TileCode: Creation of Video Games on Gaming Handhelds

Thomas Ball, Shannon Kao, Richard Knoll, Daryl Zuniga
Microsoft
Redmond, USA
{tball, shakao, riknoll, dazuniga}@microsoft.com

ABSTRACT

We present TileCode, a video game creation environment that runs on battery-powered microcontroller-based gaming handhelds. Our work is motivated by the popularity of retro video games, the availability of low-cost gaming handhelds loaded with many such games, and the concomitant lack of a means to create games on the same handhelds. With TileCode, we seek to close the gap between the consumers and creators of video games and to motivate more individuals to participate in the design and creation of their own games. The TileCode programming model is based on tile maps and provides a visual means for specifying the context around a sprite, how a sprite should move based on that context, and what should happen upon sprite collisions. We demonstrate that a variety of popular video games can be programmed with TileCode using 10-15 visual rules and compare/contrast with block-based versions of the same games implemented using MakeCode Arcade.

Author Keywords

gaming handhelds, video games, visual programming, cellular automata

CCS Concepts

•Software and its engineering → Visual languages; •Human-centered computing → Human computer interaction (HCI);

INTRODUCTION

Most video game development environments require access to a tablet, laptop or desktop machine, often with an Internet connection, putting game development out of reach for those who cannot afford such devices or don't have Internet connectivity. On the other hand, a search for "gaming handhelds" on Amazon reveals a huge variety of low-cost gaming handhelds that come loaded with hundreds of retro video games.

Furthermore, for those wishing to create their own games, the conceptual gap between the idea for a video game and its realization via a computer is non-trivial. While game mechanics, the desired interactions among video game elements, can be described informally and succinctly ("a boulder falls if there is space below it"), encoding a game's mechanics for a computer to execute may require mastering various concepts, ranging from coordinate systems and game sprites to the syntax/semantics of a programming language. Web-based programming environments such as Scratch [23] and MakeCode Arcade [6] use block-based editors [9] to simplify game programming, but still introduce a large number of concepts. These environments also require a computer with a modern web browser and Internet connectivity.

Our main goal is to enable the process of game creation to take place on low-cost gaming handhelds themselves, rather than tablets/laptops/desktops. A secondary goal is to reduce the gap between game mechanics and game programming so more people can participate in game creation.

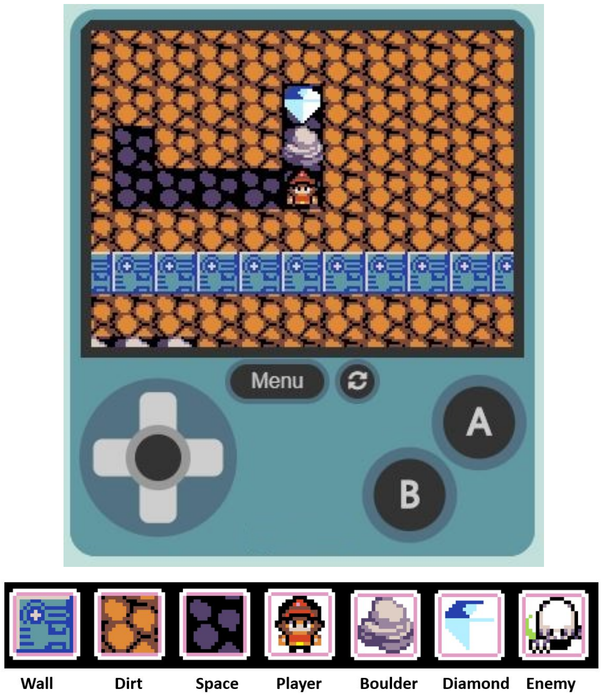
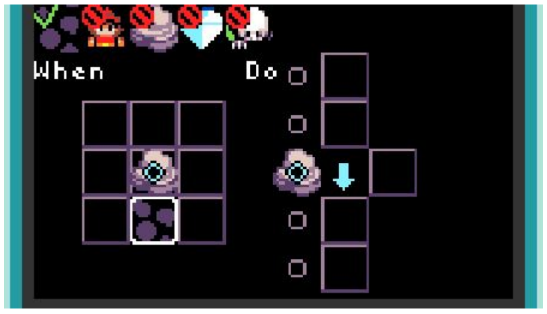


Figure 1. Screen snapshot of "Boulder Dash" game (written using TileCode) in the web-based MakeCode Arcade game machine simulator; underneath the simulator are the backgrounds (Wall, Dirt, Space) and sprites (Player, Boulder, Diamond, Enemy) in the game.

UIST '20, October 20–23, 2020, Virtual Event, USA
© 2020 Association for Computing Machinery.
ACM ISBN 978-1-4503-7514-6/20/10 ...$15.00.
http://dx.doi.org/10.1145/3379337.3415839

(a)



(b)

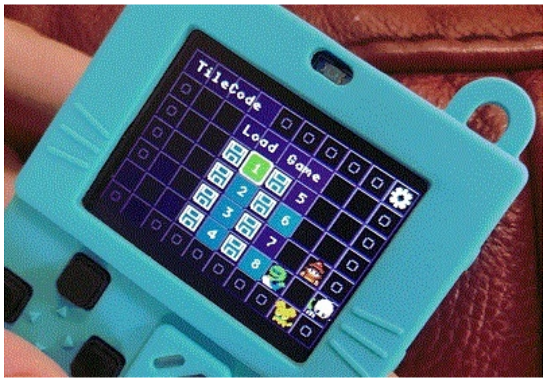


Figure 2. (a) Screen snapshot of the TileCode game creation environment with a "When-Do" rule for a boulder at rest to start moving down; (b) TC running on a MakeCode Arcade-compatible gaming handheld.

Towards this end, we have designed and developed a restricted and visual game programming model with a supporting game creation environment called TileCode (TW). TC games and its game creation environment both are based on tile maps, a foundation of many retro video games, and can run on a variety of low-cost microcontroller-based game handhelds.

A TC game has a set of sprites (such as the player, boulder, and diamond shown in Figure 1); each sprite is centered on a tile and each tile displays a background image. A sprite can move in one of four directions (left, right, up, down) to an adjacent tile. The TC game creation environment itself is implemented using tile maps as well: all editing takes place via a cursor moved with the 4-way directional pad (dpad) of the device.

A TC program is a set of rules, each of which is associated with a sprite kind (such as player or boulder). A rule takes the form of a "When-Do" pair, as illustrated in Figure 2(a). The **When** section visually describes a pattern/predicate over the *3x3 local neighborhood* around the central sprite (a boulder, in this case) to be matched against the tile map. The top row of the figure shows that the tile below the boulder should have the grey-blob (space) background and not contain any sprite. When this pattern matches on the tile map, then the boulder is sent a move down command, as shown in the **Do** section. This rule does not match on the tile map from Figure 1, but will match once the player sprite moves out from under the boulder. Other rules can specify what happens when buttons of the game handheld are pressed, or when sprites collide.

As in cellular automata [25], TC rules operate in parallel on the sprites on the tile map: the 3x3 predicate of the **When** section determines whether or not a rule applies to a sprite (note that multiple rules can apply to the same sprite). The result of a rule execution is to send commands to the sprite/tiles at the center of the 3x3 neighborhood or adjacent to the center (unlike cellular automata, which only updates the center tile). As multiple commands can be sent to a sprite/tile, conflicting commands are resolved automatically, which may result in non-deterministic/random behavior (for example, if one rule commands a sprite to move left and another commands it to move right).

TC is implemented on top of MakeCode Arcade, a web-based game creation environment for retro arcade games (160x120 pixel screen with 16 colors) that can run in the web browser or on a variety of microcontroller-based gaming handhelds [26]. Figure 2(b) shows such a handheld displaying TC's load screen. MakeCode Arcade supports both block-based and TypeScript-based programming [2] (TC is implemented in TypeScript). The code of TC and all the games described here are open source (see `https://github.com/microsoft/tilecode`).

