Aalto University
School of Science
Degree Programme in Computer Science and Engineering

Evgenia Antonova

Applying Answer Set Programming in Game Level Design

Master's Thesis
Espoo, November 30, 2015

Supervisor:	Docent Tomi Janhunen, Aalto University
Advisor:	Emilia Oikarinen D.Sc.(Tech.), Aalto University

Aalto University
School of Science
Degree Programme in Computer Science and Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Evgenia Antonova		
Title: Applying Answer Set Programming in Game Level Design			
Date:	November 30, 2015	**Pages:**	vii + 56
Major:	Foundations of Advanced Computing	**Code:**	T-119
Supervisor:	Docent Tomi Janhunen		
Advisor:	Emilia Oikarinen D.Sc.(Tech.)		

Automated content generation is used in game industry to reduce costs and improve replayability of games. Many classic and modern games have their levels, terrains, quests, and items procedurally generated. Answer set programming is a declarative problem solving paradigm which can be used for generating game content. In answer set programming, the problem is modeled as a set of rules. This encoding is fed to the solver program, which traverses the search space and outputs answer sets complying with all the given rules. The resulting answer sets are finally interpreted as solutions to the problem.

In this thesis, answer set programming techniques are applied to design and implement a level generator for Portal, a series of 3D puzzle games where the player solves each level by using objects in a certain order. All the levels in Portal games are built by human level designers. In this work, the main aspects of Portal physics and several basic game objects are modeled using answer set programming. Two alternative encoding are provided. The full encoding of the generator tracks reachability of the objects and provides a solution plan for the level. In the simplified encoding the reachability concept and the solution plan are omitted, making the generator less reliable but more efficient at creating larger levels.

Both of the encodings are able to make different looking puzzles by utilizing randomization and controlling the search for answer sets via parameters such as the amount of elevations and objects in the level or the minimum and maximum number of timesteps for solving the puzzle. The effects of varying these and other parameters and changing the frequency for randomized choices in the solver program *clingo* are studied. One of the generated levels was reviewed and tested by several Portal players, receiving mostly positive ratings.

Keywords:	procedural content generation, answer set programming, Portal game series, game level design
Language:	English

Acknowledgments

I wish to thank my supervisor Docent Tomi Janhunen for his immense help and advice. Without the Answer Set Programming course which he lectures in the Aalto University, I would not have even gotten the idea for the project, as well as the required skills for implementing it. Tomi has helped me greatly with the code of the implementation, giving valuable comments and suggesting how to fix mistakes in it.

I am also grateful to my instructor Dr. Emilia Oikarinen for her mentoring and support. She has always examined my thesis drafts carefully and regularly gave very good and helpful advice on them. She has also introduced me to LaTeX and provided useful reference materials for the thesis background.

Both Tomi and Emilia have shown a true interest in the topic of my thesis, and this fact was very supporting and motivating for me.

I would like to express my gratitude to my former employer Data Prisma, as well as my current employer RedLynx, a Ubisoft Studio. They have kindly provided me with the possibility of balancing my work and studies. I was given the necessary time off for my studies when needed, and at the same time I had a chance to participate in interesting work projects to earn my living.

I am truly grateful to my parents, Liudmila and Aleksandr, for their unconditional love and believing in me.

I would also like to thank Perttu for all his love, support and help with the project. The discussions with him on early stages of this project have helped me to make important design decisions. He has also written a helper program for the evaluation part of this project, which has saved me a large amount of time. His moral support and patience are invaluable to me.

I am thankful to Tina, my good friend, for her never-ending encouragement and ability to make me laugh even during the most stressful stages of this project.

I wish to thank Valve Corporation for making state-of-the-art computer games and inspiring me to do this project. I would also like to express my gratitude to Sebastian Bach for his timeless music, which helped me to concentrate and focus during writing of this thesis.

Finally, I am very grateful to all of my friends and colleagues, who have supported me in any way, and just for being there for me, making my life exciting and happy.

Espoo, November 30, 2015

Evgenia Antonova

Contents

1 Introduction **1**
1.1 Structure of the Thesis 3

2 Background **4**
2.1 Procedural Content Generation 4
2.1.1 PCG Classification 5
2.1.2 Required Qualities of a PCG Solution 6
2.1.3 Hard and Soft Constraints 7
2.1.4 Common PCG Techniques and their Use 8
2.2 Answer Set Programming 10
2.3 Applications of ASP in PCG 13

3 Game Description and Formalization **16**
3.1 Game Setting 16
3.2 Level Editor 19
3.3 Properties of the Generator 20
3.4 Required Qualities of the Generator 21
3.5 Formalization 22
3.5.1 Simplifying Assumptions 23
3.5.2 Identifying Constraints 24

4 Implementation **26**
4.1 Domain Representation 26
4.1.1 Room Model 26
4.1.2 Area Separation 28
4.1.3 Portals 31
4.1.4 Object Placement 33
4.2 Satisfying Constraints 35
4.2.1 Solution Plan 36
4.2.2 Goal Statement 37
4.3 Optimizing the Encoding 37

4.4 Applying Randomization 38

5 Evaluation **40**
5.1 Properties of the Resulting Solution 40
5.2 Scalability 42
5.3 Randomization Effects 44
5.4 Standard Generated Level 46
5.5 Future Development 48

6 Conclusions **51**

Bibliography **53**

List of Tables

5.1 Generation time for the complete generator encoding for different room dimensions. 43
5.2 Generation time for the simplified generator encoding for different room dimensions. 44
5.3 Average time of finding one model for different frequencies for randomized choices. 46

List of Figures

2.1 Classification of content generator properties in relation to constraints [28, Figure 1]. 7
2.2 Declarative problem solving approach. 13

3.1 Portal placement; view of the player before entering the portal end A. 17
3.2 Portal placement; view of the player after exiting the portal end B. 18
3.3 Puzzle Maker interface. 20
3.4 The approach for using ASP to produce game levels for Portal game. 23

4.1 Possibility of moving around the same floor area with normal steps. 29
4.2 Level separation into several floor areas. 30
4.3 Possibility of exiting from the player's location through a portal. 32
4.4 Possibility of entering a location through a portal. 33

5.1 Two levels with reduced proportion of solid boxes generated by the simplified encoding. 45
5.2 First level generated from a non-randomized run of the simplified encoding, view from the top and bottom. 46
5.3 Two levels generated from randomized runs of the simplified encoding with different seeds. 47
5.4 One of the levels generated by the simplified encoding with a random seed, view from different angles. 48
5.5 The effect of varying the frequencies for randomized choices on the generation time for different room sizes. 49
5.6 Level published in the workshop (generated by the simplified encoding with a random seed), view from different angles. . . 50

5.7 Players' comments and ratings of the generated Portal map (screenshot of the item page in the workshop). 50

Chapter 1

Introduction

Game industry is developing rapidly. As in other industries, technological advancement provides tools for automating various tasks and stages in the game development process. Many parts that have traditionally required human involvement can nowadays be produced automatically, even the actual content of the game. This concept is called *Procedural Content Generation* (PCG) [30]; it describes different ways and methods for automated game content creation. PCG can either be used completely independently from human level designers or simply as an aid for them to produce more diverse and reliable content. The type of generated content can vary from maps, levels, and items to quests, rules, and game stories. Some of the most common techniques include search-based approach [30], constructive generation [26], genetic algorithms [12], grammars and L-systems [9].

There are many challenges in designing a high quality game content generator. Picking a suitable method for the needed type of content is one challenge, customizing and implementing the method properly with respect to as many constraints as possible is another. The most important set of properties are called *hard constraints* and they must be satisfied above all. Generator output cannot be random as it must be *valid* for the game, *consistent*, and *playable.* For example a map should have an entrance and an exit and an item should be of valid size. Furthermore, the generated content must not prevent the player from winning the game. It should also be fair in providing similar conditions for the opponents (when applicable).

While hard constraint satisfaction implies solving the problem at hand, there exists a whole set of other properties, called *soft constraints*, which corresponds to optimizing the generator. Some of such properties are closely related to human perception of the game and therefore not trivial to assess. An ideal generator would create content which is challenging enough as well as interesting and fun for human players. It is difficult to evaluate these properties automatically and in addition they can be highly subjective and vary from one player to another. Nevertheless, it is desirable that the generator produces a high quality content rather than just any content.

Other challenges are not related to the output of the generator, but to its operation. The generator should be fast, reliable, easy to maintain and manipulate, and produce diverse content. In most cases all of the above mentioned qualities cannot be achieved simultaneously. They need to be prioritized for the specific game and content type. An appropriate balance among these qualities should be found and maintained.

Answer Set Programming (ASP) [7, 18] is a declarative programming paradigm, commonly used to search a state space to find an answer for some problem, for example, to solve a puzzle. The problem at hand is modeled using rules and the representation is then fed to the solver program. The corresponding state space is explored by the solver and the answer sets which comply with the rules of the representation, if any exist, are returned as output. Finally, the answer sets are translated to solutions to the original problem.

Recently ASP has become one of the alternative approaches to PCG, especially found suitable for maze and level generation. Thanks to the nature of ASP, there is no need to test the output content for the hard constraints. If the problem is modeled correctly, the soundness of ASP will ensure that the found answer is always valid. There are also ways to randomize the order of the answer sets returned from the solver. This makes the output diverse and potentially satisfies some of the soft constraints.

The goal of this work is to plan and implement a level generator for a fairly complex 3D puzzle game called *Portal*, published by Valve Corporation [31]. The main idea of the game is to solve logical puzzles using objects available in the level. Each Portal level is represented as a room, which has a unique combination of size, structure, number and variety of objects in it. The level structure along with the placement of the objects is the subject for automatic generation.

One of the biggest challenges in modeling Portal game level is the fact that the content itself makes up most of the gameplay. A declarative problem solving approach is used for this purpose. Game logic and some of the essential physical properties of the game are modeled and represented as an ASP program. Hard constraints are encoded as ASP rules. Soft constraints, optimization, and randomization issues are addressed as well. The qualities of the generator, such as speed, maintainability and diversity of the output are taken into account during the planning and implementation phases, as well as in the final evaluation of the generator. The resulting ASP encoding is fed to the solver and the generated answer sets are then translated to ready-to-play game maps with a custom Java program.

1.1 Structure of the Thesis

The background for ASP and PCG is given in Chapter 2, along with a review on how these two fields intersect. Chapter 3 describes the Portal game series in details, focusing on the aspects of the game which are essential for modeling. The encoding itself is presented in detail in Chapter 4. In Chapter 5, the current implementation of the generator is then evaluated from various perspectives, for instance, how efficient, complex, and scalable it is. In addition, reviews from community are presented and suggestions for the future development are discussed. Finally, the whole project along with its results is summed up and reviewed in the last chapter.

Chapter 2

Background

This chapter contains the background material necessary for understanding of the implementation and the ideas behind it. First, the meaning of procedural content generation (PCG) and its classification are reviewed in Section 2.1. Then, the basic terms and concepts of answer set programming (ASP) are explained and accompanied by examples in Section 2.2. Finally, Section 2.3 explores how ASP can be applied for content generation and reviews games which are using these techniques.

2.1 Procedural Content Generation

Procedural content generation (PCG) refers to a set of algorithms and techniques which create game content. Most of the time PCG is used in digital games, but it also can be applied to real life games, for example board games. PCG can be completely automated for some types of games and content. However, quite often it is used in parallel with manual game content design, by aiding and leading human designers.

Some authors also distinguish the concept of content from the other parts of the game, which could also be generated automatically [29]. Such game components as story management or behavior of a *non-player character* (NPC) would refer to computational or artificial intelligence (AI) rather than content generation. These algorithms are sometimes more complex and require the AI learning how to play the game. A good example of such technologies can be found in The Elder Scrolls V: Skyrim game published by Bethesda Softworks in 2011. The behavior of all NPCs in it is driven by Radiant AI, whereas new personalized quests can be considered to be game content generated by Radiant Story [4].

One of the obvious advantages of PCG is the automation of creating content for games and the reduction of the costs of game development. In many cases PCG greatly increases replayability of a game, as PCG can often produce nearly an infinite amount of levels, maps or surroundings. In some cases PCG even allows new types of games to appear, for example in games with personalized level generation [33]. In this type of games new levels are created procedurally depending on player's actions and his or her previous gameplay tactics. Another important reason why some games rely on automated content generation instead of providing it pre-rendered is memory saving. It is crucial for many games to consume as little hardware memory as possible, therefore the content is produced only at the point of the game when it is needed. Finally, PCG can result in generation of content which will not usually be obvious and "traditional" for a human level designer. Humans often create something similar to what they have done before, following their own style, taste, and preferences. PCG can help with this restriction as it is able to explore the design space more thoroughly and result in more diverse content.

2.1.1 PCG Classification

Togelius et al. [29] use the following classification to describe various properties and features of content generators.

Online or offline Some PCG methods are able to generate content dynamically during the gameplay, while other methods are used during development stages (before shipping off the product) or before the beginning a game session.

Necessary or optional Necessary content is crucial for the completion of the game or a game level, and it always has to be valid. Optional content generator does not need to comply to this rule, and the content created by it is more for decorative and game-experience complimenting purposes.

Generic or adaptive Adaptive generators analyze the player's behavior and preferences and produce the content accordingly, whereas generic PCG do not take this information into account.

Stochastic or deterministic Re-using stochastic PCG will not usually result in unique content, while deterministic generators recreate the same output under the same conditions.

Constructive or generate-and-test These properties are related to the actual algorithm and the way it evaluates constraint satisfaction (more details about constraints are given in Section 2.1.3). Constructive algorithms make sure that no action or function in it leads to invalid content. The output of a constructive algorithm does not need to be tested for compliance with the hard constraints. On the other hand, so-called generate-and-test algorithms first make a candidate solution and evaluate if it fits the necessary restrictions. If it breaks some hard constraints or has a low score on soft constraints then it is discarded and search for a better candidate continues.

Automatic or partial Automatic generators are able to produce content without any human interference, as opposed to systems which are partially dependent on human (usually player's or level designer's) input.

The choices among these properties for a content generator greatly depend on the game itself and the content type to be created.

2.1.2 Required Qualities of a PCG Solution

Togelius et al. [29] also highlight properties a high-quality PCG solution should have.

Speed Efficient generator should have a relatively high speed. The desired speed needs to be estimated individually depending on the type of content and the properties of the generator.

Reliability The generator should be reliable and produce a valid output. This quality is related to one of the most important hard constraints.

Controllability There should be some way to control and manipulate the generation (for example, by providing parameters to the generator). Often a high degree of control is preferable to produce more predictable results.

Diversity The generated content needs to be diverse, meaning that output sets should be relatively different from each other. Various randomization techniques can be used for this, depending on the selected algorithm.

Creativity Usually the content should be creative enough to look like it was made by a human level designer. This can be one of the hardest qualities to achieve for certain content types.

