Proceedings of the Twenty-First AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment (AIIDE 2025)

Game Behaviour Trees Using Tile Rewrite Rules

Kaylah Facey, Cynthia Li, Chris Martens, Seth Cooper

Northeastern University, Boston, MA, USA
{facey.k, li.cynth, c.martens, se.cooper}@northeastern.edu

Abstract

Game creation tools that minimize required resources and knowledge to use them have transformed the practice of learning and prototyping game design, especially in hobbyist and indie development contexts. However, little is known about how the different programming models found underlying these tools affect their expressiveness and usability. A recently proposed programming model allows creators to author the logic of the entire game using a single "game behaviour tree" with tile-grid rewrite rules at the leaves. We contribute to this body of knowledge by studying this recently proposed programming model. We have used it to make clones of popular games as case studies, from which we extracted a number of design patterns. To gain formative information about usability, we also conducted a small user study with people who have varying levels of experience authoring games with other tools. We find game behaviour trees capable of expressing a wide variety of 2D turn-based games. Study participants are quick to grasp the underlying concepts, but further research is needed to understand discrepancies with user intuitions that may arise from their familiarity with different programming models.

1 Introduction

The recent history of game creation tools has taught us the many-faceted value of low-barrier-to-entry prototyping tools: for novices, they offer a pathway into game development without prior programming experience; for experienced game developers, they create opportunities to rapidly prototype new ideas without investing in high fidelity design.

We aim to contribute to the design space of *programming models* that can underlie such a tool. To this end, we previously introduced Tile Rewrite Rule Behaviour Trees (TRRBT) (Zhou, Martens, and Cooper 2024). TRRBT describes 2D tile- and turn-based games in a generic way. All language constructs included in TRRBT support its two foundational concepts: *behaviour trees* and *tile rewrite rules*. In order to better understand the expressiveness and usability of this model, we undertook a dual-method evaluation. To capture the expressiveness of TRRBT, we developed case studies of a variety of different games and documented common design patterns. To investigate the ease of learning the programming model, we conducted a small formative user study with a web-based graphical interface.

We find that TRRBT is capable of describing a variety of 2D tile-based games using simple constructs. In our case studies, we found a number of useful design patterns that aid in code reuse and organization, as well as a few major limitations of the language. In our user study, we observed that participants readily understand the basic principles of TRRBT but sometimes misapply concepts from other programming models that may be more familiar to them.

The contributions of this work are:

1. An evaluation of the expressiveness of TRRBT as a suitable language for 2D tile- and turn-based games in the form of design patterns from a series of case studies;
2. Findings from a formative user study that chart a course for improving the tool's usability and give rise to questions about programming model understanding in the context of game development.

2 Related Work

2.1 Behaviour Trees

Behaviour trees (BTs) are a method of controlling agents that has been extensively used in both industry (Sekhavat 2017) and research (Iovino et al. 2022). Behaviour trees use a hierarchical tree structure of tasks and subtasks, including tasks that depend on the outcome of other tasks (Iovino et al. 2022). They are made more modular by enforcing a limited set of possible task return values ("success", "failure", [often] "running"). Behaviour trees are more scalable and extensible than finite state machines (FSMs) (Iovino et al. 2022; Sekhavat 2017), and studies using graphical BT editors have found that they are usable even by designers with minimal programming experience (Sagredo-Olivenza, Gómez-Martín, and González-Calero 2015; Becroft et al. 2011).

Most research on behaviour trees for games has focused on NPCs, including swarm behaviour and computer player strategy (Iovino et al. 2022). Our work focuses on "game behaviour trees" (GBTs), which control interrelated parts of a game that may have conflicting goals. As early as 2002 with *Façade* (Mateas and Stern 2002), behaviour trees have used specialized nodes or, in 2011, subtrees (Shoulson et al. 2011) to coordinate events between agents. The earliest example we have found of using a behaviour tree to control an entire game or game system is work by Kapadia et al. (2015), who studied "Interactive Behaviour Trees" split into subtrees that control

different parts of an "interactive narrative." They found that a GBT has lower complexity and is easier to author than an equivalent FSM-based story graph. *Villanelle* (Martens and Iqbal 2019) uses a collection of behaviour trees to control all aspects of an interactive fiction game. Recently, Normoyle, Jörg, and Hill (2024) introduced the "Curation Tree", a Unity framework using a centralized "World" behaviour tree to control an entire game. They found that using a behaviour tree to control a game makes it easier to organize and maintain game code.

2.2 Rewrite Rules

We define "rewrite rules" as a kind of programming that defines behaviour through "replacement semantics" (Martens 2015). A rule consists of two patterns, a left hand side (LHS) and a right hand side (RHS). When the rule is invoked, the game state is updated to replace the LHS with the RHS, if the LHS is found anywhere in the state. In contrast to a generative grammar, which is typically meant to produce a static artifact such as a level map (Dormans and Bakkes 2011), rewrite rules are intended to produce a set of dynamic *behaviours* for a running program. As early as the 90s, rewrite rules have been used to describe changes in a program (Repenning 1995). Rewrite rules are considered understandable enough to use in children's programming environments (Wright 2006), and thousands of people have used them to create games with the popular "tiny online game engine" PuzzleScript (Warren 2019). *Sturgeon* (Cooper 2023; Cooper and Bazzaz 2024) has used rewrite rules to generate levels with example playthroughs of games. Rewrite rules do not have to operate on a grid- or graph-like structure; rewrite rules are used by *Ceptre* (Martens 2015; Martens et al. 2023) to operate on multisets of predicates that represent game state in a genre-agnostic way.

2.3 Game Programming Tools

As noted by Van Rozen (2020), games require multiple iterations of design and testing before they can be considered complete, making tools for rapid prototyping highly valuable. In addition, game designers often lack advanced programming skills, making it infeasible for them to enact their designs in complex game-making software such as Unity without relying on programmers, which slows down iteration time. Recently, "tiny online game engines"[1] like PuzzleScript have become a popular way to create small games and prototypes quickly (Warren 2019). These tools benefit from a streamlined set of available options, allowing designers to quickly learn to use the tool. A goal of this research is to assess the suitability of TRRBT (Zhou et al. 2024) as the model underlying such a tool. Although TRRBT and PuzzleScript share the use of rewrite rules as a core concept, TRRBT introduces the innovation of using a behaviour tree structure for the control flow. In addition, whereas PuzzleScript bakes in the concepts of directions and collisions, TRRBT relies on pure pattern rewrites. This allows us to study the expressiveness of the programming model in a more generic way.

