PlaySketch: Turning Animation Sketches Into Game Logic

Richard C. Davis and Kenny T.W. Choo
School of Information Systems, Singapore Management University
80 Stamford Road, Singapore 178902
rcdavis@smu.edu.sg, kenny.choo.2012@smu.edu.sg

ABSTRACT
This paper outlines a proposed design for PlaySketch, a video game creation system that uses animation sketching and programming by demonstration techniques. A preliminary study showed that people take naturally to describing game logic with animation sketches. Our user interface design structures the video game design process as a series of sketched animations, which are run-throughs of game activity. PlaySketch will infer game logic from new run-throughs and from modifications to existing run-throughs. When making inferences, PlaySketch will rely heavily on patterns from existing game genres. Logic will be separated into three levels of increasing variety and complexity: properties, behaviors, and events. Understanding higher-level logic will require users to learn more, but there will be fewer instances in any one game. PlaySketch will also help users learn about inferences through notifications that explain them in context, and it will give them shortcuts in the form of pre-defined actors.

Author Keywords
Animation; games; sketching; artificial intelligence, pen and touch based user interfaces.

ACM Classification Keywords
H.5.m. Information interfaces and presentation (e.g., HCI): Miscellaneous.

General Terms
Human Factors; Design; Measurement.

INTRODUCTION
Video game construction is proving itself to be a great way to learn. Making games is excellent motivation for learning to program [26,27], and it can help children to learn systems thinking [9,42], critical thinking [28], media literacy [3], design [42], and even ethics [23]. Best of all, children have shown strong interest in making games, but there is a problem: the need to program limits their participation [12]. Programming intimidates many children, and it carries a social stigma [16]. This is leading some researchers to create game development environments that reduce or eliminate the need for programming [3,28].

Copyright 2014 Richard C. Davis and Kenny T.W.Choo

Some researchers have used programming by demonstration (PBD) to make game development easier for people without programming skill [18,32,33,37]. After users create concrete objects and act out their behavior, PBD systems infer the general rules that determine how the game works. Research in PBD systems for video games explored methods for editing and feedback [18,32,37] as well as improved inference methods [17]. However, many problems were never resolved, and few PBD systems for video games exist today.

More recently, sketching and demonstration have been successfully applied to animation [6,7,11,24,30,34]. While making animations is easier than making games, animation is still challenging for children. Sketching and demonstration simplify this process by taking advantage of a child's intuitive sense of space and time. This makes it possible to create short animations in minutes or seconds.

We are building PlaySketch, a video game construction environment for children that combines animation sketching with programming by demonstration. Our preliminary work has shown that children take naturally to animation when expressing video game behavior. We believe that structuring video game demonstration as a process of creating and refining animation sketches will enable children to define complex behaviors without any programming.

To help children make sense of PBD inferences, PlaySketch will use terms that match childrens' own categories for game logic. We also plan to split PBD inferences into three levels of increasing complexity. First, demonstrated motions will be used to infer properties like game genre (e.g., platformer or vertical scroller), actor roles (e.g., player, enemy, or item), and physical constants (e.g., gravity or friction). Second, PlaySketch will match demonstrated motions to pre-defined behaviors that fit the chosen genre or actor roles (e.g., 4-way direction, 2-way direction, jump, or shoot). Third, more complex events will be inferred from the ways children modify their animation sketches. Splitting inferences into three levels like this will limit what children must understand to make sense of inferences. PlaySketch also will help children through the PBD process with notifications that explain inferences and pre-defined actors that can shortcut the process.

This paper describes our preliminary work on PlaySketch. We begin with a review of related work and then briefly describe a study where we observed children using sketched

animation to design games. We then describe our design for PlaySketch and present a detailed scenario that shows how it will be used.

RELATED WORK

Animating With Sketches and Demonstration

Researchers have used sketching in numerous ways to make animation easier. Many have directed 2D or 3D character animations using a single static sketch [31,36,39] or a sequence of static sketches [4]. Static sketches have also been used to direct 2D physical simulations of rigid bodies [1] and fluid systems [41].

Static sketches cannot easily capture the relative timing of multiple object motions, nor can they capture the relative speed of different parts of one motion path. However, animation systems that use real-time demonstration to record object motions can easily capture relative timing and speed by taking advantage of users' intuitive sense of time. Serveral 2D animation systems that capture real-time demonstration of sketched objects have shown benefits for novices, including K-Sketch [6], Sketch-n-Stretch [34], and idAnimate [24]. The animation interface for PlaySketch will be very similar to these systems. We will also borrow ideas from PhysInk [30], which uses demonstrated motions to direct physical simulations.

Real-time demonstration has been used in other ways that make animation easy for novices. Igarashi and colleagues used demonstration on multi-touch surfaces to animate 2D deformable characters [11]. Barnes and colleagues used video to capture demonstrated motions of 2D paper cutouts [2], while Held and colleagues used a Kinect sensor to capture demonstrated motions of 3D objects [10]. We consider these techniques to be outside the scope of this project, but any method for demonstrating 2D animation could be integrated into our design.

Enabling Childen to Make Video Games

Game builders for children come in a variety of styles. Some are for 2D games [8,21,25,27,33], some are for 3D games [14,15,26], and a small number support both [43]. Some have a high level of polish and come with ready-made graphics and sound [8,14,15], while others rely on children to create or obtain their own content [25,26,27,33]. Some constrain they types of games allowed [8,43], others constraint the types of motion allowed [25,26,32,33], and others seek to support games of any style that are arbitrarily complex [27]. The latter type of systems are said to have "wide walls" and a "high ceiling" [27]. Because PlaySketch's interface is based on 2D animation sketching, we will focus on 2D games and child-created content, but we may allow uploading of images as well. We also seek a high ceiling and wide walls, but using PlaySketch will be easiest with games that fall into established genres.

