Coding at the Speed of Touch

Sean McDirmid
Microsoft Research Asia
smcdirm@microsoft.com

Abstract

Although programming is one of the most creative things that one can do with a computer, there is currently no way to make programs on an increasingly popular class of tablet computers. Tablets appear unable to support capable (proficient) programming experiences because of their small form factor and touch-centric input method. This paper demonstrates how co-design of a programming language, YinYang, and its environment can overcome these challenges to enable do-it-yourself game creation on tablets. YinYang's programming model is based on tile and behavior constructs that simplify program structure for effective display and input on tablets, and also supports the definition and safe reuse of new abstractions to be competitive with capable programming languages. This paper details YinYang's design and evaluates our initial experience through a prototype that runs on current tablet hardware.

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

General Terms Human Factors, Languages

1. Introduction

Kay's Dynabook [9, 10] envisioned how portable tablet-sized devices would revolutionize computer usage through accessible do-it-yourself (DIY) programming. Tablets are now becoming mainstream but most existing programming languages still rely on touch-unfriendly keyboard input to manipulate text-based abstractions. Although visual programming languages support metaphors that are potentially touch-friendly, they suffer from poor input efficiency and a *scaling up problem* [2] that inhibits productivity. A DIY tablet programming language should be both usable on tablets as well as capable of productive programming.

Onward! 2011, October 22–27, 2011, Portland, Oregon, USA.

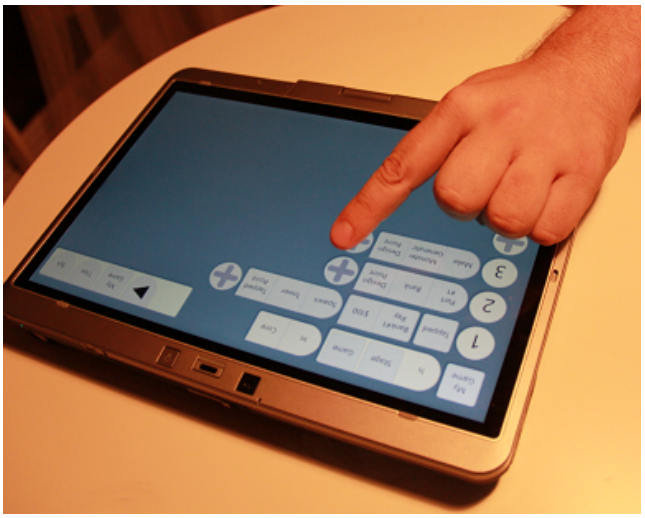


Figure 1. YinYang in use on a tablet.

Do we really need capable programming experiences on tablet devices? Given their portability and accessibility, tablets are increasingly used to create content, not just passively consume it. Their smaller screen and limited input capabilities require focused experiences with minimal features, which incidentally makes the experiences more usable; programming can benefit from a similar focus. On the other hand, computers with keyboards are ubiquitous today and programming without them seems unnecessary. Tomorrow, however, keyboards could become niche devices that are largely replaced by touch-based devices, adversely turning programming into a more niche activity [3]. Programming should become more accessible, not less, on whatever devices consumers use in the future.

Our quest to design a tablet programming experience began by studying Kodu [12], which allows children to create games for consoles on consoles. Kodu's core syntactic units are square *tiles* that are selected through context menus using the game pad. Tiles are then arranged into concurrent prioritized behaviors to define autonomous robot-like objects as inspired by Brooks' work on behavior-based robotics [1]. The tile and robot metaphors allow children to construct fun games without writing very much code. On the other hand, Kodu focuses on education where usability trumps capabil-

ity, and so programmers are limited to using a fixed grammar of built-in tiles [23].

Our language, YinYang, heavily borrows from Kodu's design being based on tiles that are structured into behaviors. However, YinYang focuses on the DIY programming of games where some usability can be sacrificed to support the abstract thinking needed for more productive programming. To this end, YinYang is open to the definition of new tiles with an object-oriented type system that ensures tiles are used in safe and meaningful ways. The type system also provides the "context" needed to generate and organize concise context menus used to edit, which cannot be organize by hand because YinYang is open to new tiles. Finally, an object's overall behavior is formed by composing multiple tiles together, where behaviors from different tiles are prioritized in a way that programmers can locally reason about.

Figure 1 shows a photo of YinYang being used on a tablet. Touch input is supported in part through language design: YinYang's syntax is very simple and focuses mostly on how tiles are arranged; YinYang code is fairly flat allowing for efficient display; and the programming model overall leads to less code to input. Additionally, the programming interface improves efficiency through menu organization, allowing programmers to make not-yet correct edits, and by allowing programmers to look beyond the given context when searching for tiles to use. Through very informal tests on our current prototype, inputting a program in YinYang with touch is still about 66% slower than using a keyboard to type the same program, which indicates that we are not too far away from our goal.

The rest of this paper proceeds as follows. Section 2 describes the design of the YinYang language while Section 3 describes how this language is paired with a touch-friendly tablet-based editing environment. Section 4 discusses YinYang's semantics as well as how it is implemented. We discuss our initial experience with a prototype in Section 5 and concurrently suggest directions for future work given lessons learned. Section 6 presents related work while Section 7 concludes.

2. Language Design

YinYang is a graphical language whose design supports tablet-based program construction with two core features. First, control flow is based on Brooks' *subsumption architecture* [1] where overall object behavior emerges from multiple arbitrated simple behaviors and overall program behavior emerges from multiple interacting autonomous objects. Second, one *tile* construct can abstract both verbal "doing" and nounal "being" logic, and is a type when checking that abstractions are used meaningfully. These features simplify YinYang's syntax, reduce how much code is written, and provide context that allows editing to be more usable and efficient. The rest of this section informally describes how to understand YinYang code; we discuss the code editing experience in Section 3 and semantics and implementation in Section 4.

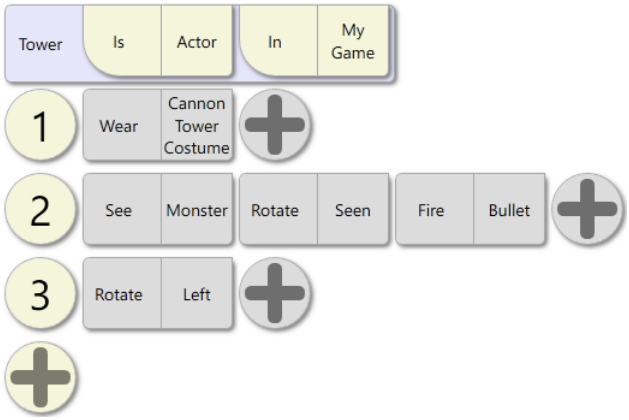


Figure 2. The definition of a `Tower` tile; the screenshot is scaled down 60% of its actual size.

Behavior-based Programming

As shown in Figure 2, YinYang's interface is dominated by square and circular boxes that contain text or simple symbols. These boxes are *tiles* that are YinYang's core unit of syntax and semantics. YinYang is object-oriented: a tile is invoked by an object to cause an object to do or be something. The YinYang code in Figure 2 defines a `Tower` tile that causes an invoking object to become a tower; the syntax used in this code is shown in Figure 3. A tile definition begins with a declaration that lists the tile's name and tiles that must be in the type of an object ("(is)") before the object can invoke the tile; e.g., a `Tower` tile can only be invoked by an object whose type contains the `Actor` tile. The tile also specifies what kind of objects must contain ("(in)") the invoking object; e.g., an object must be contained in a `My Game` object to invoke the `Tower` tile. Following a tile's declaration is its body of YinYang code that executes when the tile is invoked. This subsection describes how code in a tile's body is understood while we defer a discussion of abstraction and typing, which is related to the tile's declaration, to the second part of this section.

As described in Figure 3, an expression in YinYang resembles a simple method call whose head is a direct reference to a tile, meaning tile references are not encoded by name. An expression can have arguments: if an expression has one argument, the argument is graphically arranged flush to the expression's right; if the expression has more than one arguments, the arguments are embedded in the expression to the right of it's head. A top-level expression forms an *act* that represents an individual command and/or query, where multiple acts are then arranged horizontally on a circled numbered line to form a *behavior*. Consider behavior 3 of the `Tower` tile defined in Figure 2:

```
Tile        ::= Declaration new-line Body
Body        ::= Behavior ⊕
Behavior    ::= (#) Act ⊕ new-line
Act         ::= Expression
Expression  ::= head-tile-ref Expression
Declaration ::= tile-name Is In Args
Args        ::= argument-name type-tile-ref
Is          ::= (is) extended-tile-ref
In          ::= (in) included-tile-ref
```

Figure 3. YinYang's abstract syntax for the code in Figures 2 and 4; terminals are uncapitalized; literals are italicized; over-bars indicate zero-to-many repetition.

