Maintenance in Procedural Level Design: Lessons from Ludoscope

Daria Protsenko
University of Amsterdam
Amsterdam, Netherlands
daria.protsenko@student.auc.nl

Joris Dormans
Ludomotion
Amsterdam, Netherlands
joris@ludomotion.com

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

Abstract

Procedural level generation empowers level designers with tools for generating many levels from a single specification, while engineers maintain the level generator. Despite advances in procedural techniques, little is known about their impact on long-term system maintenance. We explore how Domain-Specific Languages (DSLs) can help improve procedural level design processes, and support maintenance by integrating level design sketches into generator-agnostic tools. This short paper examines the evolution of Ludoscope, a state-of-the-art level generator used in the games Unexplored 1 and 2. In over a decade, it has grown in complexity, with Unexplored 2's generator now containing over 20K rewrite rules.

We investigate how Ludomotion addressed maintenance challenges, and how this impacts procedural level design. Our approach combines: 1) a bottom-up analysis of Ludoscope; and 2) a top-down exploration of a generic DSL for "level blueprints". This paper contributes the first step and discusses ongoing work on a reusable framework for procedural level design. Our work takes a promising first step towards industrial-strength maintenance solutions.

CCS Concepts

• **Software and its engineering → Visual languages**; **Software maintenance tools**; • **Applied computing → Computer games**.

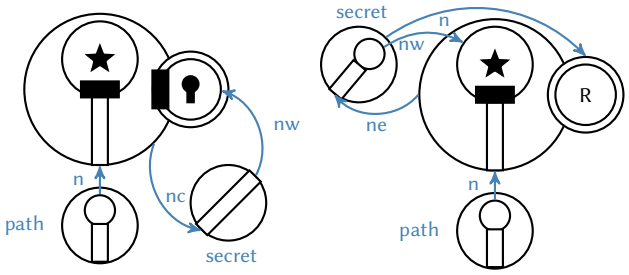
Keywords

Procedural Level Design, Sketches, Feature Modeling, Roguelikes

ACM Reference Format:
Daria Protsenko, Joris Dormans, and Riemer van Rozen. 2025. Maintenance in Procedural Level Design: Lessons from Ludoscope. In *International Conference on the Foundations of Digital Games (FDG '25), April 15–18, 2025, Graz, Austria.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3723498.3723784

1 Introduction

Procedural level generation provides tools and techniques for generating a wide variety of game levels from a single specification, leading to engaging stories, wondrous encounters, challenging quests and epic journeys [11, 17]. These tools enable level designers to specify levels independently, while software engineers focus on maintaining the generator, effectively separating concerns. Over the years, various algorithms and tools have been proposed for creating these generators, e.g., leveraging grammars (rewrite rules) [9, 22], Answer-Set Programming [15], and evolutionary algorithms [1]. These approaches have improved in usability, speed and quality [21], integrating visual notations, design patterns [2], and level sketches to improve procedural level design workflows [13]. However, due to a lack of empirical studies, little is known about the impact on long-term maintenance of level designs and generators in practice.



(a) Mine Level - Variant A **(b) Mine Level - Variant B**

Figure 1: Procedural Level Design Sketches

To shed light on this issue, we study the evolution of Ludoscope, a grammar-based level generation engine by Ludomotion based on the Missions and Spaces framework [6, 9]. Over the past decade, it has been successfully applied in creating generators for Unexplored 1 and 2, two roguelike games that center around procedurally generated dungeons, quests and adventures. During this time the code has grown significantly in size and complexity. Unexplored 2's generator, for instance, consists of over 20K rewrite rules.

We aim to understand how Ludomotion has addressed the associated maintenance challenges, how this affects the level design process, and what improvements are needed for better long-term maintenance solutions. We investigate how Domain-Specific Languages (DSLs) can help support procedural level generation and improve both level design and generator maintenance [14]. Specifically, we study how level design sketches can be integrated into generator-agnostic tools that abstract from underlying generator implementations [20]. Our approach consists of two steps.