Kelleher and Pausch's taxonomy of novice programming systems gives us a useful way of categorizing systems that empower kids to make games [13]. Many systems try to make it easier to write programs. Some do this by simplifying interaction with the programming language (e.g., with drag & drop tools), such as Scratch [27], AgentSheets [25], AgentCubes [26], and Kodu [14], and Mission Maker [3,15]. Others try to make the language more understandable. Hands [21], for example, was designed through studies of how children naturally express game behavior. We borrow many good ideas from these systems, but our goal is to help children avoid writing programs altogether, not to make writing programs easier.

When systems seek to free children from writing programs, they usually use one of two approaches for defining object behavior. The more common approach is to allow behaviors to be selected from a list of choices, usually with a menu of options for each behavior. This approach is taken by Gamestar Mechanic [8,28] and Sploder [43]. This approach does not easily support a high ceiling or wide walls, because the number of behaviors and options can become too large to search through. The less common approach is to allow definition of object behaviors using programming by demonstration (PBD), as in Stagecast Creator [33]. PlaySketch will use this approach, because we believe that combining it with animation sketching will make it easier to define a wide variety of behaviors (as explained later). However, Playsketch will allow children to select behaviors and options from a list when PBD fails.

While we seek to free children from writing programs, PlaySketch will need a way to show inferred game logic to children so they can accept or reject it. Some of the systems described here show game logic in a style that mimics procedural programming models and control structures [27]. Others use a simpler event-based style that requires all statements to be rules of the form "when <condition> do <action>" [14,21,25,26,33]. While the procedural style can be more powerful (higher ceiling), PlaySketch will use the event-based style, because it more closely matches children's natural way of expressing game behavior [22]. Such rules also match the form of inferences that can be made with PBD. Also note that all the systems here represent programs graphically, except for Hands [21], which uses text only. We will use a graphical representation because it is easier modify on tablet computers.

Finally, it is worth mentioning that some children move to more powerful tools when they go looking for a higher ceiling or wider walls. Tools like GameMaker [40], Sencyl [35], Construct [29], and Fusion [4] are intended for designers with little or no programming skill who want to create and distribute high-quality games. These tools share much in common with the other tools listed here, but the need to support a high level of polish makes them more complex. We will not seek to support this level of polish. However, Stencyl and Construct both have an interesting blend of ideas found in other tools: common behaviors can be chosen from a list, while uncommon behaviors can be

programmed. We use a similar approach in PlaySketch by separating pre-defined "behaviors" from "events," which are less common and more complex.

Programming by Demonstration

PlaySketch will use PBD techniques to free children from writing programs, but it will not free them from interpreting or modifying programs. For this reason, our approach is more similar to what Nardi calls "automatic programming by informal program specification" [20] rather than classical PBD. Children will use sketched animation to specify game requirements that are automatically turned into game programs. We borrow ideas from systems discussed previously to make game programs easy to interpret. These programs can then be refined either by sketching more animations or by modifying the program. Our contributions lie in how programs are demonstrated and inferred, how inferences are presented, and how programs are edited, all of which have been open PBD problems for decades [19].

As mentioned earlier, most PBD systems for games use demonstrated behavior to infer rules of the form "when <condition> do <action>." In Stagecast Creator [33] (formerly called KidSim [32]), all rules are graphical rewrite rules. Objects exist on a grid, and children can demonstrate rules that specify how one configuration of objects should be "rewritten" in another configuration. PlaySketch does not use this approach, because many common behaviors in video games are hard to define on a grid or with rewrite rules.

Stimulus-response [38] is a more general PBD approach that splits inference into at least two stages: conditions (stimulus) and actions (response). An important factor in these systems is the set of modes that users must be aware of. Pavlov [37] had five modes: draw, test, stimulus, response, and real-time response (for demonstrated animation). This required users to be explicit in their intention to demonstrate both stimuli and responses.

Gamut [18] is a stimulus-response PBD system with four modes: build, test, "Do Something!" (for adding new responses) and "Stop That!" (for stopping unwanted responses). Users did not need to be explicit about demonstrating stimuli, because stimuli were inferred at the moment users clicked "Do Something!" or "Stop That!" PlaySketch will be more like Gamut in this respect, but it will require only two modes: draw and run. The run mode will be used for recording animated run-throughs of a game, testing a game, and reviewing existing run-throughs. Whenever new behavior is recorded, PlaySketch may make inferences.

There are two other similarities between PlaySketch and previous stimulus-response PBD systems. Because children will be able to create multiple sketches of gameplay, PlaySketch can use Gamut's method for inferring behavior from multiple examples [17]. Also, Pavlov [37] could record demonstrated animation as PlaySketch will, but Pavlov had one timeline for each response, while PlaySketch will have one timeline for each animated run-through. There are few other notable similarities between PlaySketch and existing PBD systems. Our use of animated run-throughs together with our methods for displaying and editing game behavior will make PlaySketch very different from any exiting PBD system for games.

A STUDY OF CHILDREN DESIGNING WITH ANIMATION

To assess children's ability to design games with sketched animation, we conducted a study where we asked children to design games in three media: written words, static sketches, and animation [5]. Here we give a brief overview of this study and its major findings.

In this study, we guided children through a design process by asking four questions: how do you control the main character, what is the goal, what are the obstacles, and how do you win? Children answered all questions in one medium (written words, static sketches, and animation) before moving to the next. We looked for differences between media by comparing design artifacts. We also observed children's behavior as they worked.

