# Procedural Generation of Narrative Puzzles

by

**Barbara De Kegel, B.Sc. (Hons)**

**Dissertation**

Presented to the

University of Dublin, Trinity College

in fulfillment

of the requirements

for the Degree of

**Master of Science in Computer Science**

**(Interactive Entertainment Technology)**

**University of Dublin, Trinity College**

September 2016

# Declaration

I, the undersigned, declare that this work has not previously been submitted as an exercise for a degree at this, or any other University, and that unless otherwise stated, is my own work.

Barbara De Kegel

August 30, 2016

# Permission to Lend and/or Copy

I, the undersigned, agree that Trinity College Library may lend or copy this thesis upon request.

Barbara De Kegel

August 30, 2016

# Acknowledgments

Firstly I want to thank my supervisor Mads Haahr for his advice and the interesting discussions we had over the course of this dissertation. I also want to thank the IET class, in particular Patrick and Dan, for their camaraderie and their patience with my endless questions, and Shane, for supporting me at the end. Finally I want to thank my family for their endless support throughout my education.

Barbara De Kegel

*University of Dublin, Trinity College*
*September 2016*

# Procedural Generation of Narrative Puzzles

Barbara De Kegel
University of Dublin, Trinity College, 2016

Supervisor: Mads Haahr

Narrative puzzles involve exploration, logical thinking and progressing a story. This project proposes a system for the procedural generation of such puzzles for use in story-rich games or games with large open worlds. An extended type of context-free grammar forms the basis for both the generation algorithm and the puzzle solving. Each designer-defined rule in the grammar defines a possible behavior of item types in the game world. Puzzles, which consist of a tree of rules, are generated live on a per area basis, through recursive generation of inputs for outputs. Given a valid grammar, the backwards generation guarantees that all created puzzles are solvable. A proof of concept adventure game was developed to demonstrate some of the possibilities provided by the generation. Different playthroughs of this game resulted in different puzzles, integrated into a small 3D world.

# Contents

# List of Figures

# Chapter 1

# Introduction

Procedural content generation (PCG) has always been popular in video games; from *Rogue (1980)* to the huge success of *Minecraft (2011)*, it has often been used in the creation of unique game worlds. *Minecraft* proved that a game based on PCG could be compelling, thus setting the stage for modern game industry interest in this technique.

PCG has been increasingly used across a range of game genres and domains. *Dwarf Fortress (2006)*, a predecessor and influence on *Minecraft*, procedurally generates a detailed dwarven lineage at the start of each game, and uses hundreds of layered procedural systems to create a complex game set in a unique world. The level generation in the platform game *Spelunky (2008)* has been praised, for shifting focus from mastering a level to mastering a set of rules. In the *Civilization* series, PCG is used to promote early-game exploration of the world.

Besides world-building, PCG has also been used in the creation of AI-characters. *Crusader Kings II (2012)* procedurally generates character traits that alter the NPCs' decision-making, leading to interesting family dramas, while *Shadow of Mordor (2014)* makes players feel like they are fighting specific enemies by generating each enemy's characteristics independently.

## 1.1 Motivation

The use of PCG in the creation of puzzles has been limited, and then focused mainly on puzzles for games that are strictly puzzle games, as opposed to puzzles that are incorporated into a wider game, such as an RPG. However, it can be a challenge for game designers to fill large open worlds with engaging content, such as puzzles. This is also true for games with procedurally generated worlds, such as *Minecraft* and the recently released *No Man's Sky (2016)*. While *Minecraft* successfully profiles itself as a crafting sandbox, *No Man's Sky* has garnered criticism for being mundane in the moment-to-moment game play.

Large worlds often suffer from repetitive content, such as a single type of interaction or puzzle appearing in many different places, e.g. the bypass puzzles in *Mass Effect 2 (2010)*. Even across different games, there are generic puzzle types that will re-appear as easy add-ins. A system for adding procedurally generated puzzles into the narrative of such a world could make for a more interesting player experience.

In the context of smaller, story-driven games, puzzle generation can be used to achieve the goal of re-playability; a common complaint for these types of games is that they can only really be played once. Changing up the puzzles integrated into the game's story could change up the experience without the need for designers to write branching story lines.

## 1.2 Narrative Puzzles

Narrative puzzles can be defined as puzzles that form part of the progression of a narrative, whose solutions involve exploration and logical as well as creative thinking. They are a key component of adventure and story-driven games, and often feature in large open world games, including RPGs. Narrative puzzles can be viewed as temporary obstacles to the story's advancement; though they don't always have to be solved in a precise order, certain puzzle sequences will need to be solved before proceeding to others.

Typically narrative puzzles involve making logical connections, though in some cases players will have to combine items in ways that are not immediately obvious. A good narrative puzzle should have a satisfying solution, i.e. should make sense to the player upon being solved. Some examples of narrative puzzle patterns include: combining two items to make a third, changing the property of an item using an another, and saying the right thing to convince an NPC.

## 1.3 Objectives

In order to understand puzzles and the methods used to generate them, the first half of this dissertation presents a comprehensive review of existing methods for procedural puzzle generation, structured as a taxonomy of puzzle types. The puzzles examined as part of this are representative of the spectrum of puzzle types, involving both paper-based and digital puzzles from a wide range of game genres.

The second half proposes a grammar-based system for the procedural generation of narrative puzzles, inspired by the Puzzle-Dice system, which is the most significant current state of the art in this domain. This system is designed with the intention of being used for the creation of a range of narrative puzzle types. A small adventure game that incorporates this puzzle generation technique is designed as a proof-of-concept.

The goals for the system for narrative puzzle generation, listed in order of importance, are:

- Solvable puzzles - All puzzles created by the system should be guaranteed to be solvable.
- Expressive power - The system should allow for the expression of a range of narrative puzzle types.
- Flexible difficulty - The designer can influence the difficulty of the puzzles created by the system.
- Integration - It should be possible to integrate the system, and resulting puzzles, with existing games.

- Usability - The system should be easy to use for a game designer.