First, we take a bottom-up approach by reverse engineering Ludoscope and Unexplored 2, extracting domain knowledge from source code to identify generator-specific features [18]. Second, we follow a top-down approach to explore how a generator-agnostic DSL for "level blueprints" can bridge the gap between level design sketches and generator implementation, proposing generative solutions that target the extracted generator features. This paper contributes the first step: an analysis on the maintenance of Ludoscope and Unexplored 2.

2 Related work

We describe related work on tools that integrate visual notations, patterns, and design sketches, and relate maintenance techniques.

FDG '25, Graz, Austria

ACM ISBN /25/04
https://doi.org/10.1145/3723498.3723784

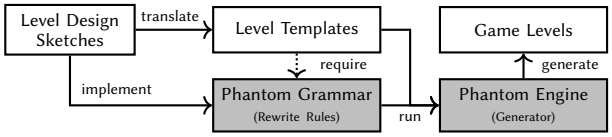


Figure 2: Ludoscope's Multi-tier Level Generation Strategy

2.1 Level Design Sketches

Level designers create high-level visual design schematics to reason about essential gameplay qualities of before the levels are generated. These designs can take the form of *sketches* that represent abstract game levels [5]. Figure 1 shows an example of a hand-crafted sketch of a Mine level for Unexplored 2, created by Ludomotion. This sketch shows two variants with distinct progressions, puzzles and secret passages. We will discuss this sketch in detail later on.

Gameplay qualities can manifest inside generated game levels in a myriad of ways. Therefore, designers need languages, tools and techniques for authoring procedural designs. Mixed-initiative and co-creative design tools address two key needs [11]. First, they automate the process of visually encoding design intent into level generators. Second, they augment manual play testing, alternating between manual and algorithmic steps. We give three examples.

Sentient Sketchbook is a tool for sketching and generating level designs as tile maps using *metrics* and genetic algorithms [13]. The tool integrates automated playtests that ensure paths are navigable meshes, and provides visual telemetry in the form of heatmaps that predict choke points and safe areas.

The Evolutionary Dungeon Designer (EDDy) is a tool that integrates patterns for visually expressing dungeon elements, such as rooms, quests and narrative structures [1, 2]. The tool provides *design patterns* that can introduce structure, and guide the evolutionary dungeon generation [2].

Ludoscope, the engine and tool we study in this paper, visually expresses level generators using rewrite rules [6, 9]. These generators step-by-step transform levels, adding details to tile maps, graphs, strings and Voronoi diagrams.

Aside from patterns and metrics, designers still lack reusable visual notations for expressing procedural level designs [20]. In this paper, we study how to integrate level design sketches into the procedural level design process. Specifically, we investigate how sketches play a crucial role for Ludoscope and Unexplored 2.

2.2 Maintaining Level Generators

Software engineers design and maintain reusable tools, engines and level generators that support the level design process and implement level designs. However, due to the large conceptual gap between level designs and implementations, creating high-quality generators is challenging. The term "procedural oatmeal" refers to the problem of generated levels being repetitive, lacking variety, and offering little replay value [4]. Additionally, the lack of reliable solutions for testing, debugging, and maintenance hinders the large-scale adoption of more advanced PCG techniques in industry.