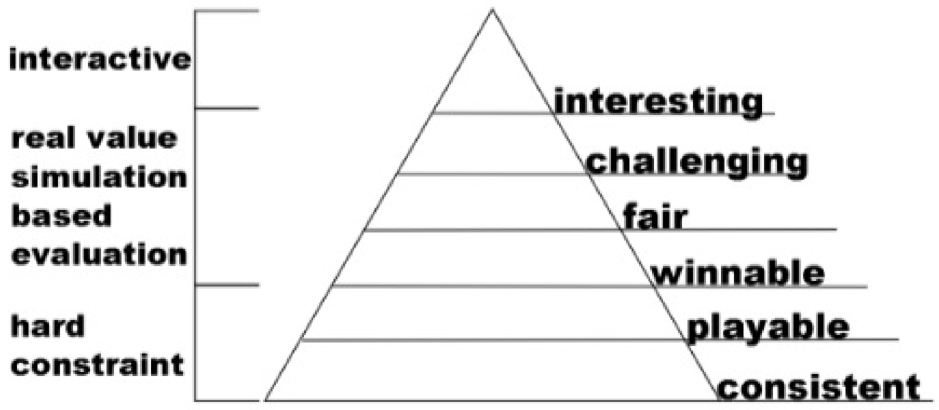


Figure 2.1: Classification of content generator properties in relation to constraints [28, Figure 1].

Often it is impossible to achieve good results in all of these qualities at the same time. Then it is important to prioritize them and balance depending on desired generator properties. For example, a fully automatic online generator for producing necessary content must be fast and reliable. Players will not be satisfied if the content takes too long to generate and to load, or if they end up receiving broken content. In this case diversity and creativity of the content is less essential than satisfying those mandatory properties. On the other hand, for the level designers using a helper tool to create offline partial content, its speed and reliability are not as essential. The designers can manually check the generated content afterwards and edit it to his or her own liking (as the produced content is meant to be partial). However in this case controllability, diversity and creativity of the generator are the top priorities, whereas sacrificing speed and reliability is acceptable.

2.1.3 Hard and Soft Constraints

Content generator properties can also be classified with the respect to constraint types (presented in Figure 1 in the article by Togelius et al. [28] and recreated here in Figure 2.1). The lower the property is in the pyramid, the easier it should be to automatically satisfy it. Hard constraints must be satisfied so that the generated content would make sense. First of all, it should be *consistent*, meaning that there are no contradictions in it (i.e., all objects must be placed inside the map, and all objects would have certain size and dimensions). It also must be playable in general. For example the starting location should exist and the player should be able to move in the map-type content. Hard constraints are easily checked and satisfied in most of the PCG techniques.

The next set of properties is important for ensuring the quality of the content generator. These properties usually need to be assessed during some time limit. It is essential that there is at least one scenario for the player to win the game and that this situation can be reached at some point in time. This condition makes the game not just playable, but winnable. The generated content also needs to contribute to the fairness of the game, providing similar conditions for the opponents. Another relevant property — how challenging the game becomes when it is using generated content — can be measured and controlled in some PCG techniques, while it is more difficult to evaluate in others.

Finally, the most difficult property to rate is how interesting the content is, as it is hard to find an appropriate algorithmic estimation for it. In most cases human interaction is needed to evaluate this criterion.

2.1.4 Common PCG Techniques and their Use

A typical use of PCG is reducing the amount of manual work needed for creating large amounts of content as well as reducing the amount of memory taken up by the content. In the 1980's and early 1990's when game development teams and budgets were quite small, games that had large amounts of content often made use of simple Pseudo-random number generation based PCG in the development phase because of the time and the costs involved in creating the content manually. Since then game development team sizes and budgets have ballooned, with manual content creation accounting for a large proportion of the increase. An average game today is longer and has significantly more content than one 20 years ago. A modern top role-playing game with an open world likely has several novels worth of writing, several movies worth of voice and motion capture acting, a couple of albums worth of music, and a massive amount of visual art assets.

The players' expectations of the content quality has also increased, which is possibly why the use of simple PCG methods in deterministic content creation has not been quite so common in recent years. Better, more advanced content generation techniques are considered an important part of game development because of their potential to reduce the cost and time of content creation.

In some game types PCG plays a central part and becomes one of the properties by which these game types are defined. The most noticeable example of these are *roguelike* games, named after the classic game called *Rogue* (1980 by M. Toy, G. Wichman, K. Arnold) [32]. This is a type of role-playing game, the main features of which are procedurally generated levels, turn-based combat system and a so-called permanent death (the player is forced to start a new game once the character dies). Level generation has been used since the first games of this type, and it is the most common type of game with generated content. Traditionally, pseudo-random number generators are used to place the content randomly within design space of a roguelike game, whereas details of the landscape are usually produced by fractal algorithms [26]. The properties of the content can depend on many factors, like player's character level and game difficulty.

Simple procedural techniques are also used in a popular role-playing shooter game *Borderlands* (2009 by Gearbox Interactive). Many attributes of the weapons in it (for example, range, ammunition type, and special effects) are generated randomly, without the use of any complex algorithm.

A popular game series *Civilization* (1991-2014[1]) has been using procedurally generated maps since the release of the first game in 1991 until the recent 2014 edition. The outlook and the properties of the maps can be controlled by the player with a set of parameters before the start of each game session. The algorithms have been improved since the first version of the game to support an increasing level of graphical complexity. Some fully automated methods were supplemented with artist-controlled randomization to make each map feel more natural and unique [19].

A part of a famous *Diablo* (1996–2012 by Blizzard Entertainment) series, Diablo III is an action role-playing game, where players explore dungeons, kill monsters, and collect loot [5]. The game employs randomized content in a number of different ways. There are three basic levels or randomization. Some area are entirely static, such as towns, very small dungeons, and plot-related areas. Overworld consists of partially randomized outdoors areas, where the general shape of level, entrances, exits and some important landmarks are static, but most of the space in the level is made of randomly generated tiles with varying content to encourage exploration. Finally, the dungeons are almost entirely randomized by the size, the shape, and the layout of the rooms and encounters. In addition to the level geometry, most of the enemy encounters and the loot gained from monsters and found as treasure are strongly randomized.

Epistory (2015 by Fishing Cactus) is a puzzle/adventure game that makes use of randomization as time saving measure in game development [3]. Unlike

[1]Games belonging to *Civilization* series were developed in different years by MicroProse, Avalon Hill, Activision, and Firaxis Games.

in Diablo, the randomness in Epistory is not intended to affect the gameplay, as all the puzzles, the geometry of the game areas, and the paths through them are created by hand. Instead, random generation is used to quickly create visual variability to the game within the constraints of the fixed game world. Creating variability in this way makes the game visually more interesting without affecting the gameplay. Furthermore, the randomness only exists in the game development phase, for the player, the game appears effectively fixed.

Qake (2015 by H. Dietz and J. Zeiser) is an old-school style arcade game where the player moves their token around the play area to partition off as large of an area as possible while avoiding obstacles such as bombs or balls that are moving around [11]. The randomness in Qake is used in determining the type and number of obstacles present in the play area. The goal is to keep the difficulty of the game increasing in reliable way, while promoting replayability. The game does this by creating a supply value that increases exponentially as the player progresses through the game. Each type of obstacle in the game is assigned a supply cost, with more difficult to deal with obstacles having a higher cost. For each level, an assortment of obstacles is randomly selected so that their combined supply costs equal the current supply value. Using randomness in this way supports replayability while maintaining game balance.

In texture generation PCG can be used to create textures using perlin-noise and other similar noise-generated techniques. Noise-generated textures are suitable for mapping on complex objects and provide a decent visual quality for objects that are not the main focus of the viewer. It is also possible to use PCG image filtering techniques to create patterns on textures or to sharpen and smooth them to imitate the level of detail of a higher resolution texture without the corresponding increase in memory usage.

In sound, PCG is rarely used for creating new content from scratch. It is however commonly employed to dynamically alter sounds to account for changes in the game environment, such as changing the direction and distance of sounds, playing sounds underwater or melding different music tracks together based on some game events.

2.2 Answer Set Programming

Answer set programming (ASP) is a *declarative* problem solving paradigm, which has roots in logic programming [21]. In traditional procedural programming the control flow of the program is provided by the programmer, who instructs the computer *how* to solve the problem. Declarative programming focuses instead on representing the problem domain and the expected result, describing *what* needs to be solved. Precise execution steps are not defined and the computation is controlled by the algorithm implementing the traversal the search space.

ASP is based on the syntax of Prolog, which is the most known logic programming language [27]. A typical ASP program consists of a set of *rules*. They are comprised of statements, called *atoms*, that can be true of false. For instance, the following expression is a *normal* rule, where exit_reachable and exit_closed are individual atoms:

```
goal :- exit_reachable, not exit_closed.
```

Full stop means the end of the rule. An atom to the left of ":-" is called the *head* of the rule. The *body* of the rule consists of all statements to the right of ":-". When all these statements are true, the head of the rule becomes true as well. The rule above states that the end of the game is reached (goal is achieved), if the exit is reachable and it is not closed. There could be another rule with the same head, but a different body, giving alternative definition for the head atom. It is enough for just one of the bodies to hold to make the head atom true for the whole model.

The *not* operator in ASP is different from the classical negation. In classical logic, negation means explicit falsity whereas in ASP negation is rather a reference to the absence of information, i.e., *failure* to derive it. In the previous example, if there are no rules deriving exit_closed in the program, the atom will be considered false.

A rule without a body is called a *fact*, indicating that the head atom of such rule is always true. In a content generator facts can be used to describe the desired properties of the output solution, like constant map dimensions, fixed object locations, or starting conditions. For instance, the following fact uses a *constant* exit and states that it is an object using an unary *predicate* object/1:

```
object(exit).
```

The head of a *choice rule* consists of several atoms, listed in the brackets. They can take true or false values, if the body of this rule is satisfied. Some rules use *variables*, which act as short-hand notations and start with capital letters (as opposed to small letters for constants). For example, the choice rule below has an empty body and is trivially satisfied, stating that there may be a solid box at a certain (X,Y,Z) location represented with the predicate solid_box/3. Thus, the room is randomly filled with solid boxes, which are freely picked from the valid X, Y, and Z coordinates.

```
{solid_box(X,Y,Z):coord_x(X):coord_y(Y):coord_z(Z)}.
```

An *integrity constraint* looks like a normal rule but without a head and it gives a set of conditions that must never be true at the same time. The first integrity constraint in the following example makes sure that if an object is placed at location (X,Y,Z), that location has a floor. The second integrity constraint simply states that the goal atom must be true. In combination with the previously presented rule with the head atom goal it ensures that the exit is reachable and not closed.

```
:- place_object(O,X,Y,Z), not floor(X,Y,Z), object(O).
:- not goal.
```

While ASP and Prolog have similar syntax, they are used for different purposes. Prolog is targeted at finding proofs, whereas ASP is used for finding models. If the problem has a solution, stable models (answer sets) are found. They do not contradict any of the given rules and they can be interpreted as the solutions to the given problem. Being more intuitive than Prolog, ASP does not presume as much knowledge in logic from the user. The order of the rules in an ASP program does not affect its execution. This is another distinction from Prolog and also from procedural programming, where the execution of the code is highly dependent on the order of the statements.

Figure 2.2 illustrates how ASP can be used for declarative problem solving [13]. At first the problem at hand is modeled as a set of rules. This formal representation of the problem is then processed by the *grounder* and the *solver* programs. The main purpose of the grounder is to transform a logic program with variables into an equivalent variable-free program. For example, let us consider the following rules:

```
object(e).
object(x).
place_object(N) :- object(N).
```

Now, the grounder removes variable N in the above rules by substituting it with constants e and x:

```
object(e).
object(x).
place_object(e) :- object(e).
place_object(x) :- object(x).
```

Gringo is one of the commonly used grounders today [10, 15]. After an ASP program is grounded it no longer contains any variables. The grounded

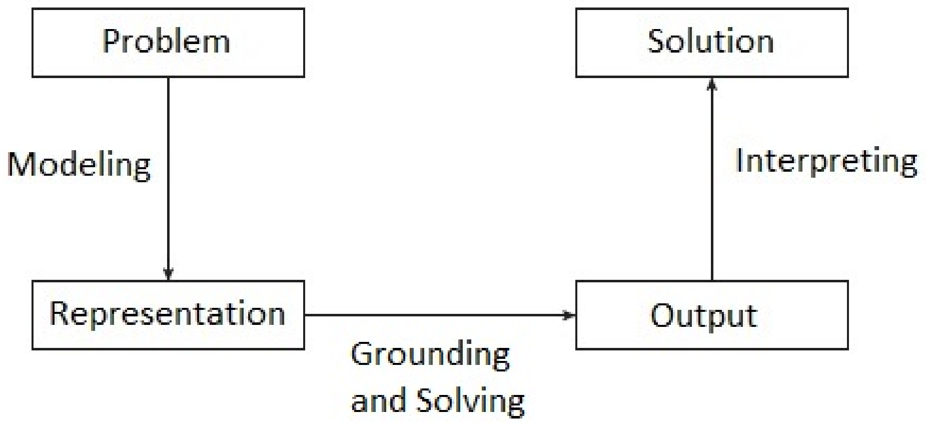


Figure 2.2: Declarative problem solving approach.

program can be fed to a solver, such as *clasp* [16]. The solver explores the corresponding state space and searches for an answer that would satisfy all the requirements of the logical representation. The produced answer sets are finally interpreted as solutions to the original problem, if any are found. The grounder and the solver can be integrated into one system. *Clingo* is one of such systems, it combines *gringo* and *clasp* and makes grounding and solving more controllable [14, 20].

The output of the solver does not depend on the choice of the tools, as they all produce the same answer sets from the grounded program. These tools can simply be viewed as "black boxes", because their internal algorithms are usually not of the interest to the programmer.

2.3 Applications of ASP in PCG

It is very common to use ASP to find an answer to a search problem, for example to solve a puzzle or make a plan of some actions. However it is also possible to take advantage of generative potential of ASP and use it to produce an arbitrary sample of an object, which satisfies given rules, like in music composition [6]. Content generation for games is another large area, where ASP could be used extensively and has already been used to some degree.

Researches have been evaluating and trying out ASP for producing various types of content, which are typically produced by other PCG techniques. For example a 2D puzzle game, called "chromatic maze", can be generated with the means of genetic algorithms and optimized with dynamic programming [2]. It is also possible to recreate the same puzzle using ASP methods [25], suggesting that ASP can be suitable for producing this type of content in a fast and reliable way.

Warzone 2100 is a real-time strategy game, where two players fight over the control of resources which are represented by oil wells. *Diorama* is a third party map generator for the Warzone 2100 and it deploys ASP techniques in two ways [25]. First, Diorama uses ASP to produce playable maps for the game where the player's base locations and the oil wells are placed in a balanced fashion. At the same time a complex terrain is generated, where movement blocking height differences are used to promote interesting and varied play. The game also features a possibility to start with pre-built bases, which Diorama handles by a second ASP process that determines the placements of the buildings within the base locations. Diorama also uses non-ASP processes to improve the aesthetics of the final map. Backtracking is not used and the final testing procedure is simply an inspection of the map by a human.

Roguelike games can also benefit from using ASP for level generation. In [26] various techniques were used in an implementation of a map generator for a roguelike game and evaluated in terms of scalability. The most efficient method included defining a distinction between the level content and the structure. Here structure denotes the basic environment of the level — components which cannot be moved, like walls and floor. The various artifacts comprise the content of the level. The objects forming the content are usually meant to directly interact with the player's character and they are placed in the structural environment of the level. The method itself consists of dividing the design space into several equal areas and therefore splitting the PCG process into two phases. During the first phase the logical content for each area is determined, whereas in the second phase the details of content placement are resolved. The phases are further divided into two stages, making sure that at first the properties for the structural objects are generated, and only then the artifacts are handled.

