/puppy/)
organize()
organize("kitten")
organize(/shrub/)
```

Gidget Code (solution):

```
function organize(objectName)
 goto objectName
 grab objectName
 if objectName:infected = true
  goto /container/
 else
  goto /basket/
 drop objectName
organize(/puppy/)
organize(/shrub/)
organize(/sapling/)
organize(/piglet/)
organize(/kitten/)
```

Gidget Goals:

```
ensure /puppy/:infected = true and /puppy/:position = /container/:position
ensure /shrub/:infected = true and /shrub/:position = /container/:position
ensure /sapling/:infected = true and /sapling/:position = /container/:position
ensure /piglet/:infected = false and /piglet/:position = /basket/:position
ensure /kitten/:infected = false and /kitten/:position = /basket/:position
```

Level 28. Are the animals infected or sick?

World Code:

```
object basket(row, column)
 set this:position to [row, column]

object container(row, column)
 set this:position to [row, column]
object piglet(row, column, inf, sck)
 set this:position to [row, column]
 set this:infected to inf
 set this:sick to sck
object bird(row, column, inf, sck)
 set this:position to [row, column]
 set this:infected to inf
 set this:sick to sck
object shrub(row, column, inf, sck)
 set this:position to [row, column]
 set this:infected to inf
 set this:sick to sck
object dog(row, column, inf, sck)
 set this:position to [row, column]
 set this:infected to inf
 set this:sick to sck
create basket(2,0)
create container(1,0)
create piglet(1,3,true,true)
create bird(4,4,true,false)
create dog(3,1,false,true)
create shrub(3,2,false,false)
```

Mission Text:

- Okay! I'm getting good at organizing things! That's going to be very useful for my cleanup duties!
- I can make even more fine-tuned decisions by using the or and and commands between two boolean expressions.
- For an or to be true, either one (or both) of the two boolean expressions surrounding it has to be true.
- On the other hand, for an and to be true, both of the boolean expressions surrounding it must be true.
- Help me organize these animals into the correct bins. Animals that are infected and sick should go in the /container/. Animals that are infected or sick should go into the /basket/.

Level 28. Are the animals infected or sick?

Gidget Code (broken):

```
function organize(objectName)
 goto thingName
 if thingName:infected = true and
thingName:sick = true
  grab thingName
  goto /shrub/
  drop
 else
  if thingName:infected = true or
thingName:sick = true
   grab thingName
   goto /container/
   drop
organize(/shrub/)
organize(/shrub/)
organize(/piglet/)
organize(/bird/)
organize(/dog/)
```

Gidget Code (solution):

```
function organize(thingName)
 goto thingName
 if thingName:infected = true and
thingName:sick = true
  grab thingName
  goto /container/
  drop thingName
 else
  if thingName:infected = true or
thingName:sick = true
   grab thingName
   goto /basket/
   drop thingName
organize(/piglet/)
organize(/bird/)
organize(/dog/)
```

Gidget Goals:

```
ensure /piglet/:position = /container/:position
ensure /piglet/:infected = true and /piglet/:sick = true
ensure /bird/:position = /basket/:position
ensure /bird/:infected = true or /bird/:sick = true
ensure /dog/:position = /basket/:position
ensure /dog/:infected = true or /dog/:sick = true
```

Level 29. Let's open the factory gates!

World Code:

```
object shrub(row, column, size)
 set this:position to [row, column]
 set this:scale to size
object sapling(row, column, size)
 set this:position to [row, column]
 set this:scale to size
object rock(row, column, size)
 set this:position to [row, column]
 set this:scale to size
object boulder(row, column, size)
 set this:position to [row, column]
 set this:scale to size
object goop(row, column, size)
 set this:position to [row, column]
 set this:scale to size
object bird(row, column)
 set this:position to [row, column]
object crack(row, column)
 set this:position to [row, column]
 set this:checkFlag to [false,false]
 when /sapling/:position = this:position and /
sapling/:scale < 1 and this:checkFlag[0] = false
  set this:checkFlag[0] to true
  remove /sapling/
 when /rock/:position = this:position and /
rock/:scale < 1 and this:checkFlag[1] = false
  set this:checkFlag[1] to true
  remove /rock/
 when this:checkFlag = [true, true]
  remove this
object hole(row, column)
 set this:position to [row, column]
 set this:checkFlag to [false,false]
 when /goop/:position = this:position and /
goop/:scale > 1 and this:checkFlag[0] = false
  set this:checkFlag[0] to true
  remove /goop/
 when /boulder/:position = this:position and /
boulder/:scale > 1 and this:checkFlag[1] = false
  set this:checkFlag[1] to true
  remove /boulder/
 when this:checkFlag = [true, true]
  remove this
create crack(3,2)
create hole(3,6)

