ScriptButler serves an Empirical Study of PuzzleScript

Analyzing the Expressive Power of a Game DSL through Source Code Analysis

Clement Julia
clement.julia13@gmail.com
University of Amsterdam
Amsterdam, The Netherlands

Riemer van Rozen
rozen@cwi.nl
Centrum Wiskunde & Informatica
Amsterdam, The Netherlands

ABSTRACT

Automated Game Design (AGD) empowers game designers with languages and tools that automate game design processes. Domain-Specific Languages (DSLs) promise to deliver an expressive means for rapidly prototyping and fine-tuning interaction mechanisms that support rich emergent player experiences. However, despite the growing number of studies that center around languages for games and play, few prototypes are ever thoroughly validated and evaluated in practice. As a result, it is not yet well understood what the costs, benefits and limitations of DSL formalisms are.

To find out, we investigate to what extent rules, affordances and play can be related by means of source code analysis. We study PuzzleScript, a language and online game engine with an active user community. We reverse engineer PuzzleScript's design and propose ScriptButler, a novel tool prototype and engine for its analysis. To validate our approach, we conduct an empirical study on the quality of the source code by performing an analysis on a curated collection of 95 games. Our results show that ScriptButler can identify bugs and helps relate PuzzleScript rules to game qualities.

CCS CONCEPTS

• **Software and its engineering → Domain specific languages**; *Integrated and visual development environments*; *Software verification and validation*; • **Applied computing → Computer games**.

KEYWORDS

automated game design, domain-specific languages, PuzzleScript, source code analysis, game design tools, reverse engineering

ACM Reference Format:
Clement Julia and Riemer van Rozen. 2023. ScriptButler serves an Empirical Study of PuzzleScript: Analyzing the Expressive Power of a Game DSL through Source Code Analysis. In *Foundations of Digital Games 2023 (FDG 2023), April 12–14, 2023, Lisbon, Portugal.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3582437.3582467

1 INTRODUCTION

Digital Games are a powerful means for creating rich interactive player experiences, e.g., for entertainment, health and learning. Automated Game Design (AGD) aims to automate game design processes by providing designers with tools they need for explorative design, creative tinkering, and iterative improvements. Domain-Specific Languages (DSLs) are a particular means to give such tools expressive power, e.g., for improving predictive accuracy and supporting mixed-initiative, co-creative and generative approaches.

DSLs have proven instrumental in raising expert productivity and improving the code quality in areas such as digital forensics, robotic engineering, and banking. Costs include learning to use the DSL and maintaining its digital infrastructure. However, how the trade off between costs and benefits applies to AGD is not yet well understood due to a lack of empirical studies on real world examples [26]. As a result, the merits of DSLs for AGD are still largely unknown. Therefore, we study which game facets are amenable to DSL development, which features can express game designs, and what the benefits and limitations of DSL formalisms are.

To obtain empirical evidence, we investigate to what extent rules, affordances and play can be related by means of source code analysis. In particular, we study PuzzleScript, an established DSL with an active user community. Developers have created a wide variety of puzzle games whose sources are available online. This presents a research opportunity. We pose two research questions:

(1) What can be observed about the quality of PuzzleScript source code in terms of a) volume in source lines of code (SLOC); of b) objects, collision-layers, win-conditions and levels; and c) rules, interactivity, affordances and play?
(2) What kind of games can PuzzleScript express, and which game elements, language features, and usage patterns do these games have in common?

To answer these questions, we conduct mixed-method research. First, we reverse engineer PuzzleScript in Section 2. We recover its design by reading manuals, studying its sources, and dissecting examples. Using the insights we obtain, in Section 3, we then design a novel tool prototype called ScriptButler, a multi-stage compiler, analyzer, engine that 1) facilitates studying PuzzleScript source code; and 2) supports designing games with immediate feedback in its IDE. Finally, we validate our approach and answer our questions by conducting an empirical study on the quality of a curated collection of 95 PuzzleScript games in Section 4. This paper contributes:

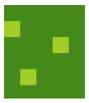
- An analysis of the PuzzleScript language and its engine.
- ScriptButler, a novel tool for PuzzleScript analysis.
- An empirical study on the software quality of 95 games.

Our results indicate that PuzzleScript is a powerful DSL, and that ScriptButler serves its purpose as a code analysis platform well. By uncovering the sources of PuzzleScript's expressive power in particular, we also gather evidence on the merits of DSLs for AGD in general. Next, we introduce PuzzleScript and discuss its design.

FDG 2023, April 12–14, 2023, Lisbon, Portugal
© 2023 Copyright held by the owner/author(s).
ACM ISBN 978-1-4503-9855-8/23/04.
https://doi.org/10.1145/3582437.3582467

```
Background
lightgreen green
11111
01111
11101
11111
10111
```

(a) Object definition



(b) Rendered object

Figure 1: Defining a grassy background object

2 PUZZLESCRIPT

PuzzleScript is an online textual language and game engine for creating puzzle games designed by Stephen Lavelle [17]. PuzzleScript has an active community of developers and enthusiasts who have constructed many different classical and original games using a surprisingly little amount of code. We study its source code in order to learn what gives it such expressive power. However, a suitable means for automating an empirical study is currently unavailable.

Therefore, we investigate the technical requirements to facilitate its source code analysis and transformation. We apply the research method of *reverse engineering* to recover its design. We perform the following activities. First, we study PuzzleScript's documentation [18], and read articles and books describing the language [2]. Section 2.1 summarizes our findings and introduces the language.

Next, we analyze PuzzleScript's codebase for recovering design intentions from the sources. We dissect the language by analyzing its compiler and features. We subject the game engine, the system under study, to various inputs to observe its behavior. In particular, we run the games bundled in its distribution to observe the effects of language features in isolation. Section 2.2 describes our findings about the engine's design. Finally, Section 2.3 discusses the technical requirements for a novel prototype that enables an empirical study.

2.1 Introduction to the PuzzleScript Language

PuzzleScript games are tile maps populated by objects, named sprites of five by five pixels that can move and collide. The game logic is defined by a set of rewrite rules. Game definitions consist of a sequence of sections whose meaning we introduce one by one. Just like the online game engine, we introduce the notation using the Simple Block Pushing Game created by David Skinner.

2.1.1 Prelude. Programs begin with a metadata section called prelude. Authors specify the game's title, author name and homepage.

```
title Simple Block Pushing Game
author David Skinner
homepage www.puzzlescript.net
```

The manual describes in detail how other tags adjust the user interaction, various timings and visual parameters [18]. We do not discuss these tags here. Instead, we introduce them when the need arises. Sections other than prelude each begin with their respective keyword, optionally surrounded by separator bars (hyphens).

2.1.2 Objects. The **OBJECTS** section defines a series of game assets called objects, sprites of 5x5 pixels that can move and collide.

Figure 1 shows an example of a background object. Its definition, shown in Figure 1a, begins with its name, followed by colors associated with numbers. The sprites directly below have a 5x5 literal notation that references these colors. When the engine renders the object, this results in the visual sprite displayed in Figure 1b.

Games typically have objects defining background, players, obstacles, enemies and exits. Objects appear in levels, can be combined to form larger structures, and are manipulated using rules.

2.1.3 Legend. In the **LEGEND** section, programmers define how symbols used in level descriptions can refer to objects to create levels.

```
# = Wall
P = Player
* = Crate
@ = Crate and Target (example aggregate symbol)
O = Target
```

One symbol can represent multiple objects called aggregates, e.g., the @ symbol in the preceding example. However, aggregates are only valid if those objects appear in different layers, i.e. objects that do not collide. An alternative (less clean) notation is writing the symbol directly behind the object name in the objects section.