The educational puzzle game *Refraction* uses ASP based PCG to generate its puzzles [24]. The player's goal in the game is to direct a laser beam to a target at the desired power. This is done by modifying the power level and the direction using a limited number of tool objects. Since the tool objects modify the beams power by simple fractions (i.e. 1/2, 1/3), the created solutions can be expressed using a mathematical formula. The developers of the game used ASP both to generate entire puzzles and their solutions, and to discover alternate solutions to already existing puzzles. The algorithm takes the available numbers and types of tool objects as input, which allows the created puzzles to be placed in a pre-determined difficulty and complexity progression. For example, a puzzle which introduces the use of a new type of tool object may only give the player one such object to use, and require that it is actually used in any possible solution.

Not all content types and game properties can be easily produced with ASP techniques. ASP aims at solving problems, for which it is easy and quick to verify the solution (they are called NP-complete problems). Declarative approach seems to be not applicable for creating interactive content, which is produced on the fly and can handle user interactions. However, in combination with other techniques from the areas related to it, ASP can form a framework capable of implementing fully functional interactive games, such as Tetris [23]. The resulting application can handle and respond properly to user actions, like stopping and resuming the game, changing the user name, and persistently storing highscores. On the other hand, the game is relatively slow (2 frames per second), not easy to debug, and not possible to implement with pure ASP.

Chapter 3

Game Description and Formalization

Sections 3.1 of this chapter describes Portal game series, giving some basic information about it, as well as focusing on the most important properties for modeling this game. Sections 3.3 and 3.4 review and classify the required properties and qualities of the generator. The latter are also prioritized, pinpointing the most important qualities for the encoding. Hard and soft constraints for the game setting are identified and simplifying assumptions are stated in Section 3.5.

3.1 Game Setting

Portal is a series of games which was developed by Valve Corporation in 2007 (Portal) and 2011 (Portal 2) [31]. These games are 3D first-player puzzles consisting of several levels of increasing difficulty. However, each of these games has a unique plot and memorable characters which contribute to the gaming experience. The game physics, rules, and level completion methods are similar for both of them, although Portal 2 introduces new objects and game modes.

Each game level is represented by a so-called "test chamber" — a room with various objects, which are to be used by the player to reach the exit and continue to the next level. Portal 2 additionally has a co-operative game mode, which requires two players to complete the levels together.

The events of both Portal games occur in a fictional "Aperture Science Enrichment Center", a research facility full of laboratories, offices, and test chambers. It is operated by a corrupted AI, called GLaDOS (Genetic Lifeform and Disk Operating System), who has killed the staff of Aperture Sci-

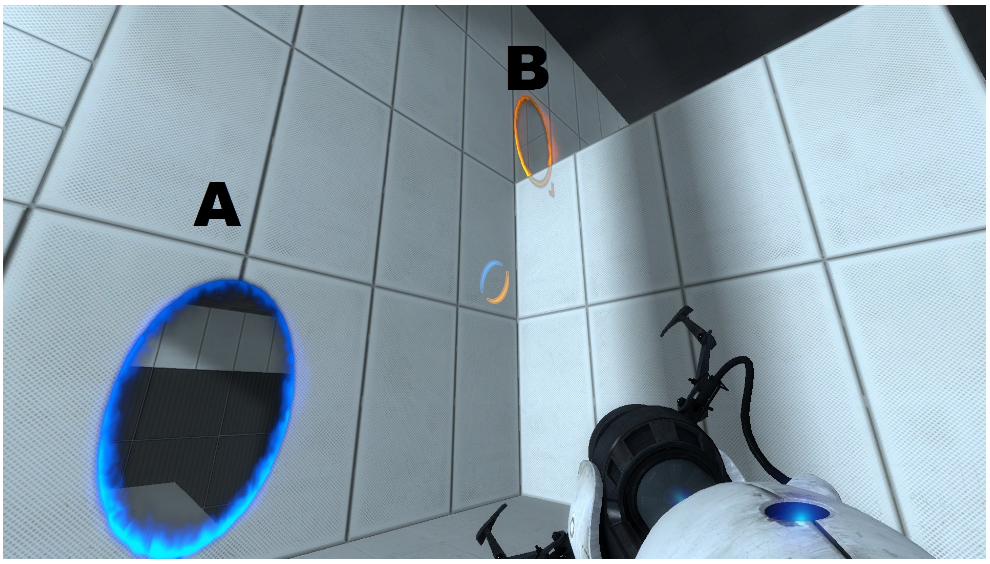


Figure 3.1: Portal placement; view of the player before entering the portal end A.

ence and who is trying to lead the player character, Chell, to her death. Main functions of GLaDOS are to maintain the facility and conduct tests, guiding the subjects through a set of test chambers.

The GLaDOS character is highly rated by both gamers and critics; it has also appeared in many top-lists of best video-game characters, for example in the *Guardian* [8]. GLaDOS is constantly commenting on the player's actions and decisions, often in a humorous and sarcastic way, becoming an essential part of the player's experience of the game series.

Portal 2 brings a new co-operative game mode, which consists of many levels to be completed by two players. Several new objects appear in both game modes in Portal 2, making levels more varied and distinct and adding unique physical laws to the game. It also introduces several new characters (two playable for the co-operative mode and one new non-playable character), but their description is omitted as they are not essential for this project.

The player can move (walk) in any direction on a horizontal plane (left, right, forward, backward, and diagonally) inside the test chamber. The player also has an ability to jump down from any height, as there is no fall damage in the game.

One unique feature of the game is the ability to use portals to move to different locations in the room and/or access objects with their help. Figure 3.1 shows an example of the situation, where one end of a portal can

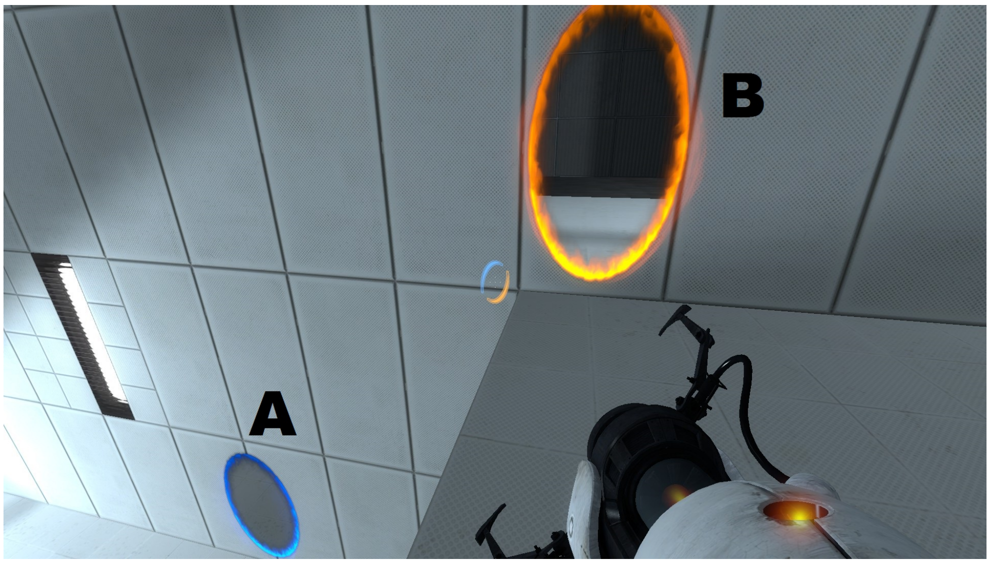


Figure 3.2: Portal placement; view of the player after exiting the portal end B.

be placed by the player on the wall next to her (marked as portal A); and another end on the visible part of the wall of some unaccessible location (portal B). The player can then step inside the portal end A and come out from the portal end B, finding herself in the previously unaccessible location (Figure 3.2). Both of these screenshots were taken during gameplay in a custom built map.

Portal ends can be placed on any surface (wall, ceiling, floor), but only on the portable white tiles. There are also tiles which are non-portable, meaning that it is not possible to place a portal on them; they are identified with gray color. The proportion of portable tiles varies from one level to another. In some cases the level designers wish to make it harder for the player to find the exact tile for placing a portal end (for moving to the next location), making many surrounding tiles around it portable as well. In other cases they might want to highlight the correct location of portal placement by making it the only portable white tile surrounded with many non-portable gray ones.

Each level starts with the player entering the test chamber through the entrance door. The goal of each level is to reach and go through the exit door. To do that, there are usually several objects which the player needs to interact with in order to reach and/or open the exit door. There is no time limit in the game and player can do as many steps as they want to solve the puzzle.

Most commonly used objects are buttons and cubes. There are a lot of other objects with special properties, but for simplicity only these two types of non-obligatory objects will be used in this project. Cube dispensers are in-game objects which produce cubes. There are several kinds of cubes, and two types of cube dispensers for each of them. Some dispensers release as many cubes as the player wants to have, and others just one cube per game session. Button is another very common Portal game object. There are several types of them (for example, button with a timer) but only the basic one is used for this project. Buttons can be connected to other objects, like the exit door or a cube dispenser. Some buttons need to be pressed only once in order to open door/release a cube. Other buttons can have a different property and require an object, usually a cube, to be holding them down, for example, for the exit door to remain open.

The complexity of the test chamber depends on several factors. The most obvious ones are the size of the room and the number/variety of objects needed to solve the puzzle. However, a clever distribution of leveled surfaces and portable/non-portable tiles can result in very complex and interesting game levels as well.

3.2 Level Editor

For developers and level designers, Valve published a so-called Hammer Editor, quite complex and detailed software for designing test chambers (game levels) for the Portal games. After the release of Portal 2 another tool was developed, called Puzzle maker. It is much simpler and easier to use than the Hammer Editor; even people without any technical skills are able to design their own test chambers using the drag-and-drop interface (Figure 3.3). Many new levels created in Puzzle Maker are still shared among players every day; summing up to more than 400,000 custom test chambers.

Recently, Puzzle Maker has also been used in schools by geometry and physics teachers. Valve is supporting the "Teach with Portals" initiative, comprising of pages and materials where teachers can get ideas of how to use the Portal games in the education process and even find concrete lesson plans made by other teachers. For example, there is a lesson plan of building a simple harmonic oscillator [22] to help students learn about periodicity, friction, and relationships between various physical qualities.

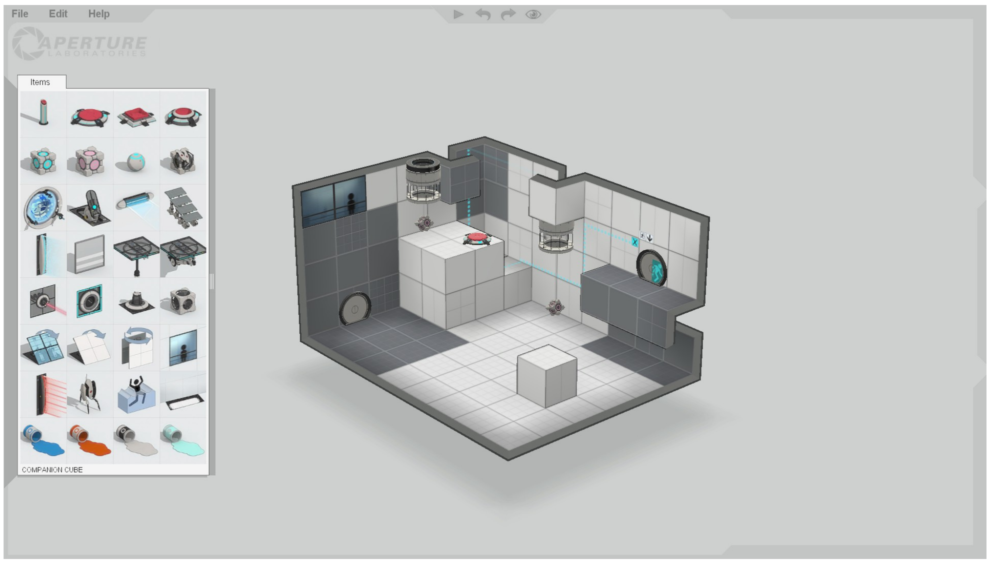


Figure 3.3: Puzzle Maker interface.

3.3 Properties of the Generator

The properties for content generator classification from Section 2.1.1 can help to specify what type of a generator is needed for Portal game levels.

Online or offline It is only possible to have an offline generator because of the content type and its properties. All elements of the Portal map are closely connected, so they must be generated at once, and not dynamically with level progression.

Necessary or optional The generator is needed to produce game levels for Portal, they have to be correct and possible to complete, classifying the content type as necessary.

Generic or adaptive The content is going to be generated offline, so it is not possible to know the player's behavior in advance, making the generator generic.

Stochastic or deterministic Usually ASP generators are deterministic, however randomization techniques can be used to make them stochastic.

Automatic or partial At the early stage the Portal level generator will be unable to produce, build, and publish maps fully automatically. It will require some parameters as an input, as well as somebody to build the output puzzle in the level editor and finally publish it. However, there can be a possibility of automating at least some of these processes later on.

3.4 Required Qualities of the Generator

The desired common qualities for PCG solutions as described in Section 2.1.2, can be used to produce a set of well-defined requirements for the generator of Portal game levels. Later these qualities can be used to evaluate how successful the implementation of the generator is.

Speed As described above, the first version of the solution is intended for an offline use, therefore the speed is not critical in this case. However, it would be good to receive an output (one game level or an "unsatisfiable" response) within a time limit of couple of minutes.

Reliability Fortunately, the nature of ASP automatically satisfies this property. As long as the game is modeled correctly and fully, there is no need to verify the validity of the resulting puzzle; one can be sure that all the rules are complied with and constraints are met. However, not all game properties are modeled in the generator encoding and thus this quality is not fulfilled automatically.

Controllability There should be a way to specify at least some parameters of the level, controlling its complexity as well as the visual appearance. The minimal set of such parameters could be the dimensions of the room (affects the size of the level) and the number of steps to reach the goal (directly corresponding to the complexity of the puzzle). Some other properties, for example the proportion of portable/non-portable tiles or the amount of various floor elevations can also have a huge effect on how complex and interesting the resulting puzzle is, therefore it would be good to have a way to control these as well.

Diversity Ideally, output level should be significantly different from each other. Due to the nature of ASP solvers, two consecutive models can be very similar to one another, for example differing only by the status of a single tile (portable/non-portable) or the location of the entrance. To solve this problem and to obtain diverse and expressive maps some randomization techniques should be applied.

Creativity In most cases it is desired that the generated content is indistinguishable from the one which was manually created by human level designers. However, in the case of map generation for the Portal game, the unique plot and prominent characters of the game can be exploited. As described previously in Section 3.1, the player is guided through the game by GLaDOS AI. In-game tests chambers were originally designed by the staff of the research facility. The provided Puzzle Maker tool also implies that game levels are created by humans. In this context it would be an interesting addition to the story of the GLaDOS character, if she would become even more sentient and learn how to design test chambers herself. This could provide a new player experience and add a unique trait to the newly generated maps. The apparent features of machine-generated content, such as unnatural symmetry, high functionality, lack of decorative elements, and neglect for the visual appearance of the map, can be considered as a plus in this scenario and therefore should not be avoided on purpose.

This list can be prioritized by emphasizing the important qualities of the Portal level generator. The most crucial quality is reliability of the map, because if the generated content is not valid then there is not much point in achieving any other qualities. The only way to ensure that the solution is valid for the current game setting is to model the problem domain correctly, therefore most of the effort should be spent on this task. Trying to make generator produce controllable and diverse output might result in sacrificing the efficiency and speed of the solution. On the other hand, diverse and varied levels can automatically contribute to the creativity of the content, potentially resulting in satisfaction of soft constraints. As creativity is a rather optional quality for this project, it is not taken into account and will possibly occur as a side effect of implementing the other generator qualities. Speed and efficiency are also secondary qualities, therefore optimization of the ASP encoding will be done in future versions of the generator.