The main contributions of our work are:

- the design and implementation of TC, a game creation environment that runs on battery-powered gaming handhelds;
- a restricted programming model with a mapping to a visual editing experience that closely matches the domain of tile-based games;
- implementations of a variety of well-known games (Boulder Dash, Snake, Bejeweled) in TC, using just 10-15 rules each;
- a comparison with block-based versions of the games implemented in MakeCode Arcade.

Our evaluation shows that the ability to pattern match against multiple tiles/sprites and to refer to the direction of a sprite's last movement is key to enabling more games to be written in a succinct fashion and without the explicit use of loops and arrays.

RELATED WORK

Turtles, Automata, and Objects

The Kara programming environment [10] introduces students to turtle-style programming (ala Papert [18] and Pattis [19]) in a tile-based world, one of many examples of the sort. Kara is a programmable ladybug that can sense local conditions and respond by executing primitive actions, like TC. Kara allows the user to structure their program using finite-state automata/machines with user-defined states; TC lacks such a capability, so we instead encode needed states in the tile map (see the TC encoding of the Bejeweled game), but has more powerful pattern matching capability.

Moving beyond finite-state automata, environments such as Playground [8] and Alice [4, 12] use object-oriented or agent-based paradigms for novice programming of interactive animations. TC's programming model is not object-oriented, though a single TC rule can match and act upon multiple sprites, a restrictive form of multiple dispatch [16].

Novice Game Programming

Our work is heavily inspired by the Kodu Game Lab [29], an introductory game making environment situated inside a 3D world of programmable objects that feature robot-like sensors and actuators. As Kodu objects operate in an environment of continuous motion, much like that of robotics, the language is extremely high level and inherently ambiguous. TC shares the "When-Do" programming paradigm of Kodu, but departs in a number of ways: TC operates in a discrete 2D world of tiles, sprites, four directions, and focuses on retro video games.

There are many systems, in addition to Kodu, to help novices create their own games. Minecraft is a popular example where "redstone" blocks in the world can be laid out to create logic circuits (in fact, complete computers have been built inside Minecraft worlds using redstone). [22] Super Mario Maker is a side-scrolling platform game that allows the user to create their own game levels within the application [13], but the game mechanics are fixed.

The Scratch [23], Stencyl [14], and MakeCode Arcade [6] environments use the Blockly [9] framework to build games by programming with structured control-flow; while Blockly helps to prevent syntax errors, there still is a large conceptual/visual gap between the block-based program and the game world. TC is based instead on pattern matching against the tile map, rather than control-flow, as the primary program structuring mechanism.

The "program by demonstration" model of novice simulation/game programming was introduced in AgentSheets [20, 21], and then championed by Cypher, Smith and colleagues in their KidSim [5, 27] and StageCraft Creator [28] systems. In these systems, the user demonstrates how a sprite should move/change in a local context. A graphical rewrite rule is the result of the demonstration, expressed in terms of "before" and "after" pictures. BlockStudio [3] is a newer demonstration environment for children to create games and animations, in the model of KidSim. TC rules are not graphical rewrite rules, but a form of guarded command [7]; the **When** predicate (guard) corresponds to the "before" picture of a graphical rewrite rule, while the **Do** section is a set of explicit commands.

ToonTalk is based on concurrent constraint programming, a fully general programming model [11]. While TC does permit rules to fire concurrently, it has a domain-specific way to resolve conflicts arising from concurrency.

Game Modeling Languages

The Video Game Description Language (VGDL) [24, 17] provides a high-level way to describe many common video arcade games, for the purpose of conducting research on learning and planning algorithms. VGDL contains a set of classes for common types of sprite motion found in games, based on an underlying ontology of video games. The user programs a game by choosing among the motion classes and specifying an action to execute when sprites collide with one another.

In contrast, TC is based on a set of lower-level primitives to encourage exploration of new game mechanics. For example, in TC it is possible to program the logic for a boulder tumbling off of another boulder (as shown in Figure 4 and discussed later); this behavior is not found in the VGDL implementation of Boulder Dash. The user would have to extend the VGDL runtime itself, written in Python, to provide a new primitive for this behavior. The Snake and Bejeweled games, discussed in the evaluation section, are not expressible in VGDL without substantial coding in Python.

Mek [30] is a language and prototyping tool for 2D turn- and tile-based deterministic games that also uses visual rules as its primary structuring mechanism. Mek has no concept of a sprite moving in a direction - rather all game state and commands are based on changing a tile's background color. While the system can be used to prototype game mechanics, it cannot produce playable games, as TC can.

Retro Game Creation Environments

PICO-8 [1] is a retro game engine supported by a virtual machine that supports a 128x128 pixel screen with 16 colors, much like MakeCode Arcade. PICO-8's goal is to provide a "fantasy console", which is like a gaming handheld but "without the inconvenience of actual hardware". The PICO-8 programming environment is available for MacOS, Windows and Linux and supports a text-based Lua code editor. The TC programming environment is much more restricted than that of the PICO-8, as it was designed to run on gaming handhelds rather than desktops.

OVERVIEW OF TileCode APPLICATION

Bob purchases a gaming handheld for his daughter Alice. To her surprise, in addition to the set of built-in games, the handheld comes loaded with TC, which allows modification of the built-in games, as well as creation of new games.

From the load screen of TC (Figure 3(a)), Alice can select a game from one of eight slots. She moves the square-shaped cursor between adjacent tiles using the dpad of the handheld, selects a tile using the A button to perform an action, and presses the B button to return to the menu of the current screen or to the previous screen.

As shown in Figure 3(a), Alice has selected the game in slot 1 (Boulder Dash), which brings up the game's home screen of Figure 3(b). The home screen has a menu bar and displays the four kinds of backgrounds and four kinds of sprites. Selecting a background or sprite opens a gallery of images that Alice may select from, not shown here. The green play button on the menu bar allows Alice to play the game, as shown in Figure 1.

The first item on the menu bar is the red map icon—Alice selects it to open the tile map editor, shown in Figure 3(c). She selects the player sprite from the menu bar and moves to the tile map (of reduced 8*x*8 tiles) to place the sprite. Alice returns to the game home screen and selects the paint icon (to the right of the map icon) to bring up the bitmap editor (Figure 3(d)), which she uses to color the player sprite's eyes green. Game assets are saved automatically in the non-volatile flash of the handheld on each transition between screens.

From the game home screen, Alice now selects the coding icon (to the right of the paint icon), which brings her to the rule selector screen (Figure 3(e)). Alice selects the boulder sprite to see what rules already exist for the boulder. The rules are

