First-Person Realtime Collaborative Metaprogramming Adventures

Riemer van Rozen[1], **Youri Reijne**[2], **Clement Julia**[2] **and Georgia Samaritaki**[2]

[1]Centrum Wiskunde & Informatica
Amsterdam, The Netherlands
rozen@cwi.nl

[2]University of Amsterdam
Amsterdam, The Netherlands
{y.reijne, clement.julia13, samaritakigeorgia}@gmail.com

CEUR-WS.org/Vol-3217/paper20.pdf

Abstract

Game developers strongly rely on programming languages and tools for creating educative, adventurous, challenging, humorous and playful interactive experiences. However, it is not yet well-understood how to construct languages and tools for improving games and play. In this position paper we discuss how to leverage language workbenches and metaprogramming techniques for creating language-centric solutions that help speed-up game design and development. We present a research vision, followed by three illustrative examples that demonstrate how the available enabling technology can be applied to tackle recurring challenges in language prototyping. Finally, we reflect on threats to validity and discuss challenges and opportunities for future collaboration.

Introduction

We are at the frontier of an emerging research area that explores what informs the design and construction of good games. This field studies to what extent languages, notations, patterns and tools, can offer experts theoretical foundations, systematic techniques and practical solutions they need to raise their productivity and improve the quality of games and play. We have conducted a systematic mapping study on *'languages of games and play'* that maps the breadth of related work (van Rozen 2021). This study gives an overview of research areas and publication venues, summarizes over 100 languages identified in more than 1400 publications, synthesizes a set of fourteen complementary research perspectives, and relates challenges and approaches to opportunities for automated game design.

Our findings indicate that languages play a central role in an increasing number of publications. Languages are necessary in distinct research areas for a variety of technical and non-technical reasons. Here we discuss a brief selection. For a more elaborate overview, we refer to the mapping study.

In practice, game developers rely on game engines, languages and tools for constructing high quality games. Authors propose languages as development tools or design aids aimed at communication, creativity, productivity or quality. Automated Game Design (AGD) proposes mixed-initiative or co-creative languages and *authoring tools* for helping domain-experts contribute in expressing a game's design. Procedural Content Generation (PCG) aims to deliver *generators* that express and produce *game content* such as behaviors, game mechanics, virtual worlds, dungeons, levels, narratives or stories. In AI, the study of *general game playing* requires Game Description Languages (GDLs) as a testbed for algorithms that play many games well. Education or Game-Based Learning (GBL) proposes languages that help educators express lessons and training material for helping learners master a subject, such as programming or game design. Languages are needed for a myriad of reasons, and are crucial components of research platforms, methodologies, experimental setups and case studies.

Unfortunately, developing the necessary language prototypes comes at a high cost. Researchers and developers spend a considerable amount of time and effort on developing and maintaining languages, generators and tools. Several recurring technical challenges present obstacles that prevent them from rapidly creating high quality language prototypes. It is not commonly known that this type programming is called *metaprogramming*, and that language prototypes for analyzing and transforming the source code of other programs are *metaprograms*. To alleviate and speed-up prototyping efforts, developers require appropriate meta-tooling.

In this position paper, we argue for collaborative multi-disciplinary research and development of languages and tools that tackle research challenges related to game design and development. In particular, we pose that *language workbenches* and *metaprogramming techniques* may be key to advancing multi-disciplinary research in this area. These tools can be leveraged to support the practice of metaprogramming. Metaprogramming techniques can help developers speed-up the construction of language prototypes, raise the quality, bridge technological spaces and combine the state-of-the-art in novel language-centric solutions.

First, we detail our position in the context of, and based on, a previously published research vision. Next, we discuss three illustrative examples that apply the langue workbench Rascal. Each example discusses the creation of a proof of concept for an ongoing master project at the University of Amsterdam in collaboration with Centrum Wiskunde & Informatica. We discuss selected challenges and demonstrate how to leverage Rascal's language features in creating relatively small prototypes that serve our research needs.

1. First, we discuss studying PuzzleScript, an online script

Copyright © 2021by the paper's authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

language and game engine that offers a concise expressive notation for designing a wide variety of puzzle games. We demonstrate how to reverse engineer its JavaScript implementation into a concise Rascal prototype for studying its design space, analyzing the qualities of existing games, and generating new ones using patterns and metrics.

2. Next, we discuss improving Ludoscope, a language and tool for level generation that has been successfully applied in the Unexplored series. We present ongoing work that aims to add support for debugging, content orchestration and improved quality assurance by combining constraints with transformative grammars.
3. Finally, we present a live interactive visualization prototype, achieved by combining meta-programming in Rascal with Unity 3D. We illustrate how the synthesis of these domain specific workbenches, and how modular language creation and analysis could support game play.

We conclude with reflections on threats to validity and challenges and opportunities for future collaboration.

Research Vision

We reiterate a concise version of the research vision that appears as part of a systematic mapping study on languages of games and play, which has been previously published as:

van Rozen, R. 2021. Languages of Games and Play: A Systematic Mapping Study. *ACM Computing Surveys* 53(6).

The reader can navigate and explore the map's interactive website at the following url: https://vrozen.github.io/LoGaP/

Languages of Games and Play

Languages of games and play are language-centric approaches for tackling challenges and solving problems related to game design and development. We propose studying existing languages and creating new ones. Two central hypotheses drive this research. We formulate a general and a specific hypothesis:

1. Languages, structured notations, patterns and tools can offer designers and developers theoretical foundations, systematic techniques and practical solutions they need to raise their productivity and improve the quality of games and play.
2. "Software" languages (and specifically domain-specific languages) can help automate and speed-up game design processes.

Languages of games and play exist in many shapes and forms. The next section describes a technical point of view that we explore in more detail in this paper, and which also details and motivates the second more specific hypothesis.

Domain-Specific Languages

We aim to deliver solutions that automate game design and speed-up game development with so-called Domain-Specific Languages (DSLs), an approach originating in the field of Software Engineering. Van Deursen et al. define the term as follows:

> *"A Domain-Specific Language is a programming language or executable specification language that offers, through appropriate abstractions and notations, expressive power focused on, and usually restricted to, a particular problem domain."* (van Deursen, Klint, and Visser 2000)

DSLs have several compelling benefits. They have been successfully created and applied to boost the productivity of domain-experts and raise the quality of software solutions. For instance, in areas like file carving in digital forensics (van den Bos and van der Storm 2011), engineering financial products (van Deursen, Klint, and Visser 2000), and controlling lithography machines (Tikhonova et al. 2013).