object fence(row, column)
 set this:position to [row, column]
 set this:labeled to false
 set this:layer to 2
object gate(row, column)
 set this:position to [row, column]
 set this:labeled to false
 set this:layer to 2
 when /hole/ = nothing and /crack/ = nothing
  remove this
create bird(0,6)
create fence(2,0)
create fence(2,1)
create fence(2,2)
create fence(2,3)
create gate(2,4)
create fence(2,5)
create fence(2,6)
create fence(2,7)
create fence(2,8)
create sapling(5,2, .9)
create rock(4,5, .95)
create boulder(7,3, 1.1)
create goop(6,1, 1.2)
```

Mission Text:

- Oh! I can see the factory! Now I just have to open the /gate/!
- I remember that mission control said that I need to fill up the spaces next to the /fence/ to get the /gate/ to open!
- We can use all the commands we've used up till now. Another useful command is not, which is used to flip a boolean expression or value.
- Let's put all the larger things (scale > 1) into the /hole/, and all the smaller things (scale < 1) into the /crack/.

Level 29. Let's open the factory gates!

Gidget Code (broken):

```
function dropSmall(thing)
 goto thing
 if not thing:scale > 1
  drop thing
  goto hole
  grab thing
function dropBig(thing)
 goto thing
 if thing > 1
  drop thing
  goto hole
  grab thing
dropSmall(/sapling/)
dropSmall(/boulder/)
dropSmall(/rock/)
dropBig(/boulder/)
dropBig(/sapling/)
dropBig(/rock/)
goto /bird/
```

Gidget Code (solution):

```
function dropThing(thing)
 goto thing
 if not thing:scale > 1
  grab thing
  goto /crack/
  drop thing
 else
  grab thing
  goto /hole/
  drop thing
dropThing(/sapling/)
dropThing(/goop/)
dropThing(/rock/)
dropThing(/boulder/)
goto /bird/
```

Gidget Goals:

```
ensure /gidget/:position = /bird/:position and /gidget/:position = [0,6]
```

Level 30. Where Will Gidget End Up?

World Code:

```
object dog(row, column)
 set this:position to [row, column]
object cat(row, column)
 set this:position to [row, column]
object bird(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]

object goop(row, column)
 set this:position to [row, column]

object bucket(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

create dog(5, 3)
create goop(2, 3)
create cat(1, 5)
create piglet(2, 0)
create bird(1, 2)
create bucket(4, 0)
create basket(3, 2)
```

Mission Text:

- Great, I'm understanding how to use booleans and if statements thanks to you!
- I'm going to calibrate my logic chip by using some booleans and need you to help me in one shot before we continue!
- Can you help me determine where I will end up after running the code? Since we only have one shot at this, I hope you get it right! Click on the tile to choose your answer!

Level 30. Where Will Gidget End Up?

Gidget Code (assessment):

```
goto /cat/
if /cat/:position = [0,5]
 goto /bird/
else
 goto /piglet/
if not /dog/:position[1] = 3
 down 2
else
 right 3
```

Assessment Question:

Where will I be after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

```
wrong:: I won't end up on this space.
wrong:1,5:The <span class='object'>/cat/</span>'s position was <span class='keyword'>not</
span> [<span class='object'>0</span>,<span class='object'>5</span>], so we went to the
<span class='object'>/piglet/</span> (instead of the <span class='object'>/bird/</span>),
then <span class='keyword'>right</span> to the <span class='object'>/goop/</span>.
wrong:3,2:The <span class='object'>/cat/</span>'s position was <span class='keyword'>not</
span> [<span class='object'>0</span>,<span class='object'>5</span>], so we went to the
<span class='object'>/piglet/</span>, then <span class='keyword'>right</span> to the <span
class='object'>/goop/</span>.
wrong:4,0:Because of the <em><span class='keyword'>not</span></em>, we moved <span
class='keyword'>right</span> <span class='object'>3</span> spaces instead of <span
class='keyword'>down</span> from the <span class='object'>/piglet/</span>.