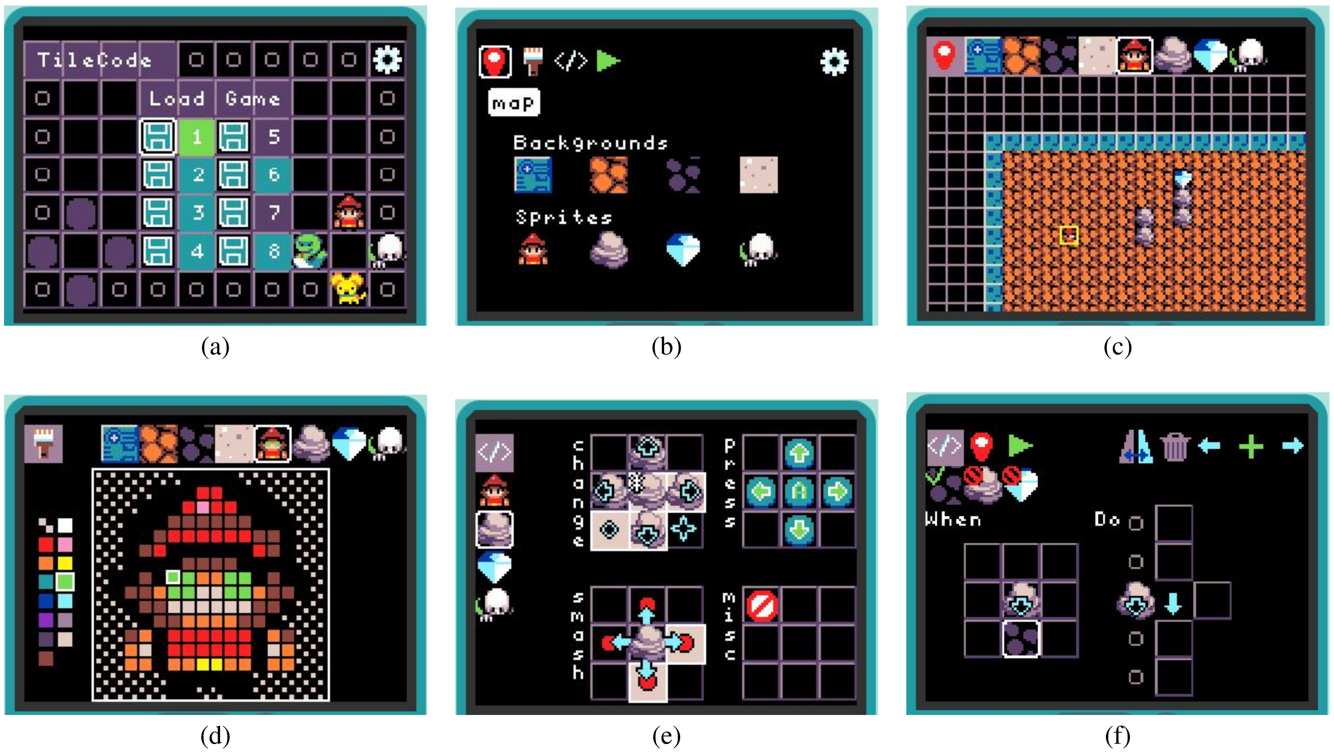


Figure 3. TileCode screens: (a) load screen; (b) game home screen; (c) tile map editor; (d) bitmap editor; (e) rule selector; (f) rule editor.

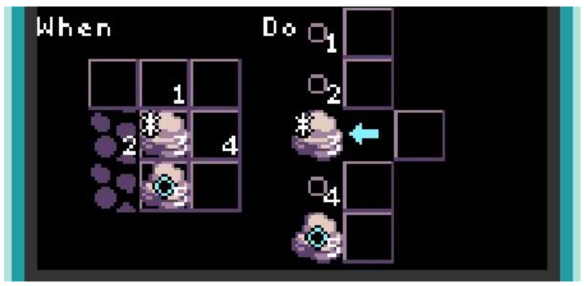


Figure 4. Rule for a boulder to fall off another boulder to the left, provided there is space available. The numbers show the correspondence between the rows of the Do section and the tiles of the When section.

summarized in four quadrants: **upper-left**–there is a **change** to a sprite's 3x3 neighborhood; **upper-right**–button **press** event (A, dpad-left, dpad-right, dpad-up, dpad-down); **lower-left**–collision (**smash**) of sprite into a tile/sprite; **lower-right**: miscellaneous events. The highlighted squares represent the rules with code.

Alice sees that there are rules in the upper-left and lower-left quadrants of the screen. She decides to investigate the rule for a boulder at rest, and selects that tile (lower-left tile of upper-left quadrant), which brings up the rule editor (Figure 3(f)). The menu bar of the rule editor provides quick access to the map editor and game play. The blue arrow on the far right of the menu indicates there are other rules for the boulder.

Alice knows that the original version of Boulder Dash allows boulders to tumble off of other boulders when there is space to the side of the boulders, but finds no such rule for this behavior. Using the rule editor, as shown in Figure 4, she creates a new rule in which the "When" section has a resting boulder under the central boulder and empty space to the left of both boulders; the central boulder is sent a move left command.

PROGRAMMING MODEL

The TC programming model has a small set of concepts and primitives, from which a host of different behaviors can be described using a visual rule-based paradigm. This section focuses on the core aspects of the programming model and its visual representation. The next section describes the editing experience.

Game State and Rounds

The basic atoms of TC are tiles and sprites that are located on a grid coordinate system. A tile must contain a background image and may contain any number of sprites. As we saw before, there are four kinds of background images and four kinds of sprites in TC (the implementation actually can accommodate up to 16 kinds of backgrounds and 16 kinds of sprites). The tile map and the set of sprites make up the entirety of the TC game state. The game screen can only display a 10-by-7 area of the tile map, so the tile map scrolls during game play to follow the player's avatar (by default, the first sprite kind).

A TC game proceeds in rounds: a round executes rules in parallel on the current state to determine if the game should proceed/end, the direction that each sprite should next take, which sprites should be spawned/destroyed, and the actions that should be taken upon a pending collision. After a round, the movements/actions are executed by the underlying game engine to update the state (all sprites move synchronously and at the same speed). A sprite stores the direction (left, right, up, down) it moved in the last round (or if it remained at rest), for inspection by the rules in the next round.

Game Rules and Events

As already seen, a TC rule is formed by a pair of a **When** predicate and **Do** commands. Rules fire in parallel on the current game state and set of events (e.g., such as button presses), which results in commands being sent to tiles and sprites; each tile/sprite stores a local log of the commands sent to it. A resolution step determines which of the (possibly conflicting) commands in each log will be executed. A new game state is produced by executing the commands.

Every rule is parameterized by an event. There are three main types of events in TC, corresponding to the quadrants seen in the rule selector screen of Figure 3(e): (1) **change** fires when there is a change to the local neighborhood around a sprite; (2) **press** fires for a button press of the dpad or the A button (the B button is not available to the game writer as it is used by TC to exit the game); (3) **smash/collide** fires when a sprite is about to collide with another sprite. Section "Evaluation I: Three Popular Games" presents examples of all the above rules.

When, Tile and Direction Predicates

A **When** predicate is a predicate on the game state that examines the *3x3 local neighborhood* around a tile/sprite. This predicate can specify the presence or absence of background/sprite kinds, as well as the direction that sprites last moved in.More precisely, a **When** predicate is a conjunction of *tile predicates*, one for each of the nine tiles in a sprite's neighborhood, including the center tile. Most of these predicates will simply be "true", corresponding to a black tile, which means that no constraints are placed on that tile. A tile predicate is defined by three non-intersecting sets (Include, Include', and Exclude) and a direction predicate:

- **Include**: the tile must contain at least one background/sprite whose kind is in this set and whose direction (in the case of a sprite) matches the direction predicate;
- **Include'**: the tile must contain at least one background/sprite whose kind is in this set;
- **Exclude**: the tile must not contain any of the backgrounds/sprites from this set.

A black tile's include and exclude sets all are empty. The second row of Figure 3(f) shows the include and exclude sets for the tile one space down from the center boulder: the Include set is denoted by green check marks; the Exclude set is denoted by red "no-entry" circle-slash signs; membership in the Include' set is denoted by a yellow dot. In this case, the space background is included in the tile and the boulder and diamond are excluded from the tile. Figure 5(a) shows the seven icons corresponding to the seven possible direction predicates (left, up, right, down, resting, moving, any). The direction predicate icon is overlaid on the sprite, as shown in the figure.