In Software Engineering, meta-programming refers to the creation of programs that analyze and manipulate other programs. Language workbenches are tools that facilitate the creation of such programs, e.g., compilers, interpreters, and generators [7]. Applying

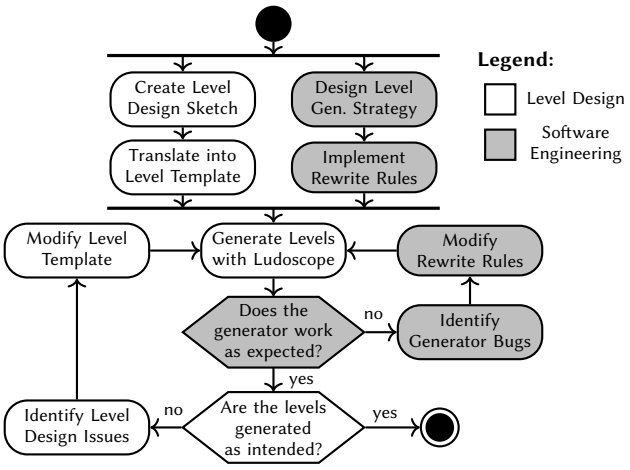


Figure 3: Procedural Level Design Process

these tools to content generation offers two major benefits. First, creating tools for source code analysis enables empirical studies on real-world generators, e.g., using software metrics. Second, developing Domain-Specific Languages (DSLs) helps bridge the gap between level design sketches and generator-agnostic solutions that are reusable across different generator back-ends. In this paper, we leverage Rascal in a reverse engineering approach [10].

DSLs for level generation are becoming increasingly common, e.g., PuzzleScript [12], Rewrite Rules [9], Behavior Trees, combinations thereof [22], to name a few. Metrics usually work on the generated levels, e.g., for expressive range analysis [16, 21], and not on the source code of the generators [19]. Empirical studies of source code repositories are still rare, with one example being the analysis of PuzzleScript [8]. Next, in our analysis of Ludoscope we describe how Ludomotion addresses maintenance challenges.

3 The Evolution of Ludoscope

Ludoscope is an authoring tool for creating level generators that evolves from the Missions and Spaces framework [6]. Over the years, Ludomotion has addressed development and maintenance challenges with a multi-tier level generation strategy called Theory of the Place [5], which is schematically shown in Figure 2. To improve the workflow, they have separated level design (white) and software engineering (gray) concerns. As a result, the team can now separately maintain level-specific designs, game-specific generator implementations (Phantom Grammar), and a fully reusable generator (Phantom Engine) for both Ludoscope and Unexplored 2.

Figure 3 shows the procedural level design process in the form of a UML activity diagram. Starting at the top (filled circle), the workflow splits activities (rounded rectangles) between level design (white) and software engineering (gray). Arrows and horizontal bars respectively indicate sequential and parallel activities, and six-sized polygons represent choice points. The process ends (circled dot) when all choice points succeed. Next, we explain this process.

3.1 Designing Level Sketches

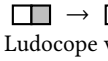
The level design process begins with creating sketches. Figure 1 shows two sketches of a simple Mine Level for Unexplored 2. The notation describes level designs in terms of places (circles), locked

```
1  encounter(name="mine", type="Destination")
2  setVariant(notVariant="ab", choose="a or b")
3  mainEncounter(chance=0.5f, encounter="RANDOM")
4  questEncounter(chance=0.1f, encounter="RANDOM")
5  addLocation(id="mainSite", direction="north"...)
6  addLocation(structure="[small,medium]|placeIsNorth"...)
7  addLocation(features="[roomIsPlace|clearing|spawnLair"...)
8  if(variantOr="a")
9     addLocation(id="secretSite", direction="northEast"...)
10    ...
11 else()
12    addLocation(id="secretSite", direction="northEast"...)
13    ...
14 endIf()
15 addConnection(variantOr="ab", start="secretSite", end="mainSite", ...
```

Figure 4: Simplified Level Template of the Mine Level

doors (closed bars), keys (key symbol), and objectives (stars), and paths that connect them (arrows and open rectangles). In variant A, the player enters a mine path and walks into the main room, but the entrance is barred. The player must find a secret room, obtain a key, and reach the level's end, marked by a star. Variant B is similar to Variant A, but with extra elements. From the main hall, a rope leads to a secret room, and players can take refuge in another room accessible from the hall. The sketches are annotated with names, relative positions and directionality, e.g., `n` and `nw` refer to north and north-west, and `secret` indicates a place needs to be discovered.