## 1.4 Roadmap

This dissertation starts with a puzzle taxonomy in Chapter 2 that provides the framework for a detailed discussion of previous methods for puzzle generation. This taxonomy includes a section on narrative puzzles, but can also be considered a stand alone review of the literature. Chapter 3 delves into the grammar-based design of the puzzle generation system created for this dissertation, and Chapter 4 describes an implementation of that system in Unity, as well as its application to a small proof-of-concept game. Chapter 5 evaluates the system, by comparing it to the Puzzle-Dice system which is the current state-of-the-art for the generation of narrative puzzles. Finally, Chapter 6 summarizes the main contributions made by this dissertation and the limitations of the system, as well as ideas for future work.

# Chapter 2

# Puzzles and Their Procedural Generation

Previous work on puzzle generation has mostly focused on specific puzzles because creating a generator requires at least some knowledge of the puzzle rules. Consequently, this chapter evaluates past research in puzzle generation in the framework of a puzzle taxonomy. Puzzles are classified according to their rules, and by extension, the cognitive skills needed to solve them. The characteristics of each puzzle type are closely linked to the challenges posed for generating such puzzles. Naturally there are also many puzzles that fall in the overlap between two or more categories; ingenuity in puzzle design often stems from making original combinations of existing mechanics.

Togelius et al. distinguish between algorithms that are constructive and those that can be described as generate-and-test [4]. Constructive algorithms generate the content once, performing validity checks at different stages of construction; the Markov chain is a typical example of such an algorithm. As described later in this chapter, some generators are generate-and-test, with levels being continuously tested, discarded, and regenerated, while others are constructive, with solvability guaranteed in the method of generation. Generate-and-test systems are more open-ended; it is difficult to compute a crisp threshold for acceptability when output evaluation is based on heuristic functions.

This chapter first looks at some fundamental differences between paper- and computer-

based puzzles as they pertain to design decisions. Then the taxonomy section runs through different puzzle types, providing a description of traits and a discussion of past procedural generation research, if any. Generation is not feasible for every type, and for others it is a trivial task that is not worth reviewing in detail.

## 2.1 Traditional vs. Digital Puzzles

Some puzzle types in the taxonomy existed on paper before they were digitalized, while others, such as physics-based puzzles can only really exist as video games, due to their interactive nature. Traditional puzzles are often digitalized as is, such as *Sudoku* and crossword puzzles, meaning that there is no change to the base puzzle mechanics and the skills required for solving. Digitalization can make traditional puzzles more accessible by providing ways to easily undo and check partial solutions.

Different past papers target puzzle generation of both traditional and digital puzzles. Some of the distinctions in the methods of generation may be linked directly to whether or not the puzzles under investigation were digital.

There are a few inherent differences between puzzles games played on a computer versus in real-life; one of the notable ones is the possibility of brute forcing. Whether players will be inclined to find a solution through brute forcing is (generally) not an issue for real-life puzzles, but some computer-based puzzles, such as the maze puzzles in *The Witness (2016)*, have an interactive feedback system that allows the solver to try all possible solutions. Brute forcing is not always a negative - *The Witness* uses it as a learning mechanism - but for puzzles in which it is undesired, the solver must be dissuaded by making the cost of brute forcing higher than that of logically arriving at a solution. There are also computer-based puzzles that do not lend themselves to brute forcing, typically they are in the path-building category, such as the area puzzles in *The Talos Principle (2014)*.

Another important difference between paper-based and digital puzzles is the existence of temporal aspects; digital puzzles can use time as a mechanic or constraint in a way that on-paper puzzles cannot. Video games can use time pressure to increase

the difficulty of a puzzle; this can come in the form of a timer or through setting the game in an environment that is changing in real-time, e.g. *Tetris (1984)*. Puzzles of this nature, i.e. those that are real-time, are often called action puzzles, a label that can be applied across many of the classification categories described below.

There is a distinction to be made between puzzles that use time as a constraint and those that use time as a puzzle mechanic. While temporal constraints push the player to quickly find a solution, they do not change the skills the player must use to arrive at that solution. On the other hand, temporal puzzles require the player to find solutions that include player-directed manipulation of time, e.g. the puzzles in *Braid*.

Some on-paper puzzles require the solver to use spatial thinking, e.g. matching a folded-open cube texture to a drawing of a cube. Spatial thinking in this context can be described as envisioning the 3D representations of object drawn in 2D. While that sort of mental exercise has no place in a 3D game world, the player would use spatial thinking to, for example, conceive of logical placements of items in space to form a solution. It should be noted that spatial thinking applies to 2D and 3D path-building puzzles, e.g. *Refraction* [5].

## 2.2 Puzzle Taxonomy

While some puzzle video games consists of levels of just one puzzle types, many others include puzzles from several of the categories described in this taxonomy, or puzzles that are best defined by the overlap of two categories. For example, *The Talos Principle* includes a mix of mazes, temporal puzzles, construction puzzles and even some physics-based elements. Puzzles are also present as mini-games in video games that would not be classified as puzzle games, e.g. hacking in *Mass Effect 2 (2010)*. To this end, the taxonomy presented here is about puzzle types, closely related to mechanics, as opposed to, and independent from, game genres.

### 2.2.1 *Sokoban*-Type Puzzles

This category of puzzles is name after the 1982 Japanese video game in which you push crates around a constrained grid-based area to get them to goal positions. A defining factor of this category is that no items/characters are ever lost or added to the board; the solution exists as a rearrangement of the original configuration. Rearrangement comprises of moving items around a confined environment, using a limited number of possible actions, i.e. crates can be only be pushed, not pulled, and only one at a time.