Sprite Witnesses

In all TC rules, the center tile predicate has a special form, which is to have a non-empty Include set that contains only sprites. This guarantees that if the predicate matches then there will be a sprite witness to execute commands against. The four tiles adjacent to the center tile also may have sprite witnesses. This is the main form of variable binding in TC. Sprite witnesses are displayed in the column to the right of the **Do** keyword, as discussed further below. The Include' set does not bind a sprite witness.

Do Commands and Conflicting Commands

If a rule's **When** predicate matches on the tile map, then the associated commands in the rule's **Do** section are sent to the objects that they address. The commands can be addressed to the center tile/sprite and to the four tiles/sprites adjacent to the center tile. Thus, there are five rows in the **Do** section. As the user moves the cursor over these rows, the corresponding tile is highlighted in the **When** section. A sprite witness is displayed at the beginning of a row. If there is no sprite witness, an empty circle is shown instead. Figure 4 identifies two sprite witnesses, the center boulder and the boulder beneath the center boulder. The numbering of the five rows in the **Do** section and the five tiles in the **When** section shows the correspondence.

To reiterate, commands are not immediately executed but sent to the addressed tile/sprite object, which maintains a command log. Conflicting commands are resolved automatically, by removing commands from the log before execution. The commands that can be sent to a sprite are: **move**–conflicts with itself–resolve by choosing a move command at random; **destroy**–no conflicts; **stop**–conflicts with move command–overrides all move commands (can only be issued by collision rules). The commands that can be sent to a tile are: **paint background**–conflicts with itself–resolve by choosing one paint command at random; **spawn sprite**–no conflicts; open a **portal** to a random tile on the tile map that has the given background and contains no sprite–no conflicts. Finally, there is a command to disable (for one round) all the rules associated with a sprite.

Collisions

Once all rules have fired and commands resolved, each sprite has a unique direction to move in (or no direction at all, if it received no move command in this round). This new set of directions is used to determine pending sprite collisions. TC gives the end-user the ability to determine what will happen for each pair of sprites about to collide: **destroy**, **stop motion**. The evaluation section presents examples of collision rules.

Progress/Termination Conditions

Many game progress/termination conditions are existence rules, such as "the game ends when a boulder falls on the player" and are easily specified using a **When** predicate. Other such conditions require that a predicate holds for every member of a set: "the player goes to the next level when every diamond has been collected from the game board". For these cases, we can often use the *negation* of **When** predicate: "the player goes to the next level when no diamond is on the

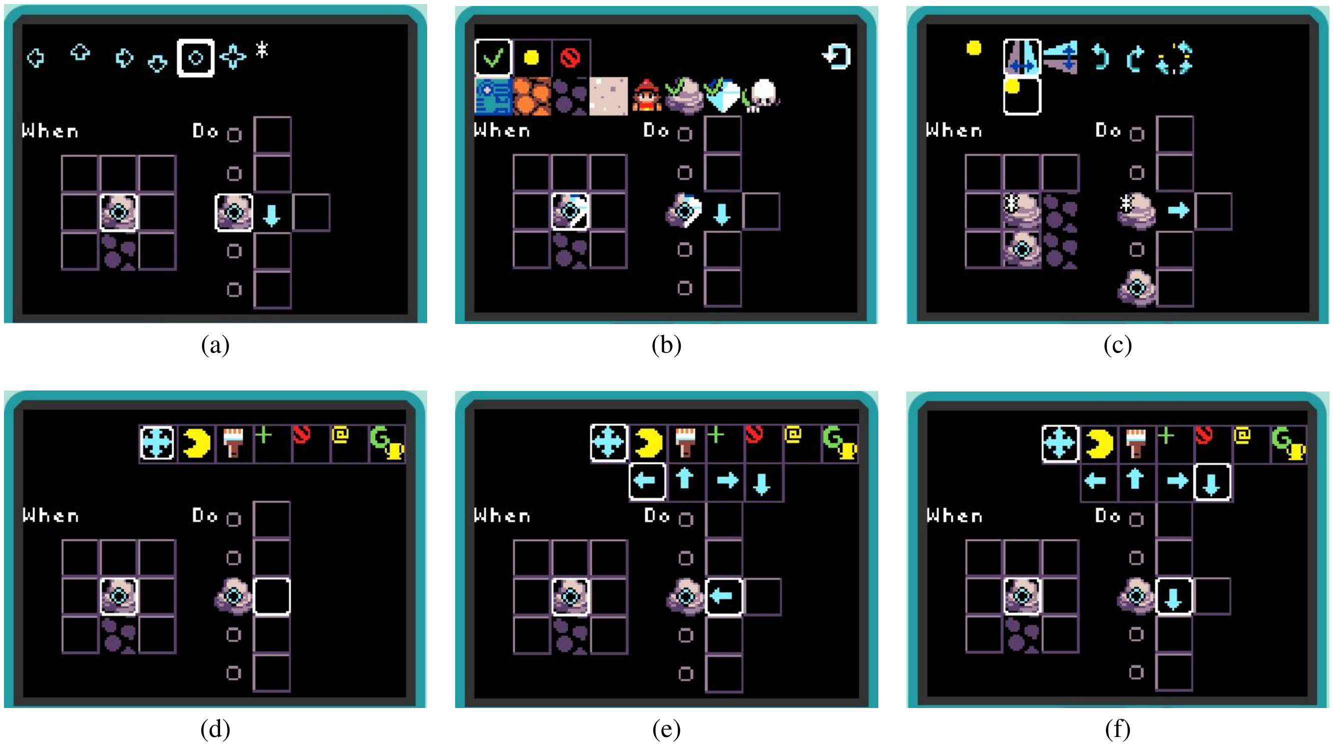


Figure 5. (a) Selecting the sprite witness in the Do section brings up the direction predicate menu; (b) A rule that applies to both diamond and boulder sprites and shows the attribute menu for specifying a tile predicate; (c) Screen for generalizing a rule by flip and rotation operations; (d) selecting the tile to the right of the sprite witness brings up the command menu; (e) selecting the move icon from the command menu; (f) selecting the move down command. The command menu icons, from left to right, are: move, destroy sprite, paint tile, create sprite, block rule, portal, game win/over.

game board". The evaluation section presents an example of a negated **When** predicate.

RULE EDITOR DESIGN

Designing the TC game creation environment so that game programming is possible on a 160 × 120 pixel screen with only A, B and dpad buttons is an interesting challenge. The tile map editor and paint editor are straightforward adaptations of standard interfaces for creating tile maps and pixel art, so this section focuses mainly on creating and editing of TC rules.

Our design was guided by the following two goals: (1) minimize the amount of hidden user-modifiable state as much as possible (though the size and complexity of a program will fight against this goal, as will be seen); (2) enable visual programming via a painting metaphor that is similar to the creation of tile maps and sprites. To meet these goals, we severely restricted the expressiveness of the programming model, as detailed in the previous section. Here we focus on the user interface to support the programming model.

Creating and Browsing Rules

Rules are identified by their event type and the kind of sprite at the center tile of the **When** neighborhood. The rule selector screen of Figure 3(e) was designed to give users an overview of the space of rules. The user can choose among the four sprite kinds on the left; the four quadrants update to display the rules available for that sprite kind. Each quadrant summarizes a related group of rules, parameterized by direction. As shown in the figure, the boulder has been selected: we can see in the lower-left quadrant that there is a collision rule for the case of the boulder moving down, as this square is highlighted. If a square is not highlighted, then there is no rule of that type for the sprite. Clicking on a blank square will create a new rule of that type and enter the rule editor so the user can edit the rule. Clicking on a highlighted square will enter the rule editor on the first instance of that rule type. Once in the rule editor, the user can browse all the rules associated with the sprite kind without returning to the rule selector screen.