This study was run in four different sessions at community centers in the United States. Fifteen children age 7-14 years old took part, some working individually and some in pairs. Of these, five individuals (3 boys and 2 girls) and two pairs (all girls) completed designs in all three media. One of these children had participated in a summer workshop for making video games, but the others had no experience making animation or video games. All children worked with writing or sketches first before using animation. Animations were created with K-Sketch after 15 minutes of practice.

Figure 1 shows the average number of game elements found in artifacts of each type. Children expressed an unusually high number of action elements when using animation. For most other types of elements, the number was comparable to written words or static sketches. This

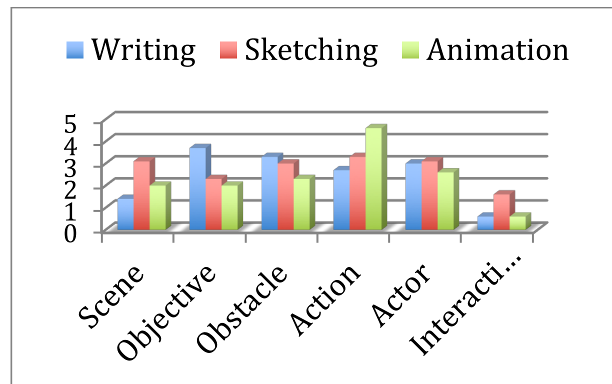


Figure 1: Average number of game elements found in teams' design artifacts, by medium and element type.

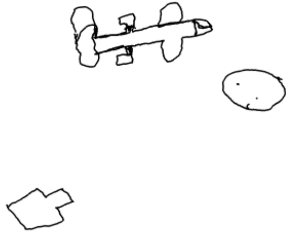


Figure 2: Child-created animation of a turret shooting a spinning ball at a moving airplane.

Figure 3: Child-created animation of a boy sneaking up behind an angry woman and stealing her food.

shows that children are able to express video game behavior with sketched animation, and it hints that sketched animation may be especially suited to capturing behavior in action games.

When we looked at how children made use of animation, we noticed three patterns: *exploring motion timing*, *storytelling*, and *collaborating*. The first pattern captures the way children iteratively refined their animations as they explored the relative timing of objects that moved simultaneously. This gave children an opportunity to experiment with different game mechanics. For example, Figure 2 shows an animation of a turret shooting a spinning ball at a moving airplane. The child started by drawing and moving the airplane, then drew the ball and made it spin, and finally moved the ball from the turret so that it missed the airplane. Later, the child modified the animation to make the ball bounce off the airplane.

In the *storytelling* pattern, children laid out the narrative of their game. For example, Figure 3 shows an animation of a boy sneaking up behind an angry woman to steal her food. These animations are usually longer and have few objects moving in parallel. Sometimes children added text labels to explain the action. There is less iterative refinement of these animations, but children did experiment by making multiple scenes that told slightly different stories. For example, the child who created Figure 3 created an earlier animation where the angry woman was hunting for the boy.

Finally, the *collaborating* pattern showed that children can easily work together through animated sketches. This sometimes appeared in the form of turn-taking while working on a shared sketch. Alternatively, a child might show an animation to a friend, get a verbal response, and quickly modify the animation. For example, the child who made the spinning ball animation in Figure 3 showed it to a friend who said, “What if the blob bounced off the airplane?” Within seconds, the child had modified the animation to show the ball hitting the airplane and bouncing off.

This study gives evidence that children can express game behavior with animation sketches about as easily as they can with words or static sketches. Animations are particularly good at capturing the relative timing of moving objects as well as the sequence of events in a story. Also, children can collaborate around animation sketches as they do around static sketches. These are indicators that sketched animation is an excellent medium for exploring a game design space and that a PBD engine will have rich data for inferring game behavior.

PLAYSKETCH USER INTERFACE DESIGN

We are building PlaySketch, a system that will help children make video games without writing programs. Figure 4 shows our concept for the PlaySketch user interface. The interface has been designed for multi-touch tablets like the iPad, which are becoming popular in educational environments. Games are divided into scenes, which can share actors. Scenes have two views. The *Scene* view (shown in Figure 4) is where Children will spend most of their time. The *Events* view is for looking “under the hood” at the most complex game logic, as explained later.

Instead of making games by writing code, assembling blocks, or selecting options, children will use animation-sketching techniques to make run-throughs of game activity. PlaySketch will use these sketched run-throughs to infer properties, behaviors, and events, which are the fundamental building blocks of game logic. To help children through this process, PlaySketch will give them notifications that explain how inferences are made and allow them to be corrected. Also, children can avoid automatic inferences with pre-defined actors. The following sections will explain these aspects of the PlaySketch user interface.

Sketching Animations

PlaySketch’s *Scene* view will be similar to K-Sketch [6]. This view has two modes: *draw* and *run*. The view opens in *draw* mode with an empty timeline. Drawing or manipulating objects in this mode changes the initial state of the scene. Children can switch to *run* mode by pressing the *run* button (Figure 4h). In *run* mode, PlaySketch will