3.5 Formalization

Declarative problem solving approach, which was illustrated in Figure 2.2, can be applied for generating levels for Portal game. Figure 3.4 shows this approach on a more detailed and specified level. Essential game logic is modeled and represented as a logic program. This encoding is fed to *clingo* solver, which will solve it and produce an answer set as output. This output is then interpreted by a Java program, kindly written for this project by

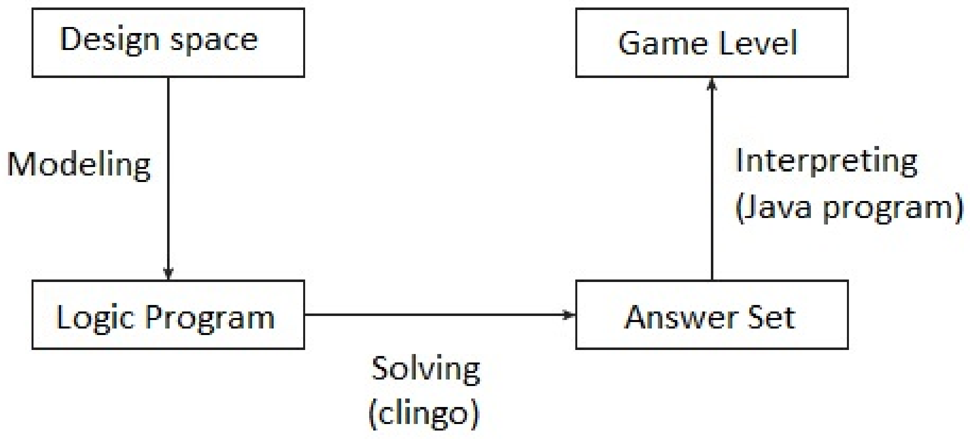


Figure 3.4: The approach for using ASP to produce game levels for Portal game.

Perttu Laukkanen. The program transforms the answer set into a ready-to-play Portal map file.

3.5.1 Simplifying Assumptions

Being a set of 3D puzzles, Portal levels are not trivial to model. In order to limit the complexity of the model, some simplifying assumptions need to be made. There are dozens of object types which can be used in each level; only two very basic ones are to be used in this project: a button and a cube dispenser. The goal room size is to be default as in Puzzle Maker ($6 \times 8 \times 4$). Usually all the elements and objects of the level are used for solving the puzzle and reaching the exit door, therefore there should not be any decorative elements.

It can be assumed that the player can always find a solution if there is one and he or she must not be able to get to an "inescapable" location (if the area can be reached, there should be a way to get out from it as well). The amount of timesteps which are used to rate the complexity of the puzzle correspond to the number of the player's transitions from one area to another, therefore only the steps leading to the correct solution are counted (and it does not matter how many times the player can go from one area to another during the actual game session).

The physics engine of the Portal game is quite complex, and allows manipulating the character's velocity by the portals using a technique called flinging. This property is difficult to model using ASP, as it requires distance and speed calculations. The use of flinging techniques is not always mandatory for finding a solution, it is more often used in advanced levels and therefore it is ignored in the current version of the generator.

Portal is played from a first-person perspective, making some objects and walls not visible for the player from certain locations. It follows the real-life line of sight concept, contributing to the complexity of some levels. Accessibility of some objects and portal placement can depend on whether the player can see the corresponding tiles from his or her location. The formula for calculating all the visible tiles and spots is quite complicated for a generator like this. Thus, it is assumed that the player can always see and use all the walls/tiles and open a portal end in them if the other conditions are met.

Another important assumption is related to the fact that there can be several solutions for some of the Portal levels. Even if the level is created by a human designer and there is an exact plan on how to solve it, some players are able to find alternative solutions. For example, certain areas in the level can be skipped or accessed with placing portals in not obvious locations or by performing flinging tricks. Especially because not all physical properties are modeled in the level generator, there are probably several ways to solve the output levels. This situation is normal and might give additional replayability value to the puzzle when players try to find alternative ways to solve it. On the other hand, in some cases it can drastically reduce the level complexity.

3.5.2 Identifying Constraints

There are many important hard constraints needed to ensure that the generated level is solvable and that it is built in accordance with the rules of the game. They are presented in the previous chapter as the base of the pyramid in Figure 2.1. Some of these hard constraints are enforced by the Puzzle Maker interface. It is not possible to remove or add certain objects, for example, there must always be exactly one entrance and one exit door in each level. Only one object can be placed on a single floor/wall tile (no overlapping objects) and no object can be placed outside of the room dimensions. Other than that, the validity of the level must be ensured by the user of the Puzzle Maker (level designer). The victory condition must be reachable by the player, therefore the exit door needs to be accessible and open at some point of the game session.

Once the hard constraints are met, the generator can be improved further by meeting the soft constraints. They are used to make the generated content interesting and challenging for players and comparable to human-made content; they are not mandatory (like hard constraints), but rather desirable properties.

The main way to rate the complexity of the generated level can be timesteps, size of the room, and the number of various objects placed in the level. The amount of timesteps that player needs to take in order to complete puzzle and the number of objects to be used directly correspond to the puzzle complexity, whereas the size of the room may have lesser effect on it. Nevertheless, all of these values should be easily maintained in the encoding, making the output content controllable and more diverse.

It is harder to evaluate how interesting the puzzle will be for players, as this criterion can be quite subjective. In general, one can assume that the players enjoy complex levels (which take some time to figure out) with a variety of objects in them. One of the simplifying assumptions states that only two types of optional objects will be used in this project, therefore the soft constraints should be met in other ways, for example by employing more complex logic in the generated levels or by making interesting structures inside the test chambers.

A further constraint comes from the name and idea of the game. It is possible to make levels where the players do not need to use portals to reach the goal, however this condition (the use of portals) should be enforced. This will give the level a more complex logic and make it better fitting to Portal's name.

Chapter 4

Implementation

Current chapter is divided into several parts, which discuss different aspects of the implementation. Section 4.1 describes how the level structure, object placement, and portal physics are encoded. Satisfaction of hard and soft constraints within the encoding is addressed in Section 4.2. Some ways to optimize and make the implementation faster are worked out in Section 4.3. Finally, Section 4.4 presents randomization techniques, which are used to make the generated content more diverse and creative.

4.1 Domain Representation

At first the physical structure of the level like walls, ceiling, and floor, along with various elevations is produced. Then several objects, which are meant to be used by player for solving the puzzle, are placed in the level structure. Finally, a solution plan is created, making sure the game is winnable and the desired complexity is reached. It should be noted that the order of the rules in the program does not affect its execution flow. The separation of the code into subsections is done purely for the sake of better readability.

4.1.1 Room Model

The complexity of the resulting puzzle partially depends on the size of the room. The default room dimensions $6 \times 8 \times 4$ offered by Puzzle maker are used and defined as constants in the beginning of the program (lines 1–3 of Listing 4.1). These and other constants can be changed later to obtain rooms with different sizes and looks, as well as less or more complex puzzles.

In the beginning it was decided to start with a very simple model of a 3D room, which has a fixed size and does not have any floor/wall elevations.

```
 1 #const width = 6.
 2 #const length = 8.
 3 #const height = 4.
 4 
 5 coord_x(1..width).
 6 coord_y(1..length).
 7 coord_z(0..height).
 8 
 9 solid_box(0, 1..length, 1..height).
10 solid_box(width+1, 1..length, 1..height).
11 solid_box(1..width, 0, 1..height).
12 solid_box(1..width, length+1, 1..height).
13 solid_box(1..width, 1..length, 0).
14 solid_box(1..width, 1..length, height+1).
15 
16 {solid_box(X,Y,Z) :coord_x(X)  :coord_y(Y)
      :coord_z(Z)} width*length*height/2.
17 
18 wall(0, Y) :- coord_y(Y).
19 wall(width+1, Y) :- coord_y(Y).
20 wall(X, 0) :- coord_x(X).
21 wall(X, length+1) :- coord_x(X).
22 
23 floor(X, Y, Z) :- solid_box(X, Y, Z), not solid_box(X,
      Y, Z+1), coord_x(X), coord_y(Y), coord_z(Z).
```

Listing 4.1: Generating basic level structure

However it quickly became obvious that the game level resulting from a simple model would not even remotely be comparable with the levels designed by humans. Additionally, converting this model to a more complicated one in the future would have probably lead to problems, forcing one to redesign the whole room model. Therefore various surface elevations were simulated already at the early stage of modeling process. The walls (including floor and ceiling) and elevations of the room are represented by solid boxes as opposed to the empty ones — the actual space of the room. Each box has a straightforward XYZ-specification to determine its location. Because the walls are fixed and do not vary for different models of the same room size, they can be expressed with facts in the lines 9–14.

Next, the surface elevations are generated using the same representation

```
1 step(0,-1 ;; 0,1 ;; 1,0 ;; -1,0 ;; -1,-1 ;; 1,1 ;;
     1,-1 ;; -1,1).
2 step_no_diag(0,-1 ;; 0,1 ;; 1,0 ;; -1,0).
3
4 same_floor(X1,Y1,X2,Y2,Z) :- floor(X1,Y1,Z),
     floor(X2,Y2,Z), step(DX,DY,_), X2 = X1 + DX, Y2 =
     Y1 + DY.
5 same_floor(X1,Y1,X2,Y2,Z) :- same_floor(X1,Y1,X,Y,Z),
     same_floor(X,Y,X2,Y2,Z).
6
7 jump_down(0,-1,0..height ;; 0,1,0..height ;;
     1,0,0..height ;; -1,0,0..height).
8 reachable_by_jump(X,Y,Z,NX,NY,NZ) :- floor(X,Y,Z),
     floor(NX,NY,NZ), jump_down(DX,DY,DZ), NX = X + DX,
     NY = Y + DY, NZ = Z - DZ, not solid_box(NX,NY,NZ+2).
```

Listing 4.2: Defining floor area separation and reachability by jumps

of solid boxes. Additionally, the overall amount of solid boxes is controlled via the aggregate operation, meaning that the proportion of empty/solid space in the room can be set (line 16 of Listing 4.1). In the lines 18–21 the actual walls of the room are identified (those solid boxes, which are not random surface elevations). They are going to be used later for correct placement of the entrance and exit doors.

Finally, an important notion of `floor` is introduced, which denotes all the solid boxes on which the player can walk and on which any game object can be placed. Naturally, only solid boxes which are not blocked by other solid boxes above them are considered valid in this respect.

4.1.2 Area Separation

The player is free to move in any direction on a XY-plane, as long as the following floor space is not blocked by a solid box. Using pooling, one step could be summed up in one atom. Pooling alternative terms can be done using ";;" symbol, for instance atom `step(0,-1;;0,1)` is expanded by grounder into `step(0,-1)` and `step(0,1)` atoms. Two versions of the rule defining steps are needed, one of them allows diagonal steps (line 1 of Listing 4.2), another one is used later for portal transitions and does not allow diagonal steps (line 2).

The level structure is then separated into several areas, depending on

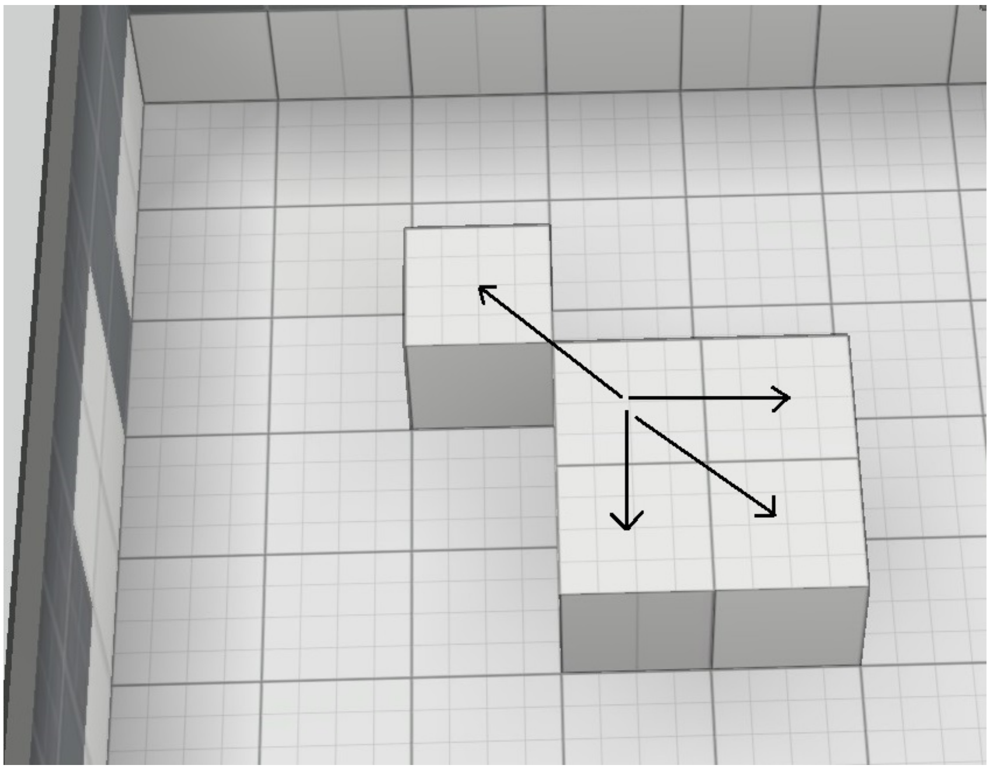


Figure 4.1: Possibility of moving around the same floor area with normal steps.

how the player can reach each specified area. All the floor tiles which are accessible with normal steps (defined above) are considered to belong to the same area. For instance, if the player is located on the central solid box in Figure 4.1, he or she can move one step to the left, one step behind, one step diagonally to top left or bottom right. As all these five solid boxes are reachable with normal steps, they are considered to belong to the same floor area.

