Say it with Systems: Expanding Kodu’s Expressive Power through Gender-Inclusive Mechanics

Teale Fristoe*, Jill Denner†, Matt MacLaurin‡, Michael Mateas*, Noah Wardrip-Fruin∗

* Expressive Intelligence Studio
† ETR Associates
‡ Microsoft FUSE Labs

teale, michaelm,
nwf@soe.ucsc.edu
jilld@etr.org
mattmac@microsoft.com

ABSTRACT
While game mechanics are a primary focus in game design and game studies, they have been little discussed in the context of introductory game creation and programming environments. But game mechanics are central here as well, with different tools supporting the elements needed for some game mechanics (and genres) but not others. Research suggests many children, especially girls, want to create games based on dynamic relationships, social interactions, and storytelling. But game creation tools aimed at beginners offer no support for game mechanics that would enable such games. This inspires our work on Kodu AI Lab, a set of extensions to Kodu Game Lab, which we are iteratively developing and evaluating with middle school girls. This paper describes our first extensions (attitudes, learning, and fuzzy logic), the principles guiding them (simplicity, understandability, and expressiveness) and the results of our first evaluation. We conclude with our next planned development: extending the “say” command into a game mechanic.

Categories and Subject Descriptors
D.3.3 [Programming Languages]: Language Constructs and Features – frameworks.

General Terms
Design, Experimentation, Human Factors, Languages.

Keywords
Kodu, gender, children, motivation, programming, computer science education, programming environments, game design, game mechanics.

1. INTRODUCTION
Within the game design and game studies communities, game mechanics have become a primary focus for understanding how games are constructed, experienced, and interpreted. Game mechanics are variously defined, e.g., as “methods invoked by agents, designed for interaction with the game state” [25] or “rules that determine the operation of the game world” [23]. But despite these different nuances, mechanics are widely understood to describe the fundamental operations of games, serve as the basis for game genres, and provide the elements for building up game meaning. The defining mechanics of a particular game or genre are often referred to as the “core mechanics.”

Despite the attention mechanics have received in game design and game studies, there has been little explicit discussion of them in another area: the design of game making tools, particularly those aimed at beginners. Perhaps this is due to the ambition of such tools to enable wide ranges of game creation, but this ambition should not distract us from the fact that tools such as Game Maker [31] and Storytelling Alice [14, 16] include programmatic elements that support some base mechanics well, and therefore support certain genres well, but not others. For example, Game Maker provides good support for the jumping-based gameplay of the platformer genre, with built in gravity and collision detection. On the other hand, Game Maker provides poor support for the elements that make up unit production, technology trees, and other core mechanics for the real time strategy genre. These tools can certainly be used to create games outside the core mechanics and genres they support by default—students of one of the authors have used Game Maker for everything from role-playing games to multiplayer racing games—but this requires extending the tool’s base vocabulary, which many beginners are not prepared to do.

This is particularly important because introductory game making tools, and the larger class of introductory programming environments to which they belong, are often imagined as a way of broadening participation in game design and computer science. When successful, such tools can open the creation of games as an avenue for expression for wider groups in the population, which may help address the overall lack of diversity in those who pursue both game development and computing generally. But though it is widely believed that game play preferences—including genre preferences—depend on social, cultural, and gendered factors [8, 13, 20] this has not necessarily been used to inform the core mechanics of game making tools. For example, it is the models and animations of The Sims that have been brought into Alice, rather than the core mechanics of the gender inclusive social simulation game genre to which The Sims belongs [16].

This paper describes our work to expand the expressive potential of Kodu Game Lab, an introductory game making tool, by developing new elements that can support new game mechanics. Our work is particularly aimed at engaging middle school girls, to empower them to express ideas in game form. We hope to show that by providing mechanics that will enable them to create the sorts of games that most interest them, they will have more positive first experiences programming. It is guided by past research results and ongoing work with girls and boys in the target
2. RELATED WORK

Our project is inherently multi-disciplinary, with computer science and game design perspectives facilitating the development of educational programming environments, which are then analyzed from the perspective of developmental psychology. Our findings from these studies then feed back into the further development of the tool, resulting in a healthy collaboration in which our project is enhanced by a variety of sources. For this reason, we have broken up our related work into two sections: Educational Programming Environments, which focuses on the computer science and game design inspirations for our project, and Middle School Students, which focuses on the developmental psychology contributions.