DSLs divide work and separate concerns by offering domain-experts ways to independently evolve and maintain a system's parts. Typically, DSLs raise the abstraction level and incorporate domain-specific terminology that is more recognizable to its users. Powerful language workbenches enable analyses, optimizations, visualizations (Erdweg et al. 2013), and foreground important trade-offs, e.g., between speed and accuracy in file carving.

Naturally, there are also costs. DSLs are no silver bullet for reducing complexity. Time and effort go into developing the right language with features that are both necessary and sufficient for its users. In addition, a DSL may have a steep learning curve and users require training (van Deursen, Klint, and Visser 2000). While DSLs help users maintain products, DSLs themselves also demand maintenance and must evolve to accommodate new requirements, usage scenarios, restrictions and laws, such as new legislation on financial transparency or privacy.

Visual Programming Languages

Here we describe our position and motivation. We aim to empower game designers with DSLs that automate and speed-up the game design process. We wish to learn how to facilitate the design space exploration and reduce design iteration times. We envision a set of complementary visual languages, techniques and tools that help designers boost their productivity and raise the quality of games and play. Challenges include offering abstractions and affordances for:

1. expressing a game's parts as source code artifacts, especially interaction-bound game elements, and modifying these at any given moment.
2. evolving 'games and play' by steering changes in the source code towards new gameplay goals for prototyping, play testing, balancing, fine-tuning or polishing.
3. obtaining immediate and continuous (live) feedback on a game's quality by continuously play testing the effect of changes on quantified gameplay hypotheses.
4. obtaining feed forward suggestions that focus creative efforts and assist in exploring alternative design decisions in a targeted way.
5. forming better mental models for learning to better predict the outcome on play.

The mapping study on languages of games and play provides a good starting point for choosing areas, formulating questions and charting research trajectories. Next, we discuss *how* the necessary language prototypes can be created.

Applying Language Workbenches

We propose an approach to Automated Game Design (AGD) that leverages language workbenches and metaprogramming techniques. Many languages exist that facilitate constructing DSLs, compilers, generators, tools and visualizations. Erdweg et al. describe the state of the art in language workbenches (Erdweg et al. 2013). Examples of metaprogramming languages and language workbenches include Epsilon, GemocStudio, Meta-Edit+, MPS, Racket, Rascal, Spoofax and Xtext. These tools and techniques are part of Programming Language research and a subject of Software Language Engineering (Lämmel 2018). We argue they provide an indispensable means in prototyping languages, bridging the gap between technological spaces and combining the state-of-the-art in novel language-centric solutions.

Previously, several authors have advocated the use of generic language technology, and demonstrated its power in relatively small, illustrative and educational examples (van Rozen 2021). For instance, Walter and Masuch (2011) discuss how to integrate DSLs into the game development process. Maier and Volk (2008) report teaching experiences on applying DiaMeta, an EMF-based language workbench for creating visual DSLs, e.g., for level editors for classic games such as PacMan and the platform game Pingus. Insights include that metamodeling has a steep learning curve and that the proposed approach speeds-up game prototyping. Flatt (2012) demonstrates in a tutorial-like manner how to create languages in Racket, and describes an illustrative text-adventure DSL for interactive fiction.

We present a more elaborate metaprogramming perspective that adds a vision on AGD. We discuss three larger language prototypes, each with distinct goals and applications, developed for case studies in collaboration with indie game developers. We share our experiences on applying Rascal.

Applying the Rascal Metaprogramming Language

Rascal is a programming language that has been specifically designed by Klint, van der Storm and Vinju to facilitate metaprogramming (2009a), and to create programs for source code analysis and manipulation (2009b)[1].

Here we discuss how its features can help tackle recurring technical challenges in language prototyping. For instance, for parsing text, processing concrete syntax, defining abstract data types, transforming data, traversing trees, tracking origins and generating code, to name a few.

Compiler Construction Keeping language designs simple and concise usually entails separating the translation into phases, a common practice in compiler construction (Aho, Sethi, and Ullman 1986). Therefore, developers have to construct staged compilers using grammars, datatypes and visitors that analyze and transform programs step by step, e.g., for parsing, checking contextual constraints, and optimizing and generating code. Rascal provides functions, pattern matching and the **visit** expression for separating code into cases, and traversing trees in a concise manner. In addition, the builtin **loc** datatype provides an vital means for *origin tracking* (van Deursen, Klint, and Tip 1993), tracing source locations over successive transformations.

Bridging the Gap between Technological Spaces Translations from one formalism to another are often needed for bridging the gap between technological spaces, e.g., for applying model-checkers, SMT solvers and ASP for calculating results or proofs, or libraries or algorithms written in a different formalism In addition, it may be necessary to separate games and tool ecosystems into modular components across platforms for reusing game engines, visual simulations, browsers, and other components.

Rascal provides the *'glue'* we need to put these components together. For instance, *string templates* provide a means to write concise code generators. Rascal offers many libraries, e.g., JSON and CSV, but others can be easily added, either natively or by wrapping Java methods. We will discuss bridging the gap between Java-based Rascal and Unity 3D, which is Mono-based, in the context of code puzzles.

Formalizing Language Semantics Unfortunately, precise program analyses are impossible without formal language semantics, e.g., for simulating the effects on dynamic user interactions and play. Existing languages may lack formalizations altogether, which complicates efforts on reverse engineering, and prevents reusing and extending tools in successive case studies. In this paper, we study two existing languages PuzzleScript and LudoScope. The prototypes we describe represents necessary steps for further study.

Creating Interactive User Interfaces Usually, intended language users are creative experts whose focus is on game design or content creation. These users need well-designed interactive visual interfaces for authoring and debugging programs, e.g., for mixed-initiative co-creative design and procedural content generation. Visual debuggers may also require embeddable game engines with APIs for remote access. Rascal enables creating web-based interfaces using its Salix framework[2]. The prototypes we will discuss currently still have an Eclipse interface that obtains syntax highlighting from the respective grammars, which is a quick way to start. Next, we discuss three illustrative language prototypes, and show that Rascal has many built-in features we need.