In the room presented in Figure 4.2 five areas are identified, starting with floor tiles at area A (adjacent to the entrance) and finalizing with the small balcony structure at area E near the exit door. The described area division is important for knowing how many logical steps it would take for the player to solve the puzzle. The idea is to place objects along with entrance and exit doors into different areas of the room. Accessing the area with a button in it implies that the player has the ability to press that button and continue to another floor area. We are not interested in the amount of normal walking steps that player takes within a single floor area, as there is no time limit for

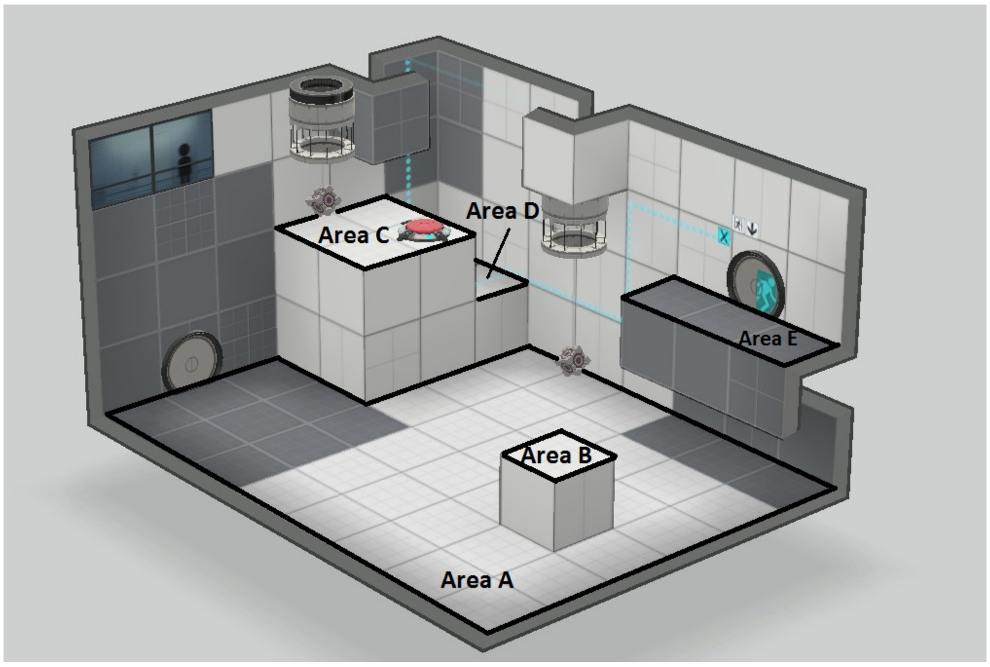


Figure 4.2: Level separation into several floor areas.

solving the puzzle and the player is free to walk around as much as he or she wants. Instead, it makes more sense to define a notion of logical timesteps, which counts transitions of player from one area to another. One transition can be done by jumping down from one area to another or by moving into a different area with a portal. As going into another area implies getting access to the objects placed in there, it becomes possible to estimate the complexity of the resulting puzzle and make the level more interesting.

Separating the level into areas can be done in several ways, for example giving each area an identity. This method has its own advantages and disadvantages, though a bit more straightforward approach is used in this project. Two floor tiles located in the same XY-plane (in other words, at the same height) have a special predicate `same_floor` to denote if they belong to the same area (line 4 of Listing 4.2). This notion is transitive in a way that if a tile at location (X, Y, Z) is reachable from tiles at locations (X1, Y1, Z) and (X2, Y2, Z), then those two tiles also belong to the same floor (line 5 of Listing 4.2).

The character of the player can also jump down to the floor boxes from elevations, obeying basic gravity laws. There is no fall damage in the game, so the jump can start from any valid height. The rule in the line 7 of Listing 4.2

```
1 width*length*height/3 {portable(X,Y,Z) :coord_x(X)
      :coord_y(Y) :coord_z(Z)}.
2 exitable_by_portal(NX,NY,NZ) :-
      same_floor(X1,Y1,NX,NY,NZ), step_no_diag(DX, DY),
      X1 = X + DX, Y1 = Y + DY, NZ = Z - 1,
      portable(X,Y,Z).
3 entrable_by_portal(NX,NY,NZ) :- floor(NX,NY,NZ),
      step_no_diag(DX, DY), NX = X + DX, NY = Y + DY, NZ
      < Z, portable(X,Y,Z).
```

Listing 4.3: Encoding portal physics

is very similar to the step rule in the line 1, but it also allows variations in Z variable, whereas in the step rule the height is fixed. After the structure of the level is generated, the possibility of reaching one floor tile from another with a jump does not change. A solid floor box can be reached from the neighboring elevation by a player's jump from the "border" of that elevation (here also the above space is checked for the absence of solid boxes blocking the jump). This concept is expressed in one rule in the line 8 of Listing 4.2. This rule checks that if the jump ends at height NZ, there is no solid box at height NZ+2. There is no need to check height NZ+1 in the same way, because the location where the jump ends must be a floor. This term makes sure that there is no solid box at NZ+1 location, according to the floor rule defined earlier.

4.1.3 Portals

Next we model the main properties of the portal physics of the game. All solid boxes are separated into two categories: portable and non-portable ones. If the box is categorized as portable, then a potential portal end can be opened in any of the surfaces of this box. Portable tiles are colored white in the game, as opposed to dark gray non-portable tiles. If there is enough lighting in the room, the player can easily distinguish between them. The rule in the line 1 of Listing 4.3 arbitrarily picks several tiles to be portable.

The portal physics can be modeled similarly to the simple steps and jumps. Consider the player located at a floor box, for example at location marked "P" in Figure 4.3. Then it would be possible for the player to open a portal and exit the current location from any neighboring portable solid box, in this case the box in front or to the right of the player. The box to the left of the player is not suitable for that, because it is not portable (colored dark

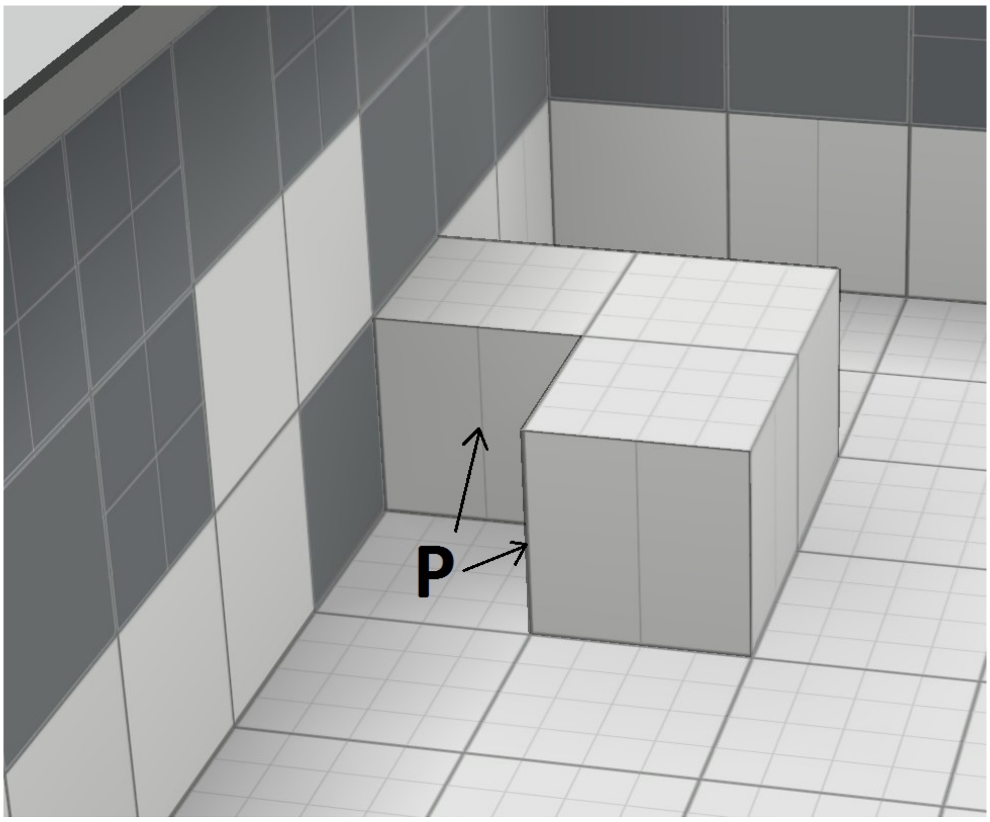


Figure 4.3: Possibility of exiting from the player's location through a portal.

gray). There is no solid box behind the player, therefore it is not possible to open a portal end there. This is modeled by the rule in the line 2 of Listing 4.3, stating that the solid box where the portal end can be opened for exit must be portable and it should be one of the higher neighboring boxes (and diagonals are not allowed).

The possibility of entering to the floor box through the other end of the portal depends on visibility of the neighboring solid boxes to the player from the current position, as well as its reachability with normal steps and jumps. The visibility condition is not modeled in the current version of the generator (as mentioned in Section 3.5.1). Entering a location from a portal is defined in the line 3 of Listing 4.3 in a similar manner to exiting one. The difference here is that entering any location from a portal does not require this location to be in the same floor area as that portal end. Also the height is not that restrictive here, in comparison with the exiting rule in the line 2 of Listing 4.3. There is no damage from falling in the game, therefore it is

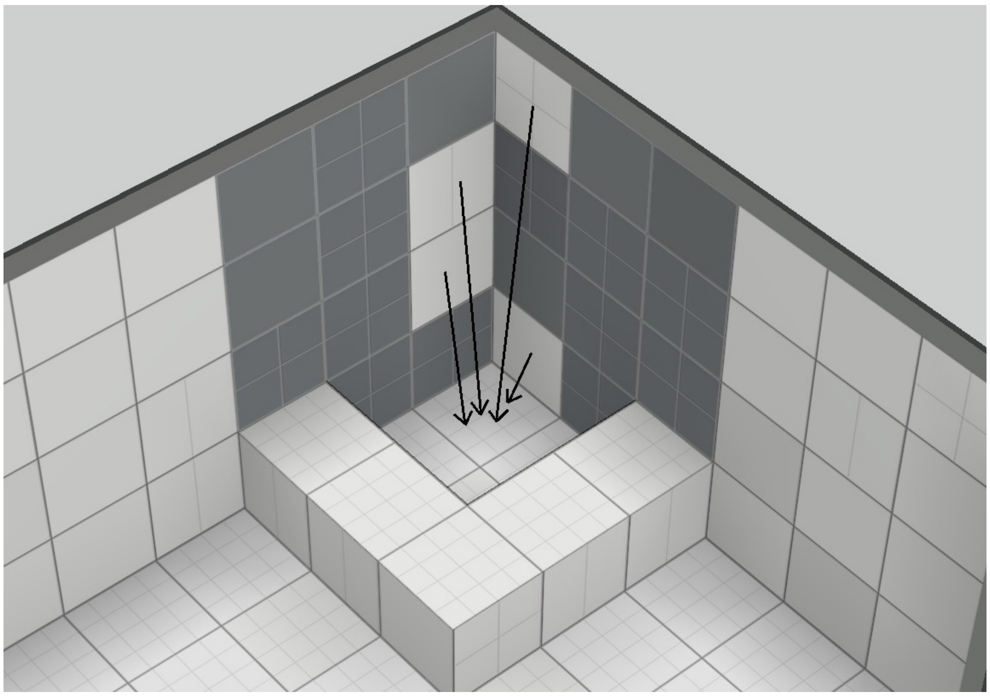


Figure 4.4: Possibility of entering a location through a portal.

possible to enter a location by dropping from a portal from any height above that location. For instance, Figure 4.4 shows how the location in the corner can be entered from a portal opened in four different locations (five, if we consider the ceiling tile strictly above that location portable as well).

4.1.4 Object Placement

For the purpose of simplicity only the basic and obligatory objects are modeled in the first version of the generator. Such objects include one entrance and one exit, and several buttons and cube dispensers. There are several types of cube dispensers in the game, some of them can produce an infinite number of cubes, others dispense just one. The latter type is modeled in the generator, making it easy to associate a dispenser with a single cube. All of the modeled objects fall under `object` type, however buttons and cubes have their own numerical identifiers. In the following, an entrance `(e)` and an exit `(x)` are defined as objects (lines 1–2 of Listing 4.4).

```
 1 object(e).
 2 object(x).
 3 
 4 potential_button(b(1..n_buttons)).
 5 {button(b(X))} :- potential_button(b(X)).
 6 :- not button(b(X)) : potential_button(b(X)).
 7 object(b(X)) :- button(b(X)).
 8 
 9 potential_cube(c(1..n_cubes)).
10 {cube(c(X))} :- potential_cube(c(X)).
11 :- not cube(X) : potential_cube(X).
12 object(c(X)) :- cube(c(X)).
13 
14 1 {place_object(O,X,Y,Z): coord_x(X): coord_y(Y):
       coord_z(Z)} 1 :- object(O).
15 
16 connected(b(1),x).
17 {connected(O1,O2)} :- place_object(O2,X2,Y2,Z2),
       place_object(O1,X1,Y1,Z1), not
       same_floor(X1,Y1,X2,Y2,Z2), button(O1).
18 
19 valid_entrance :- place_object(e,X1,Y1,Z),
       wall(X2,Y2), step_no_diag(DX,DY), X2 = X1 + DX, Y2
       = Y1 + DY.
20 :- not valid_entrance.
21 
22 valid_exit :- place_object(x,X1,Y1,Z), wall(X2,Y2),
       step_no_diag(DX,DY), X2 = X1 + DX, Y2 = Y1 + DY.
23 :- not valid_exit.
24 
25 valid_cube(c(N)) :- place_object(c(N),X,Y,Z), not
       solid_box(X,Y,Z+1), cube(c(N)).
26 :- not valid_cube(c(N)), cube(c(N)).
```

Listing 4.4: Generating and placing objects

In the line 4 of Listing 4.4 several potential buttons are generated, their number varying from one to n_buttons). Every button is identified by a number in brackets. They are arbitrarily picked by the choice rule in the line 5 and identified as objects (line 7 of Listing 4.4). Similarly, several cubes are produced and denoted as objects (lines 9–12 of Listing 4.4). Then all the

```
:- 2 {place_object(_,X,Y,Z), place_object(_,X,Y,Z)},
   floor(X,Y,Z).
:- place_object(_,X,Y,Z), not floor(X,Y,Z).
:- button(b(X)), not connected (b(X),
   c(N)):potential_cube(c(N)), not connected (b(X),x).

:- connected (O1,O), connected (O2,O), O1 != O2,
   object(O;O1;O2).
:- connected (O,O1), connected (O,O2), O1 != O2,
   object(O1;O2;O).
```

Listing 4.5: Encoding some of the hard constraints

modeled objects can be placed on exactly one location with another choice rule (line 14).

Finally, object connections are defined in the lines 16–17. Buttons are the only triggers in this generator, therefore they must be connected to other objects, like cube dispensers and an exit door. The connection of one button with an exit door is assumed to be mandatory for this generator, this connection is expressed as a fact in the line 16. In general, Portal game does not require that the exit door is always connected to an object, sometimes the exit is open from the start of the level and the solution of the puzzle is to reach it. In other cases (and under the assumptions for the generator), the exit is closed and can only be opened by placing a cube to the corresponding button. Then to solve the puzzle, the player must both find a way to open the exit and then reach it.

Placement of the entrance and the exit requires an additional set of rules to ensure that they are located near the main walls of the room. This concept is expressed with a set of rules in the lines 19–23, which define what valid entrance and exit objects are, and check that they hold for the output model by using the integrity rules. A valid placement of cubes is ensured by the rules in the lines 25–26 of Listing 4.4). Cubes are dropped by cube dispensers located right above them. Assuming the cube is placed at a certain location, there should be an empty space above it to fit the dispenser.

4.2 Satisfying Constraints

Many integrity constraints are required to ensure that the generated puzzle is valid, satisfying one of the most important hard constraints. Rules in Listing 4.5 are rather generic, as they are related to all of the modeled game

```
 1 #const min_time = 2.
 2 #const max_time = 6.
 3 time(0..max_time).
 4
 5 use(e, 0).
 6 use(O, T) :- use_jump(O,T).
 7 use(O, T) :- use_portal(O,T).
 8
 9 use_jump(O2, T+1) :- object(O2;O1), place_object(O2,
      X2, Y2, Z2), place_object(O1, X1, Y1, Z1), use(O1,
      T), time(T), reachable_by_jump(X1, Y1, Z1, X2, Y2,
      Z2).
10
11 use_portal(O2, T+1) :- object(O2;O1), place_object(O2,
      X2, Y2, Z2), place_object(O1, X1, Y1, Z1), use(O1,
      T), time(T), exitable_by_portal(X1, Y1, Z1),
      entrable_by_portal(X2, Y2, Z2), not
      same_floor(X1,Y1,X2,Y2,Z2).
```

Listing 4.6: Introducing time steps and generating solution plan

objects. For instance, line 1 states that only one object can be placed at a single location. Line 2 makes sure that each object is placed on a valid floor tile, and line 3 guarantees that every button placed in the level is connected to either a cube dispenser or the exit door. Some unique object properties, like connections, also require integrity constraints to prevent several outgoing or incoming connections to one object (the connection relation in this game is one-to-one, this is encoded in the lines 5–6).