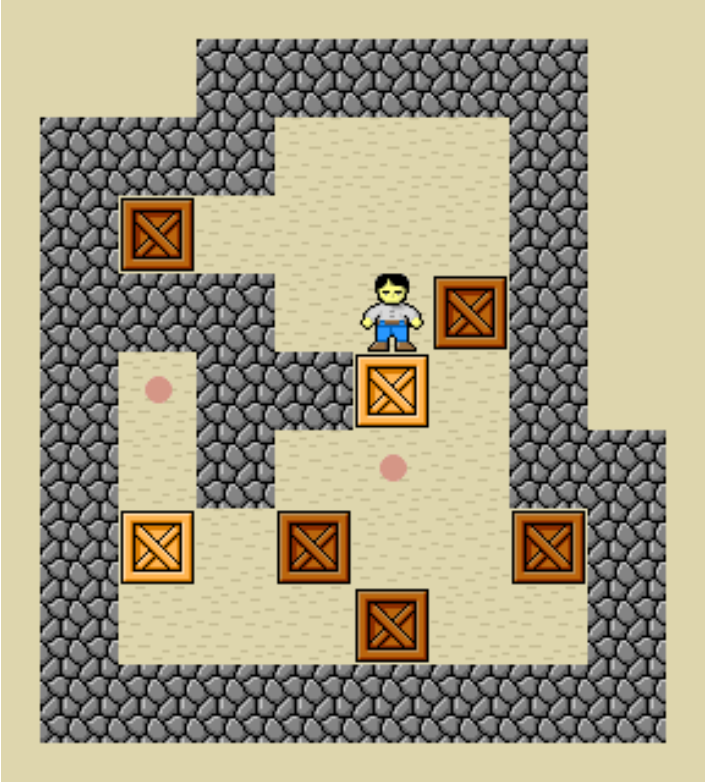

Figure 2.1: A screenshot from *Sokoban*.

Many variations (descendants) of *Sokoban (1982)* now exist, and game developer Stephen Lavelle has even created an HTML5 engine for this type of game: PuzzleScript [6]. Lavelle is also the developer of the recently released *Stephen's Sausage Roll (2016)*; a game in which you push a sausage across tiles with the goal of grilling each side exactly once on special grill tiles. The grill tiles function as a variation of the goal

tiles; the objects (the sausages) have to touch multiple goal tiles in order for the level to be completed.

Figure 2.2: A screenshot from *Stephen's Sausage Roll (2016)*.

The constraints imposed by limited space are a key characteristic of *Sokoban* puzzles. The player has to think a few moves ahead as some lines of play lead to a state from which the solution is unreachable. In *Sokoban* the player character can only push, not pull, blocks, so once a block is in a corner or against a wall, it loses (a) degree(s) of movement freedom. In practice, *Sokoban* puzzles often have an undo functionality, so that the player does not have to restart the game each time they follow a line of play to a dead end. This facilitates the player's problem-solving as it allows physical exploration of solutions. The movement mechanics vary for different puzzle games but the basic notion of moving one item at a time and avoiding unrecoverable situations is common.

Well-designed *Sokoban* puzzles must strike the balance between trivially easy and impossible, a difficult task for all but experienced *Sokoban* level designers [7]. Procedural generation can function as an aid to inexperienced puzzle designers. Generally *Sokoban* levels do not have many, if any, alternative solutions, so the player must discover a precise sequence of actions. The problem-solving process involves pursuing promising sequences, and discarding those that are futile.

Despite the simplicity of the rules, *Sokoban* can be challenging to solve [8]. Past

research has determined that solving *Sokoban* puzzles is PSPACE-complete zciteculberson1999sokoban, meaning among the most difficult one-player games [8]. Automatic puzzle solvers are not the focus of this chapter, but are relevant insofar that are often used to test the solvability of a generated puzzle.

Procedural generation of *Sokoban*-type puzzles has garnered a significant amount of research interest. *Sokoban* exhibits compelling challenges in the field of puzzle generation, including a large space of possible configurations, which hinders search algorithm traversal, and may make it difficult to guarantee solvability [8]. There is also no good method to evaluate if an initial state will lead to a non-trivial or interesting solution.

One of the earliest forays into puzzle generation was by Murase et al.; they developed a program to create *Sokoban* problems in three stages; generation, checking and evaluation [9]. In the first stage a level layout is generated by random combination of templates that are overlapped to guarantee a connected space. The templates are grids of wall and passageway tiles. Goal positions are randomly placed on legal passageway tiles, i.e. tiles onto which a player can push an object and crates are added one by one within range of at least one goal position. The player character is put on a random passageway tile.

Generated problems are not guaranteed to be solvable so, in stage two, the program uses a Breadth First Search (BFS) solver to check for a solution. However this automatic solver was only able to solve problems with short solution sequences (due to BFS), so those with long sequences were incorrectly discarded [9]. In the final stage trivial and uninteresting, albeit legal, levels are discarded according to the following criteria: the length of the solution sequence, the number of changes in direction of pushing, and the number of detours.

The results showed that out of 500 generated levels, 44 were deemed good by the program, but only 14 of those were evaluated as good problems by human experts [9]. One of the main issues with this generation program is that the restriction on solution length prevents the creation of complex problems.

Taylor and Parberry generated interesting *Sokoban* levels that are guaranteed to be solvable on the basis of working backwards from the goal positions [7]. Empty rooms are similarly generated with templates but for a wider range of dimensions; invalid or low quality rooms, e.g. ones that are disconnected or have large sections of open floor, are immediately discarded.

Next, a brute force algorithm is used to evaluate all possible combinations of goal positions. While this can lead to the discovery of compelling levels, it is obviously an expensive process; the runtime of the algorithm is exponential. For each goal placement, the system finds the furthest possible starting position, i.e. the shortest longest path, by moving in reverse - ensuring playability [7]. The value of a certain placement combination is the farthest found path, where distance is measured using the box line metric, which is the number of unrepeated box pushes. Paths are found using a form of iterative deepening twice; A* is not suitable because the target is only vaguely defined. Finally, levels are evaluated according to some heuristics.

The technique employed by Taylor and Parberry generates self-proclaimed interesting levels at the cost of a long generation time (hours or even days). The generation speed is okay for offline generation, but not great for creating mini-games on the fly, and the system cannot handle more than 6 boxes. They mention that their methodology could be applied for other puzzles, but there would be quite some effort involved in reducing the amount of game-specific information; as mentioned, a major issue in puzzle generation efforts.