Analysis and Generation of PuzzleScript

A key challenge in Automated Game Design is defining languages whose semantics directly describe playful affordances. One way to approach this challenge is studying existing *tiny online game engines* (Warren 2019) for combinatorial games such as card games and puzzle games.

Here, we discuss a study of PuzzleScript, identified as Language 91 in (van Rozen 2021), an online game engine that offers a concise expressive notation for defining a wide variety of puzzle games. We reverse engineer its JavaScript implementation (referred to here as the *'original'*) into a concise Rascal prototype for studying its design space, analyzing the qualities of existing games and generating new ones using patterns and metrics.

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

[2] https://github.com/usethesource/salix

PuzzleScript is an online textual puzzle game design language and interpreter created by Stephen Lavelle using JavaScript and HTML5/CSS[1]. PuzzleScript game levels are tile maps populated by objects (named sprites of 5x5 pixels) that can move and collide, and whose game logic is defined as a set of rewrite rules. Figure 1 shows an example where the objective is to push crates into place. When the player collides with a crate, both directionally move if possible. The sources are released under the MIT license[2]. Lim and Harell (2014) present an approach for automated evaluation and generation of PuzzleScript videogames and propose two heuristics. The first, level state heuristics, determines how close the state of given level is to completion during gameplay. The second, ruleset heuristics, evaluates rules defining a videogame's mechanics and assesses them for playability. Naus and Jeuring (2016) propose the *heuristicHint* an algorithm for optimized heuristics search, and create a DSL for expressing rule-based problems. They apply the algorithm to PuzzleScript and solve Sokoban levels. Vermeulen (2018) creates a feature model of PuzzleScript mechanics and shows he can use the model to analyze existing rules and generate new ones. Osborn et al. apply Playspecs (Language 57).

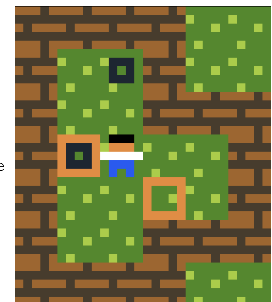
[1]https://www.puzzlescript.net (visited August 17th 2021)
[2]https://github.com/increpare/PuzzleScript (visited August 17th 2021)

publication	query	publication type	research category	note
(Lim and Harrell 2014)	110w	conference paper	validation research	
(Naus and Jeuring 2016)	~~gd~~	conference paper	validation research	citation added in this paper
(Vermeulen 2018)	–n	Bachelor's thesis	solution proposal	citation added in this paper

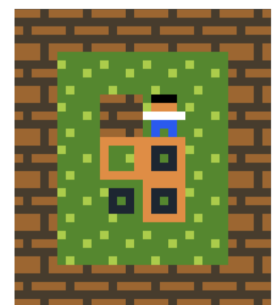
Summary adapted from (van Rozen 2021).

```
title Simple Block Pushing Game
author Stephen Lavelle
homepage www.puzzlescript.net
========
OBJECTS
========
Background
lightgreen green
11111
01111
11101
11111
10111

Target
darkblue
.....
.000.
.0.0.
.000.
.....

Wall
brown darkbrown
00010
11111
01000
11111
00010

Player
black orange white
     blue
.000.
.111.
22222
.333.
.3.3.

Crate
orange
00000
0...0
0...0
0...0
00000

=======
LEGEND
=======
. = Background
# = Wall
P = Player
* = Crate
@ = Crate and
    Target
O = Target

=======
SOUNDS
=======
Crate MOVE 36772507

================
COLLISIONLAYERS
================
Background
Target
Player, Wall, Crate
======
RULES
======
[> Player |
    Crate] -> [>
    Player |>
    Crate]

==============
WINCONDITIONS
==============
All Target on Crate

=======
LEVELS
=======
####..
#.O#..
#..###
#@P..#
#..*.#
#..###
####..

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

(a) Source code with dynamic dynamic syntax highlighting of sprites



(b) First level



(c) Second level

Figure 1: PuzzleScript tutorial: "Simple Block Pushing Game" (from puzzlescript.net, figure appears in (van Rozen 2021))

Parsing PuzzleScript

We aim to study all existing PuzzleScript games and facilitate designing new ones. Therefore, we need a parser. The original JavaScript parser is hand-crafted and line-based, and comprises 1065 Lines of Code (LOC). We instead provide a grammar for PuzzleScript, composed of just 50 LOC of Rascal code[3]. Figure 2 shows *lexicals*, symbols consisting of sequences of characters that are not interrupted by layout symbols. For conciseness we omit keywords here. Figure 3 shows the syntax, a literal description of game definitions that consist of the following sections.

[3]https://github.com/ClementJ18/PS