[1]Although TOGEs are sometimes marketed as "no-code", we take a broad view of programming that includes the domain-specific scripting and authoring languages that creators write when using them.

3 System Overview

Our system consists of two parts: (1) the underlying TRRBT language, as defined in our previous work (Zhou et al. 2024), and (2) a browser-based graphical user interface for writing, running, and editing TRRBT games.

3.1 The TRRBT Language

TRRBT combines the principles of behaviour trees and rewrite rules by using "tile rewrite rules" as the leaf nodes of a game behaviour tree. A *tile* is a string (with no spaces). A *tile rewrite rule* defines the replacement of a group of tiles, called a *pattern*, with another pattern of the same dimensions. Tile rewrite rules are used to enact all mechanics in TRRBT games.

By combining the two concepts into one language, we are able to take advantage of their synergy. Rewrite rules are a natural choice for representing mechanics in GBTs, as they give tasks a common language for enacting state changes. This reduces the number of concepts authors need to learn and increases modularity.

On the other hand, behaviour trees are a convenient control flow for tile rewrite rules, as they work well with the the use of *transform nodes* that enhance tree reuse by applying transformations hierarchically. Transform nodes do not represent any runtime behaviour, but rather modifications to be performed to the tree itself before it is run.

We can divide TRRBT nodes into three main node types: Control Nodes, Tile Nodes, and Transform Nodes. All nodes return either *Pass* or *Fail*. Unlike many behaviour trees (Sekhavat 2017), all execution is synchronous, and nodes cannot return *Running*. Nodes denoted with $^+$ have been added since Zhou et al. (2024).

Control Nodes are the behaviour tree nodes that internally control the logic flow through the tree by defining the order of operations of their children. Although some of our control nodes are analogous to the behaviour tree nodes common in other systems (Sekhavat 2017), there is no direct mapping. Using control nodes, a game author can specify sequences (*order*), when to short circuit (*all*$^+$*, none*), create loops (*loop-until-all, loop-times*), introduce non-determinism (*random-try*, *player*), or end the game (*win, lose, draw*) for a given agent.

Tile Nodes are the tile rewrite rule related nodes. Tile nodes specify tile rewrite rules (*rewrite*), check for existing patterns (*match*), change the composition of the gameboard as a whole (*set-board*, *append-rows*$^+$, *append-columns*$^+$, *layer-template*$^+$), and display the gameboard between player choices (*display-board*).

The state of the game is tracked on the *gameboard*. The gameboard is a 2D array of non-empty tiles. We have extended Zhou et al. (2024) by allowing the gameboard to have multiple named *layers*. Layers can be used to "stack" tiles in both the gameboard and tile rewrite rules, which we have found creates extensive opportunities for code simplification.

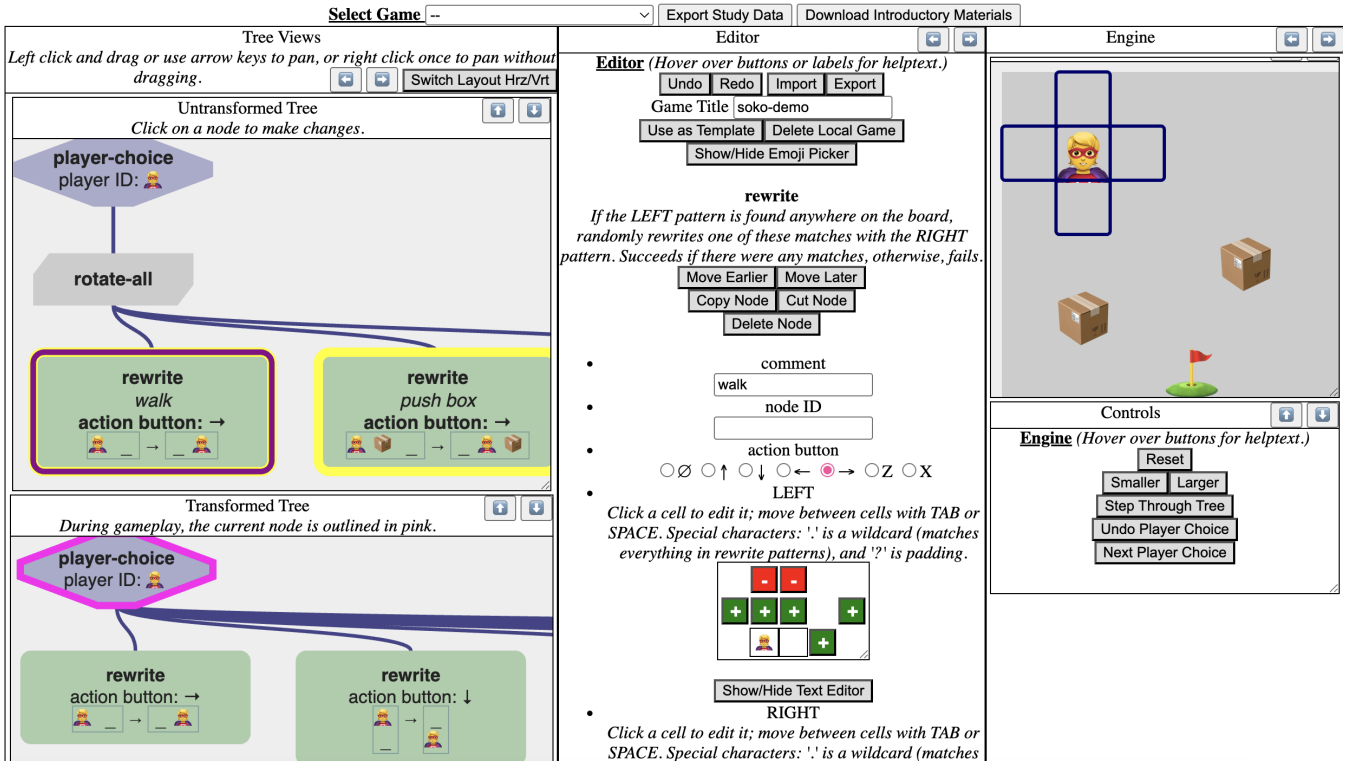

Figure 1: The browser-based interface. The user can edit the untransformed tree in the top-left, and the transformed tree is updated in the bottom-left. Editor controls are in the middle. This includes properties of the currently selected rewrite node (outlined in red). The game is played in the engine on the right; the current node is highlighted in magenta.