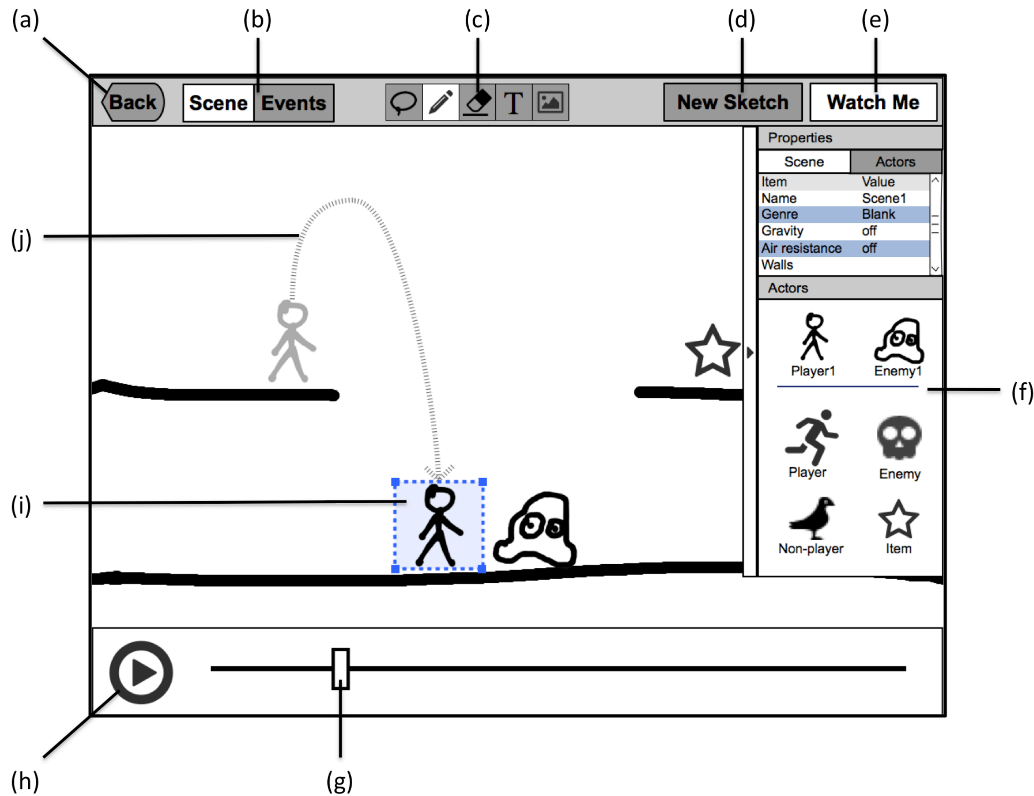


Figure 4: A mock-up of the PlaySketch interface. (a) Back to list of scenes. (b) Switch between Scene view (shown) and Events view. (c) Tool palette: select, draw, erase, text, and image. (d) Create new animation sketch. (e) Turn inferencing on/off. (f) Collapsible view showing available actors and properties for the scene or the selected actor. (g) Time slider bar for reviewing current animation sketch. (h) Run button. (i) Selected actor. (j) Motion path.

record all game behavior and object manipulations in real time. Tapping on a selected object once also causes the next drag operation to be recorded (a handy shortcut).

The behavior of *run* mode depends on the current selection and the state of the timeline (as shown in the time slider bar in Figure 4g). If no object is selected, then the behavior is straightforward. *Running* with an empty timeline will test the game in its current state, recording a run-through of game activity. Rewinding and pressing *run* again will review the recoded run-through.

If an object is selected, then manipulating that object in *run* mode may cause PlaySketch to infer new game logic. If the object is not an actor, then it will be converted into an actor. Manipulating an actor in *run* mode will override any behavior or events that would normally determine that actor's movement. Furthermore, after a run-through has been recorded, rewinding and manipulating an actor in the middle of a recorded run-through will overwrite any previously recorded activity. This will give children a way to correct an actor's behavior when they notice that it did not behave as it should have.

Inferring Properties, Behaviors, and Events

Whenever the *Watch Me* button is pressed, PlaySketch may respond to a child's actions by inferring properties, behaviors, or events. Properties include the most common game play attributes, including, runtime attributes (e.g., *position*, *health*, or *inventory*), physical constants (e.g. *gravity* or *mass*), and some other options (e.g., *can rotate* or *affected by gravity*). The two most important properties, however, are *genre* (for scenes) and *role* (for actors). These properties are listed first, and they influence how PlaySketch will make inferences from demonstrated motions. *Genre* and *role* may also constrain other properties. For example, setting the *genre* to *platformer* will turn on *gravity* for the scene.

Behaviors are special properties that can be added to determine how actors move. Behaviors are more varied and complex than properties, but each game will need only a few of them. Behaviors come in several types. Control behaviors map commands to specific types of motion (e.g., *2-way directional control* or *jump control*). AI behaviors work with control behaviors to make actors move automatically (e.g., *patrol AI*, *pursue AI*, or *evade AI*). Game behaviors specify constraints (e.g., *player cannot*

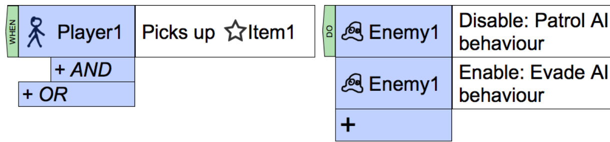


Figure 5: An event as it will appear in the *Events* view.

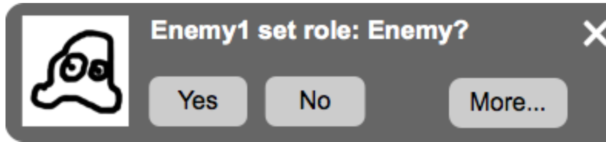


Figure 6: Notifications like this one will appear in the upper-left corner when inferences are made.

leave scene) or ending conditions (e.g., *win when enemies are dead*). These pre-defined behaviors are listed with the properties of the scene or an actor. Behaviors may have their own properties, and they may also constrain other properties. For example, the *jump* behavior has an *acceleration* property, and the *2-way direction control* constrains an actor's *can rotate* property to *false* and its *affected by gravity* property to *true*.