In the same snippet, we also introduce comments. Comments appear on one or more lines between left and right parentheses.

2.1.4 Sounds. The engine defines built-in sounds that can be associated with rules and actions. Because sounds support experiences but do not directly influence rules, we do not discuss them further.

2.1.5 Collision Layers. Each game consists of surfaces representing a z-axis on the map, e.g., for background and foreground objects. The **COLLISIONLAYERS** section describes how objects appear on maps in a sequence of vertically separated collision layers. Objects in the same layer can collide with each other.

```
Background            (background layer)
Target                (support layer)
Player, Wall, Crate   (foreground layer)
```

In the example, we see three layers. The background, support and foreground layers each appear on one line. The last line determines that player, wall and crate objects can collide with eachother.

2.1.6 Rules. In the **RULES** section specifies game mechanics and run-time behaviors as a sequence of rewrite rules. Players can interact with these rules using the arrow keys and the action key `x`. The following rule describes a crate pushing mechanism.

```
[ > Player | Crate ] -> [ > Player | > Crate ]
```

The rule's left hand side is a pattern describing the condition that must hold before applying the rule. We can read: *"if the player moves in the direction of a crate"*. Of course, a collision would normally prevent this movement. However, the rule's right hand side describes a different result of the collision. We can read: *"then the player and the crate both move directionally"*.

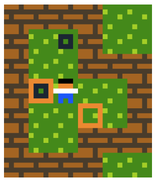
The *omnidirectional* > operator is short-hand for applying the rule in every direction. The single rule from the example therefore translates into a *group* of four *directional* rules that work in an **up**, **down**, **left** and **right** manner. Internally, the system processes:

```
  down[Crate|up Player] -> [up Crate|up Player]
+ down[down Player|Crate] -> [down Player|down Crate]
+ right[Crate|left Player] -> [left Crate|left Player]
+ right[right Player|Crate] -> [right Player|right Crate]
```

Rules are processed in groups specified using the + operator. Matching can be restricted by prefixing the rule a direction keyword.

```
####..
#.O#..
#..###
#@P..#
#..*.#
#..###
####..
```

(a) Level definition



(b) Rendered level

Figure 2: Level of the Simple Block Pushing Game

These rules trigger until the system cannot match and apply them any more, i.e. as a fixpoint computation.

Other short-hand operators include **`horizontal`** for rules that work in the directions **`left`** and **`right`**, and **`vertical`** for both **`up`** and **`down`**. In our analysis of game rules, in Section 4.3, we will discuss additional mechanisms and more advanced patterns.

2.1.7 Win conditions. The **`WINCONDITIONS`** section describes a set of conditions that work on sets of objects. They must each be true for a player to win. In the example, every crate must be on a target.

```
all Target on Crate
```

Perhaps somewhat counter-intuitive, this condition specifies that given the set of all targets, crates must occupy the same coordinates. After all, crates and targets are in different collision layers.

Aside from intersections of sets, conditions can also test for the empty set with the **`no`** keyword or a non-empty set using the **`some`** keyword. Midas poetically describes its win condition as follows.

```
some Love (Awwww!)
```

When a condition cannot become true anymore, the player gets stuck, which is the PuzzleScript way of losing a game. Normally, players can undo moves, unless the **`noundo`** tag prevents it.

2.1.8 Levels. The **`LEVELS`** section describes a sequence of game levels and messages for mixing information with puzzle progressions. Like objects, levels have a literal notation describing a tile map composed from legend symbols. Figure 2 shows an example. Each level has a regular width and height. Rows each have the same length. However, successive game levels need not be the same size.

By default, PuzzleScript games show an entire level. The tags **`zoomscreen`** `WxH` and **`flickscreen`** `WxH` can be used to show smaller areas of the map. Moving the player-controlled object between areas shifts this view. This concludes our limited introduction. Next, we discuss the design and implementation of its engine.

2.2 Overview of the PuzzleScript Engine

Here we report a partially recovered design. PuzzleScript's game engine is based on HTML5/css and JavaScript. The sources, released under the MIT license, are available on GitHub [18]. The repository contains 27 JavaScript files whose volume in source lines of code (SLOC) amounts to 15 KLOC[1]. The SLOC metric includes lines with code and curlies, but not lines that are empty or comments only.

The core of the engine consists of the three files shown in Table 1. The other sources provide features not directly tied to PuzzleScript's design such playing sounds, and saving and exporting games. Next, we discuss the parsing, compilation and run time phases.

[1]We use cloc (http://cloc.sourceforge.net) to obtain these numbers.

Table 1: Main files of PuzzleScript's engine

file	LOC	description
parser.js	1065	responsible for the parsing phase.
compiler.js	2340	responsible for the compilation phase
engine.js	2405	responsible for the run time phase.

2.2.1 Parsing. Developers create games using a browser-based IDE. This IDE uses CodeMirror, a versatile library specifically made for editing code in the browser. PuzzleScript has a line-based parser that processes the input line by line. The parser is a state machine composed of a collection of functions that produce an Abstract Syntax Tree (AST). In addition, this machine also performs contextual analysis. The IDE's syntax coloring adapts to the source code.

2.2.2 Compilation. Once a user presses "run" the game compiles and potential errors appear in the bottom right. The title screen for the game appears in the top right. The compiler prepares internal data structures the engine processes during the run-time phase. During the compilation, the compiler shows error messages as they are detected, and it aborts when a certain threshold is reached. This threshold is necessary to prevent an error cascade, a high number of dependent or "ghost" errors. By setting a low threshold, developers can address errors one by one. Error messages have a line number but omit symbol, column number and range.

2.2.3 Run time. The running game is displayed in the IDE's central area on the right. During play, the engine interprets the compiled rules as described in Section 2.1. The top bar of the IDE provides additional options for sharing and exporting the game as a standalone application and helpful links for seeking support.

2.2.4 Debugging. The **`verbose_logging`** tag activates logging in the output area of the IDE. When playing the game, this area displays the sequence of button presses and rule activations. Hovering over the events in the sequence shows the associated game state and the rule's effects by showing directional arrows. Clicking on an activated rule moves the cursor to the associated source line.

2.2.5 Level editor. Levels can be edited textually or visually using the online engine's interactive level editor. Programmers can generate animated GIFs from the editor by pressing CTRL+K.

2.3 Requirements Analysis

Here we discuss to what extent the design fulfils our research needs. The source code analysis of PuzzleScript requires treating the sources as data. Unfortunately, the JavaScript implementation is not very suitable for this. Language workbenches provide the specialized features we need, e.g., for expressing analyses, matching data structures and generating reports [10]. To study PuzzleScript, we need a research prototype that leverages this technology.