Transform Nodes are used to reduce redundancy in the tree by allowing parts to be reused. They are applied in a process we refer to as *transformation*, and we distinguish between the *untransformed* tree and the *transformed* tree. The untransformed tree is the tree edited by the game author, and the transformed tree is the tree that is actually executed by the engine.

Control-Related Transform Nodes can change the shape of the tree and thus the control flow. Using transform nodes, a game author can invoke plugins for code reuse (*x-link*, *x-file*) or organize the tree to improve readability (*x-ident*, *x-prune*).

Tile-Related Transform Nodes make changes to the tile patterns of their children. "Orientation" nodes manipulate rewrite patterns' shapes, flipping along an axis (*x-mirror*, *x-flip*), rotating (*x-rotate*, *x-spin*), or skewing them (*x-skew*). During transformation, they visit all leaf *rewrites* and *matches* and update nodes applying the given function. "Replacement" nodes change the tiles used in *rewrite* and *match* patterns, as well as the *player* player ID. As with orientation transforms, *x-replace* and *x-swap* update nodes for each child applying the swap or replacement. We extend Zhou et al. (2024) with *x-unroll-replace*$^+$, which creates sibling subtrees for each replacement, to enable dependencies between patterns (Figure 2).

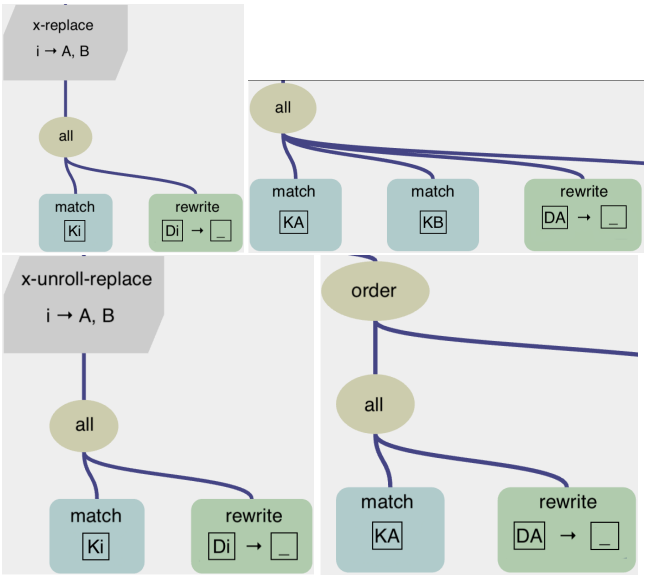

Figure 2: *x-replace* vs *x-unroll-replace* (left images, untransformed; right images, transformed): The player `P` can open a door `DK` if the matching key `Ki` is in their inventory. Using *x-replace* (top), both keys have to be in the inventory to open any door. With *x-unroll-replace* (bottom), we get the desired behaviour.

3.2 Interface Design

We developed a graphical browser-based authoring tool for editing and running TRRBT games. Our choice to develop a GUI authoring tool was motivated by the convenience of a code editor that prevents invalid syntax and provides behaviour tree visualization (Bacher and Martens 2021). Inspired by "tiny online game engines" (Warren 2019), "casual creators" (Compton and Mateas 2015), and creativity support tool research into supporting both novices and experts (Remy et al. 2020), we aim to develop an authoring tool that is both intuitive to use and highly expressive.

The graphical interface has two major components: the *editor*, for editing trees, and the the *engine*, for running games. We intend for them to be lightweight, and thus they are implemented as HTML and vanilla JavaScript in a single browser tab without reliance on large frameworks or external servers. The games themselves are stored as JSON to allow for easy import/export.

One of the biggest strengths of behaviour trees is their straightforward visualization, and our authoring tool foregrounds explorable visualizations of the tree (Figure 1). We include the transformed tree as well as the untransformed tree to show the effects of transform nodes (Figure 2). To edit the tree, authors click directly on the node they wish to make changes to. From the node edit window, authors can also create new nodes as parents or children. When the game is run using the included engine, the transformed tree highlights the currently executing node.

By default, the engine runs nodes until it needs to wait for a player choice. Players choose between available rewrites to make a move. Locations of possible rewrites are outlined on the gameboard and when moused over show what the rewrite would do. Rewrites can also have a keyboard button associated with them (↑, ←, ↓, →, Z, X). Normal gameplay only pauses on *player* and *display-board* nodes, so we include a debug "step" mode that stops execution and displays the tree after every node.

The features we chose to develop first were motivated by our intention to conduct a user study using the interface. To avoid confusing participants who would interact with our tool for a limited amount of time, we left out a number of advanced features: plugin transform nodes (*x-file, x-link*); the tile-related transform nodes *x-swap* and *x-unroll-replace* (keeping *x-replace*); and layers. Note that although editing is restricted, trees using these features can still be viewed and played in the interface if they are imported using JSON.

We do not consider our authoring tool to be complete in terms of either functionality or polish. In the future, we intend to improve its usability and eventually include the advanced features omitted for the study.

4 Methodology

We evaluate TRRBT using two methods.

1) To explore the expressiveness of TRRBT, we remade a number of popular 2D games and simulations, adding to the games that we introduced previously (Zhou et al. 2024). During the process, we have discovered a number of useful design patterns, as well as some limitations of the system.

2) To explore the usability of TRRBT and inform development of the authoring tool, we conducted a preliminary user study. Participants were given a set of tasks to perform using the tool, followed by a survey and semi-structured interview. We synthesized our observations and the survey results to get a qualitative view of the usability of the language and tool.

5 Expressiveness

A list of example games created by us is given by Table 1. All code used in this research (including the games) can be found on OSF[2].

5.1 Case Studies

To illustrate the strengths and limitations of TRRBT, we have selected a few example games to highlight here as case studies.

Lime Rick: In *Lime Rick*, the player controls a continuously growing snake that can climb over its own body or up to 4 tiles high unsupported. To grow the snake, we replace the head tile with a new body tile each time the head moves (e.g. `H _` ⇒ `B H`). To allow the head to climb 4 tiles high, we track its height state by assigning the head tile a number from 1 to 4 (e.g. `H1`), similarly to the PuzzleScript remake[3]. With an *x-replace* node, we define mechanics that are agnostic to the height of the head (*x-replace*[`H!` ⇒ `H1`, `H2`, `H3`, `H4`]), analogously to sub-typing in other languages. We use the same principle to allow the eat action to apply to multiple eatable objects. To implement gravity, we use *loop-until-all* to cause the head to fall (again creating body tiles) until there is no longer a space underneath it (*loop-until-all*[`H1` / `_` ⇒ `Bv` / `H1`]). The player wins when all the apples have been eaten, which we check for with a combination of *none* and *match* (*win*[*none*{*match*(`A`)}]). We created boards for 31 levels of *Lime Rick*. By using *x-file* plugins, we can keep the level definitions separate from the mechanics, allowing us to easily switch between levels.