Events are used for game logic that does not fit into any property or behavior. Events have the form "WHEN <condition> DO <action>". Figure 5 shows what an event will look like when viewed in the *Events* view. Note that events can be modified by adding, removing, or changing conditions and actions, which allows them to be programmed manually, should the need arise. This is the highest level of complexity that PlaySketch will reveal, but it will be kept hidden in the *Events* view so that it will not confuse children under normal conditions.

Notifications and Pre-defined Actors

While these properties, behaviors, and events may seem complex, it is important to remember that children do not need to remember or understand all of them. Children will focus most of their attention on making animations that show how the game works. When PlaySketch makes an inference, it will show a notification like the one in Figure 6. If a child dismisses the notification (by pressing "X") or presses "No", then nothing will change. If they look at the notification but do not understand it, they can press "More…" to learn the meaning of any special terms, see how their actions led to that inference, and make changes. This way, children will learn just enough detail to understand how PlaySketch can help them build their particular game, and they should quickly learn to respond "Yes" or "No" to a notification without too much thought.

Another way children can avoid complexity is by using pre-defined actors (see Figure 4f). These actors come pre-configured with a role and some behaviors. Using pre-defined actors can be much easier than programming by demonstration, but it would limit what children can do, and it could require them to sift through a many options.

There are three key ideas behind this design. First, sketched run-throughs will enable children to demonstrate how their game should function. Second, splitting game logic between properties, behaviors, and events creates three tiers of increasing complexity. Properties are the most common and simplest, behaviors more complex (but not all need to be understood), and events are the most complex and least common. These tiers limit what children must understand to use PlaySketch, because less needs to be known about the more complex tiers. Third, PlaySketch will help children through the process of demonstrating game logic with notifications that explain inferences and pre-defined objects that help them avoid automatic inferences. In the following section, we show how these ideas will play out in practice.

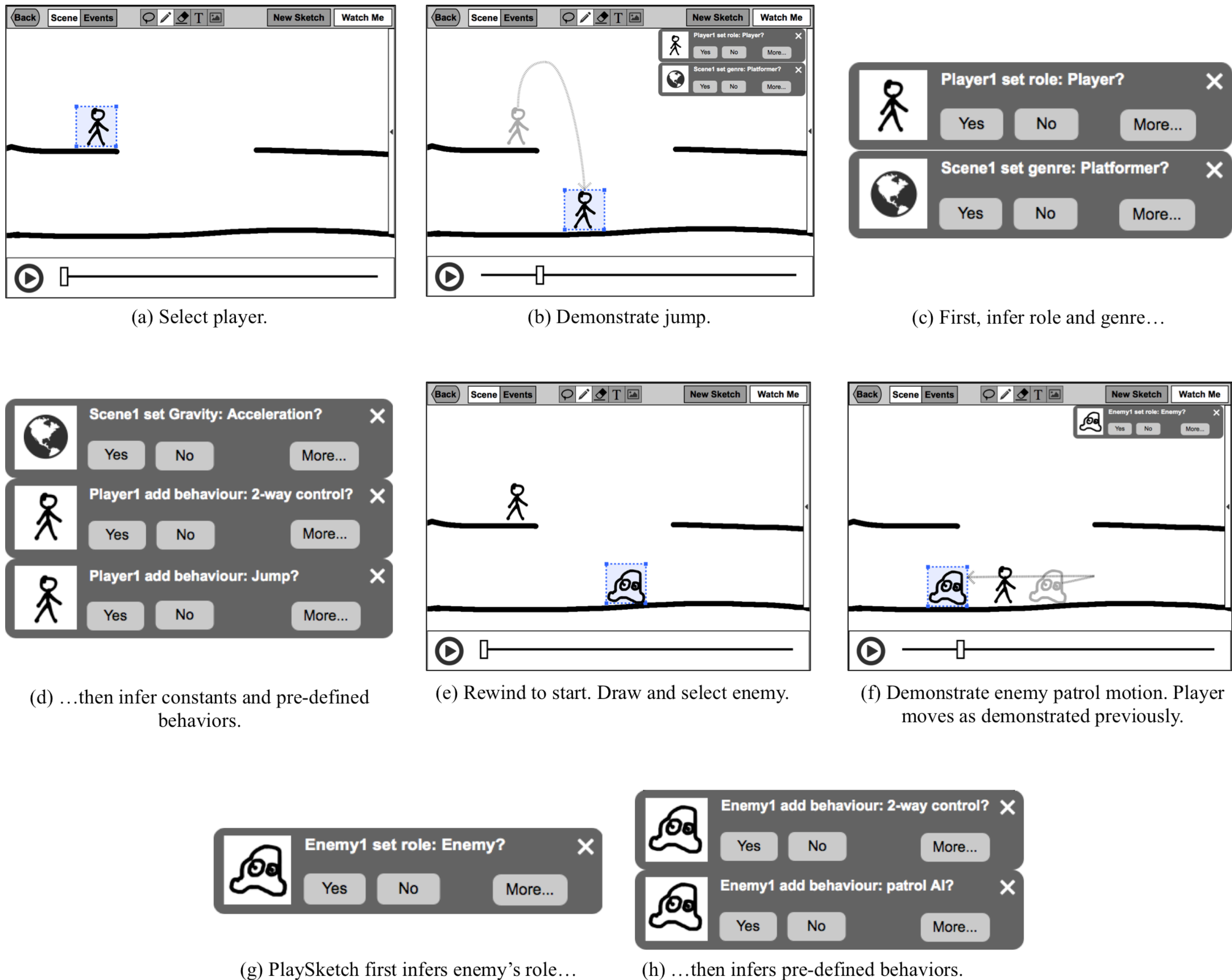
MAKING A PLATFORM GAME WITH PLAYSKETCH

Let's see how the different parts of the PlaySketch user interface will work together in the following scenario. A child wishes to make a platform game with a character that runs and jumps to avoid an monster. When her character picks up a power-up item, the monster will run away from her. In everything that follows, we assume that the child has pressed "Watch Me" and expects PlaySketch to make suggestions.