4.2.1 Solution Plan

Generation of the plan for solving the puzzle is needed to control an adequate placement of the game objects, as well as to aid in the validation of the resulting game level. As some of the game properties are not modeled (described in Section 3.5.1), the final level inspection must be made by human level designers. They would observe the level in the editor and then use the provided solution plan to verify that the puzzle is solvable.

The solution plan consists of a set of `use(O,T)` atoms, telling the user which object to use at which timestep. The amount of minimum and maximum timesteps is controlled via the constants in the lines 1–2 of Listing 4.6.

```
1 exit_openable :- use(O,T), object(O), connected(O, x),
    use(x, FT), time(T;FT), T < FT.
2 goal :- place_object(e, X1, Y1, Z1), place_object(x,
    X2, Y2, Z2), not same_floor(X1, Y1, X2, Y2, Z2),
    use(x,T), exit_openable, time(T), T >= min_time.
3 :- not goal.
```

Listing 4.7: Setting the goal

To move to a different floor area, and to get an access to the object located in it, the player must use either a jump or a portal move. The time variable T keeps a track of such transitions to different floors areas. The timer is started when the player enters the puzzle, in practice this would mean the use of entrance door at timestep 0 (line 5). An object can be used when the floor area is accessed by either jumping to it or alternatively by using a portal (lines 6–7). Using a jump to get to the floor area and use an item in it is possible when that area is reachable by jump, and an item on the first area has been used at the previous timestep (line 9). Similarly, line 11 tells the conditions of using a portal to get to an object.

4.2.2 Goal Statement

The final goal of the whole level can be defined with the rules presented in the lines 1–2 of Listing 4.7. The exit must be placed on the level, and the placement should be on a different floor area from the entrance. The exit should be reachable by jumps or portal moves, implying that it can be used afterwards. Also the exit should be opened by using a button connected to it during one of the timesteps, not exceeding the maximum number of steps.

4.3 Optimizing the Encoding

Generating a general level structure is easier than additionally finding a solution to it. The hard constraints defined in Section 3.5.2 are not very difficult to satisfy. In other words, generating a solvable Portal map only requires building a valid room structure, and making sure that the exit is accessible from the entrance.

To add any complexity to the generated levels, many of the soft constraints should also be targeted. Producing a solution plan forces the generator to take into account the amount of desired timesteps and their order. This brings some logic to the level instead of mere placement of the objects.

However, without optimizing the encoding the execution time can increase drastically.

One idea is to get rid of the Z-variable in the definition of solid boxes, leaving only X and Y coordinates. Replacing one of the three-variable rules with a two-variable one reduces the computation time, especially if the head of this rule is present in the bodies of many other rules. In practice, with this particular rule any balcony-like structures would be excluded from the map, as the height variable would no longer be present, filling all the space underneath the particular coordinate with solid boxes. This would decrease the diversity of the levels and potentially reduce their "fun" factor, while giving a slight speed up to the generator. As the speed is not a very critical factor for an offline generator, it was decided to keep Z-coordinate in that rule and not exclude these kinds of interesting structures from the search space.

In the line 1 of Listing 4.7 there is a comparison of time steps T and FT. The main concept is to have these two variables different from each other. This can also be expressed with "not-equal" predicate like T != FT. However comparing them as T < FT brings the same effect and it is more efficient, as symmetric rules are removed from the ground program. All of the "not-equal" comparisons were substituted with "less than" predicate in the encoding.

4.4 Applying Randomization

For some types of games and content randomization is used extensively as a basis of the generation algorithm, as described in Section 2.1.4. In this project randomization is not a core technique, but it is still essential for satisfying one of the important quality criteria of a content generator — diversity of the output.

The traditional ASP has the property that resulting adjacent answer sets are usually quite similar to each other, if these solutions are located near each other in the search space. In cases with complicated structures they might differ only by one non-essential predicate. When the encoding presented in this chapter is run, many of the first answer sets have exactly the same structure and object placement. The only difference comes from some tiles being portable or not, causing no effect on the actual puzzle logic and level structure.

The main purpose of using randomization techniques is to obtain diverse and varied answer sets. Ideally, players could compare two generated maps which would feel like two separate game levels and not copies of one another.

It is hard to achieve this with a limited variety of modeled objects, but randomization can affect the general look and feel of the puzzles.

To achieve this, several techniques can be used. There are tools and algorithms (e.g., *xorro* [17]) for sampling answer sets with near-uniform probability. Alternatively, there is an inbuilt *seed* and *rand-freq* options in *clingo*, which allow the user to select a random seed and frequency for random choices during the search. Setting frequency will guide the solver in its search and make it do random decisions with a certain probability. Based on a few tests, the use of this command proved to be suitable enough for this project. While being simple to use, it provides the desirable diverse output and gives a certain controllability, as well as allows the user to reproduce the results using the same seed. Changing the default solver behavior this way can cause loss of efficiency and generation speed. This is experimented on and the results are discussed in Chapter 5.3.

Chapter 5

Evaluation

In this chapter the implementation of the generator is reviewed in several ways. First, the properties of the generator are discussed in Section 5.1. Some of these properties are tested in practice, and the test results are presented in Sections 5.2 and 5.3. Then, one of the levels is published for the community to play and its reviews are gathered and analyzed in Section 5.4. Finally, Section 5.5 discusses possible future features and improvements for the solution.

5.1 Properties of the Resulting Solution

First, the generator can be evaluated by testing it against the quality conditions set and specified in Section 3.4. Whereas most of them require practical experiments to be performed, some conditions can be evaluated based on the encoding and implementation techniques. All the practical tests are made on a classroom Linux computer with Intel Xeon processor E5-4627 v2 (16M Cache, 3.30 GHz). The tool used for grounding and solving is *clingo* version 3.0.5.

Speed In general, the generation time for most cases is in a matter of minutes, which is an acceptable result for an offline content generator. The generation time increases when parameters become too restrictive. For instance, when the number of maximum steps is set to 2, it takes several minutes to find a solution for a moderately small room sizes ($5 \times 4 \times 4$). For the default $6 \times 8 \times 4$ dimension generator does not produce any result for hours. Increasing the maximum number of steps and the amount of potential objects in the room makes the requirements less restrictive and the models can be found much faster even for the standard size of the puzzle. Effects of varying these and other parameters, as well as the exact speed measurements and scalability of the encoding are presented and discussed further in Section 5.2.

Reliability As mentioned earlier, if the problem domain and all the rules are encoded correctly, then the use of ASP ensures the validity of the output. However, even though a large amount of time was spent on modeling the generator, not all of the game specific properties were represented. The quickest way to test reliability of the solution is to simply try and play some of the maps. Satisfying hard constraints is the most important issue, therefore the two main things to check during the test are consistency and playability of the puzzle. Fortunately, fulfillment of these constraints can be easily checked upon a brief visual examination of the map in the Puzzle Maker. The tool highlights objects and structures which break any of the game rules (for example, colliding items or invalid elevations). Next, the exact sequence of steps suggested by the output must be used to verify that at least one solution can be found and the goal can be reached, making the generated level winnable.

Controllability Many important parameters of the output can be controlled from within the encoding. These include the size of the room, numbers of potential buttons and cubes and the desirable amount of logical steps for finding a solution, the relative amount of elevations and empty space in the map, and the proportion of portable/non-portable tiles in it. All of these parameters can be set individually, and they have a direct impact on the complexity of the resulting puzzle, as well as its look and feel. Effects of varying some of these parameters are documented in Section 5.2.

Diversity Diversity of the generated levels is mainly reached by using the randomization technique described in Section 4.4. Satisfying the controllability requirement using various constants can also help to achieve different looking game levels. The main challenge is the limited amount of objects modeled in the generator. Even though the levels look different, their solutions are similar to each other. Details of achieving some level of diversity by using randomization are described in Section 5.3.

Creativity Most of the Portal maps designed by players use a variety of game objects in them. It is hard to produce truly creative content, when one is bound to using only a small portion of available types of objects. In addition to that, some maps with a harder difficulty level usually exploit unique Portal physics (e.g., flinging) and complex logic to make them fun and original. For instance, some levels require the players to launch themselves in the air by jumping into one portal end, while making another portal end in the location which becomes visible only in the air.

Portal map generator, developed in this project, is lacking representation of many available Portal object types (light bridges, gels, hazards, etc.). Also, the game environment and the physics are simplified (see Section 3.5.1), i.e., visibility of tiles at each map location and flinging effects are not modeled.

However the final results of the generator look quite unique and original. The logic required for solving the puzzle is simple, but the visual look of it is interesting and unusual to the players. Feedback from the players is given in Section 5.4.

5.2 Scalability

Scalability usually describes how well a system is able to handle growing quantities of data. In this project scalability mainly refers to the size of the test chamber (which can be different from the default $6 \times 8 \times 4$). While non-optimized code can have an acceptable performance for small puzzle sizes, it is likely to slow down significantly when room dimensions are increased.

First, several test runs were made with the complete generator encoding, which outputs both the design and the solution plan for the puzzle. The height, length and width of the room was varied. The number of maximum objects was relatively small (2 cubes and 2 buttons, plus the mandatory entrance and exit doors). The amount of possible solution steps was set from 2 to 5, so that it does not become too restrictive for the larger sized rooms).

Table 5.2 shows the measured results of finding one model for the full version of the generator. Grounding time (corresponding to "Prepare" column) increases gradually with the number of rules. Most of the time is usually spent during the preprocessing stage, when the ground program is modified and simplified by internal algorithms of *clingo*. Solving is then less time-consuming, however in some cases it is particularly hard to find an answer (for example, for the $3 \times 5 \times 3$ dimension). For the rooms of the same volume, it often takes more time to generate an output when the height is too restrictive. This effect can be observed for the room volumes 45, 80, and 150, where solving time is significantly higher for the room with a smaller height value. This can be related to the modeled property of some floor tiles being reachable by jump. There are more possibilities to use jumps for area transitions in the rooms with a higher ceiling, thus it can be faster to find a solution.

Starting from room volume of 75 the size of the ground program starts

Dimensions	Volume	Overall (s.)	Prepare (s.)	Prepro (s.)	Solve (s.)	Rules
$3 \times 3 \times 3$	27	2.90	0.62	1.48	0.80	181114
$3 \times 4 \times 3$	36	4.84	1.12	3.46	0.26	314527
$3 \times 3 \times 4$	36	8.34	1.12	3.64	3.58	316540
$3 \times 5 \times 3$	45	21.98	1.82	7.12	13.04	484408
$3 \times 3 \times 5$	45	10.60	1.78	7.44	1.38	489040
$3 \times 4 \times 4$	48	11.44	2.06	9.02	0.36	549132
$3 \times 5 \times 4$	60	22.78	3.28	19.00	0.50	844730
$3 \times 4 \times 5$	60	23.28	3.24	19.62	0.52	848017
$4 \times 4 \times 4$	64	30.02	3.80	23.98	2.24	958545
$3 \times 5 \times 5$	75	50.52	5.22	42.70	2.58	1303774
$4 \times 5 \times 4$	80	113.20	5.92	53.02	54.26	1481526
$4 \times 4 \times 5$	80	61.02	5.88	53.56	1.58	1479146
$4 \times 5 \times 5$	100	141.94	9.46	120.82	11.66	2284275
$5 \times 5 \times 5$	125	355.58	14.64	272.80	68.12	3539676
$5 \times 6 \times 5$	150	742.46	21.76	551.46	169.22	5073727
$5 \times 5 \times 6$	150	611.22	21.90	560.64	28.66	5047450
$5 \times 6 \times 6$	180	n/a	n/a	n/a	n/a	n/a
$6 \times 6 \times 6$	216	n/a	n/a	n/a	n/a	n/a

Table 5.1: Generation time for the complete generator encoding for different room dimensions.

to have millions of rules, and the overall generation time increases to several minutes. Unfortunately for rooms with volume larger than 150 no results were obtained. The amount of rules in this case increases so much that the machine where the code was executed ran out of memory.

Next, a simplified version of the encoding was tested. The rules associated with finding a solution plan from Listing 4.6 are left out. The reachability concepts in the lines 7–8 of Listing 4.2 and in the lines 2–3 of Listing 4.3 are omitted as well. The resulting encoding performs significantly faster, which can be observed in Table 5.2. The same room sizes were used for these tests to observe how scalable the solution can be. The simplified generator is significantly faster, but it inherits even more properties of a partial and not fully automated solution. It does not produce a complete verified level, but can give a good basis for it. Human level designers need to verify and complete the level if it happens to be unsolvable.

Controllability property of the generator is satisfied by varying several constants in the encoding. With the simplified generator code the proportion of solid boxes is normally set to at most half of the volume of the room (`width*length*height/2`). When it is reduced to one third of the volume, the levels contain less elevations and look more human-made, as many human designers leave most of the room free and empty. Examples of such

Dimensions	Volume	Overall (s.)	Prepare (s.)	Prepro (s.)	Solve (s.)	Rules
$3 \times 3 \times 3$	27	0.08	0.04	0.04	0.00	18119
$3 \times 4 \times 3$	36	0.18	0.08	0.08	0.02	33060
$3 \times 3 \times 4$	36	0.14	0.60	0.08	0.00	27628
$3 \times 5 \times 3$	45	0.36	0.14	0.20	0.02	53293
$3 \times 3 \times 5$	45	0.22	0.10	0.10	0.02	39081
$3 \times 4 \times 4$	48	0.32	0.12	0.18	0.02	50309
$3 \times 5 \times 4$	60	0.62	0.22	0.36	0.40	80838
$3 \times 4 \times 5$	60	0.50	0.18	0.28	0.04	71014
$4 \times 4 \times 4$	64	0.76	0.24	0.46	0.04	92868
$3 \times 5 \times 5$	75	0.98	0.30	0.62	0.06	113783
$4 \times 5 \times 4$	80	1.52	0.42	1.00	0.10	150915
$4 \times 4 \times 5$	80	1.20	0.36	0.76	0.08	130585
$4 \times 5 \times 5$	100	2.50	0.62	1.72	0.16	211340
$5 \times 5 \times 5$	125	5.34	1.02	3.84	0.48	344997
$5 \times 6 \times 5$	150	9.70	1.54	7.78	0.38	518504
$5 \times 5 \times 6$	150	8.04	1.38	6.14	0.50	457402
$5 \times 6 \times 6$	180	14.76	2.16	12.10	0.48	684739
$6 \times 6 \times 6$	216	27.98	3.20	23.92	0.82	1031594

Table 5.2: Generation time for the simplified generator encoding for different room dimensions.

levels where solid boxes are parametrized to fill at most one third of the room volume are shown in Figure 5.1. The proportion of portable/non-portable tiles can be controlled in a similar way.

5.3 Randomization Effects

Randomization is very important for satisfying both the diversity and the creativity qualities of the generator. The test chamber shown in Figure 5.2 is the first model resulting from running the simplified encoding (with no solution plan, only elevations and object placement). It has a clear structure, where most of the variations in the elevation happen on the ceiling of the room (most of its area is filled with solid boxes, except 2 locations). The tiles are also clustered by their type, portable tiles are located in the same sectors, mainly on the floor and on the ceiling with very little variation. All the objects are placed near each other except the entrance, which obeys the rule stating that the entrance and the exit doors should not be located in the same floor area. The resulting puzzle basis is extremely easy and boring for the players.