Rust Bucket: In *Rust Bucket*, the player must escape a dungeon with various obstacles. 63 of its 129 nodes relate to enemy pathfinding. To implement pathfinding, we use a *loop-until-all* node and a "path" layer to fill out the best direction (e.g. `>`) to reach the player from each point on the board. Since there are no special direction characters that could be rotated using *x-spin*, pathfinding code is repeated for each direction. We also use *loop-until-all* to allow every enemy to move each turn. To enforce that each enemy moves exactly once without giving enemies unique identifiers, the nested rewrite adds a "tag" character (*loop-until-all*[`e _` ⇒ `_ e*`]), and a second *loop-until-all* removes the tags. Similarly, spikes in *Rust Bucket* change heights every turn, which we track using both tags and a numeric value similar to the head height in *Lime Rick*. A third layer is used to denote the locations of items that can be walked over (such as spikes

[2] https://osf.io/btqcj/

[3] https://www.puzzlescript.net/editor.html

	Game	Description	Nodes
Top-Down	*Sokoban* (Imabayashi 1982)	A top-down block pushing puzzle game in which the aim is to push every block onto a target while avoiding getting stuck.	10
	Rust Bucket (Nitrome 2015)	A turn-based top-down dungeon crawler including spikes, traps, and switches.	129
Side-View	*Block Dude* (Solutions 2007)	A side-view block pushing game in which the player must move and stand on boxes to make their way to an exit door.	53^{+}(12)
	Lime Rick (KissMaj7 2013)	A side-view snake variant in which he player controls a "snake" that can climb over its own body to reach an exit.	42^{+}(31)
	Turn Undead (Nitrome 2014)	A turn-based side-view game in which the player uses a gun that shoots stakes to both defeat enemies and create ladders.	89
Point and Click	*Lights Out* (Mérõ 1983)	A point-and-click game about turning on all the lightbulbs in a grid. Every time each lightbulb is flipped, its 4 nearest neighbours also flip.	15
	Minesweeper (Johnson 1990)	A point-and-click game about finding the locations of hidden mines. Uncovered tiles show the number of neighbouring mines.	147
Experimental	*Conway's Game of Life* (Berlekamp, Conway, and Guy 2004)	A Turing complete cellular automata (Rendell 2011) in which cells are "born" or stay "alive" if they are adjacent to 2 or 3 living neighbours. PatternProgrammer (Wright 2006) implemented the same program in 228 rules.	110
	Lost and Found	A text-based game inspired by interactive fiction games like *Cloak of Darkness* (Nelson 2022).	66
	Pong (Alcorn 1972)	A turn-based remake of the classic tennis arcade game.	129
	Blocks World (Slaney and Thiébaux 2001)	A classic planning problem in which blocks are rearranged using a single machine arm.	13

Table 1: Example games demonstrating the expressive range of TRRBT. [Counts]$^{+}$ indicate the count of mechanics-related nodes in games with multiple level files (level count given in parentheses).

and portals). This allows us to separate movement logic from item logic. A number of items in *Rust Bucket* are paired (for example, portal exits). To avoid duplicating code for each pair, we use *x-unroll-replace* to associate the rewrites for their shared mechanics. To select which of 3 implemented levels to play, we use *random-try* to initialize the gameboard.

Minesweeper: In *Minesweeper*, uncovered tiles show the number of neighbouring mines. 54 of the 147 nodes are used to create a count function. As TRRBT doesn't support variables, each increment from 1 to 8 has a separate rewrite. To capture all 8 neighbours without double-counting any mines, the count function uses a separate subtree for each neighbour. To avoid duplicating the increment rewrites, we use *x-link* with *x-replace* and transform nodes to implement the neighbour subtrees.

5.2 Design Patterns

We looked for common TRRBT design patterns in games we implemented. These could be useful for informing future tool or language design, such as adding a library of patterns to the authoring tool or language documentation.

Gravity: Gravity can be simply implemented with *loop-until-all*, as shown in *Lime Rick*.

Simple State Tracking: We have found that simple states can be modelled directly in the gameboard by annotating tiles with different characters and using an *x-replace* node to handle shared behaviours between states. This is used for the head height in *Lime Rick* and the timed spikes in *Rust Bucket*.

Tagging: Tagging, as shown for enemy movement in *Rust Bucket* and counting in *Minesweeper*, is a variant of Simple State Tracking we use extensively. By "tagging" each pattern that is visited by *loop-until-all*, we can track which entities have already been affected.

Direction Characters: Another useful variant of Simple State Tracking, we often use characters (e.g. `>`) to track direction of movement, as shown in *Rust Bucket* pathfinding.

Sub-typing: Similarly to Simple State Tracking, we've found that *x-replace* nodes can be used to implement a kind of sub-typing to apply the same behaviours to multiple types of tiles. *Lime Rick* uses sub-typing for different eatable objects.

Layer State Tracking: Layers are useful for more complex state tracking than Simple State Tracking, since authors only have to think about layers that are relevant to the current rewrite. Layers are also easier to conceptualize: instead of having to remember the order of characters in a state-related tile, authors can use named layers (e.g. instead of representing player inventory and health as a single tile `PIH`, it can be spread across layers: a location layer with `P`, an inventory layer with `I`, and a health layer with `H`).

Background Layer: As in *Rust Bucket*, we often use layers to distinguish between a background and moving objects.

Symmetric Players: In a two-player game where players share the same moves (e.g. *Tic-Tac-Toe* or *Checkers*), the player behaviour only needs to be implemented once. The other player can be added by using an *x-swap* (e.g. *x-swap*

[X⇒O] in *Tic-Tac-Toe*) combined with an *x-link* plugin to the first player subtree.

Conditionals: Though TRRBT doesn't include any conditional nodes, we often use a combination of *match* (check condition) and *none* (proceed if false) or *all* (proceed if true) to set preconditions. We can also use *all* and *order* to approximate AND and OR, respectively. *Match* and *none* are used to check the win condition in *Lime Rick*.

Plugins: Using *x-link* and *x-file*, we can import code from elsewhere, enabling code organization and reuse. This is used for level files in *Lime Rick* and counting in *Minesweeper*.