The child begins by creating a new scene, drawing the platforms and her character, and selecting her character (see Figure 7a). The child then taps the manipulator once and drags the character to demonstrate a jumping motion (see Figure 7b). Tapping the manipulator indicates that the drag will be performed in *run* mode, so this motion is recorded and the time slider advances.

After the motion is recorded, PlaySketch attempts to infer properties, behaviors, or events. Since the scene genre and actor role have not been chosen, PlaySketch tries to infer them from the current state of the game and the interaction history. After examining the configuration of objects and the pattern of motion, PlaySketch suggests the *platformer* genre. Because this actor is the first to be moved, PlaySketch assumes its role is *player*. The child is notified of these suggestions (see Figure 7c) and asked to approve them. If any terms are unfamiliar, the child can press *More* in a notification to learn about the *platformer* genre, the *player* role, and alternatives for each. In this case, the child presses *Yes* to accept both suggestions.

Once the scene genre and the actor role have been chosen, PlaySketch will make more inferences (see Figure 7d). Since the platformer genre has gravity, PlaySketch will use the player's motion to infer gravity's acceleration in this scene. Since most players in the platform genre have a *two-way direction control* behavior, PlaySketch will suggest this behavior to the player. The motion path looks like a jump, so PlaySketch will also suggest a *jump control* behavior. The movement speed and acceleration applied when



(a) Select player. (b) Demonstrate jump. (c) First, infer role and genre…

(d) …then infer constants and pre-defined behaviors. (e) Rewind to start. Draw and select enemy. (f) Demonstrate enemy patrol motion. Player moves as demonstrated previously.

(g) PlaySketch first infers enemy's role… (h) …then infers pre-defined behaviors.

Figure 7: Creating a simple platform game in which a player must jump over a patrolling enemy. The system begins by inferring the game genre and actor roles, and then infers behaviors and physical constants.

jumping are also inferred from the motion path. The child reviews and accepts these notifications. After accepting, the player's motion path changes slightly to make it consistent with the two-way direction and jump controls.

Now it's time to add the monster. The child rewinds to the beginning of the animation, draws the monster on the bottom platform, and selects it (see Figure 7e). Now she taps the manipulator once and records a motion of the monster moving back and forth on the platform (see Figure 7f). The player also moves while she records the monster's movement, but the two actors do not interact, because actors with no role pass through other actors by default. Since the monster is the second actor that moved, PlaySketch suggests that its role should be *enemy* (see Figure 7g), and the child accepts. By default, enemies have a *hurt on touch* behavior that is configured to kill players. This behavior is added automatically, without an additional notification.

After the enemy's role has been determined, PlaySketch tries to infer more about the enemy (see Figure 7h). A *two-way direction control* behavior is suggested, since most characters in the platformer genre move this way. Because the enemy's motion is repetitive, PlaySketch suggests the *patrol AI* behavior, which uses any control behaviors to move a character in a repeating pattern. And as before, the enemy's motion path is used to infer behavior properties: the direction control's movement speed and the AI's patrol path. The child accepts these suggestions as well. Now her game has two characters, movement controls, and obstacles. Her game is already playable!

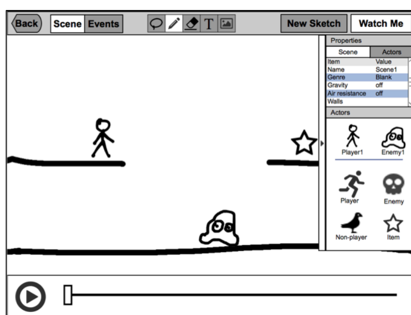
Now the child adds the power-up item that will make the enemy run away. She starts by pressing *New Sketch,* which

creates a new sketch with the same setup as the current sketch but with no recorded activity. To add the power-up, she drags a pre-defined star actor from the actor palette, because the appearance of the power-up item is not important to her (see Figure 8a). The star actor comes pre-configured with the *item* role and the *allow pick up* behavior, which means that players who touch the star will remove it from the scene and add it to their inventory. This role and behavior could also be inferred for a normal drawing by moving the payer on top of it and erasing it when they touch. Using the pre-defined actor saves her the trouble.

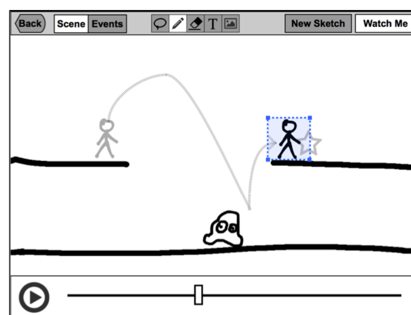
With the item added to the scene, the child can demonstrate the enemy running away after it is picked up. She selects the player, presses the *run* button, and uses the *two-way direction control* and the *jump control* to avoid the enemy and pick up the item (see Figure 8b). She then presses the *run* button again to stop recording, rewinds to the moment when the player picked up the item, and selects the enemy (see Figure 8c). After tapping on the manipulator, she can use the *two-way direction control* to record the enemy's movement away from the player (see Figure 8d). This control moves the enemy instead of the player, because the enemy was selected.