```
1  layout LAYOUTLIST = LAYOUT* !>> [\t\r\ ] !>> "(";
2  lexical LAYOUT //layout consists of
3    = [\t\r\ ] //tab, carriage return and space (but not line break)
4    | ˆComment LBS //solitary comments
5    > Comment; //comments
6  lexical Comment = @category="Comment" "(" (![()]*|Comment)")";
7  lexical LB = [\n];
8  lexical LBS = LB+ !>> "\n";
9  lexical DELIM = [=]+;
10 lexical ID = [a–z0–9.A–Z]+ !>> [a–z0–9.A–Z] \ Kwds;
11 lexical KeyChar = [a–zA–Z.!@#$%&*];
12 lexical LegendKey = KeyChar+ !>> KeyChar \ Kwds;
13 lexical SpriteLine = "\n" << [0–9.]+ !>> [0–9.] \ Kwds;
14 lexical Pixel = [a–zA–Z.!@#$%&*0–9];
15 lexical LevelLine = "\n" << Pixel+ !>> Pixel \ Kwds;
16 lexical String = ![\n]+ >> [\n];
17 lexical SndIndex = [0–9]|"10" !>> [0–9]|"10";
18 lexical Directional = [\>\<ˆv] !>> [a–z0–9.A–Z];
```

Figure 2: PuzzleScript layout and lexicals in RASCAL

Language 57 ***Playspecs*** *generic / tool / practice*

Osborn et al. introduce PlaySpecs, regular expressions for specifying and analyzing desirable properties of game play traces, sequences of player actions. PlaySpecs are validated with the PuzzleScript engine, which is itself described as Language 91, and Prom Week, a social simulation puzzle game. The TypeScript sources of PlaySpecs are released under the MIT license (visited August 17th 2021)[1].

[1] https://github.com/JoeOsborn/playspecs-js

publication	query	publication type	research category
(Osborn et al. 2015)	930w	conference paper	validation research
(Osborn 2018)	–w	PhD thesis	validation research

Summary appears in (van Rozen 2021)

Objects describe named sprites. Legend enables users to define short-hand symbols for expressing levels (rectangular tile maps) using the assigned characters. Sounds can be added for special effects. Collision layers determine which objects collide, enabling or disabling their movement. Rules define how players can interact with the objects on the tile map, and how events trigger object movement. These are rewrite rules whose left hand side is a pattern that must match on the game's current state for the rule to trigger. The right hand side specifies a result in case the rule succeeds.

Our grammar, which to the best of our knowledge is the first complete syntax formalization, greatly simplifies constructing PuzzleScript prototypes. With the exception of a few corner cases, it already passes a test suite composed of 83 existing games. Most importantly, it is more compact and readable, and easier to maintain and reuse than the original.

Providing Errors and Warnings

Developers require feedback and warnings, ideally 'live' during coding, for improving their designs. The original JavaScript implementation solely provides syntax coloring as feedback. Once the code is compiled, it provides more explicit and comprehensive feedback. However, because of the single phase parsing, errors tend to cascade forcing the compilation to abort before the game is fully checked.

We introduce a *static checker* for improving live feedback, a contextual analysis phase that runs after every change developers make to the code. On every change, it parses the code, annotates the parse tree and visualizes errors and warnings in the IDE. Developers obtain a wide variety of messages, ranging from critical errors (invalid syntax, undefined references and duplicate definitions) to informative feedback, e.g., as shown in Figure 4.

```
1  start syntax Game = game: (Prelude LBS)* {Section LBS}* LBS?;
2
3  syntax Prelude = prelude: PreludeKwd String+;
4  syntax Section
5    = objects: (Delim LBS)? 'OBJECTS' LBS Delim? (LBS Object)*
6    | legend: (Delim LBS)? 'LEGEND' LBS Delim? (LBS Legend)*
7    | sounds: (Delim LBS)? 'SOUNDS' LBS Delim? (LBS Sound)*
8    | layers: (Delim LBS)? 'COLLISIONLAYERS' LBS Delim? (LBS Layer)*
9    | rules: (Delim LBS)? 'RULES' LBS Delim? (LBS Rule)*
10   | conditions: (Delim LBS)? 'WINCONDITIONS' LBS Delim? (LBS Cond)*
11   | levels: (Delim LBS)? 'LEVELS' LBS Delim? (LBS Level)*;
12
13 syntax Object = object: ID LegendKey? LB ID+ (LB Sprite)?;
14 syntax Sprite = sprite:
15   SpriteLine LB SpriteLine LB SpriteLine LB SpriteLine LB SpriteLine LB;
16 syntax Legend = entry: LegendKey '=' ID (LegendKwd ID)*;
17 syntax Sound = sound: ID+;
18 syntax Layer = layer: {ID ','}+;
19
20 syntax Rule = rule: ID? RulePart+ '–\>' (Command|RulePart)* Message?;
21 syntax RulePart = part: '[' {RuleContent '|'}+ ']';
22 syntax RuleContent = content: (ID | Directional)*;
23 syntax Message = message: 'message' String+;
24 syntax Command = command: CommandKwd | play_sound: 'sfx' SndIndex;
25
26 syntax Cond = condition: ID+;
27 syntax Level = level: (LevelLine LB)+ | level_progress: Message;
```

Figure 3: PuzzleScript grammar expressed in RASCAL

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

(a) Sprite not 5x5

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

(b) Uneven level rows

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

(c) Unused color

```
[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 4: Errors and warnings detected by the checker

Language 36 **Ludoscope** *generic / engine / practice*

Lindenmayer systems (or L-systems) are generative grammars that were originally intended for describing plant growth patterns (Lindenmayer 1968) and are now also used for PCG. Dormans (2010) investigates strategies for generating levels for action adventure games, and proposes mission and spaces as two separate structures. He analyzes a Zelda game level, and generates its missions and spaces using transformative grammars. Ludoscope is a tool for designing procedurally generated game levels based on these principles. In Ludoscope, level transformation pipelines step-by-step transform level content, gradually adding detail, dungeons, enemies, encounters, missions, etc. These pipelines consist of grammar rules that work on content represented as tile maps, graphs, Voronoi Diagrams and Strings. Karavolos et al. (2015) explore applying Ludoscope in the design of two distinct pipelines that generate dungeons of a rogue like game and platform levels of a metroidvania game. Van Rozen and Heijn propose two techniques for analyzing the quality of level generation grammars called MAD and SAnR (see Language 58).

publication	query	publication type	research category	note
(Dormans 2010)	–w	workshop paper	proposal of solution	grammars
(Dormans and Bakkes 2011)	–w	journal article	proposal of solution	grammars
(Dormans 2011)	–w	workshop paper	proposal of solution	grammars
(Dormans and Leijnen 2013)	~~language~~	workshop paper	validation research	Ludoscope
(Karavolos, Bouwer, and Bidarra 2015)	~~language~~	conference paper	validation research	Ludoscope
(van Rozen and Heijn 2018)	–n	workshop paper	proposal of solution	Ludoscope Lite
(Heijn 2018)	–n	workshop paper	proposal of solution	Ludoscope Lite

Summary adapted from (van Rozen 2021)

The analysis consists of sub-phases with progressively stricter rules, starting with the parsing. This allows isolating errors, and ensuring they do not affect subsequent phases. Rascal's functions can be overloaded with pattern signatures to act like an extensible switch statement. We use these 'partial functions' for easily adding new checker cases.

Preventing Bad Gameplay

Predicting the outcome of changing a game's rules on play is hard in general. In PuzzleScript, Victory Conditions and Rules define how the player interacts with the game. Changing a rule can cause levels to become impossible to win or, instead make them trivial to solve. Unfortunately, it is not always possible to foresee how those changes impact levels.

As part of future work, we aim to enable making more informed design decisions by adding a *dynamic analyzer*, a process capable of playing games, solving trivial puzzles, and analyzing potentially negative impact of changes.