An empirical study of PuzzleScript entails analyzing and comparing syntax trees of many programs. Therefore, we require *bulk parsing*. However, the JavaScript parser provides at most one tree at a time. To study PuzzleScript at scale, we need a *grammar* that parses all PuzzleScript. Given a grammar in extended Backus-Naur

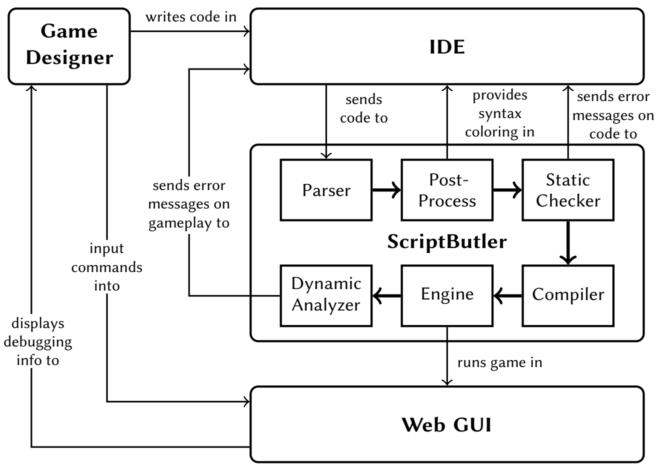


Figure 3: ScriptButler framework, components, data flow

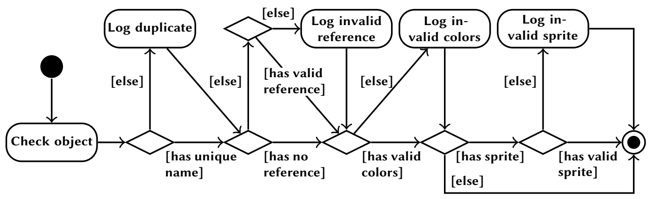


Figure 4: Activity diagram for checking objects

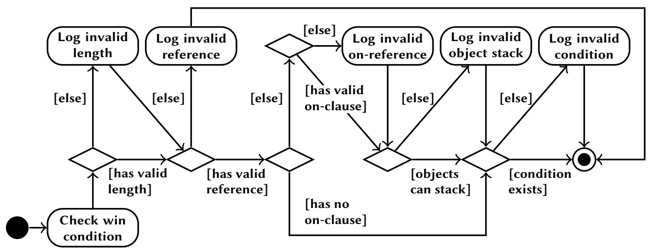


Figure 5: Activity diagram for checking win conditions

```
Player
black orange white blue
.000.
.111.
2222222222
.333.
.3.3.
```

(a) Sprite not 5x5

```
Crate
orange green
00000
0...0
0...0
0...0
00000
```

(b) Unused color

```
####
#.O#..
#..###..
#@P..#
#..*
#..###
####..
```

(c) Uneven level

```
[Eyeball| ... |Player] -> [> Eyeball | Player]
```

(d) Missing ellipsis in right hand side of rule

```
[> Player|Crate] -> [> Player] [> Crate]
```

(e) Unexpected rule part in right hand side of rule

Figure 6: Errors and warnings detected by the checker

(EBNF) form, we can obtain a more concise and maintainable language specification. In addition, an EBNF grammar would greatly simplify the analysis of its concrete and abstract syntax.

Analyzing the source code quality requires an exhaustive analysis of errors and warnings. Expressing the design as a multi-phase compiler helps prevent error cascades that pollute the data. Enabling root-cause analysis requires that each error messages has an exact *source location* that includes column numbers and range.

Next, we introduce ScriptBulter, our solution to these challenges.

3 SCRIPTBUTLER

Here we describe ScriptButler, a novel multi-purpose tool capable of parsing, validating, and running PuzzleScript games. Our goals are two-fold. The first, is to give game designers a better understanding of how changes to the code affect the game quality by providing immediate feedback in the IDE. The second is to create an extensible research platform for conducting empirical research by automating source code analyses. We use the Rascal language workbench to create ScriptButler [13]. Rascal helps generate an IDE at a low cost[2].

Figure 3 illustrates the framework. In the following sections, we explain how the multi-phase compiler and its components work. For a more detailed description, we refer to Julia [12].

3.1 Parser and post processing

For bulk parsing, we create a grammar that can parse every PuzzleScript program. At a mere 106 SLOC, our grammar is significantly smaller the original, which counts 1065 SLOC of hand-written JavaScript. However, perhaps surprisingly, the grammar itself is not necessarily easier to extend and maintain. Eliminating ambiguities in the grammar has been extremely challenging because PuzzleScript evolved as a line-based language. However, our grammar now passes every test. It is a central part of ScriptButler, and can also be reused for creating other tools for PuzzleScript.

The post processing phase simplifies the syntax trees for analysis in successive phases. By analyzing colors of sprites and modifying the concrete syntax trees, it also offers syntax coloring in the IDE.

[2]https://www.rascal-mpl.org

3.2 Static checker

The static checker performs a contextual analysis on simplified syntax trees and generates human readable error messages. Compared to the JavaScript implementation, which has 117 unique error messages, this checker maintains just 72 message types.

There are two reasons for this. First, parse errors need not be covered by the checker. Second, the design of the checker is more concise, requiring fewer messages for the same purpose.

Messages have three levels of importance. First, error-level messages indicate failures or unintended side effects that make running the game impossible. Warnings indicate dead or unoptimized code. Information-level messages involve advise on gameplay quality and best practices. Message categories are: 1) Invalid: a component is not well-formed, making it unusable; 2) Undefined: a reference/object with that name is never defined but is used; 3) Existing: a reference/object with that name already exists, but the code is trying to define it again; 3) Unused: a warning, the code defines a reference/object/sound but never uses it; 4) Misc: few other errors.

Figures 4 and 5 show UML Activity Diagrams that illustrate the data flow in the analysis of objects and win conditions. Figure 6 shows selected examples of how the IDE shows error and warning messages. Compared to the original implementation, we add checks for mutually exclusive win conditions, impossible layering, duplicate rules, and unused legend, objects and colors.

3.3 Dynamic analysis and play

The focus of this paper is on static analysis. Therefore we only briefly describe the components for dynamic analysis and play.

3.3.1 Compiler. The compiler generates low-level data structures for a game's rules, levels and objects. We ensure the source of errors can always be identified by tracking the origins of each structure. By propagating the textual source locations onto these structures, we can enable debugging and generate appropriate run-time errors.

3.3.2 Engine. The engine processes rule activations and transforms the objects appearing in the level's layers.

3.3.3 Dynamic analyzer. The dynamic analysis checks for: 1) Instant victory, a win condition is fulfilled from the start; 2) Impossible victory: a win condition requires objects that are not present and not created by any rules; 3) Rule similarity: compiled forms of rules are structurally similar; and 4) Unusable rule: prerequisites for a rule must be spawned by another rule. Developers can use this information to assess if play and win scenarios are appropriate.

3.3.4 Game UI. A web-based game UI is rendered using Rascal's Salix framework. Its debug view enables inspecting layers and rules.

3.4 Implementation and testing

ScriptButler is implemented in the Rascal language workbench [13]. The core consists of 9 files and 2317 SLOC, significantly less than the original. Its test suite verifies the system's main functions. The sources are available on GitHub under the 3-clause BSD license[3].

In this paper, we leverage ScriptButler in an empirical study. For further validation, we refer to a case study that applies ScriptButler to an evolution scenario of a game called Timothy's Adventure [12].

4 ANALYZING PUZZLESCRIPT

We perform an empirical study on PuzzleScript. The motivation for this analysis is twofold. First, we investigate the expressiveness of PuzzleScript by studying its source code. Second, we aim to validate ScriptButler and evaluate its use in answering research questions.

4.1 Methodology

The analysis of PuzzleScript requires a well-defined methodology that ensures an accurate and reproducible information extraction, categorization and comparison. We define the scope, formulate research questions, select sources and devise a review protocol.