Random Initialization: The *random-try* node can be used to randomly initialize a level, as in *Rust Bucket*. One way to do this is simply to choose from predefined levels with *set-board*; however, it can also be used with *rewrite* to randomly place tiles (e.g. enemy locations) in empty spaces. We can also randomly "shuffle" the gameboard to initialize a level for a game where player moves are reversible, e.g. *Eights* (Zhou et al. 2024), by first setting the gameboard to a win state and using *random-try* with an *x-link* to the player's choices.

Layers for Global Behaviours: We can simulate global behaviours like path planning by using intermediary *rewrites* to encode information into a layer, as in *Rust Bucket*.

5.3 Language Limitations

Several common game patterns are challenging to implement with TRRBT.

Regex: We have found ourselves wishing for regular expressions in patterns. For example, there is no single-pattern way to match a group of tiles of undefined length. TRRBT does support a simple single-tile "wildcard" that functions more like the "empty" tiles of PatternProgrammer (Wright 2006), which skip that tile in the pattern.

Numeric Variables: Because there is no support for numeric variables, it can be verbose to make changes according to a numeric function, as seen in *Minesweeper*.

Text: Because tiles are composed of strings, TRRBT can display text. However, as each tile is the same size, it is difficult to create long text that displays well (e.g. for dialogue).

Multi-Sub-typing: *X-replace* is very useful for approximating sub-types, but it becomes unwieldy when there are patterns that include multiple tiles of one type. To handle this, it is necessary to nest *x-replaces*, which is tedious and hinders readability (See Figure 3).

Multi-Transformation: Shape-related transform nodes handle the case where an author wants rewrites that are identical except for their orientation, and tile-related transform nodes handle the case where an author wants rewrites that are identical except for a given tile, but we have found that we sometimes want to write rewrites that transform in both orientation and tile. Though it is possible to accomplish this by combining *x-link*, *x-replace*, and transform nodes (as in *Minesweeper*), it is unwieldy.

6 User Study

We conducted a preliminary user study to evaluate the learnability of TRRBT and the usability of our authoring tool.

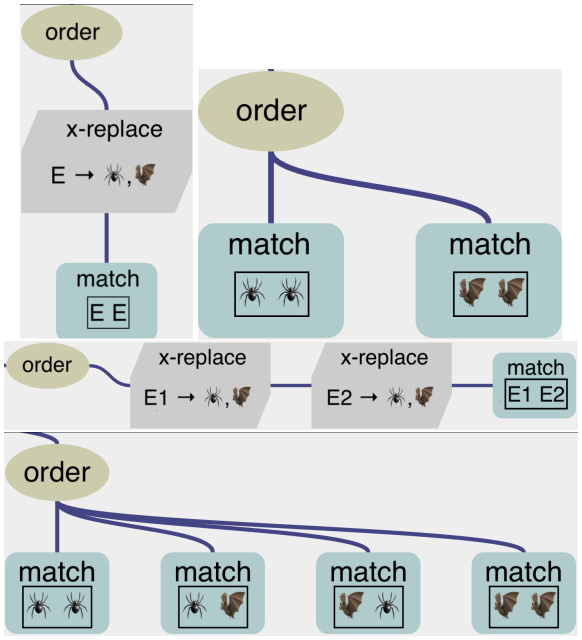

Figure 3: Multi-Sub-typing: One *x-replace* (top, untransformed shown left) fails to capture all possible configurations of enemies. Instead, two *x-replaces* must be nested (untransformed shown middle, transformed shown bottom).

6.1 Procedures

Methods were approved by the authors' IRB. Participants consented to participate in research during recruitment. They were compensated with a $20 gift card for their participation.

We recruited seven participants from game development forums for a 90 minute recorded Zoom session. For the first hour, they attempted a series of tasks using the authoring tool. This was followed by a ten minute survey covering their previous experience developing games and administering the System Usability Scale (SUS) (Brooke et al. 1996). Finally, we conducted a 20 minute semi-structured interview. Afterwards, the session videos were transcribed and reviewed by the first author.

User study materials are available on OSF[4].

Tasks: For the tasks, we followed a "Use-Modify-Create" structure (Lytle et al. 2019). We provided example games *Sokoban* (a block-pushing puzzle game) and *Connect 4* (a vertical *Tic-Tac-Toe* variant), and most tasks required modifying an incomplete implementation of *Sokoban* with added spiders that move up and down using Tagging (*Soko-Enemy*, see Figure 4). If participants gave up on a task, they were allowed to continue to the next task.

1. Before doing any tasks, we had participants download a one page guide to the editor along with a video tutorial on implementing *Sokoban*. They were given as much time as they wanted to peruse the introductory materials.
2. Use - We had participants open the *Sokoban* example

[4]https://osf.io/btqcj/

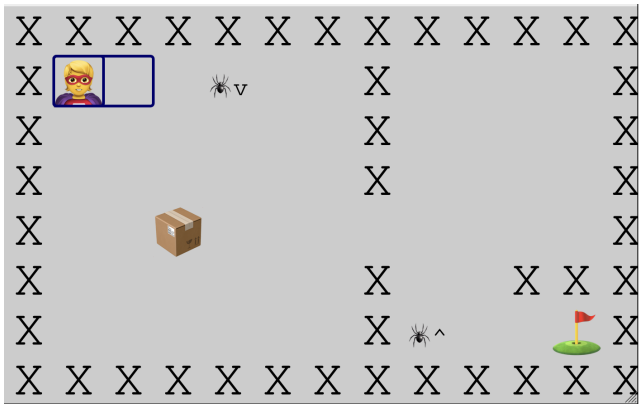

Figure 4: *Soko-Enemy*, the game used for most tasks in the user study. Available rewrites are highlighted using blue outlines. At the beginning of the study, the player avatar can only move left and right.

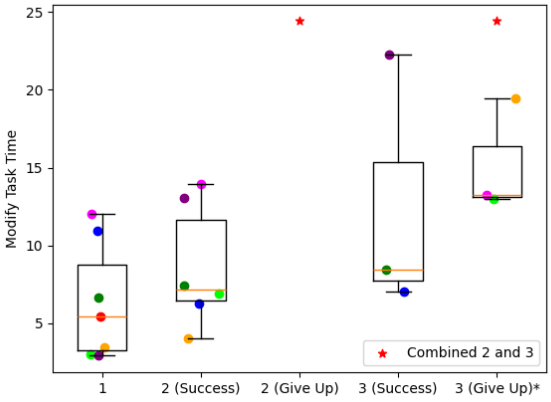

Figure 5: Task Time for Modify tasks. Each colour represents an individual participant.

game and observe it, hypothesizing about what the different tree nodes might do.