Editing Rules

The rule editor (see Figure 3(f)) has three main active areas besides the top menu bar: (1) the 3x3 neighborhood of the **When** section allows the user to define the Include/Include'/Exclude sets of the tile predicates; the column immediately to the right of the **Do** keyword identifies the sprite witnesses and allows the user to set the direction predicates for them; (3) the squares to the right of that column allow the user to define the commands associated with the central tile/sprite and its four adjacent tiles/sprites.

Editing and Visualizing Tile Predicates

Selecting a tile in the 3x3 neighborhood brings up a menu that allows the user to attribute each background/sprite kind with a green check mark (Include set), yellow dot (Include' set), or a red-slash circle (Exclude set). Figure 5(b) shows the attribute menu for the central tile (where both diamond and boulder have been added to the Include set).

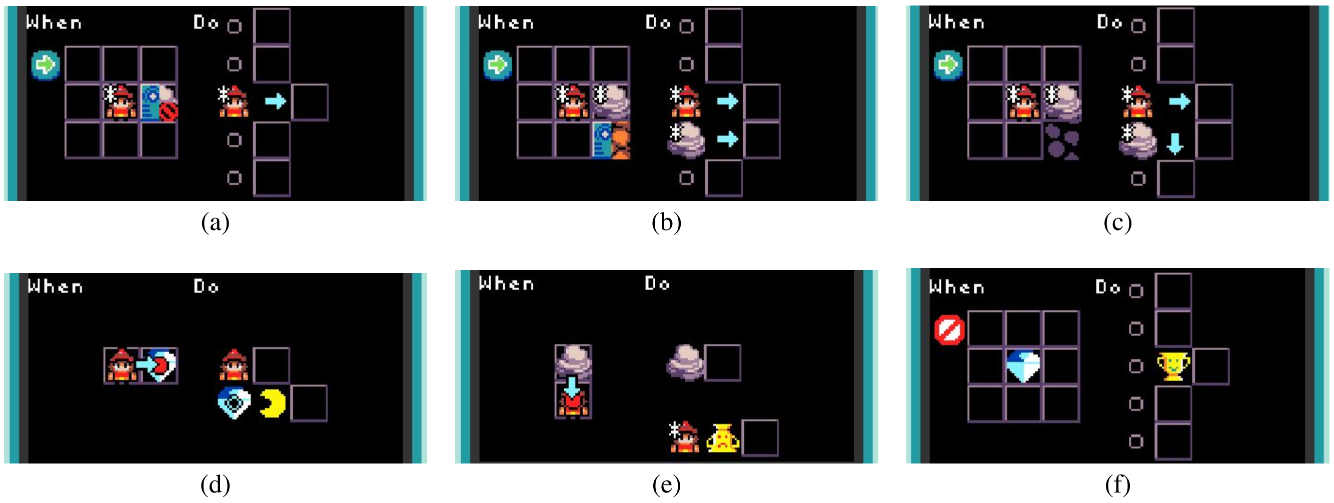


Figure 6. Boulder Dash game: rules for (a) moving the player avatar, (b-c) pushing boulders, (d-e) handling collisions, and (f) game win condition.

A main issue is how to summarize the multi-faceted attribution for a tile predicate in the 16x16 pixels of the tile. Our choice is to "accentuate the positive", as seen in Figure 3(f). The attribution for the tile below the central boulder includes the "space" background and excludes the diamond and boulder. This is summarized by showing just the space background in the tile below the central boulder (that is we prioritize visualizing non-empty Include sets over non-empty Exclude sets). The user can move the cursor over the tile to see the precise attribution, as shown just above the **When** section. If a tile's Include sets are empty but Exclude set is non-empty, we will then display the exclusion, adding the red-slash circle to make it clear (see Figure 6(a) for an example).

Direction Predicates

If a tile predicate has identified a sprite witness, then we may wish to constrain the direction of that sprite. Selecting the sprite in the column to the right of the **Do** label, brings up the direction predicate menu, as shown in Figure 5(a). The currently selected direction predicate (resting) is selected and may be changed by navigating to the desired new predicate and selecting it. Note that the direction predicate is reflected in the tile predicate in the **When** section, as the predicate takes part in the pattern matching, as described previously.

Commands

We finally come to the editing of commands in the five rows of the **Do** section, corresponding to the center tile and its four adjacent tiles in the **When** section. Figure 5(d)-(f) demonstrates the steps needed to program the move down command (solid blue arrow pointing down). As shown in Figure 5(d), we start by selecting the tile to the right of the boulder (sprite witness) in the **Do** section, which brings up the command menu at the top of the screen. We then select the leftmost menu item (the blue 4-way move icon), which brings up a sub-menu of four move commands, as shown in Figure 5(e), selecting the move left command. We move the cursor to select the move down command (Figure 5(f)). Selecting an existing command from the **Do** section allows one to change the command or delete it.

Game	Base Rules	Derived Rules	Tile Predicates	Cmds.
Boulder Dash	14	15	29	16
Snake	9	24	15	15
Bejeweled	11	14	33	38

Table 1. Table summarizing the TileCode implementations of the three games (see main text for details).

Generalizing Rules: Multiple Sprites and Directions

There are two main ways to generalize a rule in TC. The first is to have a rule apply to multiple kinds of sprites. For example, in Boulder Dash, diamonds fall just like boulders do (so far, we have shown the falling rules for just boulders). Figure 5(b) shows how we generalize a boulder falling rule to include the diamond: this is done simply by adding the diamond sprite to the Include set of the center tile. The center tile now shows half of each sprite (more than two sprites can be added to the Include set, but the visualization shows at most two).

The second way to generalize a rule is by direction. As we have seen, sprite directions are stated explicitly by the user, as shown in Figure 4. Often, when a user codes a behavior for a sprite to move in one direction (boulder tumbling to the left), they will also want to code a behavior for the opposite direction (boulder tumbling to the right).

As shown in Figure 5(c), TC provides a feature for *deriving* new rules from existing rules: flip vertically/horizontally and rotate clockwise/counter-clockwise (by 90 degrees). In this case, we have used the flip horizontal operation to create the desired derived rule. The derived rules are represented as views of the original rule, so if the user changes the original rule, the changes are propagated to the derived rules. The existence of derived rules is shown in the rule editor by a yellow dot on the flip icon (to the left of the garbage can). The parent-child relationships between rules created by this feature cannot be broken, but they can be reverted.

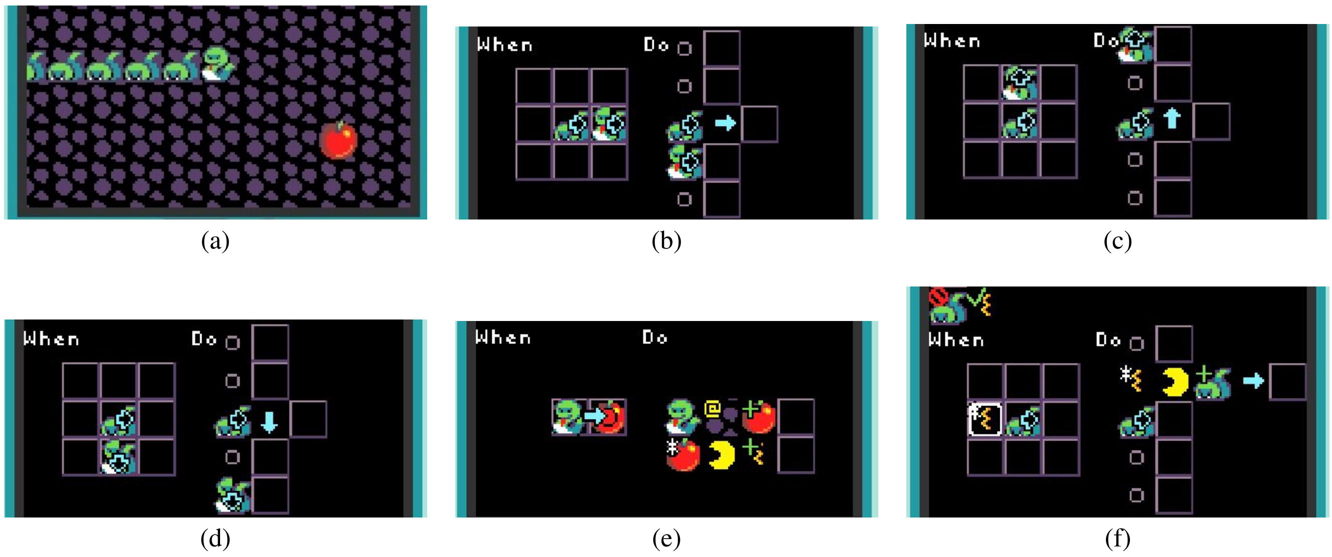


Figure 7. (a) Screen snapshot of the Snake game and (b)-(f) some TC rules of the game.