This behavior has only one act whose expression is headed by the `Rotate` tile. An object executes a behavior by executing its acts where executing an act involves invoking the tile that heads the act; e.g., an object executes the act in this example by invoking the `Rotate` tile, causing it to rotate left. Expressions that are not acts are evaluated to yield existing or new objects; e.g., the argument of the `Rotate` act is a `Left` expression whose evaluation yields an object that is passed to `Rotate`'s implementation. An object executes the behaviors of a tile when it invokes the tile; inside the tile's definition this object is referred to as the implicit *executing object*, which is comparable to the object that is bound to `this` in a Java method.

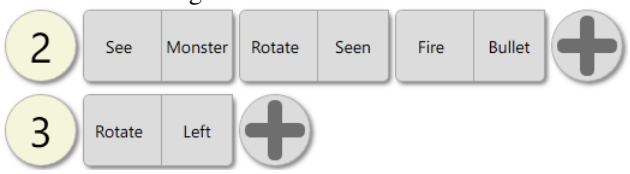
A behavior executes continuously and in parallel with respect to the executing object's other behaviors. In Figure 2, an object invoking the `Tower` tile is, in parallel, continuously wearing a cannon tower costume (behavior 1), trying to see monsters (behavior 2), and rotating left (behavior 3). The last behavior is animated: rotation causes the executing object's orientation to be updated every display frame according to the object's rotational velocity and the current frame rate. Within a behavior, acts to the right execute only on the *successful execution* of acts to the left, where successful execution is determined by the heading tile's implementation, usually according to a sensor reading or goal achievement. Consider behavior 2 in Figure 2:

The first act in this behavior computes whether the executing object can see some monster while the second act instructs the object to rotate toward the monster that was seen. If the object cannot see a monster, then execution of the `See` act fails and the `Rotate` act is not executed. When the object sees a monster, the `See` act's execution succeeds, the `Seen` tile is bound to the monster that was seen, and the `Rotate` act begins to execute, which in turn causes the object to begin orienting itself toward that monster. Once the object is oriented toward the monster, the `Rotate` act's execution succeeds and the `Fire` act begins executing, causing the object to fire bullets at the monster. If behavior 2 existed in `Tower`'s definition without behavior 3 in Figure 2, it could conceptually be translated into the following pseudo code:

```
while (true)
  if (See(Monster, &Seen))
    if (Rotate(Seen)) Fire(Bullet);
```

A behavior is always re-executed from its first act since an act whose execution is succeeding could stop succeeding at any moment while it could still have useful work to do even while it is succeeding. In our example, execution of the `See` act will stop succeeding of the seen monster disappears while the `Rotate` act will attempt to maintain the object's alignment with a moving monster even after it is initially aligned with the monster. Note that tiles that do not specify success conditions in their definitions, such as the `Tower` tile in Figure 2, succeed automatically when they are executed as acts. In this paper, only tiles that are implemented in C# such as `Rotate` or `See` have non-trivial success conditions; Section 4 describes how success conditions can be defined in YinYang code.

Conflicts that arise between behaviors are resolved with a prioritization scheme. Consider behaviors 2 and 3 in `Tower`'s definition from Figure 2:



When the `See` act in behavior 2 succeeds in its execution, both `Rotate` acts in behaviors 2 and 3 are able to execute. However, the `Rotate` tile is defined to execute *exclusively* because an object cannot rotate toward more than one orientation at the same time. In YinYang, if two acts in two behaviors can execute but execute the same exclusive tile, then the act in the higher-priority (lower numbered) behavior executes while the act in the lower-priority (higher numbered) behavior immediately fails in its execution. In our example, behavior 2 has a higher priority than behavior 3, so when the executing object sees a monster, it stops rotating left and instead starts rotating toward the seen monster. The combined behavior instructs the object to scan for monsters in its limited field of vision by continuously rotating left, and when a monster is seen, to rotate toward that monster so that the object can fire at it. The combined behavior can be realized with an `else` condition in the following pseudo code translation:

```
while (true)
  if (See(Monster, &Seen)) {
    if (Rotate(Seen)) Fire(Bullet);
  } else Rotate(Left);
```

Section 4 will describe how behavior execution is more efficient than this pseudo code translation implies; behavior re-execution occurs only when the re-execution could change the state of the program.

Because of its behavior-based programming model, YinYang's syntax is simpler and its code is flatter (i.e., less nested) as it avoids the diverse and nested control flow constructs such as if/else conditionals, event handlers, switches, loops, and so on. A simpler syntax makes it easier to input programs on a tablet while flatter code allows for more effective use of a smaller screen. Finally by mandating continuous execution, YinYang can more concisely express the reactive computations that are common in the game and user interface programs. Reducing the code needed to express a program is desirable in any context but crucial on an input-constrained tablet. On the other hand, YinYang is overall less flexible in the kinds of programs that it can express effectively; it is very limited in its support for the imperative computations or forming complex arithmetic expressions; Section 5 discusses the impact of these trade offs.

Abstraction and Typing

Figure 4 demonstrates how object behavior can be encoded in multiple tile definitions for better modularity and reuse. "Attack" logic previously encoded directly in the `Tower` tile of Figure 1 is abstracted as the `Attack` tile in Figure 4 and used in a new definition of the `Tower` tile (also Figure 4). To complement the ability to define new tiles, YinYang statically enforces that tiles are only invoked in meaningful contexts according to types that are formed from tiles. Using the (is) term, a tile can **extend** other tiles, which must be successfully executing on an object before the tile can be invoked on that object. The static type of an object then consists of the tiles that must be successfully executing on the object in order to reach some point during execution.

If a tile extends another tile, then any object executing the tile's body must also be successfully executing the extended tile. For example, the `See`, `Wear`, and `Rotate` tiles extends the `Actor` tile, meaning that invoking objects must be a graphical and animated "stage actor." Because the `Tower` tile in Figure 4 also requires the `Actor` tile, the type of an object executing the `Tower` tile's body must contain the `Actor` tile; as a result, acts can be formed in `Tower`'s definition from the `See`, `Wear`, and `Rotate` tiles. An object also includes the type of the tile whose body it is executing because the body is not executed until the tile's success condition is true. As a result, the `Attack` tile can be used to form an act inside the `Tower` tile. Tile arguments are also constrained with tile requirements; e.g., the `Attack` tile is declared with a `Target` argument that must be bound to an object whose type contains the `Actor` tile, which is satisfied by the `Seen` tile in `Tower`'s use of the `Attack` tile.

The typing properties of tiles allow them to effectively encapsulate class-like nounal "to be" logic in addition to method-like verbal "to do" logic. For example, an object that

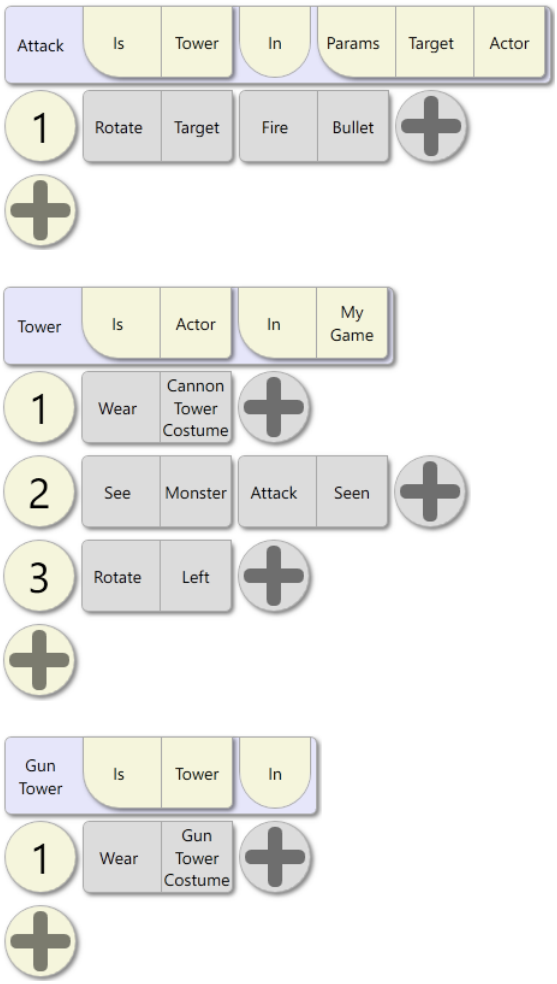
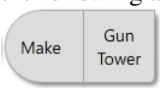


Figure 4. Definitions of the `Attack`, `Tower`, and `Gun Tower` tiles.

invokes the `Tower` tile in Figure 4 not only executes tower behavior, its type is also enhanced so that it can invoke tower-extending abstractions like the `Attack` tile. Tiles further resemble classes in the way that tile extension acts like class inheritance in that a tile can override behavior that is defined in the tiles it extends. Programmers can locally reason about overriding with tile extension in a way that similar to class inheritance: given a tile A that extends tile B, an act of a behavior defined in A will always execute with a higher priority than an act of a behavior defined in tile B. For example, because the `Gun Tower` tile in Figure 4 extends the `Tower` tile, the act in behavior 1 of `Gun Tower` will always execute with a higher priority than the act in behavior 1 of `Tower`. As a result, a gun tower object will wear a gun tower costume rather than the cannon tower costume that would otherwise be worn by a tower object. Although overriding can locally be understood by the programmer in terms of tile extension, it is realized globally during execution through a somewhat surprising right-to-left prioritization scheme that will be detailed in Section 3.