3. Modify (1) - In the incomplete *Soko-Enemy*, the player avatar can only move horizontally via *x-mirror*, and participants were tasked with enabling the player to move in 4 directions. This could be accomplished most directly by replacing the *x-mirror* with an *x-spin*, and a demonstration of this was available in both the example *Sokoban* and in the video tutorial. This was intended to test if participants could apply a basic rewrite transform.
4. Modify (2) - In *Soko-Enemy*, there are two enemy spiders moving up and down, but initially they stop when they reach a wall. The participants were asked to make the spiders turn around when they reached a wall. This could be accomplished with two *rewrite* nodes for turning the spider around when it reaches a wall ($\boxed{\begin{matrix}\text{Sv}\\ \text{X}\end{matrix}} \Rightarrow \boxed{\begin{matrix}\text{S}\hat{}\\ \text{X}\end{matrix}}$; $\boxed{\begin{matrix}\text{X}\\ \text{S}\hat{}\end{matrix}} \Rightarrow \boxed{\begin{matrix}\text{X}\\ \text{Sv}\end{matrix}}$). This was intended to test whether they could add a new rewrite rule.
5. Modify (3) - Finally, participants were asked to create another enemy moving left and right. This was intended to test that participants could transfer the principles of the vertical spiders to a horizontal enemy. In particular, we wanted to see if they could apply Tagging to prevent the spiders "teleporting" multiple steps at once or getting caught in an infinite loop.
6. Create (1) - We gave participants the option of 3 tasks: (i) add a new functionality to *Soko-Enemy*, (ii) create a platformer-style game, or (iii) create *Tic-Tac-Toe*.
7. Create (2) - If they finished all of the previous tasks, participants were invited to do whatever they wanted with their remaining time.

Survey: We surveyed participants on the following questions:

1. What is your level of computer programming experience?
   (a) No prior programming experience
   (b) I've tried programming but found it confusing
   (c) I have created some small programs or taken a programming course
   (d) I program regularly as a hobby or for work
2. What is your level of game development experience?
   (a) No prior experience
   (b) I've tried creating a game but found it confusing
   (c) I have created a couple small games
   (d) I create games regularly as a hobby or for work

We also administered the System Usability Scale (SUS) (Brooke et al. 1996). The SUS is a popular scale for assessing perceived usability that has been shown to be reliable (Bangor, Kortum, and Miller 2008). The SUS produces a single number between 0 and 100.

Interview: We conducted a semi-structured interview using the following guide:

1. What are your general impressions after using the game editor?
2. What kind of games do you think you personally could make or prototype with the editor?
3. What similar games could you not make?
4. What kinds of games do you think anyone could make or prototype with the editor?
5. What kinds of games do you think no one could make with the editor?
6. What made you feel frustrated using the editor?
7. What did you like about the editor?
8. [Ask any questions that occurred to you while observing the participant using the editor]

6.2 Results

Observations: Generally, participants demonstrated that they understood the basic concepts of TRRBT. All but one user

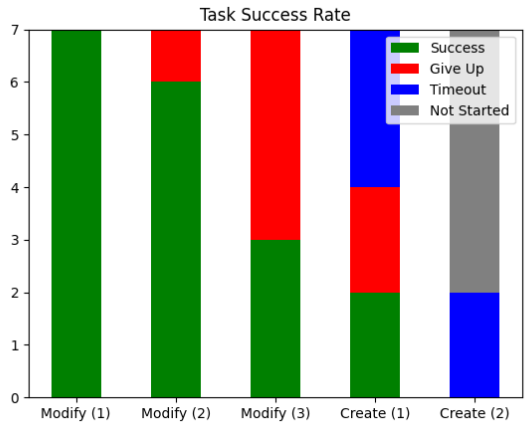

Figure 6: Task Success

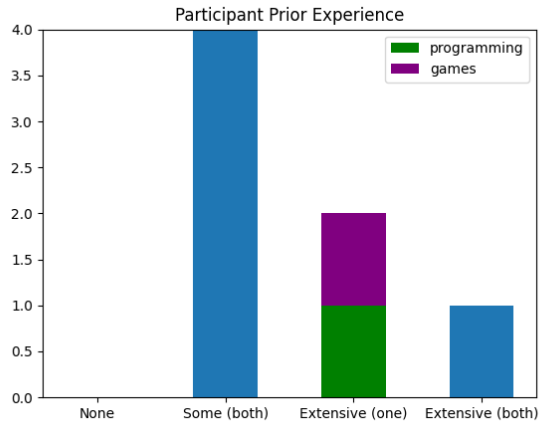

Figure 7: Prior Experience

managed to complete the first two "Modify" tasks, and most (4/7) made at least some progress on the "Create" tasks. Four of the seven participants, however, gave up on the third "Modify" task (Figure 6). Time to successfully complete tasks varied widely (Figure 5). We observed that even with an example of Tagging in the game they were editing, participants did not realize that *loop-until-all* would cause *rewrites* to continuously apply when a LHS match was found; they expected each spider to move only once each turn. When participants noticed the Tagging with ', they frequently expressed confusion: "Do I need this little mark thing? No, there's no way" [P392]. It was unclear whether the three participants who successfully completed the task truly understood Tagging or were simply copying the provided behaviour successfully.

Participants also seemed to be treating emojis on the gameboard as *objects* with their own identities, rather than as *tiles* that have no meaning outside of their rewrite patterns. Part of the confusion over movement Tagging seemed to stem from participants believing that *loop-until-all* would apply to the spiders as individuals. The clearest examples of object-based thinking are from participants who treated the player avatar as a special entity. Two participants assumed at first that any rewrites featuring the player avatar would become player choices, even outside of the *player* node. Another believed that any rewrites including the player avatar were required to be player choices, placing a gravity *rewrite* in the player subtree.