EVALUATION I: THREE POPULAR GAMES

In this section, we show that three popular video games can be expressed in TC with a modicum of rules. We informally describe each game's mechanics and show how it can be coded in TC. Table 1 summarizes the total number of rules, derived rules (via flip/rotate operations), tile predicates and commands for each game. A number of other games, including Pac-Man, have been coded in TC and can be found at the URL given in the Introduction.

Boulder Dash

The Boulder Dash game mechanics are as follows: the user guides the player's avatar (red elflike figure) through the map. The avatar (and all other sprites) cannot cross walls (blue tiles) or boulders; the player turns "dirt" (orange tiles) into "space" (dark tiles) upon moving onto them. The goal is to collect all the diamonds in the map, while avoiding falling rocks (a rock is a boulder or a diamond). Rocks fall when there is space below and can also tumble off rocks (to the left or right), as we have already seen. The player can push a boulder to the left or right when there is space available on the other side of the boulder.

We have already presented rules for expressing how boulders fall. Figure 6 shows rules for moving the player avatar and its interaction with boulders and diamonds: (a) on a right-dpad button press, when there is no wall or boulder to the right of the avatar, it is instructed to move right (rule generalized to four directions by rotation); (b) on a right-dpad button press, when there is a boulder to the right of the avatar, and no space below the boulder, both the avatar and boulder are instructed to move right (rule also flipped horizontally);[1] (c) is similar to (b) but has the boulder fall into the empty space below to make room for the player to move to the right (rule also flipped horizontally); (d) is a collision rule for a player moving onto a diamond at rest, which results in the diamond being consumed (generalized to four directions by rotation); (e) is a collision rule for a boulder falling onto a player, which results in the game ending with a loss; (f) is a negation rule that checks for the absence of diamonds on the board - in this case, the game ends with a win.

The full implementation of the mechanics described above takes a total of 14 rules, as show in Table 1.

[1]This rule optimistically assumes that there is a space to the right of the boulder; collision rules are used to determine if this is not the case and to stop the movement of boulder and player.

Snake

The mechanics of the Snake game (see Figure 7(a)) are as follows: the snake is always in motion; the user guides the head of the snake using the direction pad and each segment of the snake's body follows the one in front of it (its predecessor). If the head of the snake hits a wall or the snake's body, then the game ends. The effect of eating an apple is to increase the length of the snake by one segment, as well as to spawn a new apple at a random position.

Let's first describe the motion of the snake. We must start the snake moving with no input from the user, so we choose to create the snake oriented from head at right to tail at left, as shown in Figure 7(a), and start the head and segments moving right from rest (this rule is simple and not shown). The dpad controls the direction of the snake's head (rule not shown).

Making a snake segment follow its predecessor is an interesting challenge. Consider a segment that has just moved right into the center tile and the question: which tiles can its predecessor occupy and which direction must the predecessor have moved to now be in that tile? As shown in Figure 7(b)-(d), there are three basic cases and the move command for the center (following) segment must be in the direction that its predecessor last moved (all these rules are generalized to 4-way by rotation).

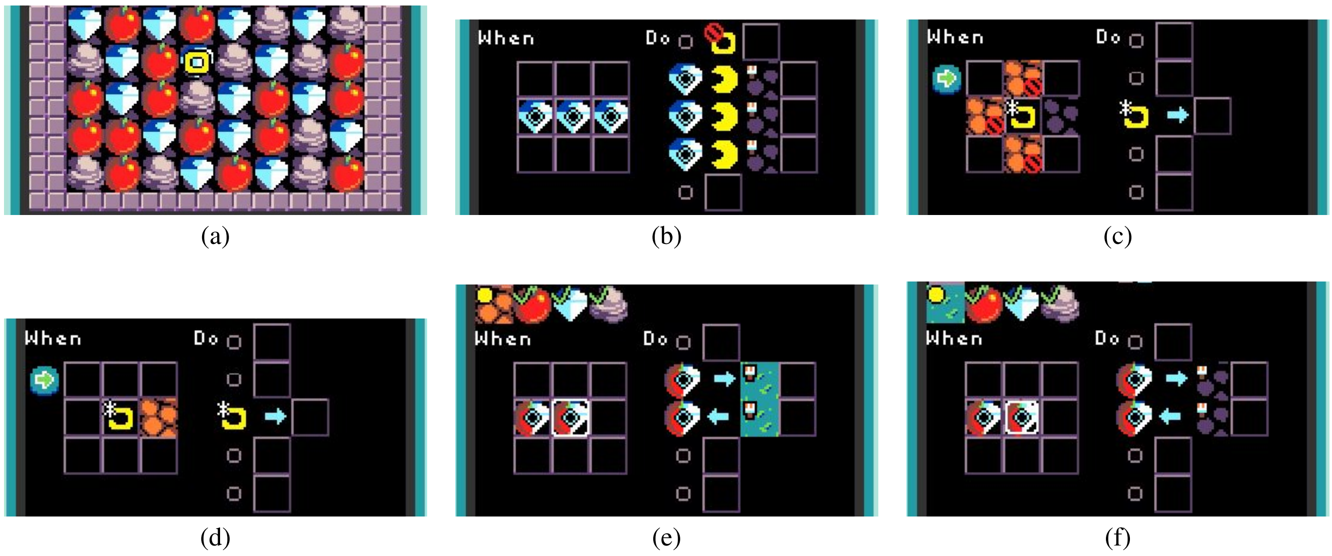


Figure 8. (a) Screen snapshot of Bejeweled game and (b)-(f) some TC rules of the game.

Growing the snake is done using two rules. As shown in Figure 7(e), when the head of the snake collides with the apple, the apple sprite is destroyed (via the pac-man "eat" command) and an apple core sprite is spawned in its place. This rule also invokes the at-sign "portal" command: this command randomly picks a tile location on the map with the given background that has no sprite on it; the command following the at-sign command (a spawn of an apple) is executed at that location.[2] As seen in Figure 7(f), when the apple core immediately follows the end of the snake, the apple core sprite is destroyed and a new snake segment is spawned, moving in the same direction as the end of the snake.

Bejeweled

Bejeweled has the following mechanics: the user navigates a cursor around a field of jewels arranged in a matrix (see Figure 8(a)); the user selects a jewel (with the A button) and then moves the cursor to an adjacent tile and presses the A button again to swap the previously selected jewel with the current jewel. If the swap operation creates a sequence of three or more jewels of the same kind then that sequence is eliminated, leaving empty space; otherwise, the swap is undone. Jewels fall into empty space. When jewels come to rest, they may create new three-jewel sequences to eliminate. Once all jewels stop moving, the user may again select a pair of jewels to swap. We do not consider the mechanic of filling empty space with new jewels here, due to lack of space.