An new object is created with a *root behavior* that defines the top-level acts that the new object will execute. Consider the following act that creates a gun tower object:

`Make` is a built-in tile (keyword) that creates a new object given a root behavior that the new object will execute. In this example, the `Actor` and `Tower` acts of the root behavior must execute before the `Gun Tower` act since the `Gun Tower` tile, as declared in Figure 4, extends the `Actor` and `Tower` tiles. As syntactic sugar, acts that invoke extended tiles can be omitted in an object's root behavior given their verbosity; the following act is then equivalent to the former:



`Actor` and `Tower` acts are automatically added to this root behavior when the `Make` act is de-sugared.

The ability to define abstractions makes YinYang a real programming language: programmers can modularize code to reduce repetition, reuse code from other parties, and form libraries for others to reuse. On the other hand, the unification of class-like and method-like behavior in tiles presents interesting usability problems: sometimes a method-like tile should be suggested before a class-like tile, and vice versa. Section 4 addresses how tiles are enhanced to deal with this and similar usability issues.

This section concludes with a complete simple game that is formed from the tiles in Figure 4 combined with additional tiles defined in Figure 5. In this game, the player starts out with $100 in a bank and can build a gun tower by tapping the game's stage (display surface) and paying $100. The game has a monster generator that creates a new monster every 2 seconds that wanders around while towers fire at it. When a monster is hit by a bullet, the player earns $10 while the monster stops moving, performs a splat animation, and then disappears. The player can build additional gun towers with the money they earn from killing monsters. The code in Figure 5 uses several features that are not core to YinYang's tablet-supporting design but are still necessary to define complete programs:

- The `Game` tile represents the entry point of a game; because the `My Game` tile extends the `Game` tile, it can be executed as a program.
- A tile can define parts as embedded objects that are exported from an executing object; e.g., the `My Game` tile declares an exported `Bank` part to keep track of how much money the user has earned and spent.
- A tile definition can access the parts of the tiles it *includes* (ⓘⓝ), where the executing object must be contained by an object that is executing the included tile. The ⓘⓝ operator replaces the need for definition nesting as used in most other languages. For example, the `Monster` tile includes the `My Game` tile to access its bank part, while the `Monster Generator` tile includes the `My Game` tile because it creates monster objects.
- The built-in `Spawn` tile creates a new object that will exist even after the successful act executions that trigger the `Spawn` act cease being successful. For example in the `My Game` tile, a new gun tower object is created whenever the user taps the game's canvas and $100 can be payed from the bank part. The new gun tower object remains after the user lifts their finger and the tap ends.
- The built-in `Start` tile executes a behavior on the executing object that will continue executing even after the successful act executions that trigger the `Start` act cease being successful. For example in the `Monster` tile, a monster object will start invoking the `Monster Dying` tile when it has been hit by a bullet. The object continues to invoke this tile after the monster is no longer being hit by the bullet.
- Some tiles can be invoked on an explicitly referenced object that is not the object that is executing the enclosing tile definition. For example, the `Pay` and `Earn` tiles, which command bank objects, can be invoked on the game's `Bank #1` part from outside of this part's root behavior. Tiles that are invoked through an object reference must be recursively non-exclusive since they cannot be meaningfully prioritized (Section 4).

3. Editing with Touch

The design of an effective tablet-based programming interface must mitigate the tablet's poor support for text input and small display size with its strength in support for direct touch. With this in mind, YinYang is a graphical language whose code is rendered in a way that supports pervasive tapping as well as a dense yet quickly scannable layout. Editing is made more efficient by leveraging YinYang's type system to generate concise **context menus** that help programmers find and select options with a speed that is competitive with typing them out. YinYang's editor also aims to be less constraining by allowing programmers to make invalid edits if the program can be fixed to accommodate the edit. Finally, YinYang fully leverages its graphical nature by supporting custom graphical code editing when appropriate. The rest of this section details how these features are realized in YinYang's programming interface.

The design of YinYang's interface is informed by two human computer interaction principles. First, Fitts' Law [5] predicts that the time needed to select an object depends on distance to the target and its size. Second, Hick's Law [7] describes the time it takes for a person to make a choice based on how many choices are visible. Kent applies these principles in an analysis of menu selection [17], which also influences YinYang's design.

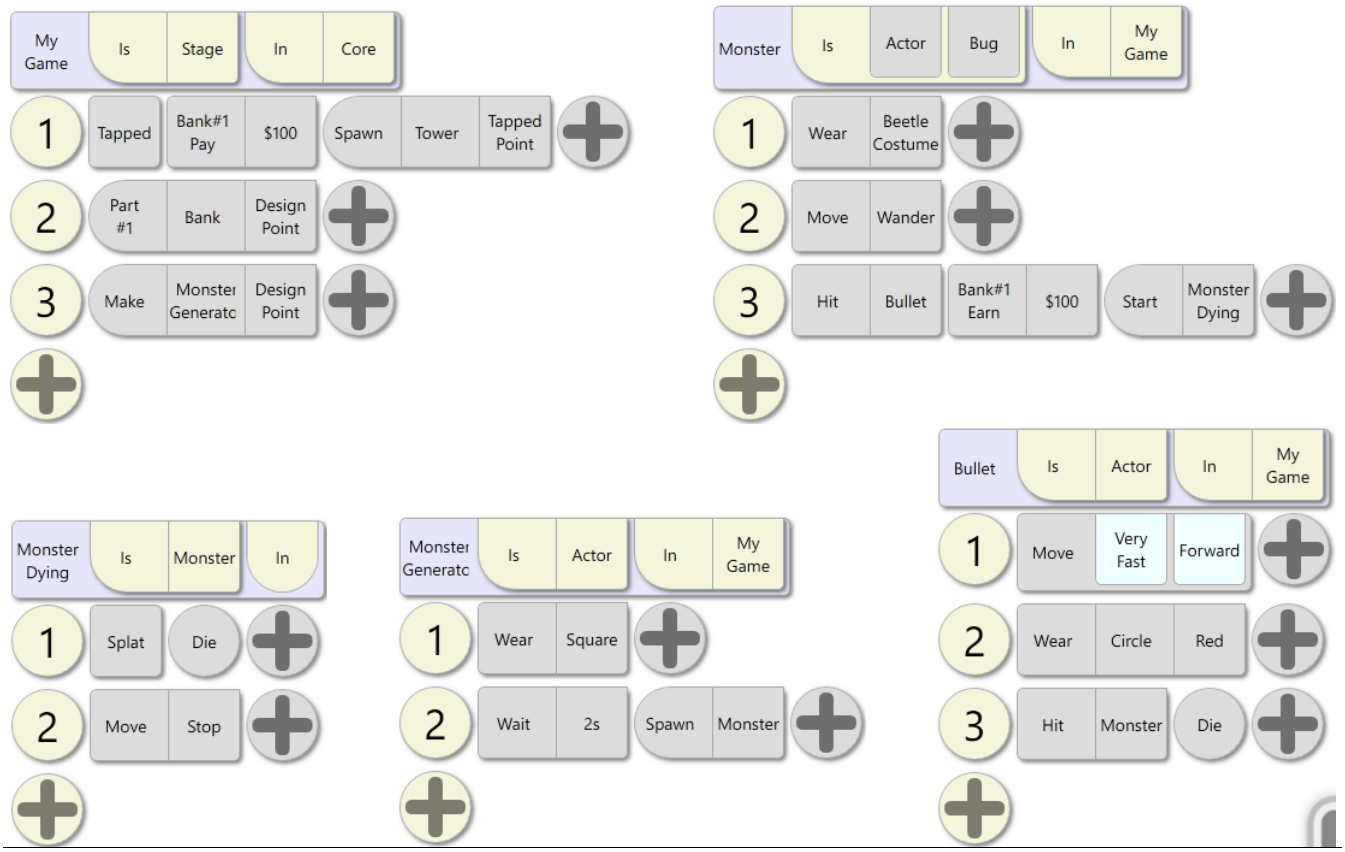


Figure 5. Definitions of the `My Game`, `Monster`, `Dying Monster`, `Monster Generator`, and `Bullet` tiles that together with the tiles in Figure 4 define a simple complete game.

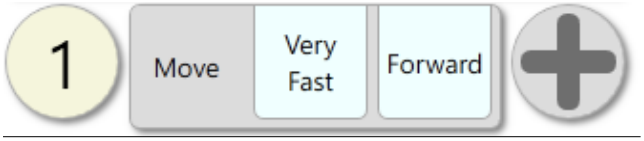


Figure 6. A YinYang behavior presenter; `Move` labels an embedded "act" presenter; `Forward` and `Very Fast` label two argument presenters embedded in the act presenter; the screenshot is unscaled.