Interviews: In their interviews, participants were mostly positive about the tool. Even those who struggled said "in terms of the concepts, the concepts are easy to get" [P843], and two participants suggested it would be a good educational tool. The most frequent negative comment we received was that participants wanted Multi-Transformation "[I wish I could] like rotate direction arrows, but also have the same [spider] emoji" [P392]. We also received a number of comments about the prototype being "a work in progress" [P606] lacking polish. participants uniformly agreed that the prototype would be

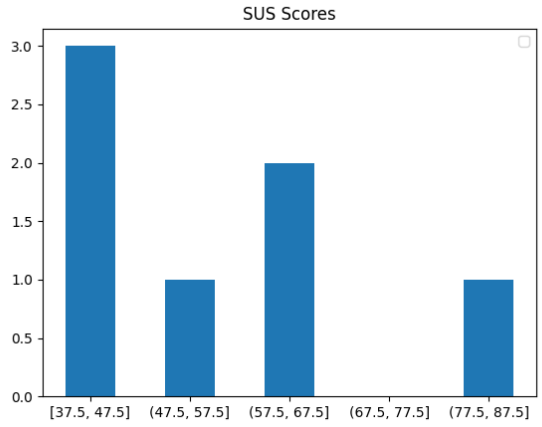

Figure 8: SUS Scores

a good tool for 2D puzzle games, and they readily suggested more experimental things that could be made, even if they might be tedious to execute (e.g. a pixel art tool, 3D chess, or frame-by-frame animation).

Post-Task Survey Responses: All the participants had previous experience both programming and creating games with other authoring tools, and three had extensive experience with one or both (see Figure 7). All participants had previously used either Unity or Godot before, and only one mentioned experience with tiny online game engines, most relevantly PuzzleScript.

A histogram of computed SUS scores is shown in Figure 8. Scores varied between 38 and 80, and most were below the SUS mean of around 70 (Bangor, Kortum, and Miller 2008).

7 Discussion

An important factor in the user study results is usability of the prototype. Despite everyone being positive about the system in their interviews, scores were generally low, with 3 of 7

SUS scores below 50. Bangor, Kortum, and Miller (2008)'s analysis suggests that a system with an average score around 50 should be considered only marginally acceptable. It is probable that some of the participants' problems would have been mitigated by using a more mature tool. However, it is difficult to say whether the main contributor to low SUS scores was tool usability or participants' difficulty establishing a mental model of the TRRBT language.

We noticed that participants frequently appeared to treat pattern tiles as game objects despite TRRBT not being an object-oriented system. This difficulty in paradigm shift might be shared with engines using entity component systems (ECS) (Härkönen 2019; Martin 2007), a programming paradigm used frequently in multiplayer online games. In ECS, rather than objects carrying their data and behaviour with them (as in object-oriented programming), "entities" are simple collections of data "components" whose behaviour is handled by different "systems". In TRRBT, we might make the loose analogy that *grid locations* are like entities, *tiles* and *layers* are like components, and *subtrees* are like systems. Although not a perfect analogy, we wonder if participants' tendencies to treat tiles and *rewrites* as intertwined reflects a broader difficulty in escaping an object-based mindset.

Overall, our findings indicate that although TRRBT is highly expressive (as indicated by our own case studies), its pattern-based model is not readily adoptable by authors used to object-based systems (as indicated by the user study). Participants' observed difficulty with more advanced tasks and low SUS ratings indicate there is more work to be done to make TRRBT accessible. More broadly, we suggest that game language designers consider the ways their language constructs might work against users' pre-existing mental models and build tooling to address discrepancies.

8 Future Work

What we discovered both in our case studies and in the user study will inform further language design. We are considering making Tagging an official language feature after both using it frequently ourselves and seeing participants assume `'` was a semantic mark. Alternatively, rewrites could have the option to apply to all matches at once rather than just one.

Similarly, we have weighed giving semantic meaning to arrows (e.g. `←`, `↑`, `→`, `↓`), to allow them to be rotated along with their patterns. However, we want to be mindful of adding special characters, as it would reduce the set of characters that game authors can use, and we want to keep TRRBT as generic as possible.

We may introduce new or changed nodes to make code easier to organize and less repetitive. In particular, participants' frustration with creating copies for every direction reinforces our intuition that TRRBT would be well-served by better support for Multi-Transformation. From our experiences developing complex games, we are also considering ways to better support Multi-Sub-typing.

We are also considering ways to change our user study design. Typically game authors spend at least several hours if not several days or weeks creating a game, whereas our participants only had an hour to familiarize themselves with TRRBT. When the interface is more polished, we would like to run a longer term study to better assess the diversity of games encouraged by the system. We also want to better evaluate users' understanding of the TRRBT concepts. We are interested in how users fare if they are not given an example of Tagging or Movement Arrows; would they solve those problems in the same way without our examples? To retrain users to think more in terms of patterns, we wonder if the introduction to TRRBT should be something without such an obvious game entity-to-emoji mapping. For example, we could have users implement a cellular automata or a game with many interchangeable tiles, like *Tic-Tac-Toe*. Wright (2006) designed a series of basic tasks intended to promote "pattern-based" rather than "agent-based" thinking; we could adopt a similar progressive tutorial system as an introduction to the system. It would also be useful to do a direct comparison with PuzzleScript, which shares the language of tile rewrite rules without the control structure of behaviour trees.

9 Research Limitations

This work has some limitations.

The demonstration games presented are limited in that they were created by the authors, who designed and are familiar with the system and its features. Games made by more individuals might highlight other shortcomings or demonstrate more variety in design patterns.

The user study suffers primarily in size; only seven participants interacted with the tool for only one hour. In addition, it is not entirely clear whether participants' struggles with Tagging and object-based thinking were caused by the inherent difficulty of those concepts or if the tasks chosen were an insufficient introduction to TRRBT and pattern-based thinking. *Soko-Enemy* is a game that includes distinct game objects represented by emojis, which could encourage users to think in terms of objects rather than patterns. It also uses `'` for Tagging, a small character that participants were slow to even notice and which was easily mistaken for a special character.

10 Conclusion

We have investigated the expressiveness and ease of adoption of TRRBT, a minimal game design language introduced by our previous work (Zhou et al. 2024). We have developed and described a lightweight, browser-based authoring tool for such games. We presented case studies demonstrating the capability of the language to describe a variety of 2D tile- and turn-based games, and we have found a number of useful design patterns to aid in code reuse and organization. We also showed through a user study that, although the basic concepts are readily adoptable, users are unused to pattern-based (as opposed to object-based) thinking. We recommend that designers of new programming models consider users' existing mental models and how they can be retrained.

Acknowledgements

We would like to thank the anonymous reviewers. This work was supported by a Northeastern University Khoury College Research Seed Grant.

References

Alcorn, A. 1972. Pong. Game [Arcade].

Bacher, J. T.; and Martens, C. 2021. Interactive Fiction Creation in Villanelle: Understanding and Supporting the Author Experience. In *2021 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 1–5. IEEE.