Figure 8(b) shows the basic rule for removing sprites: if there are three diamonds at rest in a row, they are all removed and the backgrounds painted with the empty space (there is a new command shown in the top row - more on this later). The parallel semantics of rule matching means that this single rule can eliminate a row of diamonds of any length. The rule is generalized to a column via rotation. Rules to remove sequences of apples and boulders must be coded separately. Making sprites fall into empty space is straightforward and has been covered already.

The main challenge of Bejeweled is the implementation of cursor motion, the swap operation, and its potential interaction with the removal/falling of jewels. We use the orange-dirt background (dirt, for short) to mark the tile where the user first presses the A button (rule not shown). Pressing the A button when the cursor is on a dirt tile will mark the tile empty (also not shown), to permit the user to abort the swap and move freely again. The cursor motion rules of Figure 8(c) and (d) will limit the cursor motion to the dirt tile and its four adjacent tiles.

Now, when the user moves the cursor to a tile adjacent to the dirt tile and presses the A button, another dirt tile will be set, denoting the pair of sprites to be swapped. The rule of Figure 8(e) checks for a pair of sprites that both have dirt backgrounds behind them (note the use of both the Include and Include2 sets here). The sprites are swapped and their backgrounds are changed to blue (so rule (e) will not fire again). The rule of Figure 8(f) matches on blue backgrounds behind two resting sprites and swaps the sprites back while resetting the background to empty space.

Once the sprites have been swapped the first time, the sprite removal rules will fire. Recall that the removal rules, if successful, reset the backgrounds of the removed sprites to empty space. Thus, if no sprites are removed then the blue backgrounds will remain and the second swap rule will fire. On the other hand, if a removal rule does fire then two interesting things happen: at least one blue background and perhaps both will be set to the empty space. We use an extra rule (not presented here) to deal with the case when exactly one blue background is left. The second aspect of the removal rule (Figure 8(b)) is that it disables all the rules associated with the

[2]The portal command has many other uses: when there are just two tiles of a given background *B* in the tile map, the portal command can be used to send a sprite from one of the *B* tiles to the other *B* tile.

Game	Blocks	Globals	Arrays	Loops
Boulder Dash	107	2	1	2
Snake	52	2	1	1
Bejeweled	111	5	1	9

Table 2. Table summarizing MakeCode Arcade implementations.

yellow cursor for the next round. That is, as long as jewels are being removed, the cursor will remain stationary. Once this removal rule stops firing, then the cursor will be able to move again.

Summary
We see from Table 1 that two to three handfuls of base rules suffice to encode the basic mechanics of all three games. How complex are these rules and how much reuse of the base rules do we get through generalization? The ratio of tile predicates to base rules and of commands to base rules gives us a measure of rule complexity, while the ratio of derived rules to base rules gives us a measure of rule reuse: **Boulder Dash** has, on average, 2.2 tile predicates, 1.2 commands and 1.1 derived rules per base rule; **Snake** has 1.7 tile predicates, 1.7 commands and 2.7 derived rules per base rule; **Bejeweled** has 3 tile predicates, 3.5 commands and 1.3 derived rules per base rule. From this basic analysis, we can see that the Snake game has the simplest rules and most reuse of the three games.

EVALUATION II: BLOCKS-BASED VERSIONS OF GAMES
The paper's last three authors, professional software developers on the MakeCode Arcade team, each wrote a version of one of the three games (Boulder Dash, Snake, Bejeweled) using the Blocky editor of MakeCode Arcade. The input to this process was the informal game mechanics description created by the first author, though all games were already well-known by all; the TC versions of the games also were available to play. The goal was to implement the same basic mechanics of these games using the standard blocks and APIs available in MakeCode Arcade. Each co-author implemented his/her game in isolation (not difficult in these times). Table 2 contains basic metrics about the games. For each block-based game, we discuss how it implements the basic game mechanics and compare to the approach taken in TC. Links to the games' code can be found from the URL given in the Introduction.

Boulder Dash
Unsurprisingly, the movement of boulders in Boulder Dash is the central mechanic of the game. The blocks-based version uses a recursive algorithm to compute the next state for each rock, with user-defined functions to determine if rocks should fall or tumble. In this game especially, the TC concepts of a round and internal command conflict resolution allows the state to synchronize separately from the game engine, which helps to resolve inherent conflicts between different rules before rendering the next frame. That is, there is a strict separation of model-view-controller (MVC) enforced in TC. In MakeCode Arcade, once a sprite is rendered to the screen, it's visible to the engine and treated as a valid state, which puts the onus on the programmer to maintain the MVC abstraction.

Snake
As discussed before, the main complication of the Snake game is to have the snake's body segments follow their predecessors. The block-based implementation represents the snake as an array, stored in a global variable, and loops over the array to change the direction of each segment to that of its predecessor. In TC, we used the direction each segment last moved to identify predecessor segments using three rules and pattern matching. The looping over the segments happens implicitly in TC and there is no need for a separate array.

It is worth noting that without TC's direction predicate, it would be impossible to identify the predecessor of each sprite body segment; since the snake can double back on itself, a given sprite can have more than one adjacent tile containing another body segment—in such situations, the direction is needed to distinguish which of the adjacent tiles contains the predecessor segment. In an imperative language with arrays, this problem does not arise, as the ordering of sprites in the array establishes the predecessor relation.

Bejeweled
In the TC implementation of Bejeweled, there is one rule to say "if three consecutive items match, eliminate them", about the most declarative statement of the mechanic possible. In the MakeCode Arcade version, this mechanic is implemented by using a sliding three-wide window that traverses every row and column of sprites and checks each triple of jewels for sameness. This is done using nested loops over a 2D array of sprites, with three local variables indexing into the array to track the current triple.

The TC has a runtime optimization that keeps track of which tiles/sprites changed in the previous round and only checks a rule on a sprite when its neighborhood has changed in the last round. This means that not every sprite has to be examined, as in the MakeCode Arcade version.

Summary
In all three games, we see that the pattern matching capability in TC eliminates the need for explicit use of loops and arrays, compared to the block-based implementations. While TC generally has less code, some of the TC rules take quite a bit of thinking to come up with (for example, changing the tile backgrounds to keep track of swap state in Bejeweled). On the other hand, such state seems to be required to implement the mechanics correctly, regardless of the programming paradigm chosen.

LIMITATIONS
The TC programming model was designed so that: (1) games could be created on a gaming handheld with a small screen and limited input affordances; (2) a variety of popular games could be created. This section describes the major limitations of the programming model and how this affects the kinds of games that can be created with the model.

Local State/Compute/Behavior
By design, the only state values in TC are the tile background and anonymous sprite objects (with a kind and direction), each

having a visual representation. Furthermore, as in cellular automata, predicates/commands may only access a finite 3x3 neighborhood relative to the center sprite of a rule, so that instantaneous state access at an arbitrary distance (as enabled by random access memory) is not possible.

This restriction makes certain mechanics impossible to achieve. For example, in Pac-Man, when the pac-man avatar eats a power-up all the ghosts immediately turn dark blue and run away from the pac-man, reverting to their original form after some period of time. This mechanic is not possible in TC currently. One realization would set a global countdown counter to a non-zero value upon a power-up and count down to zero. When the counter is non-zero, the ghosts would exhibit the fleeing behavior [15]. This would require an extension to TC to support global counters.

It is also difficult to program ghosts to move towards (or away from) pac-man, as there is no mechanism for comparing the locations of different sprites in TC.