The same encoding was then run several times using a random seed and

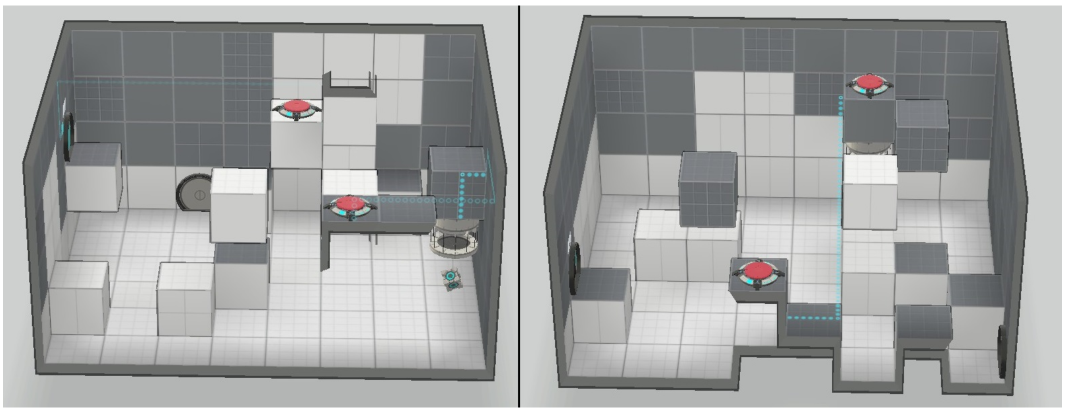


Figure 5.1: Two levels with reduced proportion of solid boxes generated by the simplified encoding.

frequency. Some resulting levels are shown on Figure 5.3. These rooms look very different from the non-randomized level in Figure 5.2, but are still similar to each other. Although most of the portable tiles are still distributed evenly in the lower part of the room, the placement of objects and elevations is clearly randomized. This puzzle is much more interesting and engaging for the players, encouraging them to explore the map. In addition, it is not clear right away how to solve it in comparison with the non-randomized puzzle from Figure 5.2, and this provides additional level complexity.

Many randomized rooms looked similar to each other, though one of the random seed runs resulted in an interesting looking level (shown in Figure 5.4). It is more similar to the non-randomized output in Figure 5.2. Like there, most of the elevations are placed at the ceiling of the tests chamber, however each object has its own floor area which is not shared with other objects. This demonstrates that not all outputs of the generator have the same chaotic structure like in Figure 5.3 even when randomization is used.

The relation of randomization with scalability was also studied. For each value of frequency for randomized choices the generator was run 10 times. The average time of finding one model for different room dimensions is presented in Table 5.3. For non-randomized output the values are taken from Table 5.2 and rounded. As expected, the generation time increases together with increasing the frequencies used for randomization. The solver is tuned to work at its best and applying random frequencies changes its behavior. For example, running the program with parameter –rand-freq=0.3 guides the search for models in such a way that the solver will make random decisions with probability 0.3. Setting this frequency too high (e.g., to 0.5–0.9) can

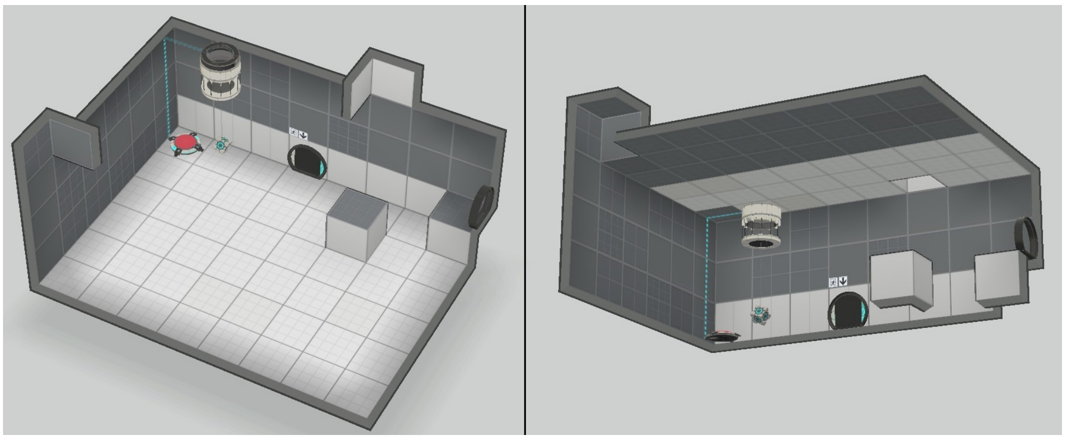


Figure 5.2: First level generated from a non-randomized run of the simplified encoding, view from the top and bottom.

Dimensions	Volume	Average overall time for freqency (s.)			
		non-randomized (0.0)	0.1	0.3	0.5
$3 \times 3 \times 3$	27	0.1	0.5	0.5	0.5
$4 \times 4 \times 4$	64	0.8	1	1	1
$3 \times 5 \times 5$	75	1	1	1	1
$4 \times 4 \times 5$	80	1	1	1	2
$4 \times 5 \times 5$	100	3	3	4	4
$5 \times 5 \times 5$	125	5	6	7	9
$5 \times 5 \times 6$	150	8	10	12	14
$5 \times 6 \times 6$	180	15	21	27	32
$6 \times 6 \times 6$	216	28	51	61	73

Table 5.3: Average time of finding one model for different frequencies for randomized choices.

slow down the generation speed significantly for large room sizes. For smaller room sizes it does not make much difference. The generation time for small rooms is relatively short and similar or the same as for non-randomized runs. A plot in Figure 5.5 is based on the data from Table 5.3.

5.4 Standard Generated Level

Levels with the goal dimensions $6 \times 8 \times 4$ can be generated with the simplified encoding, which does not check the reachability of objects and does not provide a solution plan. One of such levels is selected for further evaluation. A random seed and frequency of 0.1 is used for its generation. The

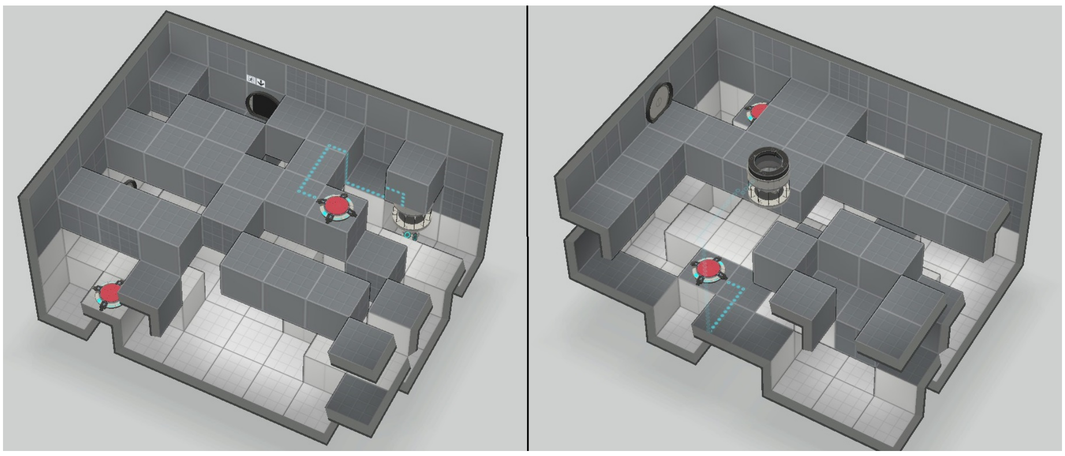


Figure 5.3: Two levels generated from randomized runs of the simplified encoding with different seeds.

number of possible buttons and cubes is set to two for each object, and the proportion of solid boxes in the room is no more than half of the volume of the room (width∗length∗height/2). Generation time for this level is relatively small, around 1 minute (but can vary depending on the seed). After examining the level corresponding to the model in the Puzzle Maker (Figure 5.6), one minor correction needs to be done, changing the tile on top of one buttons to be a portable type. This makes the puzzle solvable, and this level can be published to Portal 2 players for testing.

The level presented in Figure 5.6 was published [1] for the community to play and evaluate. Several people downloaded it and so far most of them gave positive ratings to the map. These ratings and some of the comments can be seen on Figure 5.7. Most of the players commented, that the level looked ”machine-made”, the complicated structure and surface elevations confused them in the beginning. Several players said, that it took them several minutes to wonder around the map figuring out what needs to be done. Usually Portal maps made by human designers are much more open, they do not contain that many obstacles on the way and labyrinth-like structures. Because of that many players found the generated level interesting and curious, especially because some objects appeared hidden (not in their direct view-sight).

Most of the levels created by human designers use lighting to aid players in solving the puzzle. No light sources were modeled in the generator, therefore the resulting map is very poorly lit, making it harder to find the solution (in particular, it was hard to figure out where the portal can be placed). This can be seen as an advantage for such a simple puzzle. Some players

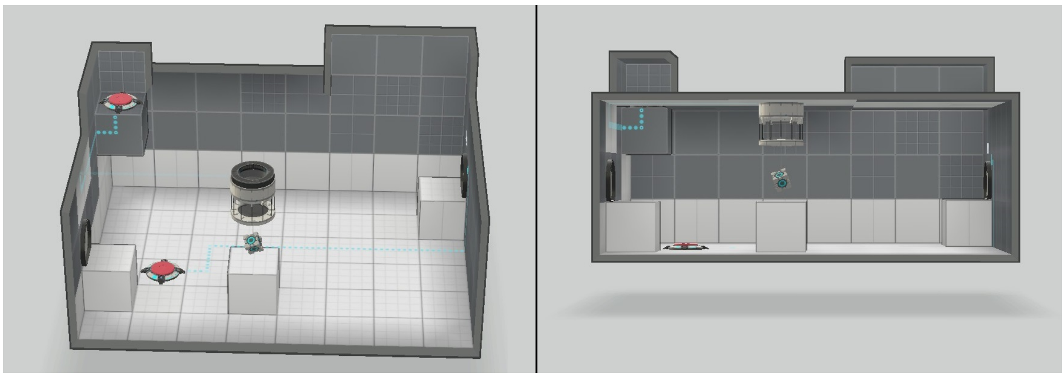


Figure 5.4: One of the levels generated by the simplified encoding with a random seed, view from different angles.

also noticed that darkness helps to create a unique atmosphere for this level, making it look more like some heartless machine created it.

Several people who played this level expressed a fear of reaching a dead end, i.e. not being able to complete the puzzle without restarting it. There are many "ditches" in the level, and the only way to get out of them is to use portal (meaning there must be portable tiles near the player and in some other area of player's vicinity). However no such cases actually happened and all players were able to solve the puzzle without getting stuck.

5.5 Future Development

Portal is a very complex and interesting game. It involves dozens of various game objects, several memorable characters, unique physics, and an outstanding plot. All of these have been used by the professional level designers to create a set of polished puzzles of progressing difficulty. Furthermore, the fans of the game were given an opportunity to create their own levels with little restrictions on their sizes and amounts/variety of objects in them.

Currently only two basic kinds of non-mandatory objects are modeled for the generator. The next course of action is to add several more objects to the problem encoding. This will increase level diversity significantly. At the same time, it is important to optimize the encoding so that it is able to generate larger maps and place more objects in them.

In the original Portal and Portal 2 games, several of the first levels have the role of a tutorial for the players. They are progressively increasing in difficulty, gradually introducing new objects and teaching the players new

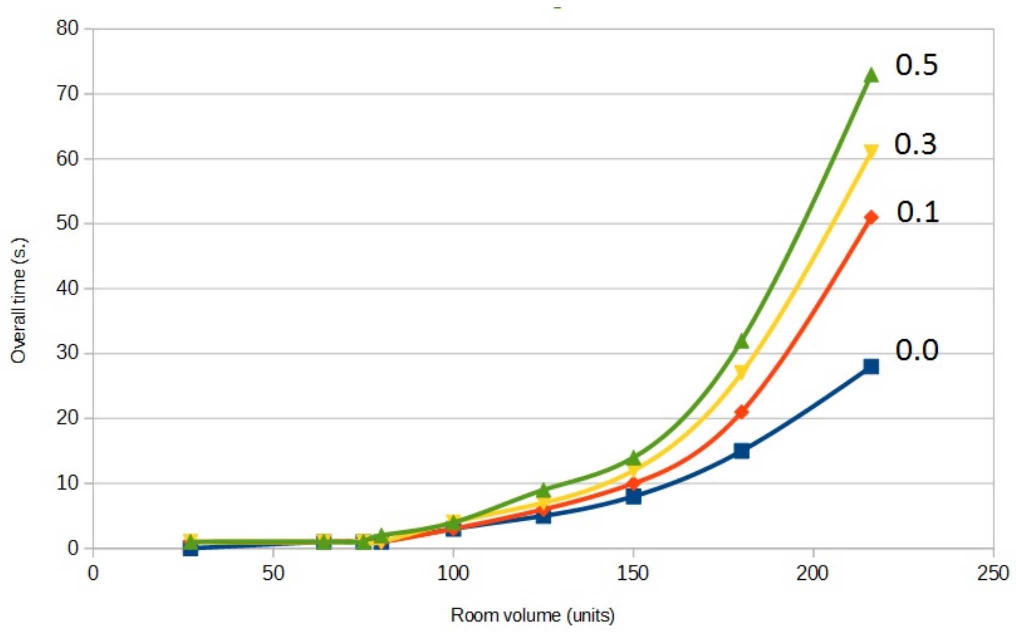


Figure 5.5: The effect of varying the frequencies for randomized choices on the generation time for different room sizes.

skills for solving puzzles. Currently the generator outputs one model at a single run, and it does not keep any track of how difficult the previous puzzle is. A further improvement is to make the generator start by creating easy levels with a few basic objects in the beginning, and gradually increase the complexity in subsequent models (for instance, similar to the ASP level generation in Refraction game [24]).

Another interesting property to implement is to try and make the generated levels look like if they were designed by humans. This effect can partially be achieved by creating additional parameters and rules. For instance, it is possible to make elevations more symmetric by applying structuring rules (for example, allowing only certain constructions in the level or forcing a minimum amount of solid boxes in one floor area). Distributing lighting around the level can also help it look like human-made.