4.1.1 Scope. We study PuzzleScript and automate this work with ScriptButler. In particular, we wish to learn what can be observed about the quality of existing games by analyzing their sources. Our goal is not to perform a critical analysis, or to construct a precise ontology that enables distinguishing between games and play. Here, the PuzzleScript source code itself is the subject of the study.

4.1.2 Questions. We address the following research questions:

(1) What can be observed about the quality of PuzzleScript source code in terms of a) volume in source lines of code (SLOC); of b) objects, collision-layers, win-conditions and levels; and c) rules, interactivity, affordances and play?
(2) What kind of games can PuzzleScript express, and which game elements, language features, and usage patterns do these games have in common?

4.1.3 Sources. We exclusively study the GitHub repository of PuzzleScript [18]. This source contains a curated collection of 95 high quality games that use different facets of the language and demonstrate its expressive power. Games include distinct recreations of existing video games (or demakes), feature demos, tutorials, and original creations that various authors have submitted to Stephen Lavelle. The repository contains the original source code, which makes it particularly suitable for our study.

The inclusion criterion is: we include games whose sources describe rules, have at least one level and have gameplay. The exclusion criterion is: we exclude duplicates, test cases and demos that illustrate one or more features in isolation. These games can easily be identified. Most have a single level and a lack of win conditions.

4.1.4 Review protocol. We review the sources and create a data set by subjecting each game to the following review protocol. The following quantitative analyses help answer Question 1.

Source code. For each game, we calculate its volume in terms of Source Lines of Code (SLOC), comment lines and blank lines. We parse the sources with ScriptButler, and record parse errors. Each game has a **`title`** and **`author`**. We identify distinct origins.

Objects. Players control an avatar, connect puzzle blocks, place props, or select and move board pieces. We record the number of objects, which object the player controls and what its name is.

Collision layers. Each game g has a list of collision layers $l(g)$. We record the number of layers $|l(g)|$. We introduce a collision metric that defines a game's maximum number of collisions $c(g)$, defined as the sum of the upper bound of collisions between two objects in each layer, or more formally $c(g) = \sum_{l_i \in l(g)} (\frac{|l_i| * (|l_i| - 1))}{2})$

Win-conditions. Common win-conditions include reaching an exit, collecting all objects of a certain kind. Other games let players freely explore without a predefined goal.

Levels. We record the amount of levels to estimate game content.

Contextual analysis. We run ScriptButler's checker and record how many warnings and errors it can identify.

Verbs, mechanisms, rules and affordances. We record how many rules a game has. In addition, we perform a qualitative analysis based on Koster's theory of fun [14]. Koster has proposed Verbs, a simple visual game design language for expressing player affordances and reasoning about fun, delight, flow and social facets [15].

We apply this conceptual lens without its visual diagrams. We play each game and interpret affordances. We relate verbs to rules

[3]https://github.com/vrozen/ScriptButler

Table 2: Syntax errors identified by ScriptButler

Title	Author	Parse error reason	Action
Ponies Jumping Synchronously	vytah	Extra ‘)’ on line 349.	Removed ‘)’
PUSH	lonebot - demake by rmmh	Missing ‘)’ on line 644.	Added ‘)’
Des Poseidons Dreizack	Stephen Lavelle	Unexpected ‘=’ in object and in level on lines 511 and 834.	Replaced ’=’ by ‘?’
Heroes of Sokoban	Jonah Ostroff	Missing line break after level on line 433.	Added line break.

and actions players can take to exert influence over in-game objects. For instance, players can move objects using arrow keys. They can push, pull or place objects. Actions include, e.g., shoot, teleport, explode. Gravity enables falling. A snake mechanism grows a line.

4.1.5 Categorization. We discuss what the games have in common in Section 4.3. From the above analyses we distill categories. We inspect the source code to in order to learn if games with similar affordances also have similar rules, thereby answering Question 2.

4.2 Results

4.2.1 Inclusion. Of the 95 descriptions we exclude 28 demos, one duplicate and a very large test case. The duplicate is: It Dies in the Light by Christopher Wells. Easy Enigma is a large test case (1432 SLOC) that covers many language features. We have used it to improve our grammar. We report results for 65 included games. The analysis of the source code reveals the following.

4.2.2 Volume. The volume of PuzzleScript game descriptions ranges from tiny, e.g., Notsnake by Terry Cavanagh at just 50 SLOC to very large, e.g., The Saga of the Candy Scroll by Jim Palmeri at 737 SLOC. Figure 7a shows a box plot that illustrates the distribution of volume in SLOC over the 65 included games. A game’s volume rarely exceeds 500 SLOC, indicating a concise notation.

In addition, authors use blank lines for improving the readability. Therefore, the sources contain many blank lines, approximately 5% on average. We find many commented out lines in relatively few games. Upon closer inspection, these are usually commented out levels. Explanations of the source code are much more rare.

4.2.3 Parsing. ScriptButler parses every syntactically correct specification. However, we identify four games that have syntax errors, listed in Table 2. We divide these errors into three categories.

First, incomplete comments, require matching opening and closing parentheses. Second, incorrect use of the reserved keyword ‘=’ in object or level definitions, requires replacing this symbol by another. Finally, we require a line break after each level definition. This is good programming practice but also serves as a disambiguation measure in the grammar. In comparison, PuzzleScript’s line-based parser only warns the programmers not to use reserved symbols as legend. We use the parse trees we obtain in the following analyses.

Table 3: Authors who contributed at least three games

author	ct.	contributions
increpare	12	Tutorials, demakes and original creations by Stephen Lavelle, the creator of PuzzleScript.
Jonah Ostroff	4	Heroes of Sokoban I, II and III.
lexaloffle	3	Love and pieces, neko puzzle, Zen puzzle garden.
David Skinner	3	Microban and simple block pushing games.

Table 4: Collision layers found in game descriptions

layers	ct.	analysis
2	4	Simple background/foreground division.
3	16	One support layer, e.g., crate targets.
4	15	Two support layers
5	14	Complex layering system.
6	13	Complex layering system.
> 6	7	Very complex layering system.

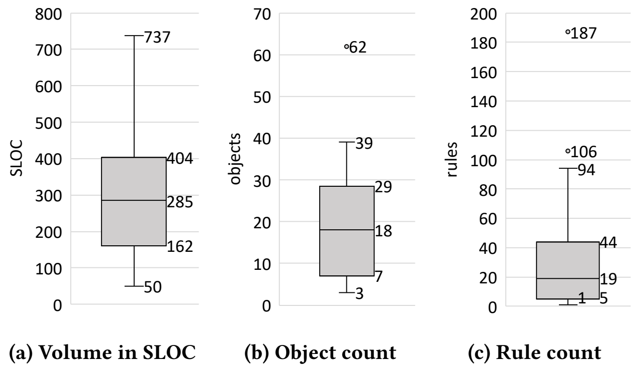


Figure 7: Box plots illustrating how volume, objects and rules are distributed over the source code of included games

4.2.4 Authors. A total number of 38 authors contribute at least one game. Table 3 shows the four most prolific authors who contribute at least three games. In addition, 8 authors contribute two games and 27 authors contributed one, indicating distinct origins.

4.2.5 Objects. Relatively simple games have up to ten objects. Games with more objects are usually fully fledged and complete experiences. Figure 7b shows a box plot that illustrates the distribution of object counts over the 65 games. Even very complex sources rarely exceed thirty objects.

4.2.6 Collision layers. Most games have between two and six collision layers, three being the most common, as shown in Table 4. Games have a background layer and a foreground layer containing player controlled objects. In addition, one or more support layers can contain items, teleport pads, pathways, destinations, etc.