No Hidden Computation

In TC, every command has a visible effect, whereas in general programming many commands have no visible effect. For certain game mechanics, it would be useful to perform an (invisible/hidden) computation over the tile map to decide whether or not to perform some visible action. A good example is the mechanic of clearing a complete row/line in Tetris: if the insertion of a Tetris piece completes a line then that line is removed. From the user's view, the determination of whether a line is complete happens instantaneously (though the actual removal of a complete line may take some time). Such an (apparently) instananeous and invisible computation currently is not possible in TC.

Synchronous Grid-based Motion

In TC, sprites may only move in one of four directions; furthermore, all sprites move synchronously and at the same velocity. This severely limits the kinds of mechanics possible, such as missiles that move faster than the player/enemy sprites and parabolic motion, such as found in Angry Birds.

CONCLUSION

We have designed and developed a novel game programming model, TileCode, for low-cost gaming handhelds that enables visual programming on small screens, yet is expressive enough to program a number of different popular video games. Its ability to pattern match on tiles, sprites, and sprite directions allows for a surprising amount of expressivity. TileCode's programming constructs are made available via an editing paradigm that is also tiled-based.

Users studies are clearly needed to evaluate TileCode. We will begin by seeing if users can successfully make small modifications to existing games, after a brief demonstration and walkthrough. We will follow-up with a set of interesting challenge problems and evaluate the solutions that novices produce.

ACKNOWLEDGEMENTS

Thanks to the MakeCode team, Peli de Halleux, Stefania Druga, Teddy Seyed, and Kimberly Ying for their input.

REFERENCES

[1] PICO-8 Fantasy Console. In `https://www.lexaloffle.com/pico-8.php`. Accessed July 2, 2020.

[2] Thomas Ball, Peli de Halleux, and Michal Moskal. 2019. Static TypeScript: an implementation of a static compiler for the TypeScript language. In *16th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. ACM, 105–116.

[3] Rahul Banerjee, Jason Yip, Kung Jin Lee, and Zoran Popović. 2016. Empowering Children To Rapidly Author Games and Animations Without Writing Code. In *15th International Conference on Interaction Design and Children (IDC '16)*. 230–237.

[4] Stephen Cooper, Wanda Dann, and Randy Pausch. 2003. Teaching objects-first in introductory computer science. In *34th SIGCSE Technical Symposium on Computer Science Education*. ACM, 191–195.

[5] Allen Cypher and David Canfield Smith. 1995. KidSim: End User Programming of Simulations. In *Human Factors in Computing Systems, CHI*. ACM/Addison-Wesley, 27–34.

[6] James Devine, Joe Finney, Peli de Halleux, Michal Moskal, Thomas Ball, and Steve Hodges. 2019. MakeCode and CODAL: Intuitive and efficient embedded systems programming for education. *Journal of Systems Architecture* 98 (2019), 468–483.

[7] Edsger W. Dijkstra. 1975. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Commun. ACM* 18, 8 (1975), 453–457.

[8] Jay Fenton and Kent L. Beck. 1989. Playground: An Object-Oriented Simulation System With Agent Rules for Children of All Ages. In *Conference on Object-Oriented Programming: Systems, Languages, and Applications*. ACM, 123–137.

[9] Neil Fraser. 2015. Ten Things We've Learned from Blockly. In *2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond)*. 49–50.

[10] Werner Hartmann, Jürg Nievergelt, and Raimond Reichert. 2001. Kara, finite state machines, and the case for programming as part of general education. In *IEEE CS International Symposium on Human-Centric Computing Languages and Environments*. IEEE Computer Society, 135–141.

[11] Kenneth M. Kahn. 1995. ToonTalk - Concurrent Constraint Programming for Kids. In *Twelfth International Conference on Logic Programming*. MIT Press.

[12] Caitlin Kelleher, Randy F. Pausch, and Sara B. Kiesler. 2007. Storytelling Alice motivates middle school girls to learn computer programming. In *Conference on Human Factors in Computing Systems, CHI*. ACM, 1455–1464.

[13] Isabelle Lefebvre. 2017. Creating with (Un)Limited Possibilities: Normative Interfaces and Discourses in Super Mario Maker. *The Journal of Canadian Game Studies Association* 16 (2017), 196 – 213.

[14] Jiangjiang Liu, Cheng-Hsien Lin, Joshua Wilson, David Hemmenway, Ethan Philip Hasson, Zebulun David Barnett, and Yingbo Xu. 2014. Making games a "snap" with Stencyl: a summer computing workshop for K-12 teachers. In *The 45th ACM Technical Symposium on Computer Science Education, (SIGCSE)*. ACM, 169–174.

[15] Richard G. McDaniel and Brad A. Myers. 1999. Getting More Out of Programming-by-Demonstration. In *Conference on Human Factors in Computing Systems, CHI*. ACM, 442–449.

[16] Radu Muschevici, Alex Potanin, Ewan D. Tempero, and James Noble. 2008. Multiple dispatch in practice. In *23rd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 563–582.

[17] Thorbjørn S. Nielsen, Gabriella A. B. Barros, Julian Togelius, and Mark J. Nelson. 2015. Towards generating arcade game rules with VGDL. In *IEEE Conference on Computational Intelligence and Games*. IEEE, 185–192.

[18] Seymour Papert. 1980. *Mindstorms. Children, Computers, and Powerful Ideas*. Basic Books, NY.

[19] Richard E. Pattis. 1981. *Karel the Robot – A gentle introduction to the art of programming*. Wiley, New York.

[20] Alexander Repenning and Tamara Sumner. 1992. Using Agentsheets to create a voice dialog design environment. In *ACM/SIGAPP Symposium on Applied Computing*. ACM, 1199–1207.

[21] Alexander Repenning and Tamara Sumner. 1995. Agentsheets: A Medium for Creating Domain-Oriented Languages. *IEEE Computer* 3 (1995), 17–25.

[22] Alexander Repenning, David C. Webb, Catharine Brand, Fred Gluck, Ryan Grover, Susan B. Miller, Hilarie Nickerson, and Muyang Song. 2014. Beyond Minecraft: Facilitating Computational Thinking through Modeling and Programming in 3D. *IEEE Computer Graphics and Applications* 34, 3 (2014), 68–71.

[23] Mitchel Resnick, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, Eric Rosenbaum, Jay S. Silver, Brian Silverman, and Yasmin B. Kafai. 2009. Scratch: programming for all. *Commun. ACM* 52, 11 (2009), 60–67.

[24] Tom Schaul. 2013. A video game description language for model-based or interactive learning. In *IEEE Conference on Computational Inteligence in Games*. IEEE, 1–8.

[25] Joel L. Schiff. 2008. *Cellular Automata: A Discrete View of the World*. Wiley-Interscience.

[26] Teddy Seyed, Peli de Halleux, Michal Moskal, James Devine, Joe Finney, Steve Hodges, and Thomas Ball. 2019. MakerArcade: Using Gaming and Physical Computing for Playful Making, Learning, and Creativity. In *Extended Abstracts of the Conference on Human Factors in Computing Systems, CHI*. ACM.

[27] David Canfield Smith, Allen Cypher, and James C. Spohrer. 1994. KidSim: Programming Agents Without a Programming Language. *Commun. ACM* 37, 7 (1994), 54–67.

[28] David Canfield Smith, Allen Cypher, and Lawrence G. Tesler. 2000. Novice Programming Comes of Age. *Commun. ACM* 43, 3 (2000), 75–81.

[29] Kathryn T. Stolee and Teale Fristoe. 2011. Expressing computer science concepts through Kodu game lab. In *42nd ACM Technical Symposium on Computer science Education (SIGCSE)*. 99–104.

[30] Rokas Volkovas, Michael Fairbank, John R. Woodward, and Simon M. Lucas. 2019. Mek: Mechanics Prototyping Tool for 2D Tile-Based Turn-Based Deterministic Games. In *IEEE Conference on Games*. IEEE, 1–8.