2.1 Educational Programming Environments

Educators have been developing educational programming environments since the first high level programming languages emerged, starting with Papert’s Logo in the late 1960s [19]. Logo not only introduced the concept of a language designed for beginners, but also established the precedent of media being the focus of such programming, with its turtle marching around, leaving a trail of light or ink in its wake.

In the subsequent decades, researchers and industry have developed countless educational programming environments, many of which continue the tradition of media creation [15]. Drawing on the appeal of digital animations and games, environments like Agentsheets [21], Alice [4], and Scratch [22] put the novice programmer in control of designing their own media that they can share with family and friends.

While many educational programming environments provide general content, assuming that the appeal of animations or games is enough to entice all students to engage with the software, some notable projects have focused on underrepresented groups. For example, focusing on middle school girls, the Storytelling Alice [14, 16] group investigated what girls like about the original Alice, and what they believe it lacks, and made modifications to the system to appeal to this audience. In the end, these changes were primarily focused on the media surface level, involving character models more appealing to the demographic and higher level built in animations that resonate more with the kinds of stories that girls that age want to program.

Another such project is RAPUNSEL, a dance game designed to teach girls programming [11]. This project was also custom made for middle school girls, drawing on past research about what appeals to this demographic, specifically using a Values at Play framework that embodies “the ideals of values such as creativity, security, and equity” (p. 265). For example, players could design their character’s clothes and dance moves, and use the chat feature to collaborate with others. While the project involved programming, its approach was more game-like, where the student was presented with a series of challenges that required programming to overcome, rather than programming a game themselves.

In addition to the legacy of educational programming environments and especially those that focus on underrepresented groups, this work draws on the research done by co-author Denner on girls programming games. This research has yielded insights into what kinds of games that girls choose to make [8], what students learn from programming games [6, 30], and the importance of leveraging girls’ existing interests in computing [7].

Research to date suggests that many youth (often girls) want to create games based on dynamic relationships, social interactions, and storytelling [8, 17, 20] but these insights have not been incorporated into a system that can be used by the average middle school student to this end. However, our understanding of the kinds of games that girls create is limited by the tools that are easily learned by middle school girls. To our knowledge, only Kelleher has actively sought girls’ input on how to modify a tool, but her focus was on the character models, animations, and level of abstraction of the language, rather than the mechanics of the games made using the tool.

2.2 Middle School Students

The middle school years (ages 11-13) are a key time to study engagement with educational programming environments because most students this age have the cognitive capacity to engage in abstraction and reflection, but they are still in the process of identity exploration so computer game design activities are still an option, even among underrepresented groups. Retrospective accounts suggest that early experiences of success (or failure) in some aspect of computing play a critical role in whether students seek out computers for their educational or career focus [18]. Relationships and identity are paramount in middle school [1, 27]. By sixth grade, students have begun to make decisions that influence their academic success; many have begun to envision career goals that presage their later careers [28]. Early middle school is a crucial stage of development—it is where students form an identity based on decisions about what they are good at, and what they like.

Previous research on underrepresentation in computing careers has largely ignored the critical question of how students learn and engage with technology. Instead, the focus has been on psychological factors, such as how low confidence and negative attitudes toward technology and IT workers limit girls’ interest in computing [5, 32]. In addition, there has been interest in the ways that social contexts and cultures of computing can support or undermine females’ motivation to persist with technology [3, 18]. However, recent data suggest that while some of these psychological and contextual barriers are decreasing, females are still less likely than males to have plans to pursue an educational path that will lead to computer science [29].