The collision metric usually yields scores between one and three. Fifteen games whose scores are above ten are indeed collision-heavy. Of course, many objects cannot collide because they do not move or do not appear in levels together. As a result, our collision metric is likely an overestimate of actual collisions.

Table 5: Win-conditions found in game descriptions

cond.	ct.	analysis
0	8	Free exploration.
1	39	Simple positive win-condition.
2	12	Dual win-condition, e.g., multiple sequential tasks.
3	6	Complex win condition.

Table 6: Games with many rules

game	author	rules	complexity analysis
Threes	Benjamin Davis	187	Tetris-like shape composition 'combining threes'.
Robot Arm	increpare	106	Controlling a robot arm to catch apples.
Coin Counter	–	94	Complex number shape manipulation maze.
Memories of Castle Mouse	Wayne Myers	93	Complex puzzle dungeon with many side-effects.

4.2.7 Win-conditions. The number of win-conditions usually ranges from zero to three, as shown in Table 5. One condition specifying the winning game state is usually enough. Additional conditions may be needed for performing more taks in a specific order. Descriptions without win-conditions imply free exploration.

4.2.8 Rules. We observe large differences in the amount of rules games have. Figure 7c shows the distribution of rules over games, in the same way we have illustrated volume and objects.

PuzzleScript's expressive power stems from the fact that few rules can concisely express playful affordances. One in five games has only one or two rules, one third has five rules or less, and still half have ten or fewer rules. However, it is not generally the case that rules and affordances can be mapped one to one. Often, multiple rules express one type of action with contextual effects.

Several complex games stand out. Table 6 shows games with a particularly high rule count. The causes of the complexity differ from game to game. For instance, composite shape manipulation or Tetris-like mechanisms can require many rules, but also games that have many actions and side-effects.

4.2.9 Contextual Analysis. Despite the high quality of the sources, ScriptButler identifies several potential issues. The main reason is the strictness of its checker. In comparison, the online engine is more forgiving. Most of the errors and warnings appear in just 25 games, whereas 28 games have no errors or just one, indicating an enormous difference in code quality. Many of the errors pertain to reserved symbols, and many of the warnings indicate dead code.

4.2.10 Levels. Unsurprisingly, we find that with some exceptions, volume usually indicates game content, i.e. lengthier game descriptions contain more levels or larger ones.

4.2.11 Verbs. Despite the concise notation, rules can express a wide variety of game mechanics. We interpret the gameplay and inspect the code of each game. Table 7 shows the verbs we identify and in at least three games. The next section discusses the game mechanics in more detail.

Table 7: Verbs and affordances

verbs	ct.	mechanism and affordance
move, walk, shift	55	move an avatar or block between tiles
reach, hug, exit, leave	27	reach a tile, avatar or exit
push	17	push an object by colliding
collect, take, obtain	8	obtain an item or object at its location
jump, climb	6	move an avatar up in a side-view
fall	6	fall due to gravity pulling down
skate, slide, run, roll	6	move an object along a line until it collides
switch, toggle, cast	6	activate an item in place
match, remove	6	match objects and remove the pattern
place, put, drop	5	put an object or piece in a place
float, fly	4	delay gravity after jumping
lose, give, rid	4	get rid of an item or object
dig, fill, extend	4	make a path to reach a location
combine, glue	4	combine objects into a composite shape
select	4	select a puzzle piece or avatar
repel	4	repel another object
pull	3	pull an adjacent object directionally
remove, cancel, kill	3	an object disappears on impact
move synchronous	3	move directionally with another object
attract	3	attract another object
redirect, bounce	3	redirect an object through a collision
grow	3	grows objects at anx avatar's location
shoot	3	activate object that directionally collides

4.3 Categorization

Here we distill categories with common pieces of code and rule patterns associated with verbs, affordances and gameplay. We illustrate these categories game summaries and code snippets.

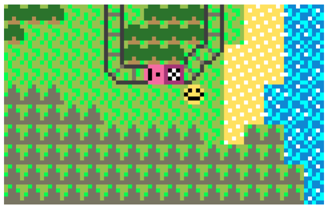
4.3.1 Map orientation. Game levels can be oriented in a top-down or side-view manner. Top down views (90%) orient levels as viewed from above, portraying dungeons, mazes, or puzzles. Alternatively, side views orient the level such that height and gravity play a role, e.g., in platform games or puzzles with falling pieces.

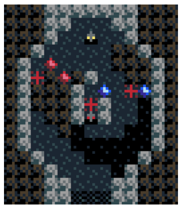
4.3.2 Platformer. Platform games orient levels in a side view of the map. Players control an avatar and reach platforms by jumping, teleporting, climbing and falling or other means. The end-goal is usually reaching the exit or collecting an item by occupying the same space. Table 7 reveals six games whose verbs indicate gravity. Falling and jumping are central to these games. The code reveals common rules that operate in the **DOWN** and **UP** direction. For instance, gravity on Crates in Lime Rick as defined as follows.

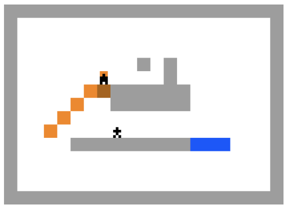
```
DOWN [ Crate | No Obstacle ] -> [ | Crate ]
```

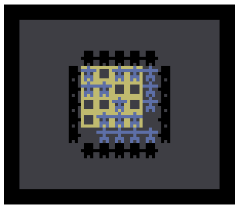
Midas (platform game). In Midas (Fig. 8c) the player has to lose the Midas' touch on water before they reach their loving partner. However, collapsing gold platforms impede Midas' movements.

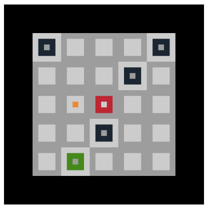
4.3.3 Snake. Snake is a classical video game genre whose rules involve moving a snake-head through a maze as its body grows. Originally, the snake becomes a rigid body that limits its movements. We identify several snake games with similar rules.

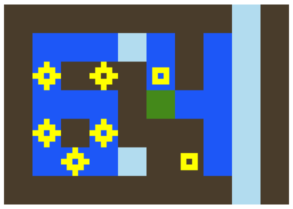

(a) Cute Train

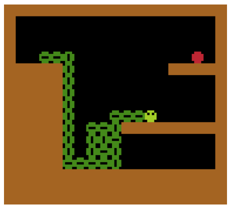

(b) Chaos Wizards

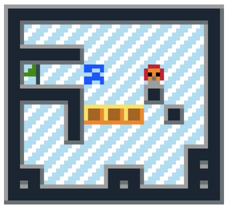

(c) Midas

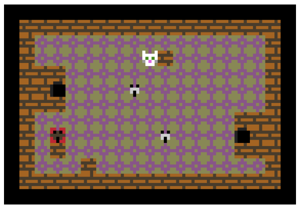

(d) Kettle


(e) Lunar Lockout


(f) Coin Counter


(g) Lime Rick


(h) IceCrates


(i) CastleMouse

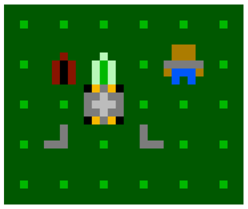

(j) Led Challenge

Figure 8: Screenshots illustrating a wide variety of PuzzleScript Games

Notsnake. In Notsnake, a top-down game by Terry Cavanagh, the player is a snake whose goal is to completely erase its own tail. However, traversing background creates a new tail instead.

```
[ > Player | No Trail ] -> [ Trail | Player ]
[ > Player | Trail ] -> [ | Player ]
```

Lime Rick. Lime Rick by Tommi Tuovinen (Fig. 8g) is a platform game in which the snake has to reach a red piece of food. However, its head can only move in one direction four spaces before it has to switch direction. The crawling pattern reminds of a limerick.