Taylor et al. performed on auditory Stroop test to investigate whether players pay as much attention to the generated puzzles from [7] as they do to hand crafted puzzles by experienced designers [10]. The experiment exploits the fact that attention is a finite resource; focusing on Sokoban will decrease participants attention on the Stroop test and vice versa. The results indicated that participants were at least as engaged with the generated puzzles as with the hand-crafted ones, which may imply that they found both equally interesting. This is an important finding in showing value in procedurally generated puzzles.

Recently, Kartal et al. developed a method for procedurally generating *Sokoban* levels of varying sizes and difficulty using a Monte Carlo Tree Search (MCTS) approach [8]. They divide the puzzle creation problem into two phases: puzzle initialization, assigning the initial room layout, and puzzle shuffling, determining the goal locations. Overall, the generation is formulated as an MCTS optimization problem because it has been successful for other problems with high branching factors and has an "anytime" property. The latter refers to the fact that the search will return a valid solution regardless of when it is interrupted.

The possible actions at each node in the search tree are: "delete obstacles", "place boxes", "freeze level", "move agent", and "evaluate level". At the start, the board is composed entirely of obstacles except for one empty tile with an agent. Puzzle initialization takes place until the level is frozen - an action which saves a start configuration. The puzzle shuffling stage consists of executing the "move agent" action, which simulates *Sokoban* game rules [8]. Like Taylor and Parberry, Kartal et al. exploit the fact that generating a puzzle through game play guarantees solvability.

The MCTS approach requires an evaluation function to guide the search. The researchers chose to use a combination of two metrics; terrain and congestion. The terrain value is calculated by adding up the number of neighboring obstacles for each empty tile. The congestion metric is based on the number of boxes, goals and obstacles between each box and its corresponding goal. Higher scores corresponded to challenging puzzles in many cases, but the metric did not capture all aspects of difficult Sokoban puzzles.

In addition to the possibility of difficulty tuning, the MCTS method eliminates the need for human input and addresses the run-time issue seen in other generation methods. This makes it suitable for online generation of mini-games. Improving the evaluation metric allows for dynamically presenting puzzles of varying difficulty, and the nature of the MCTS algorithm even allows multiple such puzzles to be output in a single run [8].

Assessment of *Sokoban* puzzles, in terms of difficulty and interest, is a closely re-

lated area of research. The ability to tune difficulty depends upon accurate evaluation, shown to be a non-trivial task for *Sokoban* puzzles. Kartal et al. provided no formal validation of their evaluation function; there is still future work to be done on evaluating what makes levels interesting.


Figure 2.3: A screenshot from *Fling! (2013)*.

*Fling! (2013)* is an example of a puzzle that falls in the overlap of *Sokoban*-type and tile-matching puzzles, though more towards the former. In this game you fling balls into each other to sequentially remove all but one ball from an empty grid. The balls act in turn as the player character, an obstacle or a crate, when compared with *Sokoban*. It is a player character when you use it to fling (push) another ball (a crate) off the board, but balls can only fling non-adjacent balls of the grid - so those that are adjacent act as obstacles.

Sturtevant looked at using large-scale breadth-first search for analysis and content generation for *Fling!* [11]. He starts by classifying different search approaches and defining the focus of the paper in the complete/uninformed category; an approach used when no guidance is available, or when the goal is to enumerate an entire state space. [11]. This generally refers to brute force searching which uses algorithms like

depth-first (DFS) and breadth-first search.

The focus of the research is the development of a tool that can analyze and explore *Fling!* puzzles, rather than generating them from scratch, though it can be used for this. The tool uses an endgame database, generated by iteratively solving all *Fling!* boards of sizes 1-10 using a retrograde search approach. For any given board the tool can determine that following metrics: the number of states legally reachable, using forward BFS; the legal moves which lead to a goal state, using DFS with endgame data; and how adding/removing pieces from the board changes the solvability. The board could be generated entirely at random because the level design rules are much less strict than for *Sokoban*. The crux is on creating interesting levels rather than just solvable ones.

The difficulty of a given *Fling!* board is most intuitively measured by the number of reachable states from an initial configuration. Experiments showed a strong correlation between levels (difficulty) and number of states in the state space. Like for *Sokoban*, more domain-specific metrics may be useful, e.g. counting the number of times the player has to switch from controlling one ball, as the player character, to another.

Overall, the motivation behind Sturtevants work is proposing the use of brute force search techniques as assistance in the design of interesting puzzle instances. This includes creating endgame databases and annotating puzzles to discover how changes to its configuration will influence solvability.

### 2.2.2 Sliding Puzzles

Sliding puzzles are closely related to *Sokoban*-style puzzles because they also involve moving items, or tiles, towards goal positions in a constrained grid-based space. There are however some differentiating characteristics that qualify this as a different category: there is no player character and items can be moved in any free direction, making unrecoverable states rare. Often the grid is square-shaped without obstacles and there are only a few open spaces to slide a tile onto. Like for *Sokoban* puzzles, the player must look a few moves ahead to determine possible winning sequences.


Figure 2.4: A screenshot from *Rush Hour*.

The most well-known examples of sliding puzzles are *Rush Hour (1996)*, the 15-puzzle and the picture-forming sliding puzzles. Since the movement rules for items on the board are much less restricted than in *Sokoban*, players are more likely to stumble across solution paths by random movement of the tiles. There are generally no "dead-end" states, which encourages interactivity. Unlike in *Sokoban*-type puzzles, sliding puzzles may have one-to-one pairings between items and goal positions, so figuring out the best mapping is not part of the solving process. In *Rush Hour* and similar puzzles, there is one item that must reach one defined goal position - specifically, the red car must reach the exit in *Rush Hour*.

The *Rush Hour* puzzle has been shown to be PSPACE-complete [12], like *Sokoban* and exhibits similar challenges in determining the difficulty of a puzzle. Limited work has been done on the generation of *Rush Hour* puzzles [13]. Block-sliding puzzles such

as *Rush Hour* can conceivably be generated by starting from the end configuration and working backwards, in a similar fashion to some of the generation methods described in the previous chapter. Also like *Sokoban*, determining what makes a difficult initial configuration is not straightforward. The number of moves used to play backwards to a start layout could be greater than the shortest path, so it is not a good metric for difficulty.