What kinds of programming environments motivate girls and other underrepresented groups to engage in more difficult computing concepts? As mentioned above, research to date points to games based on dynamic relationships, social interactions, and storytelling, especially for girls. However, few tools accessible to middle school students offer any support for games of this type, and none for dynamic, playable models of relationships, social interactions, and storytelling. Existing child-friendly authoring environments generally only support simple interactions based on movement and collision detection. The interest in relationships and real-world issues is consistent with current research findings on game play preferences [12]. Thus, although there is very little research on how children use, and want to use, game authoring tools, our work is informed both by the existing research in this area and on play preference research.
3. THE PROGRAMMING ENVIRONMENT

We chose to focus on Microsoft Research’s Kodu Game Lab (KGL) to explore the relevance of game mechanics for novices learning to program using game creation software. Before going into detail about our contributions, we will give a basic overview of KGL to show why it is well suited as a test bed for introducing youth to programming and self expression through game creation.

3.1 Kodu Game Lab

Though developed in part to expand participation in computer science, KGL takes the approach that implementation details are unimportant to learning computer science; the developers of KGL believe that users will learn computer science concepts even if they don’t learn vocabulary. This allows them to ignore popular trends in language development and to use metaphors that are more appropriate for a young audience.

KGL is heavily inspired by robotics, which influences both the programming context and language. In the environment, all programming takes place within characters; no global level programming is possible, though certain language constructs, such as creatables (a basic class system) and scores (global variables) exist outside of individual characters. Because of this, nearly all programming controls character behavior.

The programming language itself departs from most popular languages in a number of ways. Like many educational programming environments, it is graphical to avoid syntax errors, ensure that all commands are available without having to memorize them (through menus), and be accessible to children who have not mastered written English and typing. But the most striking difference from popular languages is that it is rule based as opposed to imperative. We will describe the language with the aid of figure 1, which displays the programming for a character.

Each line of code in KGL is a rule connecting sensors and possibly filters to actuators and possibly modifiers. In figure 1, there are two lines of code. The first line, which reads as “when see apple, do move toward” and is written in text as “see – apple : move – toward”, uses the “see” sensor to indicate that the rule fires when the agent sees something. “Apple” is used as a filter, to indicate the property that something that is seen must have to fire the rule. In this case, the rule will fire if the agent sees something that is an apple. When a rule fires, the agent will perform the behavior specified by the actuator, in this case “move”. “Toward” is used as a modifier, which refines the behavior of the actuator.

So, the rule can be summed up as: when the agent sees an apple, the agent moves towards that apple. The second line, “bump – apple : eat”, indicates that the agent should eat any apples that it happens to collide with, something that should happen fairly regularly given the first line of code.

Figure 1. KGL code to make a character eat apples.

For a further explanation of KGL’s language features, see [26].

3.2 Mechanics in KGL

When first exploring KGL, one might wonder why shooting is such a fundamental part of the programming language. While one character shooting another is not quite the “Hello World!” of KGL, it is pretty close, and is by far the easiest way for two characters in the system to interact with each other. Shooting is at the core of KGL, but why? Is it that the developers are obsessed with the shooting games of their youth?

One of the authors was surprised to hear the answer from Stephen Coy, the lead developer of KGL. Originally, the team didn’t include shooting in the environment. However, during an early user test, a little girl got bored very quickly with the environment. Back then, characters could only move around and eat things, and it didn’t keep the girl’s interest for very long. She declared “You should allow shooting. That would be awesome!” The developers agreed, and children since then have been thrilled that it is so easy to shoot things in KGL, even if most teachers aren’t [Coy, personal communication].

While KGL is not Turing complete, it has powerful programming features, such as global variables, state machines, and object factories. The state-of-the-art in KGL games includes a clone of Portal and crime mystery puzzle games. The children who use KGL have incredible power at their fingertips, and yet many of their first games end up looking like chaotic battlefields with characters shooting missiles in all directions at each other. Why is this?