Dungeon Janitor. In Dungeon Janitor by Farbs, the player has to clean a top-down dungeon infested by slime creature that leaves slime in its wake. However, the head of this creature can continue crawling from any slime-infested area. The goal is cleaning all slime by trapping the creature in a corner. Slime grows as follows.

```
(Grow Slime)
[Head | NO Slime NO Wall NO Player] -> [Head | Slime]
```

4.3.4 *Sokoban.*

Sokoban is a classical video game and genre where the player acts as a warehouse keeper. The four games called Sokoban each revolve around moving crates into places by pushing, pulling or otherwise shifting the location of the crates, and reaching certain locations. Levels contain a series of puzzles that require the player to deduce how to use the available space without getting stuck. Pushing is a very common mechanism.

```
(Push Crate)
[ > Player | Crate ] -> [ > Player | > Crate ]
```

Sokoban is surprisingly complex, and has been the subject of complexity research and motion planning problems [8], and continues to be a subject for planning algorithms [11].

4.3.5 *Adventure.*

By default, maps are fully visible. We identify seven extensible top-down views, indicated by keywords **`zoomscreen`** and **`flickscreen`** keywords and WxH dimensions. These maps, which are partially hidden, enable visiting smaller areas of larger levels one by one, revealing details as part of an exploration or an adventure. Players progress through puzzles, messages and quests.

Closet and the Castle (top down, storytelling). This game by HeskHwis and Holly Hatter centers around game narrative. The player explores a house containing various pieces of furniture and appliances. By interacting with these objects, and reading messages, they learn what motivates their character. When they finally reach the car outside, the goal of leaving this house for good becomes clear.

Legend of Zokoban. In The Legend of Zokoban by Joshua Minor, the player navigates a top-down dungeon populated by "baddies" that can catch them. By obtaining a sword, players can instead best the baddies. Pushing rocks on water opens a path to different dungeon areas, and enables reaching the dungeon's exit.

Chaos Wizards. In Chaos Wizards (Fig. 8b), a top-down dungeon crawler, players have to use wizard abilities to clear the path and reach the exit, e.g., by teleporting and causing explosions.

Cute Train. In Cute Train by Mark Wonnacott, the player can explore an island by driving a train on a railway track. By flipping switches, the tracks direct the train to different parts of the island. The player decides what the goal is.

4.3.6 *Puzzle block games.*

Puzzle block games revolve around shifting, pushing and moving puzzle pieces or blocks into place. Unlike crates in Sokoban-likes, these blocks are composite structures. The rules usually involve collision effects, or gluing blocks together. Although gravity can play a role, in the examples top-down views are more common. In some cases, blocks arranged in special patterns, such as rows, remove those blocks to make space for other blocks, in a way similar to Tetris. We give four distinct examples.

In Kettle (Fig. 8d the police have to move the crowd into a designated area. Each level shows increasingly grim messages.

In Led Challenge (Fig. 8j), the goal is turning leds on by pushing wires and leds near a power unit. Beware of the electric current.

In Threes, the player has to combine matching blocks of threes by colliding them. Combining blocks raises the score.

In Coin Counter (Fig. 8f), collecting coins changes the shape of a maze. The goal is reaching the exit, and not losing one's footing.

4.3.7 Board games. In board games, the tile map is a board which contains pieces that can be selected for making moves. However, selection mechanisms are relatively hard to express.

In Lunar Lockout (Fig. 8e) the goal is getting the captain to the destination by strategically colliding with helper pawns.

Similarly, in Bouncers, the player ensures balls reach the exit by first placing bouncer pieces in strategic places.

4.3.8 Directional collisions. Instead of moving an object just one tile at a time, some games continue the movement until the object collides. Associated verbs we identify are skate, roll and run. We give two examples of games with such movements.

In Ice crates (Fig. 8h), the player skates across an ice court. They reach the destination by colliding with boundaries and crates.

In Memories of Castle mouse (Fig. 8i), Wayne Myers recounts memories of the original game. The goal is letting running mice escape trough exits on the boundaries of a castle. Mice fear cats, cats fear dogs, etc. Each level the player controls a different animal that runs directionally, scaring other animals that also run in response.

4.3.9 Synchronous movement. Instead of controlling just one avatar or block, some games let players control multiple at the same time. Of course, each of the object occupies its own space and encounters separate collisions. We count four games with synchronous movements, e.g., Ponies jumping synchronously. The verbs attract and repel indicate other kinds of synchronicity, e.g., By your side.

In Ebony and Ivory, players move two synchronous red things in white and black spaces. By pushing, they can extend white and black spaces such that the red things move adjacent to each other.

5 DISCUSSION

We have conducted mixed-method research. Here we discuss the benefits, limitations and threats to validity of our approach.

5.1 Reverse engineering approach

The key advantage of reverse engineering is one can redesign software for new purposes. In the case of PuzzleScript, the tradeoff between the costs and the benefits is positive. Of course there are also risks. We may have overlooked important details, made incorrect assumptions or misinterpreted design decisions. Overall, through reverse engineering, we have been able to construct our own prototype that satisfies additional requirements.

5.2 ScriptButler

We have proposed ScriptButler as a novel multi-purpose tool capable of parsing, validating, and running PuzzleScript games. Our method, design research, requires that we iteratively design, implement, test and improve our prototype in practice. This process is ongoing. For accurate results, verifying and validating the tool itself is essential. Despite our best efforts in testing, undoubtedly some bugs remain that could affect the results of Section 4.

In addition, we have validated ScriptButler in an empirical study on high quality source code that covers many language features. However, its focus is solely on the static analysis of complete games. This may not be a good indicator for use by developers in improving and debugging the source code as it evolves. Julia describes an evolution scenario that investigates how its feedback can help, also using dynamic analyses [12]. Finally, we have not yet conducted a user study to validate ScriptButler's usability.

5.3 Empirical study

We have conducted an empirical study of PuzzleScript source code that consists of two parts, a quantitative and a qualitative analysis.

5.3.1 Method. We have defined a precise method to ensure accurate and reproducible quantitative results. By using ScriptButler to automate the data analysis, we can account for quality. We have made ScriptButler and the data of the study available on GitHub[4].

5.3.2 Scoping the area of interest. We have scoped the area of interest narrowly by selecting a single source of high quality games. As a result, there may be a selection bias. Adding additional sources could uncover a wider design space. We have opted not to do so. The repository contains diverse show cases by many different authors.

5.3.3 Rules, complexity and quality. Rules are a prime indicator of a game's complexity but usually just take up a fraction of the code. However, a game's complexity is not exclusively determined by its rules. We acknowledge that rigid bodies, collision layers, and the levels, that also determine how the rules work.

5.3.4 Verbs. In our qualitative analysis we have relied on our own ability to interpret the games, but of course, gameplay is highly personal. As a result, the analysis of verbs, and our categorization may be biased towards our own views and understanding. We acknowledge someone else might interpret the gameplay differently, and may not use the same verbs to describe the playful affordances.