3.2 Implementing Level Generators

Developers implement the generators using Ludoscope, which expresses level transformation pipelines [9, 19]. Its notation, Phantom Grammar, groups rewrite rules into modules and provides recipes to apply them. An example tile map rewrite rule that adds a door (green) in an east wall (gray) is:  . A recipe would for instance apply this rule twice. Ludocope visualizes this process, including intermediate results. For conciseness we have to omit a detailed description here, and instead refer to Karavolos et al. [9]. Unexplored 2's generator is complex with over 20K rewrite rules.

3.3 Creating Level Templates

There is a significant conceptual gap between level design sketches and the grammar rules that implement the level generator. To bridge this gap, Ludomotion has introduced an intermediate notation called Level Templates. Level Templates raise the abstraction level, providing a declarative notation that abstracts from specific grammars and pipelines. Instead, these programs use abstract features to describe level properties. Figure 4 shows the template of the Mine Level from Figure 1. Ludomotion currently maintains Unexplored 2's level designs in the form of 84 level templates, with a combined volume of 2601 source lines of code (SLOC).

3.4 Technical Challenges

Despite their innovative solutions, Ludomotion still faces several important technical challenges. We aim to tackle these challenges with DSLs and tools that bridge the gap between level sketches, level templates, and generator implementations.

For software engineers, maintaining generators that fail to work as expected is difficult due to missing feature descriptions. Loading templates via Unexplored 2's experimental modding interface can lead to run-time errors or incomplete levels. Debugging is manual, requiring step-by-step code inspection. Maintainers need a DSL to:

- R1 Specify generator features explicitly.
- R2 Verify that the generator implements its features.

For level designers, translating level sketches into templates is labor-intensive and error-prone due to the lack of IDE support and syntax highlighting. Level templates are hard to read. Designers need a visual DSL, similar to level sketches, that enables them to:

- R3 Create level designs using places, objectives, keys, and paths.
- R4 Check if designs conform to generator implementations.
- R5 Generate level templates from level designs.
- R6 Debug designs by tracing issues back to their source code through origin tracking [20].

However, a requirement for DSL design is a detailed analysis of the application domain and a mapping of the features [18], which is currently missing. Therefore, we perform the necessary analysis.

Feature	Ct.
setLocation	876
addLocation	308
travelFortune	284
complication	267
encounter	252
...	
setVariant	29

(a) Top-level features

Feature	Ct.
features	68
direction	36
id	36
themes	33
structure	32

(b) `addLocation`

Feature	Ct.
choose	29
region	4
notVariant	4
specialRegion	3

(c) `setVariant`

Figure 5: Frequently occurring features and sub-features

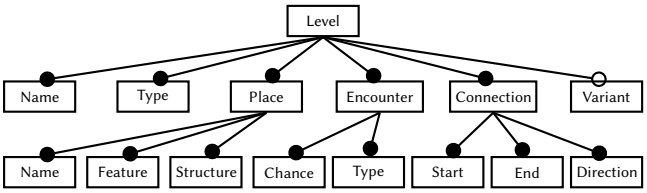


Figure 6: Partial Feature Model of the Unexplored 2 generator

4 Reverse Engineering

The Level Templates of Unexplored 2 contain a wealth of information. To understand, maintain, and evolve the system, we recover its conceptual Domain Model by identifying key entities, relationships, and constraints that define the problem space of level design. We apply a reverse engineering approach that extracts features from the Level Templates of Unexplored 2's level generator.

4.1 Setup and Results

Using Rascal, we construct an EBNF grammar, a formal description of the syntax of Level Templates. From this grammar, we obtain a parser that produces syntax trees for all 84 templates, and an IDE with syntax highlighting. We identify frequently occurring features in these trees by creating a meta-program that counts keyword occurrences. Figure 5 shows a summary of the most common ones.