Once the enemy's movement is complete, PlaySketch tries to infer why the child made this change. Since the movement began at the moment when the player picked up the item, PlaySketch infers that this may have caused something to happen. The enemy's motion path is consistent with running away from the player, so PlaySketch suggests adding an *evade AI* behavior to the enemy and an event that switches from *patrol AI* to *evade AI* when the item is picked up (see Figure 8e). After the child accepts these suggestions, the event will be visible in the Events view (see Figure 8f). Events like these are the most complex type of inference that PlaySketch will be able to make, and they are similar to the inference made by Gamut [17].

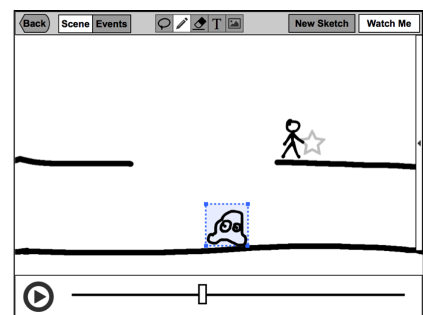
This scenario could continue with the child demonstrating how the item should disappear from the player's inventory after a few seconds and how the enemy should return to patrol behavior. All of these can be demonstrated easily by changing properties or demonstrating motions. The child in this scenario needs to understand only a small number of properties and behaviors and only one event. Notifications help her to understand the inferences that PlaySketch made, and a pre-defined actor give her a short-cut to avoid a longer inference process.



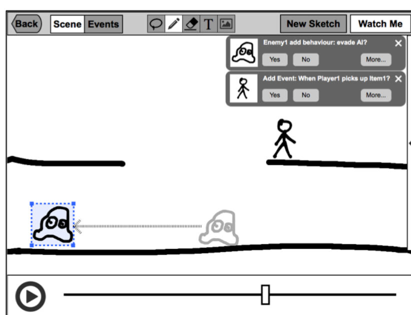
(a) Create a new sketch. Add item from actor palette.



(b) Run game and use player controls to pick up item.

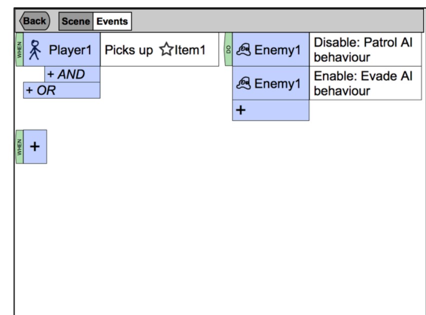


(c) Rewind to moment when item is picked up and select enemy.



(d) Demonstrate enemy moving away from player.

(e) PlaySketch infers a new evade behavior and a rule for switching between behaviors.



(f) The new rule shown in the events view.

Figure 8: Adding a power-up item that makes the enemy run away. Creating a new sketch copies the initial setup from the previous sketch. The pre-defined star actor comes with a role (Item) and some behaviors. Demonstrating the enemy's movement causes an event to be inferred.

FUTURE WORK

There are three challenges that we must overcome to realize the vision presented in this paper. First, we must define a set of properties and behaviors that match children's own categories as closely as possible. Second, we must develop sketch-understanding techniques that will match demonstrated motions to properties and behaviors. Finally, we must develop rules that will allow a programming by demonstration system to infer events from common modifications that children make. We have only begun to tackle these challenges, and we are looking for collaborators.

CONCLUSION

We have presented a preliminary design for PlaySketch, a tool that will allow children to build video games through animation sketching and programming by demonstration. Our preliminary study shows that children take naturally to expressing game behavior with animation sketches. Our user interface design frames the video game building process as a series of animation sketches. Sketches are created by drawing game objects and then demonstrating their movement, by recording a normal run-through of the game, or by modifying an existing run-through.

The game logic that PlaySketch infers will use terms that are similar to children's own categories for such logic. There will be three types of logic. Properties, the simplest type, will include high level attributes like scene genre and actor role, as well as physical constants. Behaviors will be more complex and more varied, but each game will need only a few of them. Events will be the most complex type of game logic, but they will also be the most rare. Splitting inference types into these three levels will limit what children must understand to make sense of the inference process.

PlaySketch will notify children whenever an inference is made. These notifications will give them an opportunity to learn more about what property and behavior terms mean and about how inferences are made. PlaySketch will also give children a set of pre-defined actors that will free them from going through the inference process in some situations.

We are now in the process of defining PlaySketch's properties and behaviors. We hope to find collaborators that can help us build the sketch understanding and PBD components that will help us to turn this vision into reality.