A game's mechanisms are not always immediately clear. For instance, in Poseidon's Dreizack we initially interpreted whales that spout water as a laser-guided security system. Inspecting the code has revealed the author's intentions and the rule's effects.

5.3.5 General results. Caution is advised when generalizing the results. An infamous example is the long-standing assumption of a strong linear correlation between Cyclometic Complexity and SLOC, which has been directly contradicted by Landman et al. [16]. By requiring root-cause analysis that relates metrics to the source code, we ensure the results can be traced back to empirical evidence.

6 RELATED WORK

Here we give an overview that relates our mixed-method approach to existing research areas, methods, approaches and tools.

6.1 Automated Game Design

Automated Game Design (AGD) is a research area that proposes and applies various techniques to automate game design processes [22], e.g., algorithms, procedural content generation, and human computer interaction. Cook describes the need for a software engineering discipline for AGD [7]. In a comprehensive survey of languages and tools for game design and development, van Rozen identifies PuzzleScript as one of 108 languages in over 1400 academic publications [26]. The study systematically maps related work, research areas and languages, and distills fourteen research perspectives,

[4]https://github.com/vrozen/ScriptButler

including one on AGD that identifies opportunities for advancing area. We refer to that study for summaries of game DSLs [26].

Several works aim to relate game mechanics to meaning. Summerville et al. propose Gemini, a system for analysis and generation of a game's mechanics [24]. Dormans describes prescriptive patterns for Machinations, a visual notation for game economies [9] which has evolved into a DSL with a pattern-based editor [25].

6.2 Empirical Software Engineering

Empirical software engineering uses empirical research methods to prove or falsify hypotheses about real-world software phenomena. Examples include user studies, analysis of comments, and mining software repositories. Chen et al. study software clones in open source game software [6]. Lee et al. study the impact of game modifications (mods) on the code quality [19].

Our approach is part of the source code analysis and manipulation domain [13], which leverages meta-programs to analyze the qualities of other programs [10]. To the best of our knowledge, our empirical study on a game DSL is the first of its kind.

6.3 PuzzleScript

We have introduced PuzzleScript in Section 2 as output of the reverse engineering method. Here we describe other related work.

Educational applications include teaching game design [5] and programming in PuzzleScript [2]. Many games can be found online[5]. Several games appear on Itch.io[6]. Finally, a branch of the main sources adds several features, notably a solver that attempts to beat games automatically[7]. In contrast, ScriptButler is not a branch.

PuzzleScript has been used in case studies in technical games research. For instance, Lim and Harell present an approach for automated evaluation and generation of PuzzleScript video games and propose two heuristics [20]. Naus and Jeuring propose a DSL for expressing rule-based problems and use generic search algorithms to solve these problems [21]. Osborn et al. introduce PlaySpecs, regular expressions for specifying and analyzing desirable properties of game play traces, sequences of player actions.

6.4 Tiny Online Game Engines

Warren identifies the area of "tiny online game engines" and examines three: Twine, PuzzleScript, and Bitsy [27]. The examination include the engine's interface, its design philosophy, a sampling of games, and information gathered from users. Our works have in common that we aim for better tools for game development. However, the methodologies differ. Our mixed method comprises reverse engineering, design research and an empirical study of the source code. Interviewing users is not part of this work.

6.4.1 Bitsy and Twine. Twine is an open-source tool for telling interactive, nonlinear stories[8]. Bitsy is *"a little editor for little games or worlds"* by Adam Le Doux et al.[9]. The creation environment offers documentation, a room editor, an object editor, a color palette and import/export functions. The interface does not require programming experience, which may help beginners. One avatar, and multiple tiles, sprites and items, their dialogues and behaviors can each be created through menus. Exporting games results in a single HTML file with data and large quantities of JavaScript. Unlike puzzleScript, Bitsy lacks a textual DSL that enables off-line analyses. Instead, the rules are integrated in the menus. As a result, it is not straightforward to reproduce this study on Bitsy.

[5]https://philschatz.com/puzzlescript/ – last visited November 8th 2022
[6]https://itch.io/games/made-with-puzzlescript – last visited November 8th 2022
[7]https://github.com/Auroriax/PuzzleScriptPlus/ – last visited November 8th 2022
[8]https://twinery.org – last visited November 26th 2022
[9]https://make.bitsy.org (last visited November 23rd 2022)

6.5 Patterns, ontologies and typologies

Björk et al. have created a game design pattern catalogue that relates game rules and gameplay [3]. Our work represents first steps in creating an executable pattern catalogue for PuzzleScript. Ontologies and typologies categorize a game's elements in order to critically analyze and distinguish their parts [1, 28]. From an ontological perspective, describing additional facets of user interaction and player experience may prove useful for relating rules and affordances.

7 CONCLUSION

Automated Game Design studies how to empower game designers with languages and tools that automate game design processes. We have investigated to what extent rules, affordances and play can be related by means of source code analysis of DSLs. In particular, we have studied PuzzleScript, an established DSL with an active user community and many available sources.

We have addressed the following questions: 1) what can be observed about the quality of PuzzleScript source code in terms of a) volume in source lines of code (SLOC); of b) objects, collision-layers, win-conditions and levels; and c) rules, interactivity, affordances and play?; and 2) what kind of games can PuzzleScript express, and which game elements, language features, and usage patterns do these games have in common?

To answer these questions, we have conducted mixed-method research that contributes: 1) an analysis of the PuzzleScript language and its engine; 2) ScriptButler, a novel tool for PuzzleScript analysis; and 3) an empirical study on the software quality of 95 games.

Our results indicate that PuzzleScript is a powerful DSL and that ScriptButler serves its purpose as a source code analysis platform well. Although ScriptButler is not yet thoroughly validated as a tool for AGD, its IDE also supports creating games by providing immediate feedback on the code quality. By uncovering the sources of PuzzleScript's expressive power in particular, we also gather evidence on the merits of DSLs for AGD in general. PuzzleScript is a valuable asset in learning how to create better game design tools. Ultimately, research on game DSLs can help advance AGD.

7.1 Future work

We have identified design patterns in Section 4.3 whose analysis may be automated. Future work includes studying how Feature Models [4, 23] can express PuzzleScript's design space, and how these models support pattern-based procedural content generation.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their valuable feedback and for their suggestions that helped improve this paper. We also thank Daria Polak for proofreading this paper.

REFERENCES

[1] Espen Aarseth and Pawel Grabarczyk. 2018. An Ontological Meta-Model for Game Research. In *Proceedings of the 2018 DiGRA International Conference: The Game is the Message, DiGRA 2018, Turin, Italy, July 25–28, 2018*. Digital Games Research Association. http://www.digra.org/digital-library/publications/an-ontological-meta-model-for-game-research/

[2] Anna Anthropy. 2019. *Make Your Own PuzzleScript Games!* No Starch Press.