Presenters

Figure 6 demonstrates how YinYang code is rendered. A tile or keyword expression in YinYang is rendered with a *presenter* that starts with a finger-sized ($\sim$ 1.5 cm^2) unobstructed *tap space* for easy tapping, where the larger target can be more quickly touched as informed by Fitts' Law [5]. The ability to directly tap any presenter eliminates the cognitive distance between input and output; no cursor is required as programmers simply tap on the presenter of an expression they want to manipulate.

A presenter's width increases beyond its tap space to contain nested argument expressions. Likewise, a presenter's height increases a few pixels per level of nesting that occurs inside the presenter so that expression tree structure is visually apparent. As a visual optimization, flush right-adjacency is used to display an expression's only argument; otherwise argument expressions are spaced and embedded in the presenter. Because YinYang code is typically not very nested, lines of presenters (e.g., displayed behaviors) can often be laid out vertically without wasting very much space. A text label is displayed in a presenter's tap space with an extra large font size ($\sim$ 14 points) so that programmers can scan more quickly through dense arrangements of presenters. By convention, tiles are named with up to three short words to allow their names to fit in a presenter's tap space.

Additional presenters are placed in the editor as placeholders for editing tasks that are not directly anchored to existing code. The insertion (⊕) presenter at the end of the behavior in Figure 6 is used to append a new act to the behavior, while another insertion presenter is used to append a new behavior to the end of a tile body. Each behavior is also preceded by a numbered placeholder (① in Figure 6) that identifies its priority and can be tapped to delete or move the behavior.

Context Menus

To use space effectively and utilize the rich context provided by YinYang's type system, editing in YinYang is based

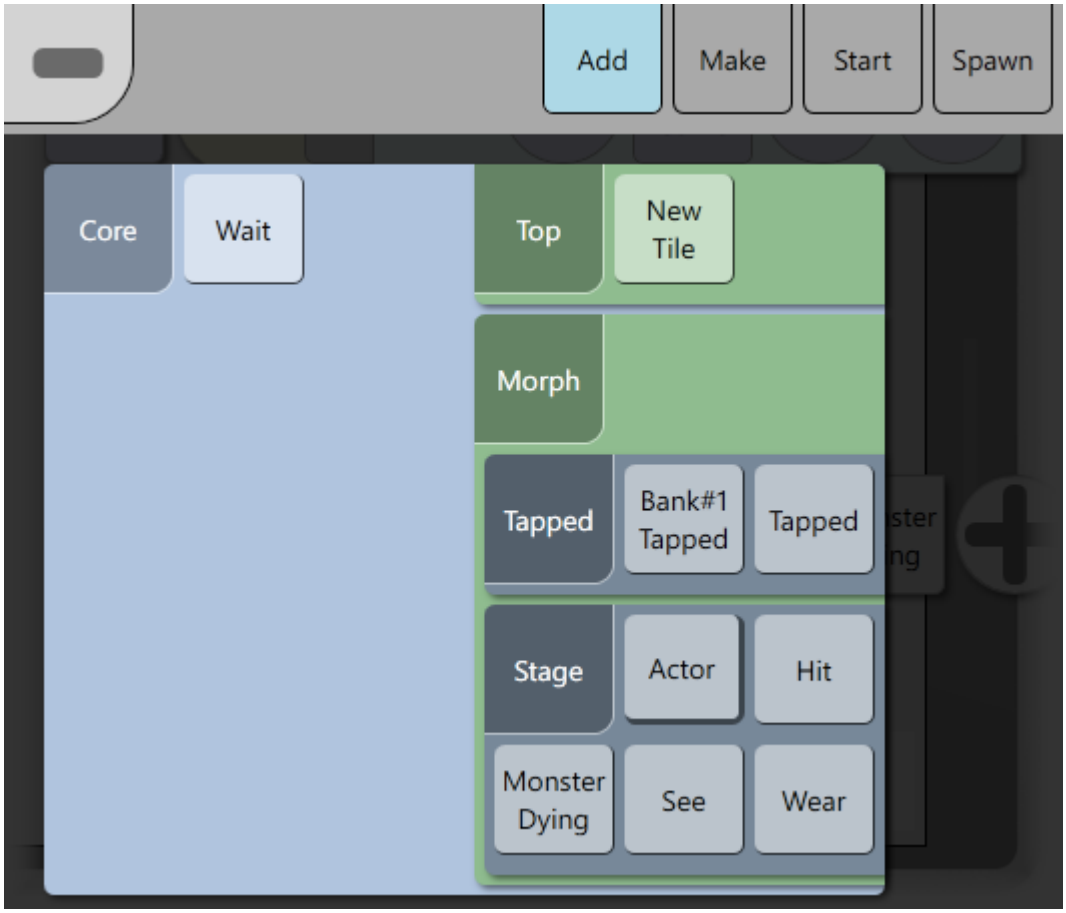
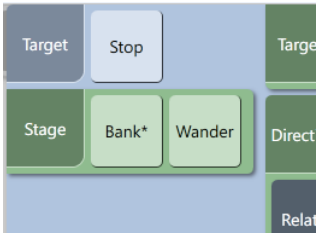


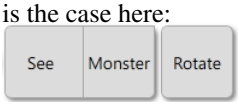
Figure 7. The root context menu (unscaled) for adding a new act in a `Monster` tile that extends the `Actor` tile. When the menu is active, programming content is grayed out. The menu has two parts: a top-positioned bar of menu modes, actions, and menu-wide options, and a menu of options that appears close to where the user tapped. The top level of `Core` panel of the menu contains the second-level `Top` and `Morph` panels; the second-level `Morph` panel contains the third-level `Tapped` and `Stage` panels.

on *context menus* (aka popup menus) that open whenever programmers tap presenters. A context menu presents tiles for an edit that are filtered according to the context at the tapped presenter's expression (or placeholder). The context of a menu is formed from what tile is being defined, what tiles are extended and included by the defined tile, and, if in a behavior, what acts are to the left of the tapped expression. For example, the context menu for an insertion (⊕) presenter will contain tiles that can be used to form a new act at the insertion's location according to what tiles are known to be in the executing object's type at that location. Figure 7 shows such a context menu for the definition of the `Monster` tile in Figure 5 that extends the `Actor` tile, but cannot include the `Attack` tile since `Monster` does not extend the `Tower` tile. As another example, consider the following behavior:

Tapping on the `Rotate` presenter will activate a menu that includes rotation directions:



The `Seen` tile does not appear in this menu because there is no `See` act successfully executing before the `Rotate` act, as is the case here:



Tapping on the `Rotate` presenter will now activate a menu that contains the `Seen` tile because the `See` tile is now in the context of the edit:

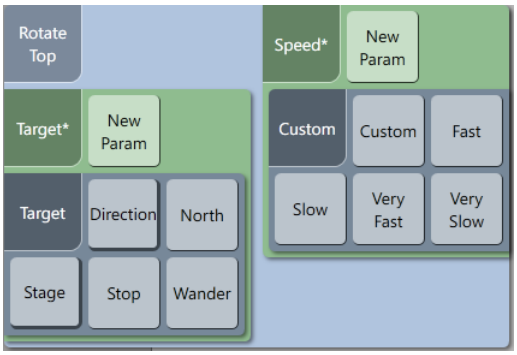
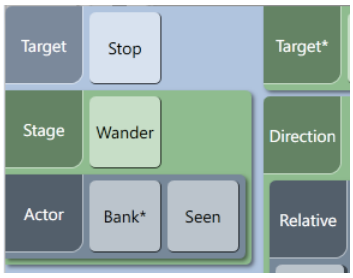


Figure 8. A context menu that appears after a `Rotate` tile has been selected to form an act; argument binding tasks are expanded as panels in this context menu.



How context is formed for an edit is described in Section 4.

As informed by Hick's Law [7], if the tiles of a context menu are ordered alphabetically then finding a desired tile can be at best logarithmic. However, programmers might not know the name of a tile, the name might not be distinct or informative enough, and there still might be too many tiles to fit into a single menu. As a result, YinYang organize tiles into a *semantic hierarchy* based on the tiles they extend and include. Context menus can then present tiles hierarchically with sub-menus that avoid overwhelming programmers with too many unorganized tiles in one flat menu. For example, if the `Monster` tile extends the `Actor` tile, it appears under the `Actor` sub-menu in a context menu, which opens above the context menu when activated.

Unfortunately, each level of hierarchy in a menu has a significant cost [17]: an extra tap; an additional scan of the new menu's option; and the need to backtrack if a mistake is made. YinYang displays multiple levels of menu hierarchies using *tree maps* [22] that visualize hierarchical information using nested rectangles. Tile hierarchy is expanded to fill a six-by-six panel (36 options) where expanded hierarchy boundaries are preserved by panel nesting as shown in Figure 7. To avoid confusing programmers with either too much expanded hierarchy or too many options in one level, only two additional levels of hierarchy beyond the one root level are expanded, while each expanded panel is limited to five unexpanded options. The left-top title of a nested panel can be tapped to activate a sub-menu to reveal all of its options.