For one thing, the children are imitating the sorts of games that they are used to–to–many of them, games have shooting. But another reason is that so much logic is built into the simple Shoot command. The developers of KGL wrote rich physics, collision detection, health, and particle systems, which are all hidden behind the unassuming shoot command. Given that it is so easy to harness all of that logic, why would inexperienced programmers do anything else?

Despite the openness of a language, programmers of all types, but novices especially, tend to employ the game mechanics that the language makes easy to access, such as the multiple logics a programmer harnessed for free using the shoot command. Thus, what the environment makes easy to do will have the biggest impact on the experience users have and the artifacts they produce with it. For educational programming environments in particular, this is a crucial concern: what a language provides will define many children’s first experience with computer science and programming, and have a major impact on their perception of the field.

3.3 Kodu AI Lab

We have developed new elements for KGL that we hope will expand the mechanics students can include in their games. We call our modified version of KGL Kodu AI Lab (KAIL) because a number of the features are inspired by Artificial Intelligence techniques. While we want KAIL to include powerful new language features, we have to balance power and accessibility.

One of KGL’s strengths is its intuitive and understandable level of abstraction. Everyone has an idea of what words like “see”, “bump”, “move”, and “away” mean, and the system takes care of the details a programmer would normally have to worry about. This allows anyone to pick up a controller and immediately begin to program meaningful experiences. One of the challenges with our project was to maintain this level of understandability and
intuitiveness even when dealing with what are sometimes seen as complex AI concepts and algorithms. For this reason, from the beginning, we designed our new features with three goals in mind: *simplicity, understandability*, and *expressiveness*. *Simplicity* means that the terminology we use is easy to grasp. *Understandability* means that the new language constructs will not result in unexpected behavior (a common challenge with many AI algorithms). *Expressiveness* means that the new features will allow users to intentionally create experiences that would have been difficult or impossible without the features.

We developed three new language constructs for KAIL. The first feature we call *attitudes*, which allows characters to put other characters into subjective classes and reference them as such. The second is *learning*, which gives characters the ability to dynamically classify other characters based on their behavior and other events in the world. The final feature is *fuzzy logic*, which allows characters to classify other characters to different degrees. In the following sections, we describe each of these features and discuss how well they satisfy our goals of simplicity, understandability, and expressiveness.

### 3.3.1 Attitudes

In KGL, characters have almost no internal state. Furthermore, sensors can only identify characters based on their external state. Our first new element, attitudes, extends KGL’s power to maintain and sense internal state.

Attitudes allow characters to classify other characters and objects. We include four attitudes for our initial tests: “friend”, “enemy”, “scary”, and “tasty”. Additionally, there is the “unknown” attitude, which all characters are in before they have been otherwise classified. These attitudes are appropriate for many games, but there is no limit to the kinds of attitudes that are possible. Furthermore, while the attitudes have meaningful names, they have no inherent meaning; their meaning only derives from the way programmers use them when coding their characters.

The new verb, “think”, is used with an attitude modifier to indicate that a character, when sensed, should become a member of the given category. For example, “see – red – cycle : think – scary” indicates that whenever the character sees a red cycle, it should think of that cycle as scary. Attitudes can also be used as filters to specify what a character should react to. For example, “see – scary : move – away – quickly” indicates that whenever the character sees something scary, it should run away from the scary thing.

Thinking and using the category filters and modifiers both build on common concepts and language patterns already established in KGL. Furthermore, the resulting behavior is straightforward: after something is thought as a member of a category, it will trigger sensors filtering that category. Because attitudes are determined using existing sensors and filters and can only trigger existing actuators, they don’t fundamentally extend the expressiveness of KGL; they do, however, make new levels of expression achievable in practice. For example, programmers can much more concisely and intuitively express rules about social relationships. Attitudes especially improve the potential for authoring complex behaviors, making them more tractable and likely for our target audience to develop. Additionally, attitudes make code much easier to understand.