REFERENCES

1. Alvarado, C. and Davis, R. Resolving ambiguities to create a natural sketch based interface. (2001), 1365–1371.
2. Barnes, C., Jacobs, D.E., Sanders, J., et al. Video puppetry: a performative interface for cutout animation. ACM (2008), 1–9.
3. Buckingham, D. and Burn, A. Game Literacy in Theory and Practice. *Journal of Educational Multimedia and Hypermedia 16*, 3 (2007), 323–349.
4. Clickteam LLC. Fusion. http://www.clickteam.com/clickteam-fusion-2-5.
5. Colwell, B., Davis, R.C., and Landay, J.A. *A study of early stage game design and prototyping*. University of Washington, Seattle, WA, 2008.
6. Davis, R.C., Colwell, B., and Landay, J.A. K-sketch: a 'kinetic' sketch pad for novice animators. *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ACM (2008), 413–422.
7. Davis, R.C., Steppe, K., Guan, M., Khoo, J.T., Zhang, R., and Koh, Q.B. Flexible Grouping and Multiple Centers for Preserving Simplicity and Flexibility in Animation Sketches. *Proceedings of The Asia-Pacific Conference on Computer Human Interaction*, (2013).
8. E-Line Media. Gamestar Mechanic. http://gamestarmechanic.com/.
9. Games, I.A. and Squire, K. Design thinking in gamestar mechanic: the role of gamer experience on the appropriation of the discourse practices of game designers. *Proceedings of the 8th international conference on International conference for the learning sciences - Volume 1*, International Society of the Learning Sciences (2008), 257–264.
10. Held, R.T., Gupta, A., Curless, B., and Agrawala, M. 3D Puppetry: A Kinect-based Interface for 3D Animation. (2012).
11. Igarashi, T., Moscovich, T., and Hughes, J.F. As-rigid-as-possible shape manipulation. ACM Press (2005), 1134–1141.
12. Itō, M. and Bittani, M. Gaming. In M. Itō, ed., *Hanging out, messing around, and geeking out : kids living and learning with new media*. Cambridge, Mass. : MIT Press, c2010., 2010, 195–242.
13. Kelleher, C. and Pausch, R. Lowering the Barriers to Programming: A Taxonomy of Programming Environments and Languages for Novice Programmers. *ACM Comput. Surv. 37*, 2 (2005), 83–137.
14. MacLaurin, M.B. The design of kodu: a tiny visual programming language for children on the Xbox 360. *SIGPLAN Not. 46*, 1 (2011), 241–246.
15. MAGiCAL Projects. MissionMaker. http://www.immersiveeducation.eu/index.php/missionmakerm.
16. Margolis, J., Estrella, R., Goode, J., Holme, J.J., and Nao, K. *Stuck in the Shallow End: Education, Race, and Computing*. MIT Press, 2008.
17. McDaniel, R.G. and Myers, B.A. Building applications using only demonstration. *Proceedings of the 3rd international conference on Intelligent user interfaces*, ACM (1998), 109–116.
18. McDaniel, R.G. and Myers, B.A. Getting more out of programming-by-demonstration. *Proceedings of the SIGCHI conference on Human Factors in Computing Systems*, ACM (1999), 442–449.

19. Myers, B.A. Demonstrational Interfaces: A Step Beyond Direct Manipulation. *Computer 25*, 1992, 61–73.
20. Nardi, B.A. Interaction Techniques for End User Application Development. In *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.
21. Pane, J.F., Myers, B.A., and Miller, L.B. Using HCI Techniques to Design a More Usable Programming System. (2002), 198–206.
22. Pane, J.F., Ratanamahatana, C. "Ann", and Myers, B.A. Studying the language and structure in non-programmers' solutions to programming problems. *International Journal of Human-Computer Studies 54*, 2 (2001), 237–264.
23. Peppler, K. and Kafai, Y.B. What videogame making can teach us about literacy and learning: alternative pathways into participatory culture. (2007), 369–376.
24. Quevedo-Fernández, J. and Martens, J.-B. idAnimate: A General-Purpose Animation Sketching Tool for Multi-Touch Devices. (2013), 38–47.
25. Repenning, A. and Ambach, J. Tactile Programming: A Unified Manipulation Paradigm Supporting Program Comprehension, Composition and Sharing. *Visual Languages, IEEE Symposium on*, IEEE Computer Society (1996), 102.
26. Repenning, A. Making Programming Accessible and Exciting. *Computer 46*, 2013, 78–81.
27. Resnick, M., Maloney, J., Monroy-Hernández, A., et al. Scratch: programming for all. *Commun. ACM 52*, 11 (2009), 60–67.
28. Salen, K. Gaming Literacies: A Game Design Study in Action. *Journal of Educational Multimedia and Hypermedia 16*, 3 (2007), 301–322.
29. Scirra, Ltd. Construct 2. https://www.scirra.com/construct2.
30. Scott, J. and Davis, R. Physink: sketching physical behavior. *Proceedings of the adjunct publication of the 26th annual ACM symposium on User interface software and technology*, ACM (2013), 9–10.
31. Shen, E.Y.-T. and Chen, B.-Y. Toward gesture-based behavior authoring. *Computer Graphics International Conference*, IEEE Computer Society (2005), 59–65.
32. Smith, D.C., Cypher, A., and Spohrer, J. KidSim: programming agents without a programming language. *Commun. ACM 37*, 7 (1994), 54–67.
33. Smith, D.C., Cypher, A., and Tesler, L. Programming by example: novice programming comes of age. *Commun. ACM 43*, 3 (2000), 75–81.
34. Sohn, E. and Choy, Y.-C. Sketch-n-Stretch: sketching animations using cutouts. *IEEE Computer Graphics and Applications 32*, (2012), 59–69.
35. Stencyl, LLC. Stencyl. http://www.stencyl.com/.
36. Thorne, M., Burke, D., and van de Panne, M. Motion doodles: an interface for sketching character motion. ACM Press (2004), 424–431.
37. Wolber, D. Pavlov: an interface builder for designing animated interfaces. *ACM Trans. Comput.-Hum. Interact. 4*, 4 (1997), 347–386.
38. Wolber, D.W. and Myers, B.A. Stimulus-Response PBD: Demonstrating "When" as well as "What."In H. Lieberman, ed., *Your Wish is My Command: Programming by Example*. Morgan Kaufmann, San Francisco: CA, 2001, 321–344.
39. Yonemoto, S. A Sketch-based Skeletal Figure Animation Tool for Novice Users. (2012), 37–42.
40. YoYo Games, Ltd. GameMaker. https://www.yoyogames.com/studio.
41. Zhu, B., Iwata, M., Haraguchi, R., et al. Sketch-based Dynamic Illustration of Fluid Systems. (2011).
42. Zimmerman, E. Gaming Literacy: Game Design as a Model for Literacy in the 21st Century. *Harvard Interactive Media Review 1*, 1 (2007), 30–35.
43. Sploder. http://www.sploder.com/.