### 2.2.3 Tile-Matching Puzzles

The player's object in tile-matching games is to manipulate tiles on a grid in order to make matches [14]. When a match is made, the corresponding set of tiles disappears, and the player scores points. Common matching criteria include shapes, colors and symbols. Puzzles of this category are relatively simple - they have very few rules - and are often categorized as casual games.

The most popular subcategory of tile-matching game is match-three games, e.g. Bejeweled, in which players swap the positions of tiles to make a row or column of at least three matching tiles. This category of puzzles has a low status, perhaps due to their low barrier to entry, or the large number of similar games that now exists [14]. Most tile-matching puzzles have an element of time pressure which introduces a fail state; without a timer the puzzles would be too easy. Fail states are important for making the player feel a sense of achievement.

The initial layout of tiles on the grid, and/or the choice of tiles/piece that appear during the game are likely always procedurally generated using random number generators. The generation of these simple tile-matching puzzles is a rather trivial task.

Rychnovsky procedurally generated all the levels for his game *Fruit Dating (2014)*, a more complex tile-matching puzzle that draws elements from *Sokoban* and sliding puzzles *Artic1:online*. *Fruit Dating* consists of moving items on a grid-based board with walls and obstacles, like in *Sokoban*, with the goal of getting matching characters onto adjacent tiles. Unlike *Sokoban*, it lacks a player character, and each action can affect between zero and all items on the board. Items are moved by swiping in one of


Figure 2.5: A screenshot from *Fruit Dating*.

the four cardinal directions; each swipe will move all objects in the chosen direction, while they are not blocked by obstacles. This mechanic is similar to that in Threes, another popular mobile puzzle game.

*Fruit Dating* overlaps with the *Sokoban* and sliding puzzle categories because it requires similar cognitive skills for solving; players must anticipate the outcome of moves to determine if it will be possible to reach a solution. It shares the characteristic that objects must be rearranged to progress the game.The room of obstacles is a *Sokoban* element, while active exploration of the solution space without easily reaching a dead end is an element of sliding puzzles not shared by *Sokoban*. As is a common characteristic of tile-matching games, a matched pair is eliminated, and there are a few items on the board with special abilities. The challenge for *Fruit Dating* stems from discovering a sequence of moves; while this is useful to some extent in tile-matching games there is generally not a single correct sequence.

Rychnovsky developed a level editor that has the capability to generate new levels and evaluate the playability and difficulty of any given level. The automatic solver uses a breadth first search approach with pruning and returns the shortest path. The level editor interface means that the designer can tweak a generated level, and then re-test it to make sure it is still playable.

The process of generating a *Fruit Dating* level is divided into two steps; generating the level structure and placing the on-board items [15]. The level structure is created by first randomly placing the external wall tiles, i.e. those connected to the border of the grid, and then the internal ones, i.e. those surrounded by 8 empty tiles. The game objects, including the pair(s) of fruits that must be matched, are placed randomly according to predefined weights assigned to empty tiles. The weights are different for each type of object.

All levels are checked for solvability after generation, and those that are not solvable are simply discarded. The generation algorithm runs quickly so that is an acceptable loss. The main issue with this generation approach is that there is no way to control difficulty, other than by simply adding more items to the board.

### 2.2.4 Mazes

Mazes are defined in this taxonomy as puzzles that require the solver to find a valid path from a starting point (entry) to and ending point (exit). A huge variety of puzzles could be created from rules for safe movement in an otherwise hazardous environment [1]. These puzzles could have explicit barriers, such as in more traditional mazes, where the path is obstructed by physical walls, or implicit boundaries, such as in some grid-based puzzles. Many of the puzzles from the recent video game *The Witness (2016)* fall under the second category; the player has to traverse a path through a grid according to some logical rules.

For most 2D mazes the player can observe the entire puzzle area, and thus it is possible to locate the solution path before starting to trace it. However 3D mazes may require the player to physically traverse partial paths to obtain all the information required to find the solution.

Obstacle course navigation can be classified as a type of maze; the player must navigate an area along a correct path to reach an endpoint without taking too much damage. Many obstacle courses are a cross between a maze and a path-building puzzle,

where a player must utilize a number of items to create the desired path between entry and exit.

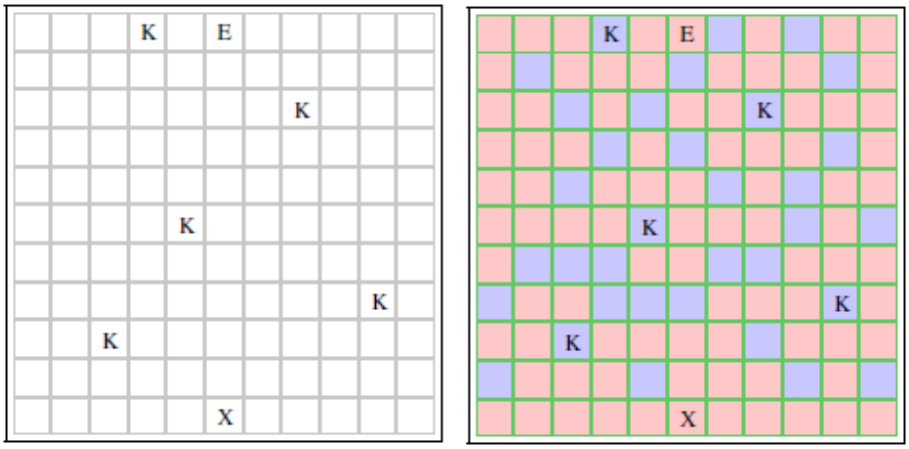


Figure 2.6: Example of a chess maze for a rook from [1]; E represents the entry tile, X the exit tile, and the K tiles have knights on them. The left image is what appears to the player, the right image has the obstructed squares marked in blue.

Ashlock uses an evolutionary algorithm as a puzzle generator for chess-based and chromatic mazes of varying levels of difficulty [1]. Evolutionary algorithms are well-suited to creating a large collection of unique puzzles because they can quickly locate many diverse optima in a complex function. The fitness function, which returns the value of puzzles in a generation, uses dynamic programming to calculate the minimum number of steps required to traverse the maze. Dynamic programming is a method for solving a complex problem by decomposing it into simpler sub-problems, and storing the solutions to those problems.