### 3.3.2 Learning

Allowing characters to classify each other based on external characteristics was the first step; we next wanted to enable characters to respond to each others’ actions--to learn from their environment.

Attitudes introduce rudimentary learning into the system, and give us a good starting point for more advanced learning mechanisms that users can actually understand and use in a consistent manner. We already have language features that allow characters to change their attitudes about other characters based on their external characteristics, so we introduce new ways to indirectly experience other characters. This allows characters to classify one another in non-direct ways.

However, sophisticated learning mechanisms can produce bizarre behavior when left unchecked. For example, we might want characters to be able to change their attitudes about classes of characters, such as fearing all cycles when a green cycle is hostile towards it. The problem is this will not always be the case, such as when some cycles are friendly towards it, but all green characters are hostile towards it. When adding learning capabilities, we want to ensure that programmers have enough control to avoid such potentially unwanted results. By focusing on characters changing their attitudes about other specific individual characters, the programmer knows exactly what will cause a character to learn and how the learning will affect the character's behavior. But this still offers enough power for interesting results.

We implemented three new filters to facilitate characters learning through indirect experience: “take damage”, “heal”, and “gift from”. As their names indicate, the first two deal with the character's health; “take damage” fires when the character loses health and “heal” fires when the character gains health. The last, “gift from”, fires when a character receives a gift from another character. While these filters do not have to be used for learning, they allow for interesting character behavior when they are. For example, “gift from : label – it – friend” makes a character identify any other character that gives it a gift as a friend and react to it as such (e.g. show a stream of hearts). As seen in this rule, language constructs using the new filters sacrifice perfect correspondence with English to ensure they fit with the existing KGL language requirements.

Our implementation of only three new filters to facilitate learning is a first step, and it illustrates the power of the concept. While we continue to make our implementation simpler, it is currently understandable, and its greatest strength lies in the significant additions to expressiveness it allows.

Even with only three constructs, children can now program characters with rich personalities. Take for example this code for a shy character: “see – unknown : move – away; gift from : label – it – friend; see – friend : move – towards”. This timid character will move away from other characters it doesn't know, but as soon as another character makes a friendly gesture by giving it a gift, the character will consider the other character a friend and will move towards it to give it a hug. A personality such as this one, impossible to write using the language constructs available in KGL, allows students to explore games similar to *The Sims*, where character interaction is the fundamental mechanic. We believe dynamic relationships offer many new and exciting programming opportunities.
3.3.3 Fuzzy Logic

Fuzziness is a natural extension to attitudes with the addition of learning mechanisms. Fuzzy logic is attractive because it helps smooth out the perceived abruptness of Boolean logic in character behaviors.

Before introducing fuzzy logic, a character might have many attitudes about another character (e.g., it might think a character is both a friend and scary). With fuzzy logic, different attitudes can be specified to an even smaller granularity. Instead of the all or nothing of Boolean attitudes, a character can hold each attitude to any degree from 0 to 1. This means that a character might be a little scary but a good friend to another character, and can even change its level of friendship as the game progresses.

Programmers need both filters (to react to characters with a minimum level of attitude membership) and modifiers (to think attitude membership of various degrees). However, revealing the numbers underlying fuzzy logic could be confusing. Instead, we use qualitative descriptions for different levels of fuzziness. For example, the phrase “a lot” indicates fuzzy membership of .7, “somewhat” indicates fuzzy membership of .5, and “a little bit” indicates fuzzy membership of .3. Using English words rather than numerical values to describe states for the programmer makes fuzziness more accessible for young students.

To facilitate the discussion of our implementation of fuzzy logic consider the following code for a character:

damage – take damage : think – it – scary – a little bit
see – unknown – a lot : move – towards
see – unknown – a lot : shoot
see – scary – somewhat : shoot
see – scary – a lot : move – away – quickly

We will call the character that has this programming the bully character. The first line says that when another character hurts the bully character, the bully character considers it a bit scary. This is cumulative, so if the bully character already considers it a bit scary, it will consider it a bit scarier when this rule fires. The next two lines implement aggressive behavior, making the bullying character attack characters it hasn't experienced before. However, the final two lines show that the bully character is a coward at heart: when it encounters a character that stands up for itself a bit, it will stop approaching that character, and when the character continues to stand up for itself, the bully will flee.