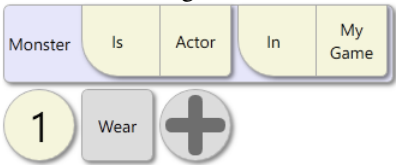
To further streamline editing and improve input efficiency, common editing tasks can appear in succession so programmers can more rapidly perform sequences of edits. For example, when the programmer creates a new `Rotate` act, the context menu does not close. Rather, additional editing tasks are shown on the menu to bind `Rotate`'s argument as shown in Figure 8. The rotate tile has two arguments, a mandatory `Target` argument and an optional `Speed` argument, that are expanded in two panels on the context menu. The menu will only dismiss itself when no common editing tasks remain; the programmer can also dismiss the menu manually and reactivate later by tapping the edit site again.

Freedom and Fuzziness

The primary advantage of a graphically-edited language over a textually-edited language is the ability to guide programmers in making syntactically and type correct edits via a graphical interface. Unfortunately, programmers often dislike graphical languages because they constrain the order in which they can make edits, which disrupts their focus [11, 15]. For example, a symbol must typically be defined before use even if the programmer would rather use the symbol first and define it later to preserve their focus on the current code. In contrast, free-form text editing allow programmers to write incorrect code that they can fix later at their convenience.

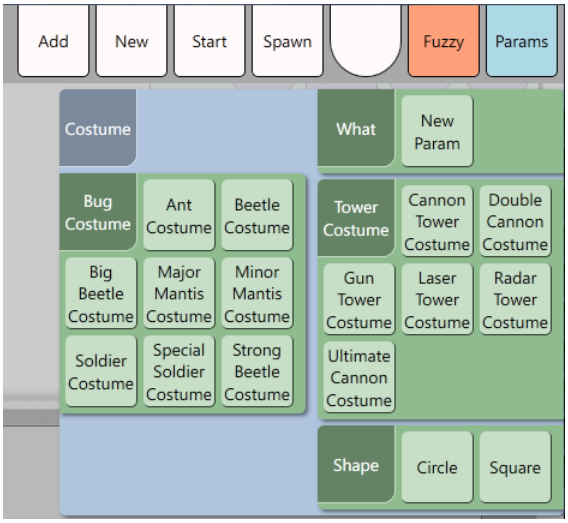
YinYang uses two techniques to give programmers more freedom during editing. First, context menus provide the option of defining a new tile to satisfy an edit. The programmer does not specify the new tile's name and can later, at their convenience, navigate to the new tile by tapping its use to fill in its name and body, which is already primed by the context of the edit. In Figure 7, the programmer can choose to define a new tile rather than select an existing tile to form an act; specify a binding for the new act by creating and binding a new argument; and then later name the tile "`Attack`" and add attack logic with its extended `Tower` tile already filled in along with its `Target` argument.

Second, each context menu can become **fuzzy** on demand so that it presents tiles for selection even if they are not exactly appropriate for the edit's context. Instead, selecting such a tile requires making multiple edits to the program that might not be local to the edited code. As an example, consider writing the code for the `Monster` tile from Figure 5 in Section 2, starting with this definition:

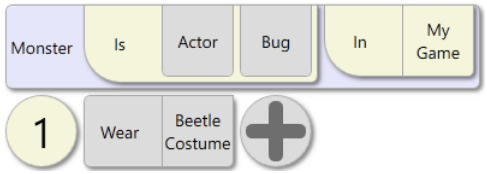


The programmer opens the `Wear` act's context menu to specify its costume:

The programmer wants the monster to look like a bug, but does not see the appropriate costume in the menu. Bug costumes require, with an include requirement, that wearing actors also extend the `Bug` tile so that special animation behavior (`Splat`) can be accessed in an actor that is wearing the costume. However, since the `Monster` tile has not yet extended the `Bug` tile, bug costumes are not visible in this context menu. By selecting the `Fuzzy` option, additional costumes become visible in the menu:



By selecting a bug costume, the definition of `Monster` is automatically updated to extend `Bug`:



Because the `Monster` tile now extends the `Bug` tile, bug-specific tiles like `Splat` will appear in `Monster` act menus without needing to select the fuzzy option.

YinYang's type system enables fuzzy edits with its simplicity and support for expressing semantic tile relationships. Enhancing context automatically in YinYang is straightforward: the additional context requirements are propagated up as new extend and include clauses. Care must be taken to rank fuzzy choices appropriately according to the existing context. Only basic heuristics are currently applied to these rankings; Section 5 will discuss how the organization of these menus could be improved.

Custom Editing Support

Beyond using context menus uniformly in the editor, YinYang also supports the versatility of a visual language to accommodate customized editing experiences through two facilities. First, a tile can be defined with a configuration menu that is activated when the tile is chosen in a context menu. For example, when the `Custom Amount` tile is selected, a virtual keypad-like widget pops up as a configuration menu where the programmer can conveniently input a dollar amount. Other configuration menus can include sliders, radial widgets, and any other kind of user interface widget. After a tile has been used in an act, the `Update` option in the context menu of the use allows it to be edited through its custom configuration menu.

Second, a tile can define custom behavior for use in the *designer* of a program to enable editing by direct manipulation [21]. For example, extending the `Game` tile creates a phone-like canvas in the editor of the tile definition that supports the direct addition and position of actors. In the `My Game` tile of Figure 5, the programmer can tap on the canvas of the game in the designer to add and position a `Bank` actor object.

4. Technical Overview

YinYang involves interesting technical details in two areas. First, YinYang's **type system** ensures tiles are used in proper contexts, which is essential in supporting the reliable definition of new tiles. Whereas type checking ensures that an expression is correct, YinYang's type system can also provide a list of all tiles, possibly unbounded given the context of a menu used for editing. Second, YinYang's **execution engine** executes a program continuously with an emphasis on correct prioritization so programmers can locally reason about how programs will behave. Behavior execution eventually reaches **native tiles**, which are written in C#. This section describes YinYang's type system and execution engine.

Types

As discussed in Section 2, an object is typed by the tiles that it is currently executing successfully. Statically, a tile `E` is known to be successfully executing on the executing object at position `P` in `T` if:

- Position `P` is an act in a behavior, and `E` forms an act that occurs earlier in that behavior.
- `E` is extended directly or indirectly by tile `T`.

We refer to the tiles that an object is statically known to be executing at position `P` by these criteria as the set $\overline{\texttt{E}}$. Additionally, we refer to the set of tiles that tile `T` includes (ⓘ) as $\overline{\texttt{I}}$, which specifies what tiles that the executing object's containing objects must successfully be executing. Given a tile `S` that extends a set of tiles $\overline{\texttt{F}}$ and includes a set of tiles $\overline{\texttt{J}}$, `S` can be used to form an act at `P` if $\overline{\texttt{F}} \subseteq \texttt{T} \cup \overline{\texttt{E}}$ and $\overline{\texttt{J}} \subseteq \overline{\texttt{I}}$; i.e., the extended tiles of `S` are known to be executing successfully on the executing object while `S`'s included tiles are known to be executing in containing objects.

Creating a new object is a bit different. Given that the new object will be contained by the executing object, `S` can

be used in the root behavior of an object-creating expression if the tiles it includes ($\overline{\texttt{J}}$) are either executing in the executing object or a containing object ($\overline{\texttt{J}} \subseteq \texttt{T} \cup \overline{\texttt{E}} \cup \overline{\texttt{I}}$). The tiles extended by `S` ($\overline{F}$) exist as acts in the root behavior before tile `S` is used; however, as mentioned in Section 2, these acts can be omitted for brevity, in which case they will be inserted automatically by the runtime.

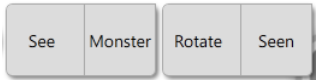
An expression used as an argument whose type is tile `R` must evaluate to an object that is successfully executing tile `R`. An argument expression is one of the following:

– An object-creating expression in the form just mentioned; an explicit `Make` tile does not need to be specified. For example, consider:

The `Beetle Costume` expression evaluates to a new object with `Beetle Costume` in its root behavior. The new object then acts as an argument to the `Wear` act's `Costume` argument, which is valid as the `Beetle Costume` tile extends the `Costume` tile. The `Beetle Costume` tile itself includes `Bug` tile, which is satisfied by the `Monster` tile because it extends the `Bug` tile;

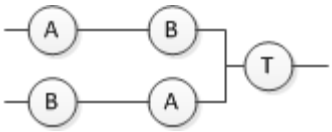
– An argument of the current tile definition; or

– An export of either a tile in $\overline{\texttt{E}} \cup \overline{\texttt{I}}$, or recursively an object expression, an argument, or another export. For example, consider:



The `Seen` tile can be used as an argument in the `Rotate` act because: `Seen` is exported from the `See` tile; the executing object is successfully executing the `See` tile; and the `Seen` tile extends the `Actor` tile that is supported as a target argument for the `Rotate` tile.