correct:2,3:Starting at the <span class='object'>/cat/</span>, the expression is <span
class='keyword'>not</span> <span class='object'>true</span>, so I go to the <span
class='object'>/piglet/</span>, then the expression is <span class='object'>true</span>
(but we flip it because of the <em><span class='keyword'>not</span></em>), so I go <span
class='keyword'>right</span> <span class='object'>3</span> spaces.
```

Level 31. Where Will Gidget End Up?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]

object goop(row, column)
 set this:position to [row, column]

object bucket(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

object bird(row, column)
 set this:position to [row, column]

object kitten(row, column)
 set this:position to [row, column]

object piglet(row, column)
 set this:position to [row, column]

object button(row, column)
 set this:position to [row, column]

create basket(3, 0)
create kitten(3,0)
create bucket(1, 3)
create goop(1, 3)
create puppy(4, 1)
create piglet(0, 2)
create bird(3, 2)
create button(4, 3)
```

Mission Text:

- Okay, I think I'm getting the hang of this and I want to try most of this by myself. I want to make sure my understanding of booleans and if statements are correct.
- Can you just help me by verifying what will happen by choosing from the options on the right?

Level 31. Where Will Gidget End Up?

Gidget Code (assessment):

```
goto /piglet/
if /gidget/:position = /piglet/:position and
/goop/:position = /bucket/:position
 goto /bird/
else
 goto /button/

if /goop/:position = /bucket/:position or /
kitten/:position = /basket/:position
 up 3
else
 left 2
```

Assessment Question:

After running the code (assuming I have unlimited energy), I will eventually end up on the:

- /piglet/
- /puppy/
- /goop/ & /bucket/
- /kitten/ & /basket/

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

```
wrong:<span class='object'>/puppy/</span>:Since the first <span class='keyword'>if</span> statement (with an
<span class='keyword'>and</span>) is <span class='object'>true</span>, I go to the <span class='object'>/bird/
</span> and <span class='keyword'>not</span> the <span class='object'>/button/</span>.
wrong:<span class='object'>/goop/</span> &amp; <span class='object'>/bucket/</span>:Since the first <span
class='keyword'>if</span> statement (with an <span class='keyword'>and</span>) is <span class='object'>true</
span>, I go to the <span class='object'>/bird/</span> and <span class='keyword'>not</span> the <span
class='object'>/button/</span>.
wrong:<span class='object'>/kitten/</span> &amp; <span class='object'>/basket/</span>:When using an <span
class='keyword'>or</span>, only one of the <span class='dictionaryTerm'>boolean</span> statements around it can
be <span class='object'>true</span> for the <span class='keyword'>or</span> to be <span class='object'>true</
span>.

correct:<span class='object'>/piglet/</span>:The first <span class='keyword'>if</span> statement is <span
class='object'>true</span> since both <span class='dictionaryTerm'>boolean</span>s around it are <span
class='object'>true</span> for the <strong><span class='keyword'>and</span></strong>. The next <span
class='keyword'>if</span> statement is <span class='object'>false</span> since both <span
class='dictionaryTerm'>boolean</span>s around it are <span class='object'>true</span> for the <strong><span
class='keyword'>or</span></strong> (only one can be <span class='object'>true</span> for an <span
class='keyword'>or</span> to be <span class='object'>true</span>). So we go to the <span class='object'>/bird/
</span>, then <span class='object'>/piglet/</span>.
```

Level 32. Let's remove all these nasty goops!

World Code:

```
object goop(row, column, size, angle)
 set this:position to [row, column]
 set this:scale to size
 set this:rotation to angle

create goop(0,0, 1, -45)
create goop(0,2, 1, -20)
create goop(0,3, 1, 0)
create goop(0,4, 1, 10)
create goop(1,0, 1, -20)
create goop(1,1, 1, 25)
create goop(1,2, 1, -5)
create goop(1,4, 1, 100)
create goop(2,0, 1, 7)
create goop(2,2, 1, 15)
create goop(2,3, 1, 180)
create goop(2,4, 1, 30)
create goop(3,1, 1, 30)
create goop(3,3, 1, 0)
create goop(3,4, 1, 10)
create goop(4,0, 1, 50)
create goop(4,2, 1, 15)
create goop(4,4, 1, 30)
```

Mission Text:

- Wow, there are so many /goop/s piled up here at the entrance! No wonder there's so many /goop/ outside too!
- It looks like it might take a long time to clean all this up....Oh! I remember that I could use a "looping" function called a while to help me out!
- For a while, I just have to use that command, followed by a boolean expression. It will keep looping (repeating) around the tabbed code until that expression becomes false!
- The goal of this level is to remove all the /goop/s! Try out my starting code to see how it works!