Outlook and Ongoing Work

Our PuzzleScript prototype is more maintainable and extensible than the original JavaScript implementation. In particular Rascal's pattern matching and meta-programming capabilities proved useful in creating a concise prototype. We continue to develop a tool that empowers game designers in writing more robust code and creating better games.

A Ludoscope Debugger

Procedural level generation studies how to generate high quality game levels populated with items, quests, enemies for unlikely encounters, astonishing stories and epic adventures. LudoScope, identified as Language 36 in (van Rozen 2021), is a grammar-based language and tool that offers powerful capabilities for creating diverse generators.

Language 58 **MAD** *generic / engine / practice*

Van Rozen and Heijn study Ludoscope (Language 36) and address quality issues of grammar-based level generation. They propose two techniques for improving grammars to generate better game levels. The first, Metric of Added Detail (MAD), leverages the intuition that grammar rules gradually add detail, and uses a detail hierarchy that indicates for calculating the score of rule applications. The second, Specification Analysis Reporting (SAnR) proposes a language for specifying level properties, and analyzes level generation histories, showing how properties evolve over time. LudoScope Lite is a prototype that demonstrates the techniques[1].

[1] https://github.com/visknut/LudoscopeLite

publication	query	publication type	research category
(van Rozen and Heijn 2018)	–n	workshop paper	solution proposal
(Heijn 2018)	–n	Master's thesis	solution proposal

Summary adapted from (van Rozen 2021)

Currently, due to the lack of research, the tool has very limited debugging capabilities. As a result, level designers have to spend a great deal of time 'eyeballing' generated game levels. We aim to improve grammar-based generators in general, and study LudoScope in particular. Here, we discuss ongoing work on LudoScope Lite, a prototype written in Rascal for studying grammar-based generators.

Creating a Manageable Language Prototype

Instead of supporting every Ludoscope language feature, we redesign a core portion of the language to study its properties in isolation. We focus on transforming tile maps and graphs, and design an understandable textual syntax for creating pipelines for study, e.g., Figure 5. Therefore, we have

```
1  pipeline {
2      alphabet {
3                  floor f #010101;
4                  wall * #111111;
5                  exit  x #000000;
6          }
7          options {
8                  size: 4x4;
9                  tiletype: f;
10         }
11         module firstModule {
12                 rules {
13                         addWalls:
14                         ffff,
15                         ffff,
16                         ffff,
17                         ffff
18                         ->
19                         ****,
20                         *ff*,
21                         *ff*,
22                         ****;
23                         addExit:
24                         f* -> fx;
25                 }
26                 recipe {
27                         addWalls;
28                         addExit;
29                 }
30                 resolvable constraint b: count(x) == 1;
31         }
32 }
```

Figure 5: Level transformation pipeline in Ludoscope Lite that creates a 4x4 tilemap, and adds an exit.

no need for its visual interface and we only support the language features relevant to our question. Our code base is not yet feature-complete but has a manageable size of 1.3 KLOC compared to Ludoscope's 21 KLOC of C#.

Integrating ASP with Grammar-Based Approaches

Grammar-based approaches for level generation enable great flexibility, but are notoriously hard to control. Answer Set Programming (ASP) on the other hand, guarantees constraints are met. For instance, Smith and Mateas (2011) demonstrate through the Chromatic Maze written in ASP (called design space) how users can redefine the game spaces by iteratively adding constraints to the AnsProlog program. We want the best of both worlds. Even though the current version of our prototype does not yet fully combine the two, our study borrows concepts from ASP. We still consider translating grammars to ASP using Rascal.

Adding Language Features for Error Handling

One crucial issue of LudoScope remains the lack of error handling. When a level generation pipeline generates 'useless' content, e.g. a map that cannot be crossed by the player, it is impossible to know how or where the error occurred. We borrow the concept of constraints from ASP to guide the generation and offer a way to trace back errors by logging them when they happen.

This is achieved by the constraint seen in line 30 in Figure 5. This feature can not only stop the generation from happening, but also alter it at times if the designer wishes so. At the same time, the constraints act as a form of fail-safe for when the generation derails in an unwanted direction.

Constraints that orchestrate content generation, are labelled as resolvable. We offer these along with their respective handlers as a way for the designer to bypass unexpected/unwanted content at run time. The constraints are periodically checked and whenever a module breaks one an error is shown. The error logs are enhanced by Rascal's capability of storing source locations and look like this:

```
Error message: addWall in module firstModule at
    [@location=|project://generatorv2/src/tests/test1/test.lm|
    (934,1,<68,24>,<68,25>)] destoyed path:path
```

Outlook and Ongoing Work

LudoScope Lite's new syntax, facilitated by Rascal's capabilities, paves the path to more accessible tools for content generation and transformative grammars. The addition of constraint-based checking gives the designer the control and freedom to explore and create without the risk of accidentally degrading quality or creating unplayable levels. These constraints can make modules more composable and allow for better content orchestration. Ultimately, our approach can significantly improve the state-of-the-art in error handling and debugging for grammar-based level generation.

A Framework for Educational Code Visualizations and Game Mechanics

A key challenge in programming education is providing visualizations that assist learners in rapidly obtaining feedback, exploring alternatives, learning from mistakes, improving code quality, and forming mental models. We hypothesize that suitable interactive 3D visualizations of code structures and execution can help novice programmers to form these mental models more easily.

Here, we introduce the live interactive code puzzle visualization (linco-puvi) framework[4,5], a research platform that enables experimentation with novel methods for education. The framework offers an introductory programming language for describing code exercises It furthermore features a language independent visualization format and editor, and live visualization application

Educators can use these formalisms to compose `code puzzles` as a combination of code exercise and live interactive visualization. Learners can play code puzzles as games with fixed mechanics whose educational content can be adjusted and varied for improving learning trajectories.

State-of-the-art in visual assistance

We briefly describe related work that uses visual components in aiding programming. The first related area is learn-

[4] Textual Components: https://github.com/reijne/solvey

[5] Visual Components: https://github.com/reijne/codeVisuals

ing and teaching through game play and development. Findings have shown that computer games can be used to acquire certain cognitive abilities and improve understanding in presented topics (Tang, Hanneghan, and El Rhalibi 2009).