4.2 Domain Analysis

Guided by this knowledge, we reverse engineer what the features mean, describing them one by one. We apply Feature Modeling, a proven technique for expressing variability of software products [3].

```
1  enum Loc {North, South, East, West}
2  enum Size {Small, Large}
3  root struct Level {
4    String name;
5    set[Place] place; ... }
6  struct Place {
7    String name;
8    Size size;
9    opt Loc location; ... }
10 ...
```

(a) Generator Features

```
1  Level {
2    name = "mine";
3    place {
4      name = "mainSite";
5      type = Site;
6      if (variant == a){ location = North; }
7      else{ location = South; }
8      ...
9    } ...
10 }
```

(b) Partial Mine Level Design

Figure 7: Mental Maps descriptions of features and levels

Figure 6 shows a partial Feature Model, a structured analysis of the capabilities of the level generator. We compare statements to establish hierarchies of mandatory (closed circle) and optional (open circle) sub-features. At its core, a level consists of one or more locations. Levels may have multiple variants, each featuring at least one location and main encounter, but differing in complications, travel fortunes, and other elements. Locations vary by theme and terrain. Some features are ranges, e.g., Location (North, South, East, West). For conciseness, we have to stop the analysis here.

5 Discussion

We describe lessons learned and discuss ongoing work on a DSL for level design that tackles the challenges of Section 3.4.

5.1 Lessons Learned

Our study on the evolution of level generators is the first of its kind, yet the lessons from Ludoscope are familiar. Over time, source code inevitably grows, making maintenance increasingly challenging. To manage this complexity, developers need better tools. Raising the level of abstraction and separating concerns improves the maintainability, but debugging remains a persistent challenge. In particular, tracking the origin of design elements through the implementation is essential for understanding and improving generator behavior.

Meta-programming is a powerful enabler for addressing these challenges. For instance, creating a grammar for Level Templates was straightforward, requiring only a few hours, a minimal investment compared to the efforts of long-term maintenance. We also discovered what is already well-known, namely, that the design of Domain-Specific Languages requires feature descriptions [18].

5.2 Towards Mental Maps

Based on our analysis Section 4, we create a preliminary language design for Mental Maps, a generator-agnostic DSL for procedural level design. Using the DSL, engineers and designers specify generator features independently from level designs, though not yet visually, addressing challenges R1 and R3. Figures 7a and 7b respectively show a partial definition of Unexplored 2's generator features, and a conformant level design of the example Mine Level.

By tracing feature definitions and uses, the IDE can instantly check if mandatory features are present and type-correct (R3). In addition, we have developed a template approach to generate Level Templates (R5) from Mental Maps, which requires mapping features to code snippets at a relatively low effort. The validation of the approach is ongoing. Initial results show it generates valid Level Templates for simple level designs. However, currently there is no solution yet for verifying the generator implements all of its features (R2). We have found that not all feature configurations result in playable levels when loading templates via the modding interface. Explicitly defining feature dependencies may be a crucial next step for making the system more robust. From these positive initial results on a real-world generator, we conclude it is feasible to further integrate sketches into procedural level design processes.

6 Conclusions

We have explored how Domain-Specific Languages (DSLs) can help improve procedural level design processes. We have investigated how Ludomotion addressed maintenance challenges, and how this impacts the level design process. We have performed a bottom-up analysis of Ludoscope, and discussed a top-down generator-agnostic DSL for "level blueprints". Based on preliminary results, we conclude that this approach is feasible. Our work takes a first step towards reusable industrial-strength maintenance solutions.

References

[1] Alberto Alvarez et al. 2018. Fostering Creativity in the Mixed-initiative Evolutionary Dungeon Designer. In *Foundations of Digital Games, FDG 2018*. ACM.