Level 32. Let's remove all these nasty goops!

Gidget Code (broken):

```
while not /goop/ = nothing
 goto /goop/
 say "I'm at the goop"!
 right 2
 up 2
 left 2
 down 2
remove /gidget/
```

Gidget Code (solution):

```
while not /goop/ = nothing
 goto /goop/
 remove /goop/
```

Gidget Goals:

```
ensure /goop/ = nothing
```

Level 33. Let's gather materials and clean up the goops!

World Code:

```
function
AllMaterialScaleGreaterThanOne()
 set myList to /material/s
 for num in myList
  if num:scale < 1
   return false
 return true
object material(row, column)
 set this:position to [row, column]
 set this:scale to 0.6
object container(row, column)
 set this:position to [row, column]
object bucket(row, column)
 set this:position to [row, column]
object goop(row, column, size)
 set this:position to [row, column]
 set this:scale to size
create material(7,7)
create material(2,3)
create material(4,5)
create material(5,0)

create container(2,5)
create bucket(5,1)
create goop(0,0, 1.2)
create goop(0,3, 1)
create goop(0,6, 0.7)
create goop(3,1, 0.8)
create goop(1,2, 1)
create goop(5,2, 0.9)
create goop(1,6, 1)
create goop(2,0, 1.5)
create goop(2,4, 1.3)
create goop(6,5, 1)
create goop(2,6, 0.9)
AllMaterialScaleGreaterThanOne()
```

Mission Text:

- It's a mess here too! How can there be so many /goop/s? There must be a major leak somewhere nearby! I'll use my energy reserves to make sure I have enough to pass this level.
- The while loop in the previous level was very useful! There's another useful repeating command called for, which we can use to go through lists of objects more easily.
- It looks like there is /material/ here that I can start collecting to use for repairs using for loops.
- The goals of this level are to get the /goop/s into the /bucket/ and enlarge the /material/ before putting them into the /container/. Try running my starting code first to get an idea about how for works!

Level 33. Let's gather materials and clean up the goops!

Gidget Code (broken):

```
set materialList to /material/s
for m in materialList
 goto m
 grab m
 goto /bucket/
set myGoops to /goop/s
for g in goopList
 goto g
 set g:scale to 1.5
 grab g
goto /bucket/
```

Gidget Code (solution):

```
for m in /material/s
 goto m
 grab m
 set m:scale to 2
 goto /container/
 drop m
for g in /goop/s
 goto g
 grab g
 goto /bucket/
 drop g
```

Gidget Goals:

```
ensure # /material/s on /container/ = 4
ensure AllMaterialScaleGreaterThanOne()
ensure # /goop/s on /bucket/ = 11
```

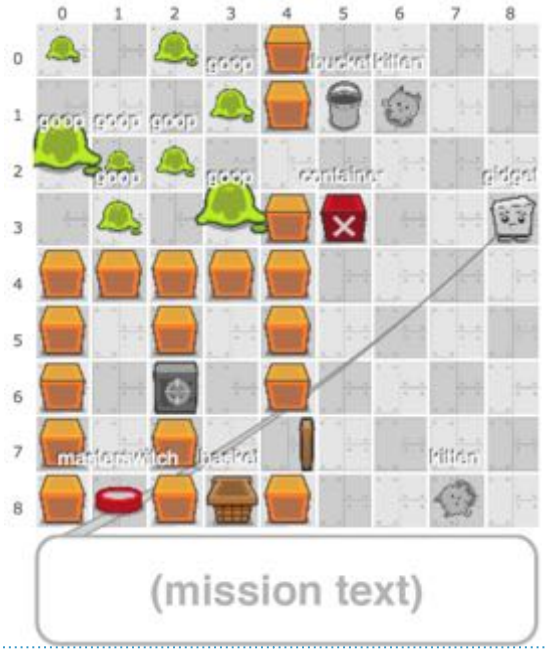
Level 34. Let's shut down the goop factory!