This is supported by *mental model matching* theory, which states that humans develop mental models of games through play, which can transfer to understandings of academic contexts (Boyan, McGloin, and Wasserman 2018). Current approaches using game playing and/or designing often utilize fully fletched general programming languages, sometimes more than one per course (Leutenegger and Edgington 2007), which are not tailored for introductory programming (McIver and Conway 1996).

Another approach for visual assistance is augmenting development environments by presenting the user with a visual interface beside one for altering the program. Alice (Conway et al. 2000) and Kodu (MacLaurin 2011) offer a visual aid in 3D, and utilize a set of possible methods to change the current state of the visualization. Scratch (Maloney et al. 2010) is a visual programming language using a drag and drop system. Its set of command blocks offers control over 2D-graphical objects in a separate view.

The state-of-the-art showcases the effect of code execution by coupling the execution of commands to static and/or dynamic parts of the visualization. In contrast, our approach also enables visualizing the structure of the code itself.

The live visualization is created directly from the abstract syntax tree (AST), of the student's written code.

Divide and Conquer

The approach is subdivided into (A) the textual component, for which we have chosen the metaprogramming language workbench Rascal, which allows the creation of modularized domain specific languages, and offers tools for code extraction, analysis and transformation (Klint, van der Storm, and Vinju 2009b). These properties make for an approach that enables maintaining and evolution of the presented framework. (B) The visual component, for which we have chosen the 3D game engine Unity. This well-documented, performant, popular and free game engine allows for the creation of custom game worlds and in-game or editor-like interfaces.

Bridging the gap between Rascal and Unity

In order to accurately visualize the live state of code we need to bridge from Rascal over to Unity. This is accomplished by employing clever extraction of data through Rascal metaprogramming, which is condensed into string format. The string parameter is then used to create a JSON remote process call, using the sophisticated JSON library. This call is sent over TCP to the running application (see Figure 6).

```
1   @doc {Rascal: Create a Remote Process Call to send over the Socket,
        using JSON rpc format.}
2   str remoteCall(str method, str param, bool close) {
3     rpc = ("method": method,
4            "param": param,
5            "close": close);
6     return asJSON(rpc);
7   }
```

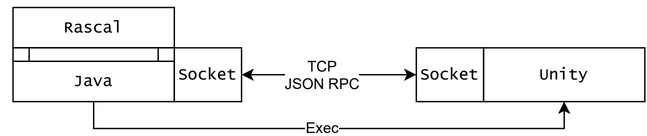


Figure 6: Architectural diagram of connected components

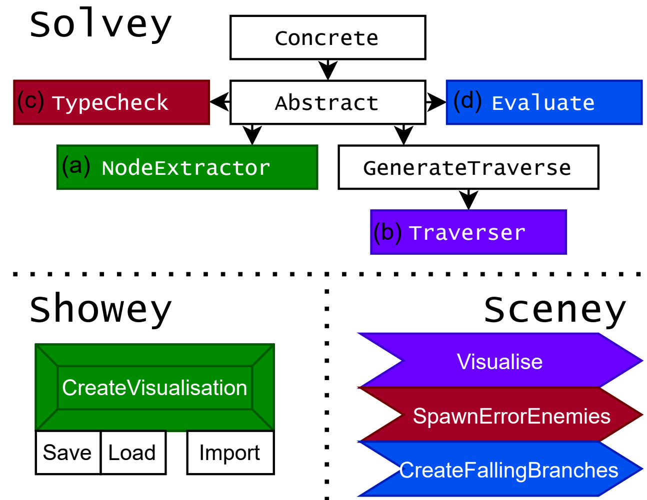


Figure 7: Modular Solvey: Introductory programming DSL, and the connections to Showey and Sceney.

Separation of Concerns

Separating the concerns into modules is where Rascal shines (see Figure 7). We have designed a modular DSL for introductory programming, with the necessary modules to allow for (a) the creation of custom visualizations. This requires the recursive extraction of the components of the language Algebraic Data Type, which is done using the visit expression[6] in Rascal, that features built in traversals of a tree.

The (b) traverser of an AST is generated using Rascal's powerful string templating[7] that allows for inclusion of statements. In the (c) Type Checking module the errors are gathered along with their (source) location[8], which in turn allows the game to be updated by spawning enemies where the errors occur. And finally the (d) Evaluation module interprets the code, and is used to gather which control flow is taken. This information lets the branches in the visualization fall when stepped on if that branch is not existent in the executed path.

Full pipeline and Interaction

The puzzle solver is presented with the two before mentioned views, namely a textual editor and an interactive

[6] https://docs.rascal-mpl.org/unstable/Rascal/#Expressions-Visit

[7] https://docs.rascal-mpl.org/unstable/Recipes/#Common-StringTemplate

[8] https://docs.rascal-mpl.org/unstable/Rascal/#Values-Location

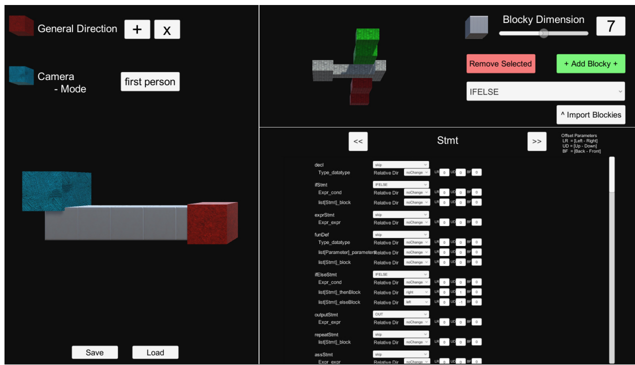


Figure 8: ShoweyBuilder, a visualization editor to create a definition how the scene should look.