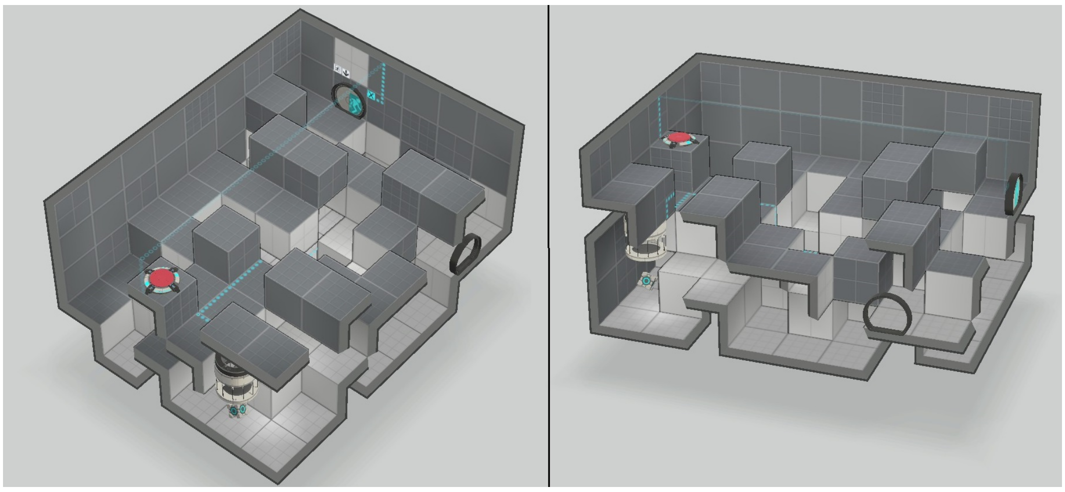
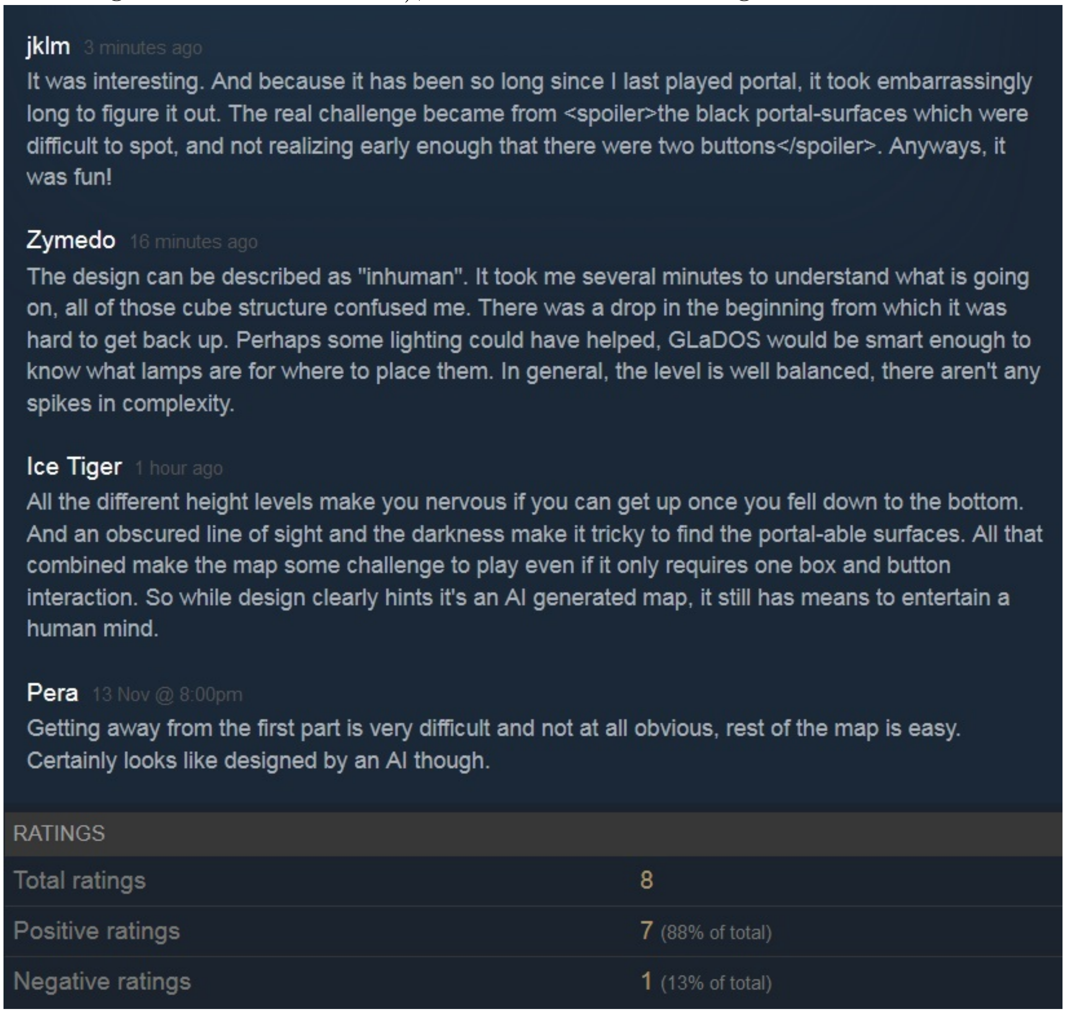


Figure 5.6: Level published in the workshop (generated by the simplified encoding with a random seed), view from different angles.



jklm 3 minutes ago
It was interesting. And because it has been so long since I last played portal, it took embarrassingly long to figure it out. The real challenge became from <spoiler>the black portal-surfaces which were difficult to spot, and not realizing early enough that there were two buttons</spoiler>. Anyways, it was fun!

Zymedo 16 minutes ago
The design can be described as "inhuman". It took me several minutes to understand what is going on, all of those cube structure confused me. There was a drop in the beginning from which it was hard to get back up. Perhaps some lighting could have helped, GLaDOS would be smart enough to know what lamps are for where to place them. In general, the level is well balanced, there aren't any spikes in complexity.

Ice Tiger 1 hour ago
All the different height levels make you nervous if you can get up once you fell down to the bottom. And an obscured line of sight and the darkness make it tricky to find the portal-able surfaces. All that combined make the map some challenge to play even if it only requires one box and button interaction. So while design clearly hints it's an AI generated map, it still has means to entertain a human mind.

Pera 13 Nov @ 8:00pm
Getting away from the first part is very difficult and not at all obvious, rest of the map is easy. Certainly looks like designed by an AI though.

RATINGS	
Total ratings	8
Positive ratings	7 (88% of total)
Negative ratings	1 (13% of total)

Figure 5.7: Players' comments and ratings of the generated Portal map (screenshot of the item page in the workshop).

Chapter 6

Conclusions

Procedural content generation techniques are applied in many classic and modern games. Pseudo-random number generators, search-based approaches, grammars, and fractal algorithms, among others, are used to automatically produce game levels, landscapes, items, quests, and maps in many kinds of games. PCG techniques benefit the game development process in several ways. The costs and the resource requirements for developing a new game or creating new content for an existing game are decreased by the use of PCG. Many of the PCG techniques greatly increase replayability of the game, thus helping to make the game more successful.

Answer set programming, a declarative problem solving paradigm, can also be used for content generation. It has proven efficient for producing simple 2D puzzles and maps, as well as levels for roguelike games. In combination with other frameworks it can even be used to generate content based on real-time feedback.

The main goal of this project was to design a Portal map generator and to encode it in ASP. The results show that it is in fact possible to make a partial offline generator for a complex logic game like Portal. Only the basic properties and physics aspects of the game are modeled in this implementation, and several simplifying assumptions are made. This makes the solution fast, but puts limitations on some of the other quality conditions. For instance, reliability is guaranteed by the nature of ASP, meaning that the output of the solver will always comply with the rules of the encoding. However, if the problem to be solved is not fully modeled in the encoding, it is not guaranteed that the solutions obtained will satisfy all the hard constraints. The generator in this project focuses only on the main physics aspects and does not take into account other factors, like visibility. Thus, the validity of the content must be checked by a human level designer.

The current version of the generator creates interesting and unusual level structures, as shown by several reviews of the players. At the same time ASP ensures that objects are placed on different floor areas, the connections between them are valid and the objects are all reachable for the player. Many different parameters of the generator can be controlled from the encoding, i.e., the amount of objects, room size, the amount of elevations, and the proportion of portable/non-portable tiles. Altering these parameters can make the generated levels look different from each other and provide the required qualities of diversity and creativity.

The generator can already be used as a partial offline level creation tool. In cases where a more complex level is required, the designer can use the generator output as a basis for his or her work. Various structures will already be present in the map and the reachability of the basic objects will most likely be ensured by the generator. Many new unique Portal levels can possibly be inspired from such a basis. The solution cannot yet fully replace human level designers for the Portal game, but it can help and guide them to create maps, which look and feel different from their usual style.

Overall, the resulting solution behaves as expected. Optimizing the encoding further will allow for the modeling of other game objects and physics properties, making the generator more complete and independent from human designers. Modeling all of the aspects of the game would make the generator reliable, resulting in each level being playable and winnable, and eliminating the need for a designer's inspection afterwards.

Bibliography

[1] Antonova, E. Steam Workshop: GLaDOs designs her first test chamber, October 29, 2015. `http://steamcommunity.com/sharedfiles/filedetails/?id=544183630`. Accessed 22 Nov 2015.

[2] Ashlock, D. A. Automatic generation of game elements via evolution. In *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games, CIG 2010, Copenhagen, Denmark, 18-21 August, 2010*, pp. 289–296.

[3] Bailly, D. Handmade vs randomized level design in Epistory, May 12, 2015. `http://www.indiedb.com/games/epistory/news/handmade-vs-randomized-level-design-in-epistory`. Accessed 22 Nov 2015.

[4] Bertz, M. The Technology Behind The Elder Scrolls V: Skyrim, January 17, 2011. `http://www.gameinformer.com/games/the_elder_scrolls_v_skyrim/b/xbox360/archive/2011/01/17/the-technology-behind-elder-scrolls-v-skyrim.aspx`. Accessed 22 Nov 2015.

[5] Blizzard Entertainment, Inc. Diablo III Official Game Site. `http://us.battle.net/d3/en/`. Accessed 22 Nov 2015.

[6] Boenn, G., Brain, M., Vos, M. D., and Fitch, J. P. Automatic music composition using answer set programming. *TPLP 11*, 2-3 (2011), 397–427.

[7] Brewka, G., Eiter, T., and Truszczyński, M. Answer set programming at a glance. *Commun. ACM 54*, 12 (December , 2011), 92–103.

[8] Chatfield, T. The 10 best video-game characters, August 8, 2010. `http://www.theguardian.com/technology/2010/aug/08/10-best-video-game-characters`. Accessed 22 Nov 2015.

[9] Dart, I. M., De Rossi, G., and Togelius, J. SpeedRock: Procedural Rocks Through Grammars and Evolution. In *Proceedings of the 2Nd International Workshop on Procedural Content Generation in Games* (New York, NY, USA, 2011), PCGames '11, ACM, pp. 8:1–8:4.

[10] Delgrande, J., and Faber, W., Eds. *Proceedings of the Eleventh International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR'11)* (2011), vol. 6645 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag.

[11] Dietz, H.-P. Controlled Chaos: Randomization in Level Design, April 18, 2015. `http://jandh.org/resources/blog/jandh_blog_2015-04-18_2_en.pdf`. Accessed 22 Nov 2015.

[12] Ferreira, L., Pereira, L., and Toledo, C. F. M. A multi-population genetic algorithm for procedural generation of levels for platform games. In *Genetic and Evolutionary Computation Conference, GECCO '14, Vancouver, BC, Canada, July 12-16, 2014, Companion Material Proceedings* (2014), pp. 45–46.

[13] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. *Answer Set Solving in Practice*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2012.

[14] Gebser, M., Kaminski, R., Kaufmann, B., and Schaub, T. *Clingo* = ASP + control: Preliminary report. In Leuschel and Schrijvers [20]. Theory and Practice of Logic Programming, Online Supplement.

[15] Gebser, M., Kaminski, R., König, A., and Schaub, T. Advances in *gringo* series 3. In Delgrande and Faber [10], pp. 345–351.

[16] Gebser, M., Kaufmann, B., and Schaub, T. Conflict-driven answer set solving: From theory to practice. *Artif. Intell. 187* (2012), 52–89.

[17] Gebser, M., Schaub, T., Schneider, M., Thiele, S. xorro : Near Uniform Sampling of Answer Sets by Means of XOR. `http://sourceforge.net/p/potassco/code/HEAD/tree/branches/xorro/XorroDocu/manual_xorro.pdf`. Accessed 22 Nov 2015.

[18] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Logic Programming, Proceedings of the Fifth International Conference and Symposium, Seattle, Washington, August*

15-19, 1988 (2 Volumes) (1988), R. A. Kowalski and K. A. Bowen, Eds., MIT Press, pp. 1070–1080.

[19] Kloetzli, J. Procedural Terrain Generation in Sid Meier's Civilization V, March 11, 2013. `http://www.firaxis.com/?/blog/single/procedural-terrain-generation-in-sid-meiers-civilization-v`. Accessed 22 Nov 2015.

[20] Leuschel, M., and Schrijvers, T., Eds. *Technical Communications of the Thirtieth International Conference on Logic Programming (ICLP'14)* (2014), vol. 14(4-5). Theory and Practice of Logic Programming, Online Supplement.

[21] Niemelä, I. Answer set programming: A declarative approach to solving challenging search problems. In *Proceedings of the 41st IEEE International Symposium on the Multiple-Valued Logic (ISMVL 2011)* (Tuusula, Finland, May 2011), IEEE, pp. 139–141.

[22] Pittman, C. Teach With Portals: Building an Oscillator, June 29, 2012. `http://www.teachwithportals.com/index.php/2012/06/building-a-simple-harmonic-oscillator/`. Accessed 22 Nov 2015.

[23] Schüller, P., and Weinzierl, A. Answer Set Application Programming: a Case Study on Tetris. In *Proceedings of the Technical Communications of the 31st International Conference on Logic Programming (ICLP 2015), Cork, Ireland, August 31 - September 4, 2015.* (2015).

[24] Smith, A. M., Andersen, E., Mateas, M., and Popovic, Z. A case study of expressively constrainable level design automation tools for a puzzle game. In *International Conference on the Foundations of Digital Games, FDG '12, Raleigh, NC, USA, May 29 - June 01, 2012* (2012), pp. 156–163.

[25] Smith, A. M., and Mateas, M. Answer Set Programming for Procedural Content Generation: A Design Space Approach. *IEEE Trans. Comput. Intellig. and AI in Games 3*, 3 (2011), 187–200.

[26] Smith, A., Bryson, J. A Logical Approach to Building Dungeons: Answer Set Programming for Hierarchical Procedural Content Generation in Roguelike Games. In *Proceedings of the 50th Anniversary Convention of the AISB.* `http://doc.gold.ac.uk/aisb50/AISB50-S02/AISB50-S2-Smith-paper.pdf`. Accessed 22 Nov 2015.

[27] Sterling, L., and Shapiro, E. Y. *The Art of Prolog - Advanced Programming Techniques, 2nd Ed.* MIT Press, 1994.

[28] Togelius, J., Justinussen, T., and Hartzen, A. Compositional Procedural Content Generation. In *Proceedings of the The Third Workshop on Procedural Content Generation in Games* (New York, NY, USA, 2012), PCG'12, ACM, pp. 16:1–16:4.

[29] Togelius, J., Shaker, N., and Nelson, M. J. Introduction. In *Procedural Content Generation in Games: A Textbook and an Overview of Current Research*, N. Shaker, J. Togelius, and M. J. Nelson, Eds. Springer, 2014.

[30] Togelius, J., Yannakakis, G. N., Stanley, K. O., and Browne, C. Search-Based Procedural Content Generation: A Taxonomy and Survey. *IEEE Trans. Comput. Intellig. and AI in Games 3*, 3 (2011), 172–186.

[31] Valve Coproration. Official Portal 2 Website, 2011. `http://www.thinkwithportals.com/`. Accessed 22 Nov 2015.

[32] Wichman, G. R. A brief history of Rogue, 1997. `http://www.wichman.org/roguehistory.html`. Accessed 22 Nov 2015.

[33] Yannakakis, G. N., and Togelius, J. Experience-Driven Procedural Content Generation. *T. Affective Computing 2*, 3 (2011), 147–161.