Context menus are formed by organizing all known tiles into a graph and then filtering that graph based on the context of the edit site. Each tile is placed into a graph based on the tiles it extends and includes given that such semantic relationships will help programmers locate the tile in a context menu. If tile `T` extends or includes tile `A`, then it must be a child of `A`. This graph is directed and acyclic (DAG), but not a tree, meaning that a tile can have multiple parents: if `T` extends or includes both unrelated tiles `A` and `B`, then it will appear as children to both `A` and `B`. Finally, all types included or extended by a tile must be in all paths to that tile, so additional nodes are added to `T`'s graph as follows:



The extra nodes are needed to encode all semantic relationships in the graph. The graph also has as nodes all possible exports as well as qualified tile accesses, where these parts of the graph are built lazily since they are potentially unbounded.

The graph is traversed efficiently by skipping traversals on nodes that are unrelated to the context of the edit site. For example, in generating a menu for a `Costume` argument, nodes in the graph that are not on the path to or from `Costume`, which lead to non-costumes, can be skipped safely, while nodes that are not supported by context, such as an included `Bug` tile when `Bug` is not in the current context, can also be skipped. When the fuzzy menu option is activated, context requirements are relaxed if context can be repaired to support the node; e.g., by adding a `Bug` tile to the list of tiles extended by a `Monster` tile that wants to create a `Beetle Costume`. Since the graph is unbounded, traversal is done lazily where the result is a new graph of edits that will be displayed directly in the context menu.

Although technically type safe, a well-formed YinYang program can contain constructions that are not very meaningful and should either be filtered out of or deemphasized in context menus. These problematic constructions are:

– The unification of method and class-like behavior in tiles is problematic for menu construction. For example, consider a `Tower` tile and a class-like `Gun Tower` tile that extends it: although technically safe, `Gun Tower` should not appear in a context menu that is used to form an act in the `Tower` tile's body. Likewise, a method-like `Attack` tile, which also extends `Tower`, should not appear in the context menu used to create a new object, although again this is technically safe.

– Many tiles successfully execute only periodically at discrete points in time. Consider the `Hit` tile, which succeeds when the executing actor object collides with another actor. The "hit" event only occurs for a small infinitesimal amount of time, and therefore it does not make sense to do an activity whose effect only occurs over finite time, like `Attack` or `Move`.

– Often, argument expressions should only be bound to stateless objects (e.g., costumes) or objects that already exist. For example, a context menu should not propose creating a new `Actor` object to satisfy the target argument of a `Rotate` or `Attack` tile.

These constructions are avoided by tagging tiles to indicate their intent, which are then used to further filter context menus. For example, a `Frame` tag indicates that the tile is used to create a stateful object and should not be used to

create objects to satisfy an argument tagged `Value`; or the `Event` tag indicates that the tile represents an event and should not be followed by a tile with the `Activity` tag, which indicates that the tile performs work over time. Both `Frame` and `Value` tags indicate class-like tiles while the `Event` and `Activity` tags indicate method-like tiles. Tags only assist with menu organization and filtering; they do not have any effect on how code is executed.

Execution

When considering the execution of YinYang code, programmers primarily need to be concerned with is how priorities are assigned across tile boundaries. As mentioned in the second part of Section 2, programmers can expect that behaviors in a tile will execute with a higher priority than behaviors in any tile that it extends. However when an object is created, these extended tiles must appear as acts somewhere to the left of the extending tile's act in the object's root behavior. As a result, it appears counter-intuitive that behaviors in a tile would execute with a higher priority than behaviors in the tiles it extends. However, behavior and tile execution are actually divided into two phases: the first phase executes each tile's *core behavior* left-to-right according to the containing behavior's act order; while the second phase executes the tile bodies (numbered behaviors) right-to-left for the successfully executing acts in the behavior. This execution is sketched out in the pseudo code of Figure 9.

Execution of a tile's core behavior initializes internal state, binds exports, and determines act execution success. A *native tile*, which is implemented with C# code, consists only of a core behavior while a non-native tile can optionally have a core behavior in addition to its body of numbered behaviors. No non-native tiles in this paper have explicit core behaviors, and so succeed in their execution by default. As shown in Figure 9, left-right execution of an act acquires a lock on the act's tile (if exclusive) and executes its core behavior. After left-to-right execution finishes, execution proceeds back right-to-left to execute the bodies of tiles whose acts are executing successfully. Body execution (`ExecBody`) proceeds from lowered numbered to higher numbered behaviors as described in Section 2. Priority is lowered (`Weaken`) after each iteration, while a new element is appended to the priority when executing a behavior or body.

The prioritization of an object created from the `Gun Tower` tile in Figure 4 is shown in Figure 10. Because of right-to-left prioritization, the priority of the `Wear` act in `Gun Tower`'s body is 6.1.1, which is stronger than the 7.1.1 priority of the `Wear` act in `Tower`'s body (lower numbers indicate stronger priorities)s. Additionally, tile bodies inherit the right-left priority of their acts; e.g., the `Rotate` act in the `Attack` tile's body has a priority of 7.2.3.1, which is stronger than the 7.3.1 priority of the `Rotate` act in the `Tower` tile's body. This matches the programmer's intuition that the execution of a tile body should inherit the priority of the corresponding tile invocation.

```
bool Exec(Object On, Behavior B, Priority P) {
  | Append a new part to a priority;
  | e.g., 7.1.1 = Append(7.1, 1)
  P = Append(P, 1);
  | Left-to-right phase
  for (i = 0; i < B.Acts.Length; i += 1) {
    | Fail if tile is exclusive and a higher
    | priority act has already locked the tile.
    if (B.Act[i].Tile.IsExclusive)
      if (!Acquire(On, B.Act[i].Tile, P)) break;
    | Execute core behavior; sets up state,
    | binds exports, and determines act success.
    if (!Exec(On, B.Acts[i].Tile.CoreBehavior, P))
      break;
    | Weaken the priority by one; e.g.,
    | 7.1.2 = Weaken(7.1.1),
    | where 7.1.1 is a higher priority than 7.1.2
    P = Weaken(P);
  }
  | Begin right-to-left phase.
  for (j = i - 1; j >= 0; j -= 1) {
    | Execute tile bodies of numbered behaviors;
    | this cannot fail.
    ExecBody(On, B.Acts[j].Tile.Body, P);
    P = Weaken(P);
  }
  | Behavior succeeds if all acts succeed;
  | only relevant for root and core behaviors.
  return i == B.Acts.Length;
}
void ExecBody(Object On, Body Body Priority P) {
  P = Append(P, 1);
  | Begin top-down (or lower-higher) phase.
  for (i = 0; i < Body.Behaviors.Length; i += 1) {
    | Call above Exec, ignore the result since
    | this is not a core or root behavior.
    Exec(On, Body.Behaviors[i], P);
    P = Weaken(P);
  }
}
```

Figure 9. Pseudo code for executing a behavior; argument evaluation is not shown.

Because behavior execution in YinYang is continuous, behaviors must be re-executed when their execution can change the state of the program. This is accomplished by dynamically tracking the dependencies of a behavior and conservatively re-executing the behavior whenever these dependencies change. For example, if the behavior acquires a lock for a tile on the executing object at some priority `P`, then it will be re-executed when another behavior acquires the lock on the same object at a higher priority `Q`. When a behavior is re-executed, resources from the previous execution, such as freshly created objects, are reused as needed or are cleaned up if they are not reused during the re-execution. For example, if behavior execution initially acquires a lock

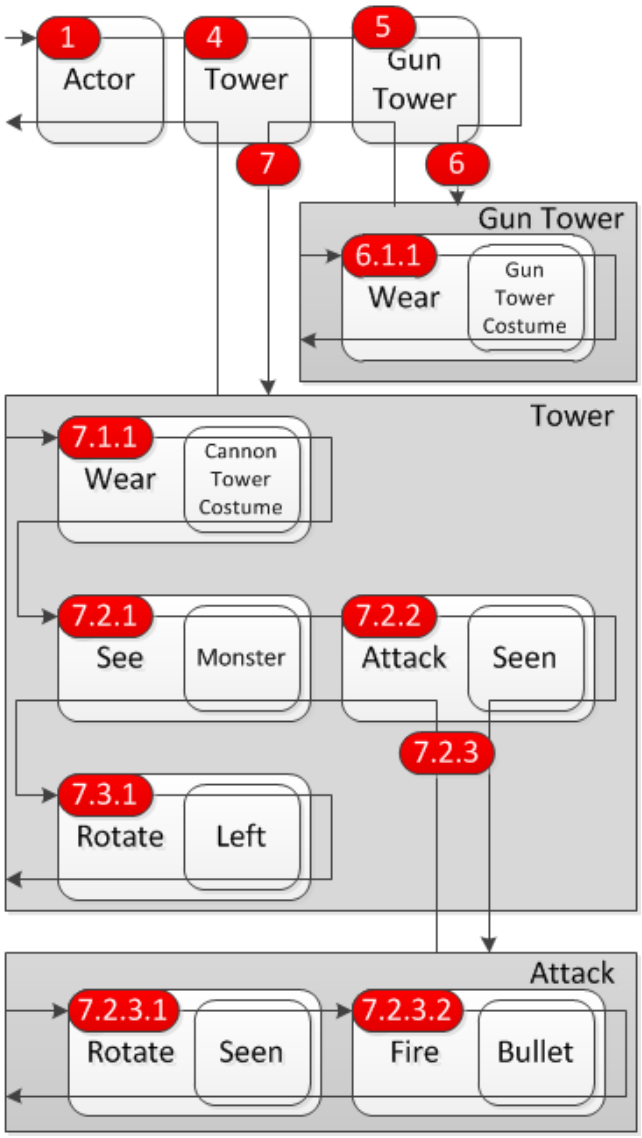


Figure 10. Trace of how prioritized execution works in YinYang for an object created from the `Gun Tower` tile in Figure 4; higher numbers indicate weaker priorities.

but does not re-acquire the lock on re-execution, the lock is released while other behaviors waiting on the lock will be re-executed. Additionally, tiles whose execution are time-dependent, such as the native `Rotate` and `Move` tiles, are marked as dependent on the frame display. On every frame update, frame-dependent tiles are executed to perform animations.