Bangor, A.; Kortum, P. T.; and Miller, J. T. 2008. An empirical evaluation of the system usability scale. *Intl. Journal of Human–Computer Interaction*, 24(6): 574–594.

Becroft, D.; Bassett, J.; Mejía, A.; Rich, C.; and Sidner, C. 2011. Aipaint: A sketch-based behavior tree authoring tool. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 7, 2–7.

Berlekamp, E. R.; Conway, J. H.; and Guy, R. K. 2004. *Winning ways for your mathematical plays, volume 4*. AK Peters/CRC Press.

Brooke, J.; et al. 1996. SUS-A quick and dirty usability scale. *Usability evaluation in industry*, 189(194): 4–7.

Compton, K.; and Mateas, M. 2015. Casual Creators. In *ICCC*, 228–235.

Cooper, S. 2023. Sturgeon-MKIII: Simultaneous Level and Example Playthrough Generation via Constraint Satisfaction with Tile Rewrite Rules. In *Proceedings of the 18th International Conference on the Foundations of Digital Games*, 1–9.

Cooper, S.; and Bazzaz, M. 2024. Sturgeon-MKIV: constraint-based level and playthrough generation with graph label rewrite rules. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 20, 13–24.

Dormans, J.; and Bakkes, S. 2011. Generating Missions and Spaces for Adaptable Play Experiences. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3): 216–228.

Härkönen, T. 2019. Advantages and Implementation of Entity-Component-Systems. *Tampere University, Tampere, Finland*, 6.

Imabayashi, H. 1982. Sokoban. Game [NEC PC-8801].

Iovino, M.; Scukins, E.; Styrud, J.; Ögren, P.; and Smith, C. 2022. A survey of behavior trees in robotics and ai. *Robotics and Autonomous Systems*, 154: 104096.

Johnson, C. 1990. Minesweeper. Game [Microsoft Windows].

Kapadia, M.; Zünd, F.; Falk, J.; Marti, M.; Sumner, R. W.; and Gross, M. 2015. Evaluating the authoring complexity of interactive narratives with interactive behaviour trees. In *Proceedings of the 10th International Conference on the Foundations of Digital Games (FDG 2015)*.

KissMaj7. 2013. Lime Rick. Game [https://www.kongregate.com/games/KissMaj7/lime-rick]. Accessed: 2024-07-03.

Lytle, N.; Cateté, V.; Boulden, D.; Dong, Y.; Houchins, J.; Milliken, A.; Isvik, A.; Bounajim, D.; Wiebe, E.; and Barnes, T. 2019. Use, modify, create: Comparing computational thinking lesson progressions for stem classes. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, 395–401.

Martens, C. 2015. Ceptre: A language for modeling generative interactive systems. In *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, volume 11, 51–57.

Martens, C.; Card, A.; Crain, H.; and Khatri, A. 2023. Modeling game mechanics with Ceptre. *IEEE Transactions on Games*, 16(2): 431–444.

Martens, C.; and Iqbal, O. 2019. Villanelle: An authoring tool for autonomous characters in interactive fiction. In *Interactive Storytelling: 12th International Conference on Interactive Digital Storytelling, ICIDS 2019, Little Cottonwood Canyon, UT, USA, November 19–22, 2019, Proceedings 12*, 290–303. Springer.

Martin, A. 2007. Entity systems are the future of MMOG development – part 1. https://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1.

Mateas, M.; and Stern, A. 2002. A behavior language for story-based believable agents. *IEEE Intelligent Systems*, 17(4): 39–47.

Mérõ, L. 1983. XL-25. Game [Handheld Electronics].

Nelson, G. 2022. *Cloak of Darkness*. Game [Inform 7]. Accessed: 2024-08-30.

Nitrome. 2014. Turn Undead. Game [Flash].

Nitrome. 2015. Rust Bucket. Game [iOS/Android].

Normoyle, A.; Jörg, S.; and Hill, J. 2024. The curation tree: A lightweight behavior tree framework for implementing puzzle and narrative games. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*, 1–4.

Remy, C.; MacDonald Vermeulen, L.; Frich, J.; Biskjaer, M. M.; and Dalsgaard, P. 2020. Evaluating Creativity Support Tools in HCI Research. In *Proceedings of the 2020 ACM Designing Interactive Systems Conference*, DIS '20, 457–476. New York, NY, USA: Association for Computing Machinery. ISBN 978-1-4503-6974-9.

Rendell, P. 2011. A Universal Turing Machine in Conway's Game of Life. In *2011 International Conference on High Performance Computing & Simulation*, 764–772. IEEE.

Repenning, A. 1995. Bending the rules: steps toward semantically enriched graphical rewrite rules. In *Proceedings of Symposium on Visual Languages*, 226–233. IEEE.

Sagredo-Olivenza, I.; Gómez-Martín, M. A.; and González-Calero, P. A. 2015. Supporting the collaboration between programmers and designers building game AI. In *Entertainment Computing-ICEC 2015: 14th International Conference, ICEC 2015, Trondheim, Norway, September 29-Ocotober 2, 2015, Proceedings 14*, 496–501. Springer.

Sekhavat, Y. A. 2017. Behavior trees for computer games. *International Journal on Artificial Intelligence Tools*, 26(02): 1730001.

Shoulson, A.; Garcia, F. M.; Jones, M.; Mead, R.; and Badler, N. I. 2011. Parameterizing behavior trees. In *Motion*

in Games: 4th International Conference, MIG 2011, Edinburgh, UK, November 13-15, 2011. Proceedings 4, 144–155. Springer.

Slaney, J.; and Thiébaux, S. 2001. Blocks world revisited. *Artificial Intelligence*, 125(1-2): 119–153.

Solutions, D. 2007. PuzzPack v2.0. Game [TI-83+].

Van Rozen, R. 2020. Languages of games and play: A systematic mapping study. *ACM Computing Surveys (CSUR)*, 53(6): 1–37.

Warren, J. 2019. Tiny online game engines. In *2019 IEEE Games, Entertainment, Media Conference (GEM)*, 1–7. IEEE.

Wright, T. 2006. PatternProgrammer: yet another rule-based programming environment for children. In *Proceedings of the 7th Australasian User interface conference-Volume 50*, 91–96. Citeseer.

Zhou, J.; Martens, C.; and Cooper, S. 2024. Authoring Games with Tile Rewrite Rule Behavior Trees. In *Proceedings of the 19th International Conference on the Foundations of Digital Games*, 1–4.