Both the chess and chromatic mazes are grid-based and have implicit barriers; players must figure out which tiles are safe according to the puzzle rules. Entry and exit squares are marked on the grids. Some tiles in the chess maze contain chess pieces and the rule is that the player may not traverse any tile that is occupied or covered according to the behavior of those chess pieces. The player is also assigned a chess piece, and so must move according to that agent type. In the chromatic puzzle a safe move consists of continuing onto a tile whose color is adjacent in the color wheel to the

tile the player is currently standing on.

These puzzles can work on paper or as part of a video game. For the latter, there would need to be some kind of repercussion for moving incorrectly or onto an illegal tile. Punishment can be used by video games to prevent brute forcing. The difficulty of both puzzles could be determined by the number of moves needed to travel from the entry to the exit.

The dynamic programming used in determining fitness puzzles works by traversing a network and recording the cost of arriving at each node. This search allows for computing the path with the minimum number of moves. During evolution, the aim is to maximize this number of moves. The evolutionary algorithm is straightforward; each generation, seven population members are randomly selected, and the two most fit out of those are picked for reproduction, which includes crossover and mutation.

Experiments had a zero rate of duplication of solutions, showing the diversity that can be achieved with an evolutionary algorithm. The size of the sample space aids in this, but for smaller sample spaces diversity promoting measures could easily be introduced. Ashlock succeeded in demonstrating the potential of evolutionary computation with a dynamic programming fitness function for generating puzzles [1]. By adjusting the dynamic programming code, the techniques described could be used for other types of puzzles.

### 2.2.5 Path-Building Puzzles

This type of puzzle requires the player to build a path from a point A to a point B using a number of provided items. The path could be built for an entity in the world to traverse, like an enemy AI; tubes, like the *BioShock (2007)* mini-games; a laser, like in some *Portal (2007)* levels; or for the player character themselves.

Path building is somewhat related to mazes but is differentiated mainly by the nature of the game environment; in a maze, the player cannot change the environment, only find the best path through it, while in a path-building puzzle, the objective is to

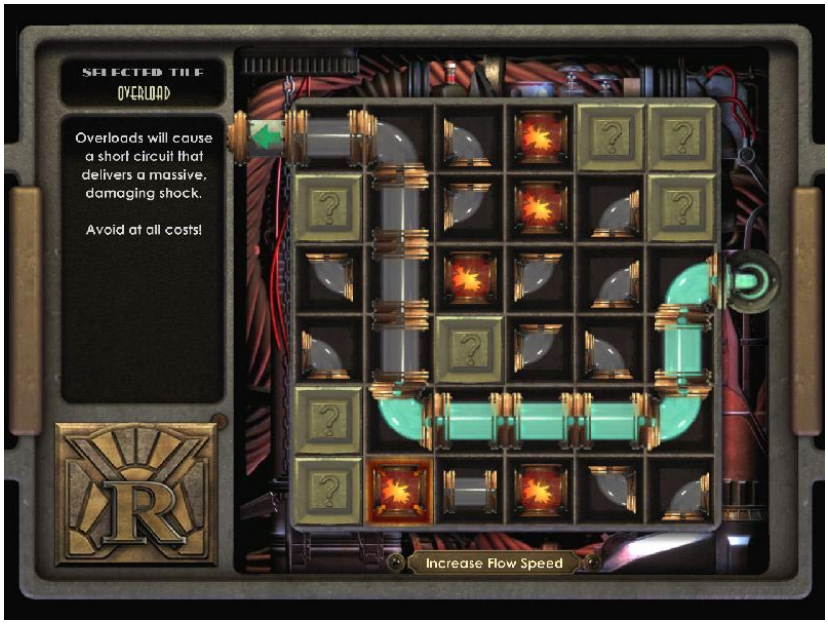


Figure 2.7: A screenshot from *BioShock (2007)*

create a new path by altering the environment at the hand of tools or items. For some of these puzzles the challenge stems from figuring out the correct placement of a limited number of items while in others there are more items than needed, i.e. decoy items, and the player must determine which ones are most suitable, in addition to placement. The latter usually lends itself to multiple possible solutions.

Smith et al. studied the problem of hard constraints in procedural generation, e.g. that a generated puzzle is necessarily solvable, in the context of the path-building game *Refraction* [5]. Many generators have not incorporated a way to guarantee that certain constraints, including aesthetic or complexity concerns, are satisfied in their output. In other words, the constraints require the game not only to be solvable, but solvable under certain prescribed conditions.

*Refraction* is an educational game that aims to teach proportional reasoning through a puzzle about re-directing laser beams towards a spaceships. The beams can be bent, split, etc using different components, the placement of which also trains spatial problem solving abilities. The game is set in a constrained space formed by asteroids walls.


Figure 2.8: A screenshot from *Refraction*

Smith et al. present two different implementations of three constraint-based artifact generation tools. The tools are for: mission generation, responsible for creating a general outline with possible level solutions; grid embedding, i.e. translating a mission to a geometric layout; and seeking alternative solutions to existing puzzle designs [5].

The overall level generator uses the structure defined by Dorman and Bakkes which distinguishes between missions and spaces; missions are the logical order of goals the player must accomplish, while spaces are the physical layouts of the levels [16]. This distinction is valid for many puzzle games, and may be especially useful for approaching puzzles set in 3D space, such as those in *The Talos Principle (2014)*. While the current version of *Refraction* is a 2D grid-based video game, the high-level formulation of the different components could lend themselves to a 3D adaptation - mission generation would not need to change.