Native tiles are currently implemented as a set of call backs that define what happens when the tile is first executed (initialize resources), re-executed (update resources), and when it stops executing (cleanup resources). Native tiles have declarations just like YinYang tiles, and so can specify extended and included tile requirements that the type system then ensures are satisfied at run-time. Combined with the ability to specify exclusive-mode execution, new native tiles can safely be added to the system without breaking existing ones.

As mentioned in Section 2, a behavior can `spawn` a new object or `start` a new behavior whose execution are detached from the former behavior. While a spawned object only continues to execute independently, a started behavior will also execute with a very high priority that will only be superseded by the next started behavior of the executing object. Such prioritization is meant to model imperative programming where later statement executions supersede the effects of earlier statement executions. Finally, some tiles can be invoked "qualified" on objects other than the executing object; e.g., one can instruct a `Bank` part to earn $100. However, because this execution occurs from outside of the object, their is no reasonable way to prioritize the tile execution. As a result, only tiles that are not exclusive, and whose implementation's do not execute exclusive tiles, can be invoked qualified.

5. Experience and Future Work

Our YinYang prototype implementation consists of about 6000 lines of interpreter and 2000 lines of library (native tile) C# code. The prototype runs on Window PCs with touch screens, such as Tablet PCs, and does not require a keyboard or mouse to use. The native tiles that we have built allow programmers to express various action-oriented mobile 2D games, although YinYang can easily be extended safely with new native tiles to address other game or UI domains. This section describes our initial experience with using this prototype and, given lessons learned, concurrently discusses directions for future work.

As a small informal case study, consider using YinYang to construct a simple but real game whose code is shown in Figure 12; two screenshots of play are also shown in Figure 11. The game starts with a certain number of monsters on its stage where the player has $100 with which to tap a monster. Once tapped, the player loses their $100 and the monster begins to explode by becoming an orange ball that grows to a very large size. As the monster is exploding, any other monster that touches it will also start exploding while the player gains $100. The goal is for the player to explode the monsters in as few taps as possible.

Input Efficiency and Usability

As a rough unscientific measurement of input efficiency, the program in Figure 12 takes a little bit less than 2.5 minutes for us to input from memory, while we can type the text-equivalent program using a real keyboard in about 1.5 minutes; overall, touch input is about 66% slower than keyboard input for this test. Although programmers spend more time thinking, learning, and searching than they do on input, input efficiency still matters, as slow input can disrupt

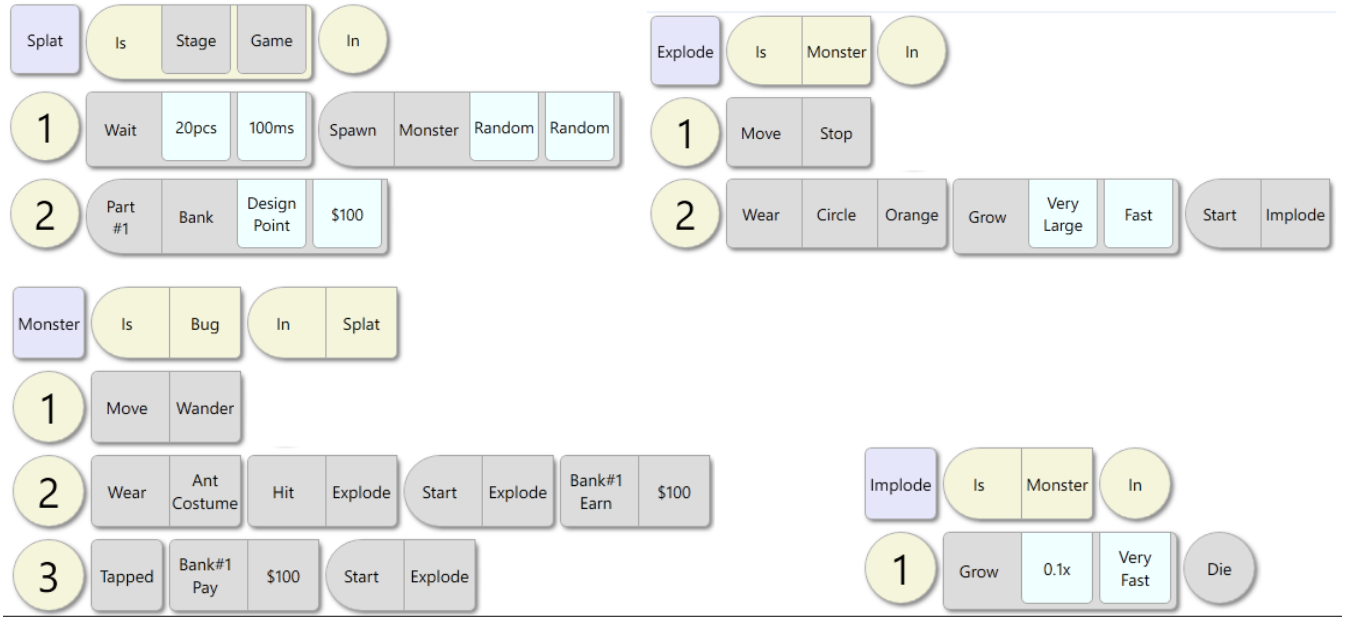


Figure 12. YinYang code for the splat game; act insertion anchors are elided from these screen shots.

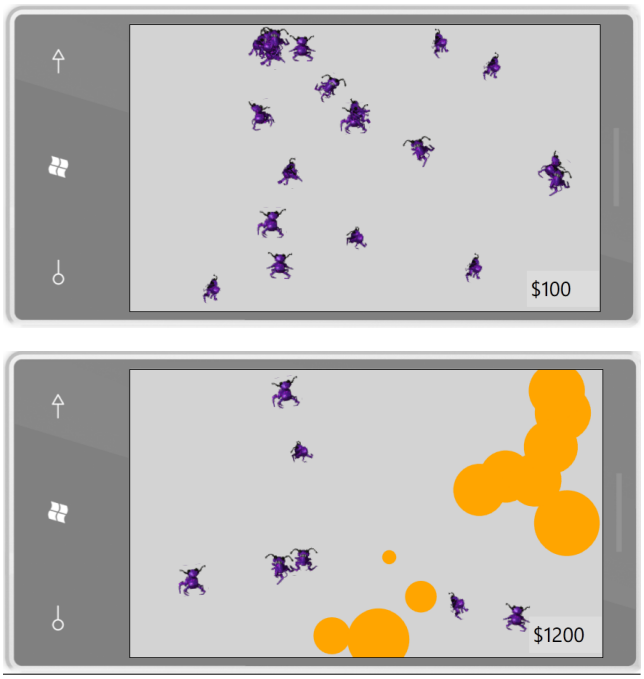


Figure 11. The Splat game in action. Top: monsters wandering around at the start of the game waiting to be tapped. Bottom: the player has just tapped a monster setting off a chain reaction that has already exploded 12 additional monsters.

the programmer's focus and rhythm, further reducing their efficiency.

Many things work well in our current prototype in the areas of input efficiency and usability. For example, the tile that we want to select is often in the first or second menu, meaning that it does not take more than a few seconds to select it. Once a tile is selected to create a new expression, that its arguments immediately appear in the menu is useful in maintaining rhythm without needing to call up another menu. YinYang also benefits from being graphical and having menus display what tiles are available at each edit site. In this respect, YinYang is like programming solely with autocompletion (aka Intellisense) in a textual language, except that the menus have more organization and are not just linear lists of completed text options. On the other hand, the most significant problems that we have identified with our current prototype are as follows:

- Editing always involves scanning and navigating the menus. Scanning each menu takes about a couple of seconds, and often one or two additional sub-menus must be accessed.
- Given that each menu is generated from edit context, the layout of one menu to the next can differ even if they share many of the same tiles. As a result of this inconsistency, the programmer cannot learn where to tap without first scanning.
- Although YinYang attempts to maintain focus by keeping the programmer's focus on one tile definition (Section 3), activating a menu itself disrupts rhythm and focus as the menu obscures content.