As the example illustrates, our implementation of fuzzy logic in KAIL sacrifices simplicity and some understandability for much greater expressiveness. Programmers gain much more fine grained control over their characters’ behaviors. What results is a potential for game characters with rich, unique personalities that can change dramatically as a game progresses.

4. THE STUDY

We developed a research plan in order to test whether the new features that we thought would be attractive for middle school girls not only appealed to a range of girls, but that the higher level programming concepts could be easily implemented by girls with a range of computer experience. The research uses an experiment design that is commonly used in educational technology research [2]. It involves developing or modifying a tool, using it with a small group of students and collecting feedback, modifying the tool, and testing with a new group of students.

4.1 Research Plan

Our research plan draws on prior studies that include children as design partners. Druin describes several roles that children can take to inform the development of new software or technologies, ranging from using an almost-finished product to participating in the early conceptualization [9]. Our plan includes girls in the initial discussions of how the tool would be modified. The first step is to work with children as informants [24] to provide input into the direction that development should take. The second step is to involve “children as testers” to identify which parts they like, which are confusing, and which are boring [10]. In the third step, we will conduct a mini study to compare students using the new version of KAIL vs. students using one of the existing game development tools, such as Storytelling Alice, measuring interest and engagement with computing.

To date we have run two sessions with “children as informants”, each of which lasted six hours over the course of a week. The participating girls were self selected and agreed to stay after school for two hours three days during the week of the session. Students took surveys at the beginning of the first class and the end of the last class. For the first half of the session, students learned how to use KAIL. One of the researchers acted as teacher, going over the curriculum (described below), while students either followed along to program examples with the teacher or completed programming activities after a brief lecture on the current material. During the second half of the session, students made a game of their own, starting from scratch. They had free reign over what they made, which ensures that all code in the final games is students’ own, and gives us the opportunity to determine what students value by looking at what programming constructs and content they use in their games.

4.2 Curriculum

In running KAIL classes for middle school age girls, we had two main goals in mind. First, we wanted to test our new elements, to determine if they appeal to our target audience and whether they were understandable by most of the students. Second, we wanted to gain a better understanding of what elements would be appropriate to develop in future versions of KAIL. A third minor goal was to avoid the use of violence in our lectures and examples, finding alternatives to common game mechanics like shooting and fighting, in order to focus on exploring new expressive spaces for game design.

Our curriculum was designed to test out our new elements as soon as the class started. From the very first example code, we incorporated attitudes, making characters think apples are tasty before moving towards and eating tasty things. We emphasized the new “gift from” filter during instruction, quickly incorporating picking up and giving objects, which we felt would introduce a non-violent alternative mechanic to shooting that students could build games around. Additionally, a giving action had the natural advantage of working well with developing friendships and introducing the girls to the concept of learning based on events. After giving, we attempted to incorporate fuzzy logic into the lessons before the girls went off to create their own games, but limited time was spent on this topic.

4.3 Evaluation

To evaluate how effective the new elements were in engaging middle school girls, we used two general strategies. First, we asked them which elements they liked. Second, we closely examined the final games they created to determine if they used...
the elements we had introduced. As mentioned above, since the students built their final games from scratch with completely free reign over what they created, this offered a good method for determining which constructs they understood and found compelling.

We faced mixed results when we used these methods to determine how effective the new elements are. We often heard positive reports about the elements from students—they seemed to like the idea of them, and said they would like to use them. However, when we examined the artifacts they produced, we found that few made use of the new elements. For example, no students used fuzzy logic. While about half of the students attempted to incorporate attitudes into their games, we found that they often used them incorrectly, for example looking for a friend but never assigning anything as a friend, or vice versa. This indicates that even if the students like the idea of making use of the new elements, they don’t fully grasp the way the elements work.