[3] Staffan Björk, Sus Lundgren, and Jussi Holopainen. 2003. Game Design Patterns. In *Digital Games Research Conference 2003, 4-6 November 2003, University of Utrecht, The Netherlands*. http://www.digra.org/digital-library/publications/game-design-patterns/

[4] Filipe M. B. Boaventura and Victor Travassos Sarinho. 2019. A Feature-Based Approach to Develop Digital Board Games. In *Entertainment Computing and Serious Games - First IFIP TC 14 Joint International Conference, ICEC-JCSG 2019, Arequipa, Peru, November 11–15, 2019, Proceedings (LNCS, Vol. 11863)*. Springer, 175–186. https://doi.org/10.1007/978-3-030-34644-7_14

[5] Alexander Card, Wengran Wang, Chris Martens, and Thomas W. Price. 2021. Scaffolding Game Design: Towards Tool Support for Planning Open-Ended Projects in an Introductory Game Design Class. In *IEEE Symposium on Visual Languages and Human-Centric Computing, VL/HCC 2021, St Louis, MO, USA, October 10-13, 2021*. IEEE, 1–5. https://doi.org/10.1109/VL/HCC51201.2021.9576209

[6] Yaowen Chen, Iman Keivanloo, and Chanchal Kumar Roy. 2014. Near-miss Software Clones in Open Source Games: An Empirical Study. In *IEEE 27th Canadian Conference on Electrical and Computer Engineering, CCECE 2014, Toronto, ON, Canada, May 4–7, 2014*. IEEE, 1–7. https://doi.org/10.1109/CCECE.2014.6901018

[7] Michael Cook. 2020. Software Engineering for Automated Game Design. In *IEEE Conference on Games, CoG 2020, Osaka, Japan, August 24–27, 2020*. IEEE, 487–494. https://doi.org/10.1109/CoG47356.2020.9231750

[8] Joseph Culberson. 1997. *Sokoban is PSPACE-complete*. Technical Report TR97-02. University of Alberta.

[9] Joris Dormans. 2012. *Engineering Emergence: Applied Theory for Game Design*. Ph. D. Dissertation. University of Amsterdam.

[10] Sebastian Erdweg, Tijs van der Storm, Markus Völter, Meinte Boersma, Remi Bosman, William R. Cook, Albert Gerritsen, Angelo Hulshout, Steven Kelly, Alex Loh, Gabriël D. P. Konat, Pedro J. Molina, Martin Palatnik, Risto Pohjonen, Eugen Schindler, Klemens Schindler, Riccardo Solmi, Vlad A. Vergu, Eelco Visser, Kevin van der Vlist, Guido Wachsmuth, and Jimi van der Woning. 2013. The State of the Art in Language Workbenches – Conclusions from the Language Workbench Challenge. In *Software Language Engineering - 6th International Conference, SLE 2013, Indianapolis, IN, USA, October 26–28, 2013. Proceedings (LNCS, Vol. 8225)*. Springer, 197–217. https://doi.org/10.1007/978-3-319-02654-1_11

[11] Dieqiao Feng, Carla P. Gomes, and Bart Selman. 2020. A Novel Automated Curriculum Strategy to Solve Hard Sokoban Planning Instances. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6–12, 2020, virtual*.

[12] Clement Julia. 2022. *ScriptButler: Leveraging Meta-Programming Principles to facilitate the Software Evolution of Digital Games*. Master's thesis. University of Amsterdam. https://scripties.uba.uva.nl/search?id=record_52797

[13] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Ninth IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009, Edmonton, Alberta, Canada, September 20-21, 2009*. IEEE Computer Society, 168–177. https://doi.org/10.1109/SCAM.2009.28

[14] Raph Koster. 2013. *Theory of Fun for Game Design*. O'Reilly Media, Inc.

[15] Raph Koster. 2016. The Limits of Formalism. In *Presentation delivered at the BIRS Workshop on Computational Modeling in Games*. Raph Koster's Website. https://www.raphkoster.com/games/presentations/the-limits-of-formalism/

[16] Davy Landman, Alexander Serebrenik, Eric Bouwers, and urgen J. Vinju. 2016. Empirical Analysis of the Relationship between CC and SLOC in a Large Corpus of Java Methods and C Functions. *J. Softw. Evol. Process.* 28, 7 (2016), 589–618. https://doi.org/10.1002/smr.1760

[17] Stephen Lavelle. 2015. PuzzleScript. https://github.com/increpare/PuzzleScript Last visited October 26th 2022.

[18] Stephen Lavelle. 2015. PuzzleScript Documentation. https://www.puzzlescript.net/Documentation/documentation.html Last visited October 26th 2022.

[19] Daniel Lee, Dayi Lin, Cor-Paul Bezemer, and Ahmed E. Hassan. 2020. Building the Perfect Game - An Empirical Study of Game Modifications. *Empir. Softw. Eng.* 25, 4 (2020), 2485–2518. https://doi.org/10.1007/s10664-019-09783-w

[20] Chong-U Lim and D. Fox Harrell. 2014. An Approach to General Videogame Evaluation and Automatic Generation using a Description Language. In *2014 IEEE Conference on Computational Intelligence and Games, CIG 2014, Dortmund, Germany, August 26–29, 2014*. IEEE, 1–8. https://doi.org/10.1109/CIG.2014.6932896

[21] Nico Naus and Johan Jeuring. 2016. Building a Generic Feedback System for Rule-Based Problems. In *Trends in Functional Programming - 17th International Conference, TFP 2016, College Park, MD, USA, June 8–10, 2016, Revised Selected Papers (LNCS, Vol. 10447)*. Springer, 172–191. https://doi.org/10.1007/978-3-030-14805-8_10

[22] Mark J. Nelson and Michael Mateas. 2007. Towards Automated Game Design. In *AI*IA 2007: Artificial Intelligence and Human-Oriented Computing (LNCS, Vol. 4722)*. Springer Berlin Heidelberg, 626–637. https://doi.org/10.1007/978-3-540-74782-6_54

[23] Victor Travassos Sarinho, Gabriel S. de Azevedo, and Filipe M. B. Boaventura. 2018. AsKME: A Feature-Based Approach to Develop Multiplatform Quiz Games. In *17th Brazilian Symposium on Computer Games and Digital Entertainment, SBGames 2018, Foz do Iguaçu, Brazil, October 29 – November 1, 2018*. IEEE Computer Society, 38–47. https://doi.org/10.1109/SBGAMES.2018.00014

[24] Adam Summerville, Chris Martens, Sarah Harmon, Michael Mateas, Joseph C. Osborn, Noah Wardrip-Fruin, and Arnav Jhala. 2019. From Mechanics to Meaning. *IEEE Trans. Games* 11, 1 (2019), 69–78. https://doi.org/10.1109/TCIAIG.2017.2765599

[25] Riemer van Rozen. 2015. A Pattern-Based Game Mechanics Design Assistant. In *Proceedings of the 10th International Conference on the Foundations of Digital Games, FDG 2015, Pacific Grove, CA, USA, June 22–25, 2015*. Society for the Advancement of the Science of Digital Games. http://www.fdg2015.org/papers/fdg2015_paper_79.pdf

[26] Riemer van Rozen. 2020. Languages of Games and Play: A Systematic Mapping Study. *Comput. Surveys* 53, 6 (Dec. 2020). https://doi.org/10.1145/3412843 Interactive version: https://vrozen.github.io/LoGaP/.

[27] Jonah Warren. 2019. Tiny Online Game Engines. In *Decision and Game Theory for Security - 10th International Conference, GameSec 2019, Stockholm, Sweden, October 30 – November 1, 2019, Proceedings (LNCS, Vol. 11836)*. Springer, 1–7. https://doi.org/10.1109/GEM.2019.8901975

[28] José P. Zagal, Michael Mateas, Clara Fernández-Vara, Brian Hochhalter, and Nolan Lichti. 2005. Towards an Ontological Language for Game Analysis. In *Digital Games Research Conference 2005, Changing Views: Worlds in Play, June 16–20, 2005, Vancouver, British Columbia, Canada*. http://www.digra.org/digital-library/publications/towards-an-ontological-language-for-game-analysis/