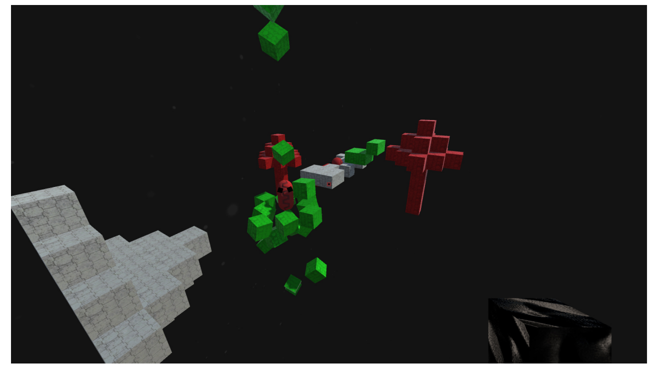


Figure 9: Sceney instance representing the code snippet from the Modulo assignment, showcasing an *Error Enemy* destroying the code and aiming at the player.

game. The Solvey language is used to code out (simple) exercises using in- and output of expressions. The following snippet is used to demonstrate a simple exercise, where the solution contains an error. A modulo operation will result in a number variable, which does not fit the conditional expression of an if statement. The solver will be presented with an error message along with a source location, while at the same time having the interactive game reflect the code by spawning the visual representation based on the AST piped through the Showey definition supplied by the puzzle creator.

```
1  // Output true if the received number is
     an even number and false otherwise.
2  number received
3
4  received := input()
5
6  if (received % 2)
7    output(true)
8  else
9    output(false)
10 end if
```

A ShoweyDefinition determines what a scene looks like. It contains general variables, a list of created 3D artifacts (blockies) and the mapping from code to blockies. This definition is created using the interface shown in Figure 8.

In this instance we have chosen to launch the shooter case study. The game, built on top of the Showey visualization, spawns *Error Enemies* at error locations. These enemies break the code and attempt to hit the player, see Figure 9.

Outlook and Ongoing Work

We have described a novel, modular, extensible research platform for static and dynamic code visualization that combines Rascal and Unity. Educators can use its notation for creating a wide variety of code exercises, study how effective adjustable visualizations are, and iteratively improve learning trajectories.

We are opening the following possible paths into the future: (a) Find effective visualizations and mechanics to aid the programmer in understanding of underlying code concepts. This would be possible by performing action research, including this framework into the curriculum of an introductory programming course. (b) Creating entire games, where the user gets to explore coding in a novel manner. And (c) enable a bi-directional transformation such that the game world can trigger source code transformations, effectively making it a 3D visual programming language.

Discussion

Here we briefly reflect on threats to validity and on how to obtain a roadmap to collaborative research and development.

Threats to Validity

Our position is based on an extensive mapping study. However, we have discussed just three examples that leverage Rascal in language prototyping. We highlighted selected challenges and design decisions for illustrating benefits. Here, we reflect on costs and threats to validity.

Like other languages, Rascal has a learning curve, and mastering its concepts can be challenging. Rascal is used in research and education at several institutions, e.g., at Centrum Wiskunde & Informatica, the University of Amsterdam, the Open University, the University of Groningen and the TU Eindhoven, to name a few.

In general, it is hard to choose the right tool for the job. When programming in Rascal, finding the right language feature can challenging, especially to novices. The built-in Tutor, provides a language reference and recipes that help choosing an approach to a particular challenge[9]. Rascal's developers also actively answer questions on stack overflow.

In addition, Rascal itself is an academic language that regularly undergoes overhauls and maintenance. Naturally, this can break features and existing programs. Spin-off SWAT.engineering uses Rascal in an industrial context.

Finally, it is not clear to what extent our experiences with Rascal hold for language workbenches in general. Comparing language workbenches has been an activity of the language workbench challenge (Erdweg et al. 2013). Comparing them on game languages could be an opportunity.

[9]The Tutor is also available online: https://docs.rascal-mpl.org

Of course, our argument can be further strengthened with additional studies and comparisons that weigh costs and benefits. Next, we discuss on how to approach this.

Challenges, Opportunities and Road Map

Here we reflect on obtaining a *road map* that brings together researchers, developers and designers in multi-disciplinary language-centric research that explores what informs the design and construction of good games.

Languages often represent months or years of research and development. Unfortunately, an analysis of success factors in revealed a *'grave yard'* of dead language prototypes (van Rozen 2021). Key success factors include: 1) multiple research angles and language reuse; 2) collaboration with industry; 3) available language prototypes; 4) Wiki pages, blogs, example materials, tutorials and workshops; 5) active online user communities; and 6) mature evaluation research, following solution proposals and validation research.

Opportunities for languages in AGD exist at the intersection of research areas (van Rozen 2021). Ontologies, typologies and game design patterns are sources on *what* games are, and how they can be understood. Content-centric challenges include: 1) integrating subject matter knowledge; 2) balancing and fine-tuning game mechanics; 3) generating varied and interesting game levels; 4) authoring realistic NPC behaviors; and 5) integrating narratives and story plot.

Finally, technical view points on how to develop languages and tools include 1) leveraging gameplay metrics and analytics to relate content, player actions and gameplay; 2) devising understandable visual notations inspired by educative languages; 3) leveraging AI, in particular general game playing, for analysis and play testing; 5) applying and reusing techniques from model-driven engineering; and 6) leveraging language work benches, as we have also argued.

Each of these areas, subjects and approaches present opportunities for future research and development.

Conclusion

In this position paper, we have proposed studying what informs the design and construction of good games by means of metaprogramming techniques and language work benches. We have argued that these tools can help language developers speed-up the construction of language prototypes, raise the quality, bridge technological spaces and combine the state-of-the-art in novel language-centric solutions. We have discussed tackling technical challenges in three ongoing master projects that leverage Rascal for rapid language prototyping. Finally, we have reflected on challenges and opportunities for embarking on first-person realtime collaborative metaprogramming adventures.

Acknowledgments

We thank the organizers for hosting the 1st PLIE workshop. In addition, we thank the anonymous reviewers for their insightful comments that helped improve our paper. Finally, we thank the Master of Software Engineering of the University of Amsterdam for supporting our participation.

References

Aho, A. V.; Sethi, R.; and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley. ISBN 0-201-10088-6.

Boyan, A.; McGloin, R.; and Wasserman, J. A. 2018. Model matching theory: A framework for examining the alignment between game mechanics and mental models. *Media and Communication* 6(2). doi:https://doi.org/10.17645/mac.v6i2.1326.

Conway, M.; Audia, S.; Burnette, T.; Cosgrove, D.; and Christiansen, K. 2000. Alice: lessons learned from building a 3D system for novices. In *Proceedings of the SIGCHI conference on Human factors in computing systems*.