Additionally, despite our push for the girls to use giving as a central mechanic in their games, we found that they rarely made use of the mechanic, focusing instead on collecting and eating objects rather than picking them up and manipulating them. When students did attempt to use giving, they often struggled with the relatively complex logic necessary for it. The sample code from the curriculum appears in figures 2 and 3. Unlike shooting, where much of the logic has been built into the language construct, giving requires explicitly specifying the logic as rules.

Reflecting on these results, we believe that we tried to expose the girls to a relatively complex, compound task too early in their educational experience. We believe that with more time they would be able to master the logic necessary to make games based on giving, but for their earliest exposure to programming, we need to find a mechanic like shoot, where more of the logic is built into the language construct.

5. FUTURE WORK: THE SAY MECHANIC

One element that appealed to the girls immediately and strongly is "say," an action as simple as shoot, that allows characters to engage in the appearance of social interaction. The girls seemed to find it attractive because it offers unlimited customizability, allowing them to express themselves in a simple and completely open way (i.e. through dialogue). Unfortunately, say also currently has no functional qualities; it is the equivalent of print, displaying text to the screen, but not making this text available to game characters. We plan to harness the inherent appeal of say but endow it with functional power that will enable its use in expressive systems. The goal is for say to be the game mechanic giving couldn’t be, driving the games that the girls make. We envision say controlling conversations between characters, determining the flow of dynamic relationships between characters as the game progresses, and doing so in an arbitrarily customizable way, allowing each girl to put her own personality into her creation.

Say appears to be fertile ground for exploring new game mechanic possibilities. An incident from one of our classes demonstrates one way more powerful conversation mechanics could be very effective at engaging children. One of the girls in the class had played Final Fantasy. Despite claiming that she didn’t know what game to make, she had big ambitions for a complex story and an introduction cut scene to go along with it. Unfortunately, KGL has no built in features for managing conversations like she wanted, and despite the best efforts of the researcher to help her cobble together a conversation system using some of KGL’s other features, they were unable to finish the cut scene in time. Had there been a conversation system in place from the beginning, this girl would have been able to realize her goals in a timely fashion, and would have better understood the system she put into place.

With motivation and goals for our advanced say system, we now have to decide how to implement it. First, we will keep the three goals for new features in mind: simplicity, understandability, and expressiveness. We will also make sure it fits into the existing framework for KGL language features.

Characters can already say things, but all this currently entails is that their words display on the screen for the player to read. We want characters to be able to talk to one another, so we need to keep track of all of the things being said at any given time. This will be taken care of internally behind the scenes so programmers of KAIL do not need to worry about it; they just program a character to say, and whatever it says will be captured by our system.

In order to facilitate actual conversations between characters, one character must be able to listen for things that another character says. There is already a hear sensor, which is very similar to “see” because it is assumed that all objects (animate and inanimate) in KGL are making noise at all times. To allow for conversations using say, we will develop a new modifier for the existing hear sensor, so that a student can specify that a character is listening for a spoken utterance from another. However, simply specifying that a character is listening for something said is not enough. We want to allow students to specify what kind of utterances the character is listening for.

The most obvious approach would be to allow a programmer to write arbitrary text in the hear modifier, just like she can write arbitrary text in a say actuator. This will allow her to make one character say something (perhaps “Hello!”) and make another
character listen for and respond to that same phrase. In this way, children can program conversations where one character listens for a particular utterance and then responds to it appropriately, only to have the response responded to. In fact, children can even assign special utterances to different buttons, and allow the player to choose what her character says, which other characters can respond to as the programmer sees fit.