– The programming interface provides no cues to the programmer about what they can do next without activating a context menu; the programmer must know where to start editing or what tile to create afresh. This is not intuitive to programmers who are unfamiliar with the language.
– Menus are organized via tile semantic relationships (extend and include) that are not always intuitive.

When comparing against textual languages, keyboard-input allows a programmer to quickly refer to a construct as long the programmer knows its name, while programmer focus is preserved since there are few menus to deal with and the edit cursor keeps track of where they are. As far as usability is concerned, textual languages often offer no cues on how to get started, although "wizards" often fill this role in the programming environment. Finally, the programmer is expected to know where most constructs are without much assistance; auto-complete menus are typically ordered only alphabetically and do not show completions for more than one level of hierarchy.

Our current prototype makes progress towards being an efficient and usable programming interface but we have not yet realized what we believe is achievable on a tablet. Some ideas for improvement from our experience in using the current prototype are as follows:

– Probability could be used to ensure that likely options are organized closer to the initial menu than less likely options, which can more acceptably be accessed through deeper sub-menus. Probability models could be based on edit site context as well as the history of the programmer as well as a community of programmers.
– Re-introduce the concept of a cursor and have content and context menus appear on the screen at the same time; the context menu shown on the screen then depends on the cursor that advances automatically as edits are made.
– Allow menu categories to be encoded more independently from semantics relationships.
– YinYang does not currently utilize the multi-touch gestures that tablets support, we should explore how such gestures can enhance the programming experience.
– Associate tiles with mnemonics that, when learned, can quickly be inputted on the tablet to make a quick selection; e.g., a gesture.

Language Design

The YinYang encoding of the game in Figure 12 is fairly concise primarily due to YinYang's behavior-based programming model. Conciseness comes from specializing in the encoding of reactive behaviors, while YinYang is not suited to expressing many other kinds of computations, which must instead be accessed through C# native tiles. YinYang lacks the control flow constructs or explicit variable declarations necessary for expressing many kinds of fundamental computations; such as searching for something in a list. Also, although it is possible to encode arbitrary expressions in YinYang, the programming interface is not suited for it: writing even simple arithmetic expressions using context menus would be very tedious and heavy nesting coupled with long streams of argument expressions invalidate the assumptions of our interface design.

Ideas for making YinYang more expressive include supporting array programming [8] and/or point-free functional programming that reduce the need for diverse control flow constructs and explicitly named variables. Tangible functional programming [4] has explored applying the latter approach in the context of a graphical language. Additionally, YinYang could be integrated with additional graphical language/user interface for expressing different kinds of computations; e.g., a calculator-like language for expressing math.

6. Related Work

As mentioned before, YinYang is directly influenced by Kodu [12]. In contrast with Kodu, however, YinYang is redesigned for touch and supports user defined abstractions. YinYang is also influenced by our previous experience with SuperGlue [14], which is a textual, reactive, and object-oriented language for expressing user interfaces and animations. We have found that structuring programs into Brooks' autonomous objects [1] is easier and more expressive than Superglue's structuring of programs into objects that are connected together through data-flow signals.

YinYang is closely related to graphical languages that are based on structured syntax. Scratch [13], like Kodu, is a graphical language that makes programming accessible to beginners and children. Programmers in Scratch drag blocks from palettes and drop them in place to form the behavior of an actor. Scratch's model has been expanded in Google's App Inventor [6] with support for user defined blocks to support more capable DIY programming. In contrast to Scratch's procedure-like blocks, YinYang tiles are more powerful given their continuous execution semantics and their ability to act like classes as well as methods.

In contrast to graphical structured languages, syntax-directed editing [24] leverages built-in knowledge of a textual language to enable direct editing of syntax trees with a graphical editor. Unfortunately, syntax-directed editing is often inefficient and cumbersome [16] when compared to free-form text editing. Bringing syntax-directed editing to tablets does not solve its problems. However, some techniques such as leveraging the type system to provide context and allowing not-yet correct edits could be used to improve syntax-directed editing in general.

Numerous visual languages such as Prograph [19], LabVIEW [25], and AgentSheets [20] leverage visual spatial relationships and/or direct manipulation [21] to enable a more immersive programming experience. Visual languages have

traditionally suffered from scaling-up problems [2], where directness often conflicts with support for abstraction, and readability problems [18] where the practical density of visual elements is less than textual elements. YinYang is very much a structured (graphical but not visual) language, although it does support direct manipulation in appropriate cases, such as the direct placement of user interface elements.

7. Conclusion

Programming has been chained to the typewriter for long enough: as computing moves toward touch-based devices, we should rethink our programming experiences lest they get left behind as niche tasks that require niche computing hardware. This paper has demonstrated the viability of one possible programming experience for tablets. YinYang tiles naturally fit the tablet's form factor and support for direct touch, while we have designed a type system around tactile tile placement that enhances expressiveness and safety while actively supporting input tasks. The resulting programming experience shows promise that a tablet language can be competitive with the capabilities and input efficiency of keyboard-based programming languages.

Acknowledgments

We thank Jonathan Edwards, Matthew Flatt, Lidong Zhou, and many others for reviewing this paper, as well as Stephen Coy and Matthew MacLaurin at Microsoft Fuse Labs for inspiring us with Kodu.

References

[1] R. A. Brooks. Planning is just a way of avoiding figuring out what to do next. In *MIT Artificial Intelligence Laboratory Working Papers*, September 1987.

[2] M. M. Burnett, M. J. Baker, C. Bohus, P. Carlson, P. J. V. Zee, and S. Yang. The scaling-up problem for visual programming languages. Technical report, Oregon State University, 1994.

[3] B. X. Chen. The iPad falls short as a creation tool without coding apps. *Wired Magazine*, March 2011.

[4] C. M. Elliott. Tangible functional programming. In *Proceedings of ICFP*, pages 59–70, 2007.

[5] P. M. Fitts. The information capacity of the human motor system in controlling the amplitude of movement. *Experimental Psychology*, 47(6):381–391, 1954.

[6] Google Labs. App Inventor for Android. `http://appinventor.googlelabs.com/`, 2010.

[7] W. E. Hick. On the rate of gain of information. *Experimental Psychology*, 4(1):11–26, 1952.

[8] K. E. Iverson. *A programming language*. John Wiley & Sons, Inc., New York, NY, USA, 1962. ISBN 0-471430-14-5.

[9] A. Kay. A personal computer for children of all ages. In *Proceedings of the ACM National Conference*, August 1972.

[10] A. Kay and A. Goldberg. Personal dynamic media. *Computer*, 10:31–41, March 1977.

[11] A. J. Ko, H. H. Aung, and B. A. Myers. Design requirements for more flexible structured editors from a study of programmers text editing. In *Proceedings of CHI*, pages 99–102, 2005.

[12] M. B. MacLaurin. The design of Kodu: a tiny visual programming language for children on the Xbox 360. In *Proceedings of POPL*, pages 241–246, January 2011.

[13] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The Scratch programming language and environment. *TOCE*, 10(4):16, 2010.

[14] S. McDirmid. Living it up with a live programming language. In *Proceedings of OOPSLA Onward*, pages 623–638, October 2007.

[15] P. Miller, J. Pane, G. Meter, and S. Vorthmann. Evolution of novice programming environments: the structure editors of Carnegie Mellon University. In *Interactive Learning Environments*, volume 4, pages 140–158, 1994.

[16] L. R. Neal. Cognition-sensitive design and user modeling for syntax-directed editors. In *Proceedings of CHI*, pages 99–102, 1987.

[17] K. L. Norman. *The Psychology of Menu Selection: Designing Cognitive Control at the Human/Computer Interface*. Greenwood Publishing Group Inc., Westport, CT, USA, 1991. ISBN 089391553X.

[18] M. Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Commun. ACM*, 38(6):33–44, 1995.

[19] T. Pietrzykowski, S. Matwin, and T. Muldner. The programming language PROGRAPH: Yet another application of graphics. In *Proceedings of Graphics Interface*, pages 143–145, May 1983.

[20] A. Repenning. *Agentsheets: a tool for building domain-oriented dynamic, visual environments*. PhD thesis, University of Colorado at Boulder, Boulder, CO, USA, 1993. UMI Order No. GAX94-23532.

[21] B. Shneiderman. Direct manipulation. a step beyond programming languages. *IEEE Transactions on Computers*, 16(8):57–69, August 1983.

[22] B. Shneiderman. Tree visualization with tree-maps: A 2-d space-filling approach. *ACM Transactions on Graphics*, 11: 92–99, 1991.

[23] K. T. Stolee. Kodu language and grammar specification. Technical report, Microsoft Research, Redmond, WA, USA, 2010.

[24] T. Teitelbaum and T. Reps. The cornell program synthesizer: a syntax-directed programming environment. *Commun. ACM*, 24:563–573, September 1981.

[25] G. M. Vose and G. Williams. LabVIEW: Laboratory virtual instrument engineering workbench. *Byte*, pages 84–92, September 1986.