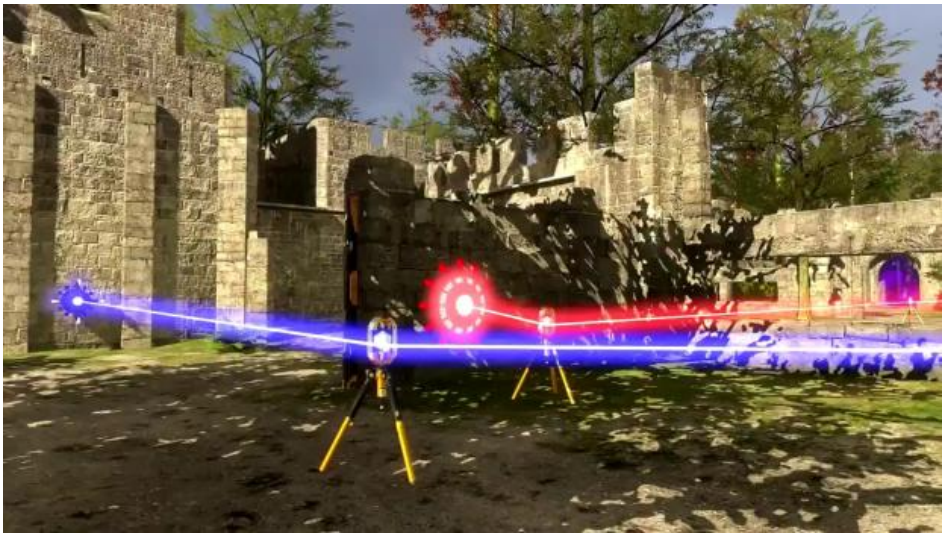

Figure 2.9: A screenshot from *The Talos Principle (2014)*, showing path-building with lasers.

Smith et al. made two implementations of each of the three tools; the initial implementation is based on constructive or complete-search techniques, such as bounded depth-first search, while the second, newer implementation is based on answer set programming (ASP), a form of declarative logic programming, with a Prolog-like syntax, that targets difficult search problems [5]. The ASP-based tools were designed using choice rules, deductive rules and integrity constraints. Smith et al. found that declarative languages can be a powerful expressive tool for reliable, controlled puzzle generators. Constraint-focused generator design can allow aesthetic failures to be treated the same as game play failure. Most other generators described in this paper have not reached the stage of taking aesthetics into account.

In a later study, motivated by the sequel *Refraction 2*, Smith et al. looked at constraining undesirable solutions, which they also refer to as shortcut solutions [17]. In path-building puzzles the addition of so-called distractor pieces can introduce alternative solutions. Since *Refraction* is an educational game, the designers require tight control over the puzzle progression; and thus easier alternative solutions are unacceptable. Even in non-educational games, unintended solutions can be a designer concern. Smith et al. added two extensions to the ASP approach developed in [5] in order to

solve the high-complexity problem that results from constrained formulations.

### 2.2.6 Packing Puzzles

A tiling puzzle is a packing problem; a number of shapes must be assembled into a larger shape, without overlap or gaps. Generally all the provided shapes must be used to achieve this. The process of fitting the pieces into the larger shape may be entirely based on shape-matching, or it could involve creating a picture out of the images printed on the pieces, as is the case for jigsaw puzzles.


Figure 2.10: A screenshot from *The Talos Principle (2014)*, showing a packing puzzle.

Unlocking new areas in *The Talos Principle* requires solving a tiling puzzle with tetromino pieces. *Tetris* is an overlap of a tiling and tile matching puzzle (see next section); the goal there is to assemble pieces into rectangles without overlap, and matching tiles into a line shape will cause them to disappear. *Tetris* is set apart from other tiling puzzles by its time pressure element, and the inability of the player to move pieces once they have been placed. Time pressure is a label that can be applied across categories, rather than being a definition on its own.

A famous example of this category is the *Eternity* puzzle created by Christopher Monckton in 1999. Composed of assembling 209 irregular polygons of the same color into a dodecagon, this puzzle is extremely difficult and came with a 1 million prize for

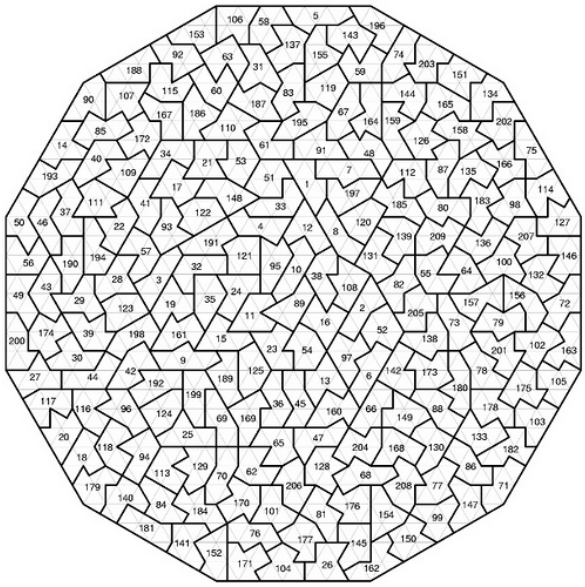

Figure 2.11: The solution to the *Eternity* puzzle, copyright 1999 by Christopher Monckton.

whoever could solve it in the first four years. The prize was awarded to two Cambridge mathematicians in October 2000.

### 2.2.7 Narrative Puzzles

This category represents puzzles that require the solver to make logical connections between objects and/or concepts. Players find solutions by exploring the environment and investigating ways in which objects can be manipulated and combined. These types of puzzles commonly function to progress a story, as indicated by the name of this category, and are thus a staple in many adventure games.

Adventure games have many sub-genres, including text adventures, point and click adventures and escape the room games. Not all of the puzzles in this broad range belong in the narrative puzzle category; adventure games could choose to include any of the puzzle types mentioned in this taxonomy. Good escape the room games tend to incorporate a variety of puzzle types into a common theme. However, in many adventure games, the player advances by solving puzzles set within a narrative.

The Puzzle-Dice system was developed by MIT media lab Singapore for the purpose of generating narrative puzzles [2]. Their motivation was to add re-playability to adventure games, which frequently only have one linear path, as well as develop puzzle generation tools that would be accessible to designers. To this end, the system focuses on the design problems associated with procedural generation. The use and development of the Puzzle-Dice system is demonstrated through two proof-of-concept games: *Symon* and *Stranded in Singapore*.