There is one concern we have with the scheme described above: it introduces syntax errors to a language that has striven to avoid them at almost all costs. Do we really want a girl’s first experience with syntax errors to be strings not matching in a say and a corresponding hear tile, especially when the programming environment can have no idea whether or not it is intended? A warning for hear tiles without a matching say tile would be possible, but is still undesirable for an early programming experience. While it might allow for a teachable moment, it will also almost surely result in confusion and frustration, and is therefore problematic. This doesn’t mean we should eliminate it altogether, especially without trying it with actual students, but it does mean we should explore alternatives that might mitigate some of these problems without diminishing the expressiveness of the system.

One possibility is to use modifiers to tag a speech utterance, and then listen for specific tags to respond to. For example, when our hypothetical character says “Hello!”, we might specify that this is a greeting, and that it is friendly. Then, instead of listening for “Hello!” another character could listen for a friendly greeting (using filters with the hear sensor) to respond to the first character. To the player of the game, it appears that the second character is responding to the first’s specific words, just like with our former system, but behind the scenes, we have a more robust conversation representation that is less likely to suffer from syntax errors or break when the exact phrasing is changed. With this system, we have enabled conversations and still remained true to the KGL style.

While this conversation system would enable the girl we described above to easily program her cut scene without fear of problems, making the characters patiently wait to hear what other characters have to say before responding to them appropriately, it also enables new gameplay that revolves around interpersonal communication, especially when combined with attitudes, learning and fuzzy logic. By listening for speech acts of a general type (e.g. greeting) and the way the speech act was delivered (e.g. friendly), characters are released from the burden of having to listen for specific utterances that fall into classes (e.g. “Hello”, “Hi”, “What’s up”, etc), and can therefore respond to other characters generally. What this allows is games where characters respond to each other in intelligent and interesting ways. We might have a vain character that becomes better friends with characters that give her flattering compliments, while a shy character responds in a completely different way. We might have a mean character that goes around giving insults to others, and the responses of other characters depends on their programming. The possibilities are endless, and we hope that they not only inspire girls to explore new gameplay that incorporates the other elements we already introduced, but also to explore their ever changing social environments, allowing them to make real world connections to programming that might inspire them for years to come.

6. CONCLUSION

While many game scholars and designers recognize the importance of game mechanics to create meaningful play experiences, developers of game creation software designed for novices have rarely focused on the mechanics that are easily enabled by their software. Those mechanics will have a major impact on the games beginners make and therefore the first experience those people have with game programming and design. Because some gameplay appeals more to some demographics than others, game authoring tools are implicitly demographic specific. Without paying attention to which groups are likely to embrace the tool, it’s very possible that members of important target groups will not find it appealing.

In this paper, we have described our initial attempts to customize the mechanics available to users of Kodu Game Lab so it will appeal more to middle school girls. We added several new language features, including attitudes, indirect learning, and fuzzy logic to facilitate the creation of games based on social interactions and dynamic relationships between characters. While we received generally positive feedback about the features, we found that girls often did not use the features, likely because—in combination with the multi-part giving and receiving mechanic we introduced—they were too complex given the amount of time we had for instruction and for them to program their games.

However, we have taken the girls’ feedback and planned our next round of modifications: making verbal communication between characters a central mechanic. We believe our target group will find this very appealing, because they already enjoy using the say command to make characters speak to the player, and because it will allow for more customizable games that are focused on social interactions, especially when combined with the KAIL features already implemented. By catering to their interests as well as challenging them, we believe we can meaningfully engage this underrepresented segment of the population in game design and programming.

7. ACKNOWLEDGMENTS

We would like to thank the Kodu team for their support and assistance: Stephen Coy, Brad Gibson, Mark Finch, Eric Anderson, Rachel Schiff, Alex Lehman, Brian Bosworth, Kristin Stecher, Josh Tabak, Daryl Zumiga, Brad Klocksiem, and Lili Cheng. We would also like to thank the members of ETR and school teachers and administrators who helped make our classes possible: Gina Lepore, Valerie Quandt, Irene Medina, Laura Hamby, Trish Hucklebridge, and Amy Thomas. This project was made possible with the support of NSF award DRL-1042944.

8. REFERENCES