World Code:

```
object goop(r,c,size)
 set this:position to [r,c]
 set this:scale to size
object bucket(r,c)
 set this:position to [r,c]
object basket(r,c)
 set this:position to [r,c]
object container(r,c)
 set this:position to [r,c]

object kitten(r,c,angle)
 set this:position to [r,c]
 set this:rotation to angle

object masterswitch(r,c)
 set this:position to [r,c]
 set this:activated to false
 set this:saidComment to false
 function press()
  set this:activated to true
 when this:activated = true and this:saidComment = false
  set this:saidComment to true
  say "masterswitch has been activated! Goop factory is now
shut down!"
object block(r,c)
 set this:position to [r,c]
 set this:layer to 2
 set this:labeled to false
object door(r,c)
 set this:position to [r,c]
 set this:layer to 2
 set this:labeled to false
 when # /goop/s on /bucket/ = 2 and # /goop/s on /container/
= 6
  say "Goops have been organized. Medium contamination risk.
Safe to open control room door."
  remove this
object vaultdoor(r,c)
 set this:position to [r,c]
 set this:layer to 2
 set this:labeled to false
 when # /kitten/s on /basket/ = 2
  say "All goops have been organized. No contamination risk.
Safe to open control vault room door."
  remove this
create block(4,0)
create block(4,1)
create block(4,2)
create block(4,3)
create block(4,4)
create block(0,4)
create block(1,4)
create block(3,4)
create block(5,4)
create block(6,4)
create block(8,4)
create block(5,0)
create block(6,0)
create block(7,0)
create block(8,0)
create goop(3,3,1.5)
create goop(0,0,0.7)
create goop(0,2,0.9)
create goop(1,3,0.9)
create goop(2,1,0.6)
create goop(2,2,0.8)
create goop(3,1,0.9)
create goop(2,0,1.6)
create block(5,2)
create door(7,4)
create vaultdoor(6,2)
create block(7,2)
create block(8,2)
create kitten(1,6,25)
create kitten(8,7,-120)
create masterswitch(8,1)
create container(3,5)
create bucket(1,5)
create basket(8,3)
```



Mission Text:

- Wow, we've finally made it to the /goop/ factory's control area! This is the source of all the leaking and we need to shut it down!
- The security system is trying to avoid /goop/ contamination in the other rooms, so we'll have to do a few things in sequence to open the brown /door/.
- The first /door/ should automatically open once we've organized the /goop/s by size. Small /goop/s (scale < 1) should go into the /container/. Large /goop/s (scale > 1) should go into the /bucket/.
- The /vaultdoor/ should open automatically once we've rotated the /kitten/s back to 0, and moved them to the /basket/!
- Once we're in the vault control room, we need to use the /masterswitch/ button's built-in function to shut down the factory!

Level 34. Let's shut down the goop factory!

Gidget Code (broken):

```
function workGoops(test)
 if test = "small"
  for g in /goop/s
   goto g
   if g:scale = 1
    grab g
 else
  for g in /goop/s
   goto g
   if g:scale > 1
    remove g
workGoops("small")
goto /bucket/
drop
doGoop("large")
goto /bucket/
drop
for k in /kitten/s
 goto k
 set k:scale to 1.6
 grab k
goto /bucket/
/masterswitch/:push()
```

Gidget Code (solution):

```
function workGoops()
 for g in /goop/s
  goto g
  if g:scale < 1
   grab g
   goto /container/
   drop g
  else
   grab g
   goto /bucket/
   drop g
function workKittens()
 for k in /kitten/s
  goto k
  set k:rotation to 0
  grab k
  goto /basket/
  drop
workGoops()
workKittens()
goto /masterswitch/
/masterswitch/:press()
```

Gidget Goals:

```
ensure # /goop/s on /bucket/ = 2
ensure # /goop/s on /container/ = 6
ensure # /kitten/s on /basket/ = 2
ensure /kitten/s:first():rotation = 0 and /kitten/s:last():rotation = 0
ensure /masterswitch/:activated = true
ensure /gidget/:position = /masterswitch/:position and /gidget/:position = [8,1]
```

Level 35. Where Will Gidget End Up After a Victory Dance?

World Code:

```
object puppy(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]
object bird(row, column)
 set this:position to [row, column]
object piglet(row, column)
 set this:position to [row, column]


object basket(row, column)
 set this:position to [row, column]

create puppy(5, 3)
create kitten(3, 5)
create piglet(3, 0)
create bird(1, 2)
create basket(4, 4)
```

Mission Text:

- Hmm... I'm thinking I'm getting the hang of this because of your help!
- I made some temporary adjustment to my logic chip and I want to try this level out myself in one shot!
- Can you help me determine where I will end up after running the code? We only have one shot at this so let's try our best! Click on the tile to choose your answer!