[2] Alexander Baldwin, Steve Dahlskog, José M. Font, and Johan Holmberg. 2017. Mixed-Initiative Pocedural Generation of Dungeons using Game Design Patterns. In *IEEE Conference on Computational Intelligence and Games, CIG 2017*. IEEE.

[3] David Benavides et al. 2010. Automated Analysis of Feature Models 20 Years Later: A Literature Review. *Information Systems* 35, 6 (2010).

[4] Kate Compton. 2019. Getting Started with Generators. In *Procedural Storytelling in Game Design*. AK Peters/CRC Press.

[5] Joris Dormans. 2021. The Theory of The Place: A Level Design Philosophy for Unexplored 2. *Gamasutra* (Oct. 2021).

[6] Joris Dormans and Sander Bakkes. 2011. Generating Missions and Spaces for Adaptable Play Experiences. *IEEE Trans. Comput. Intell. AI Games* 3, 3 (2011).

[7] Sebastian Erdweg et al. 2013. The State of the Art in Language Workbenches. In *Software Language Engineering, SLE 2013 (LNCS, Vol. 8225)*. Springer.

[8] Clement Julia and Riemer van Rozen. 2023. ScriptButler Serves an Empirical Study of PuzzleScript. In *Foundations of Digital Games, FDG 2023*. ACM.

[9] Daniel Karavolos, Anders Bouwer, and Rafael Bidarra. 2015. Mixed-Initiative Design of Game Levels: Integrating Mission and Space into Level Generation. In *Foundations of Digital Games, FDG 2015*. SASDG.

[10] Paul Klint, Tijs van der Storm, and Jurgen J. Vinju. 2009. RASCAL: A Domain Specific Language for Source Code Analysis and Manipulation. In *Source Code Analysis and Manipulation, SCAM 2009*. IEEE.

[11] Gorm Lai, Frederic Fol Leymarie, and William Latham. 2022. On Mixed-Initiative Content Creation for Video Games. *IEEE Transactions on Games* 14, 4 (2022).

[12] Stephen Lavelle. 2013. PuzzleScript. https://github.com/increpare/PuzzleScript Last visited January 30th 2025.

[13] Antonios Liapis, Georgios N. Yannakakis, and Julian Togelius. 2013. Sentient Sketchbook: Computer-aided game level authoring. In *Foundations of Digital Games, FDG 2013*. SASDG.

[14] Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. When and How to Develop Domain-Specific Languages. *Comput. Surveys* 37, 4 (2005).

[15] Adam M. Smith and Michael Mateas. 2011. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Trans. Comput. Intell. AI Games* 3, 3 (2011).

[16] Gillian Smith and Jim Whitehead. 2010. Analyzing the Expressive Range of a Level Generator. In *Procedural Content Generation in Games, PCG 2010*. ACM.

[17] Roland van der Linden, Ricardo Lopes, and Rafael Bidarra. 2014. Procedural Generation of Dungeons. *IEEE Trans. Comput. Intell. AI Games* 6, 1 (2014).

[18] Arie van Deursen and Paul Klint. 2002. Domain-Specific Language Design Requires Feature Descriptions. *Compututing and Information Tech.* 10, 1 (2002).

[19] Riemer van Rozen and Quinten Heijn. 2018. Measuring Quality of Grammars for Procedural Level Generation. In *Foundations of Digital Games, FDG 2018*. ACM.

[20] Riemer van Rozen, Georgia Samaritaki, and Joris Dormans. 2022. Debugging Procedural Level Designs with Mental Maps. In *Foundations of Digital Games, FDG 2022*. ACM.

[21] Oliver Withington, Michael Cook, and Laurissa Tokarchuk. 2024. On the Evaluation of Procedural Level Generation Systems. In *Foundations of Digital Games, FDG 2024*. ACM.

[22] Jiayi Zhou, Chris Martens, and Seth Cooper. 2024. Authoring Games with Tile Rewrite Rule Behavior Trees. In *Foundations of Digital Games, FDG 2024*. ACM.