Dormans, J. 2010. Adventures in Level Design: Generating Missions and Spaces for Action Adventure Games. In *Proceedings of the 2010 Workshop on Procedural Content Generation in Games, PCG2010*. ACM. doi:10.1145/1814256.1814257.

Dormans, J. 2011. Level Design as Model Transformation: A Strategy for Automated Content Generation. In *Proceedings of the 2nd Workshop on Procedural Content Generation in Games, PCG 2011*. ACM.

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). doi:10.1109/TCIAIG.2011.2149523.

Dormans, J.; and Leijnen, S. 2013. Combinatorial and Exploratory Creativity in Procedural Content Generation. In *Workshop Proceedings of the 8th International Conference on the Foundations of Digital Games*. SASDG.

Erdweg, S.; Van Der Storm, T.; Völter, M.; et al. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering*, volume 8225 of *LNCS*. Springer. doi:10.1007/978-3-319-02654-1_11.

Flatt, M. 2012. Creating Languages in Racket. *Communications of the ACM* 55(1). doi:10.1145/2063176.2063195.

Heijn, Q. 2018. *Improving the Quality of Grammars for Procedural Level Generation: A Software Evolution Perspective*. Master's thesis, University of Amsterdam.

Karavolos, D.; Bouwer, A.; and Bidarra, R. 2015. Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation. In *Proceedings of the 10th International Conference on the Foundations of Digital Games, FDG 2015*. SASDG.

Klint, P.; van der Storm, T.; and Vinju, J. J. 2009a. EASY Meta-programming with Rascal. In Fernandes, J. M.; Lämmel, R.; Visser, J.; and Saraiva, J., eds., *Generative and Transformational Techniques in Software Engineering III - International Summer School, GTTSE 2009. Revised Papers*, volume 6491 of *LNCS*. Springer. doi:10.1007/978-3-642-18023-1_6.

Klint, P.; van der Storm, T.; and Vinju, J. J. 2009b. 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. IEEE Computer Society. doi:10.1109/SCAM.2009.28.

Lämmel, R. 2018. *Software Languages: Syntax, Semantics, and Metaprogramming*. Springer. ISBN 978-3-319-90798-7. doi:10.1007/978-3-319-90800-7. URL http://www.softlang.org/book.

Leutenegger, S.; and Edgington, J. 2007. A games first approach to teaching introductory programming. In *Proceedings of the 38th SIGCSE technical symposium on Computer science education*.

Lim, C.; and Harrell, D. F. 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*. IEEE. doi:10.1109/CIG.2014.6932896.

Lindenmayer, A. 1968. Mathematical Models for Cellular Interactions in Development. *Journal of Theoretical Biology* 18(3). ISSN 0022-5193. doi:10.1016/0022-5193(68)90079-9.

MacLaurin, M. B. 2011. The design of Kodu: A tiny visual programming language for children on the Xbox 360. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.

Maloney, J.; Resnick, M.; Rusk, N.; Silverman, B.; and Eastmond, E. 2010. The scratch programming language and environment. *ACM Transactions on Computing Education (TOCE)* 10(4).

McIver, L.; and Conway, D. 1996. Seven deadly sins of introductory programming language design. In *Proceedings 1996 International Conference Software Engineering: Education and Practice*. IEEE.

Naus, N.; and Jeuring, J. 2016. Building a Generic Feedback System for Rule-Based Problems. In *International Symposium on Trends in Functional Programming*, volume 10447 of *LNCS*. Springer. doi:10.1007/978-3-030-14805-8_10.

Osborn, J. C. 2018. *Operationalizing Operational Logics*. Ph.D. thesis, UC Santa Cruz.

Osborn, J. C.; Samuel, B.; Mateas, M.; and Wardrip-Fruin, N. 2015. Playspecs: Regular Expressions for Game Play Traces. In *Proceedings of Artificial Intelligence and Interactive Digital Entertainment, AIIDE 2015*. AAAI Press.

Smith, A. M.; and Mateas, M. 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Transactions on Computational Intelligence and AI in Games* 3(3). doi:10.1109/TCIAIG.2011.2158545.

Tang, S.; Hanneghan, M.; and El Rhalibi, A. 2009. Introduction to games-based learning. In *Games-based learning advancements for multi-sensory human computer interfaces: Techniques and effective practices*. IGI Global.

Tikhonova, U.; Manders, M.; van den Brand, M.; Andova, S.; and Verhoeff, T. 2013. Applying Model Transformation and Event-B for Specifying an Industrial DSL. In *Proceedings of the 10th International Workshop on Model Driven Engineering, Verification and Validation MoDeVVa 2013*, volume 1069. CEUR-WS.org.

van den Bos, J.; and van der Storm, T. 2011. Bringing Domain-Specific Languages to Digital Forensics. In *Proceedings of the 33rd International Conference on Software Engineering, ICSE 2011*. ACM. doi:10.1145/1985793.1985887.

van Deursen, A.; Klint, P.; and Tip, F. 1993. Origin Tracking. *J. Symb. Comput.* 15(5/6): 523–545. doi:10.1016/S0747-7171(06)80004-0.

van Deursen, A.; Klint, P.; and Visser, J. 2000. Domain-Specific Languages: An Annotated Bibliography. *ACM SIGPLAN Notices* 35(6). doi:10.1145/352029.352035.

van Rozen, R. 2021. Languages of Games and Play: A Systematic Mapping Study. *ACM Computing Surveys* 53(6). doi:10.1145/3412843.

van Rozen, R.; and Heijn, Q. 2018. Measuring Quality of Grammars for Procedural Level Generation. In *Proceedings of Foundations of Digital Games, FDG 2018*. ACM. doi:10.1145/3235765.3235821.

Vermeulen, M. 2018. *Automated Game Generation met Gebruik van Meta-Programming*. Bachelor's thesis, Hogeschool van Amsterdam.

Walter, R.; and Masuch, M. 2011. How to Integrate Domain-Specific Languages into the Game Development Process. In *Proceedings of the 8th International Conference on Advances in Computer Entertainment Technology*. ACM. doi:10.1145/2071423.2071475.

Warren, J. 2019. Tiny Online Game Engines. In *Decision and Game Theory for Security - 10th International Conference, GameSec 2019*, volume 11836 of *LNCS*. Springer. doi:10.1109/GEM.2019.8901975.