Level 35. Where Will Gidget End Up After a Victory Dance?

Gidget Code (assessment):

```
set loop1 to 2
set loop2 to 3
say "Yippe!"

while not loop1 = 0
 left 2
 down 2
 right 1
 up 2
 set loop1 to loop1 - 1
while loop2 > 1
 up 1
 down 2
 set loop2 to loop2 - 1
```

Assessment Question:

Where will I be after I execute the current code (assuming I have unlimited energy)? Please click on the grid, then press the button below to check!

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

```
wrong:: I won't end up on this space.

correct:4,3:I loop around 2 times in the first loop,
then another 3 times in the second loop. The second
loop only happens 2 times because the <span
class='dictionaryTerm'>boolean</span> is <span
class='object'>false</span> when it checks <span
class='object'>1</span> <span class='object'>></
span> <span class='object'>1</span>.
```

Level 36. Where Will Gidget End Up After a Victory Dance?

World Code:

```
object sapling(row, column)
 set this:position to [row, column]

object basket(row, column)
 set this:position to [row, column]

object bird(row, column)
 set this:position to [row, column]

create basket(3, 1)
create bird(1, 3)

create sapling(2,2)
create sapling(1,1)
create sapling(4,3)
```

Mission Text:

- Okay, I think I'm getting the hang of this. I'm so excited to see /sapling/s already beginning to sprout so much since we shut down the factory!
- I want to try counting the leaves mostly by myself. Can you just help me by verifying what will happen by choosing from the options on the right?

Level 36. Where Will Gidget End Up After a Victory Dance?

Gidget Code (assessment):

```
set leaves to 0
set plants to /sapling/s
for s in plants
 if /bird/:position = [1,3]
  set leaves to leaves + 3
 else
  set leaves to leaves + 2
goto /bird/
grab /bird/
goto /basket/
```

Assessment Question:

After running the code (assuming I have unlimited energy), the variable leaves will be equal to:

- 9
- 0
- 8
- 7
- 6

Can you tell me how you arrived at your answer? It will help me with my logic chip repairs!

Solution Code & Responses:

```
wrong:8:I don't move the <span class='object'>/bird/</span> until after the
loop, so it always adds <span class='object'>3</span> until the loop stops.
wrong:7:I don't move the <span class='object'>/bird/</span> until after the
loop, so it always adds <span class='object'>3</span> until the loop stops.
wrong:6:I don't move the <span class='object'>/bird/</span> until after the
loop, so it always adds <span class='object'>3</span> until the loop stops.

correct:9:The <em>for</em> loop executes 3 times (the number of saplings)
before it stops and I move the <span class='object'>/bird/</span>.
wrong:0:There wasn't an error with the loop, so the value changed from its
original of <span class='object'>0</span>
```

Level 37. Let's help Gidget run out of energy!

World Code:

```
object shrub(row, column)
 set this:position to [row, column]
object kitten(row, column)
 set this:position to [row, column]
object dog(row, column)
 set this:position to [row, column]
object basket(row, column)
 set this:position to [row, column]
create kitten(3,2)
create dog(2,4)
create basket(4,3)
create shrub(4,1)
create shrub(0,1)
create shrub(1,3)
```

Mission Text:

- Great! Thank you so much for helping me shut down the factory and save all the animals! I couldn't have done it without your help!
- My logic chip is almost fully calibrated now, and it needs to reboot. Since I won't be able to use it while it's rebooting, please help me solve this last mission without any starting code!
- Now that my mission is complete, I want to take these animals home with me! I'm so excited, I have too much energy! Please help me put them in the /basket/, and get rid of my excess energy!
- I should reduce my energy to less than 15 (but not all the way down to 0!), and you can use any of the commands you have mastered to get me there!

Level 37. Let's help Gidget run out of energy!

Gidget Code (broken):

Gidget Code (solution):

```
goto /kitten/
grab /kitten/
goto /dog/
grab /dog/
goto /basket/
drop
while /gidget/:energy > 15
 goto /shrub/
 goto /basket/
```

Gidget Goals:

```
ensure /gidget/:energy < 15
ensure /kitten/:position = /basket/:position
ensure /dog/:position = /basket/:position
```