Some examples of narrative puzzle patterns, as outlined in [2], are:

- Figuring out which item a character desires, usually leading to a reward in exchange.
- Logically combining two objects to change their properties, or to create a new object.
- Disassembling an object into useful components.
- Saying the right thing to convince a character to provide aid.
- Acquiring a key to open a new area.

Logical associations are a common thread through these puzzles, and is also a cognitive skill in the closely related analogy and odd one out puzzle categories. The use of the label "narrative puzzle" could be debated because it is possible to switch out the specific puzzles without affecting the overall narrative - the focus of the research in [18]. However, the label has been used in past research and it fits when viewing story progression as centered on player action. A different take on a narrative puzzle is Framed, a puzzle game set up like a noir comic in which the player must rearrange the order of events to prevent the protagonist from getting caught or shot by the police.

*Symon* was an early prototype with a dream-like setting that allowed for fantastical logic, i.e. using something cold to change an object's color to blue. It was small in scope but succeeded at demonstrating the generation of a short, re-playable game. The initial system for *Symon* used a puzzle map that concatenates narrative puzzle patterns, into the structure of a game. Both this map and the possible puzzle patterns


Figure 2.12: A screenshot from *Stranded in Singapore* [2]

need to be specified by a designer; the generation process then inserts characters and objects into patterns, and patterns into the map, using a brute force approach. This grammar-based method of generation ensures that all puzzles will be solvable - the generation is constructive.

The full Puzzle-Dice system is more flexible than the initial one built for *Symon* and gives designers more control. It is based on puzzle building blocks that can be customized by designers; a modular approach that aims to allow for expansions. The puzzle map in this version is a map structure resulting from the combination of building blocks and defines the chronology of actions the players must undertake, in terms of dependencies. Building blocks can be either puzzles, which represent a single set of actions with input and output, or areas, the physical rooms in the game world.

Puzzle maps are paired with a database of designer-defined game items that have attributes and relationships to other items. The relationships are used to determine which items can fit into certain puzzle building blocks. For example, each item has a *madeby* property that links it to two other items in the database that must be combined

to create that item. Building blocks perform generation of narrative puzzles independently; given a desired output. The generation process is composed of three steps: generating the output item with the desired properties, generating inputs, themselves building blocks, and finally creating a relationship between the input and outputs - relevant to the type of building block. The possible relationships include: combine, property change, insertion, request and area connection.

The goal behind the research in [2] is threefold; they wanted to create a tool that guarantees solvability in its output, is accessible to designers, and is general enough to allow for a range of narrative-type puzzles. The tool is more like a framework than an out of the box puzzle generator as both the item database and puzzle map must be constructed by the designer.

Dart and Nelson worked on adventure-game puzzle generation using smart terrain causality chains [18]. They focused on creating a drop-in solution for the issue of re-playability: generating variations on puzzles that fit in the same place in an existing story line. In other words, this technique introduces re-playability without high-level narrative variation, and stands in contrast to the common branching story lines approach. The puzzles do change the low-level narrative significantly, and thus fit this category of the taxonomy.

The puzzle generation relies on a database of smart terrains items to create causality chains that form puzzle solutions. Smart terrain items, introduced by Will Wright, developed of *The Sims (2000)*, have a set of associated actions and properties that determine how they behave, and how the environment can affect them. In contrast to the Puzzle-Dice system, items are not aware of their specific relationships to other items. This also implies items have no single, specific rule, but can be used flexibly. Items actions can be either *active actions* or *passive actions*. The first occurs as the result of player input, i.e. picking an action from an action menu. The second occurs without input, e.g. heat causing nearby objects to increase their temperature.

Interactions between objects, that are not aware of each other, are executed through use of a physics simulation. Dart and Nelson used the Unity physics engine for this.

Objects can be modified through transitions, e.g. phase transitions such as melting or freezing, and can affect each other through direct collision or indirect energy, which is modeled in different forms [18].

A smart terrain causality chain (STCC) is a directed graph that defines dependencies between all the objects in the scene, and the objective of the puzzle [18]. The sequence of items represented by the STCC corresponds to the sequences of actions that the players must take to solve a puzzle. A list of possible actions for an item, along with the causes and effects, allows creating an STCC using a backwards chaining algorithm. The generation starts from a set of scene objectives and runs till it reaches primitive smart-terrain objects, which are then placed in the scene. This is a constructive process that guarantees at least one solution, and that each item in the scene is relevant for at least one solution.

Dart and Nelson tested their generation method in the adventure game *Space Dust*. Players have to replay this game several times to acquire all the information needed to win, and on each playthrough, had to solve different puzzles to progress. They discovered that 70 percent of players found the puzzle scenarios easier when there were more possible, parallel solutions, and most players said that longer causality chains corresponded to more difficult puzzles [18]. Their experiments also showed that players found the game engaging, despite the repetition in overall story.

Object placement in the physical game world is currently not automated, but could be added in by also storing semantic information about the environment. Another area of future work is actions/effects inference; similar to in the Puzzle-Dice system, all cause and effect information has to be manually specified by a designer, but it would be desirable to automate this process. Finally, the scope of procedural generation could be expanded to generate more than just the puzzles - for example, the environments, stories or items.

### 2.2.8 Search Puzzles

Search puzzles compose a common and easily-defined category. The solutions of these puzzles require the solver to search a physical space for a number of items. The most basic form of this kind of puzzle is a 2D or 3D area in which a number of objects have been discretely placed. In order to solve such a puzzle, the player just has to do what is referred to as "pixel hunting". It can be debated whether such a puzzle should even be classified as a puzzle, but they are included here due to their prevalence in games, especially adventure games. Search puzzles are one of the easiest ways in which to add basic interactions to narrative-driven games, e.g. in *Life is Strange (2015)*. Some search puzzles, such as those in *The Room* game series, and other games about escaping a room, require the player make logical connections about where to look for certain items.

### 2.2.9 Physics Puzzles

Players need to use games' physics to complete the puzzles belonging to this category. Physics in games is (most often) modeled after real-life physics, and so finding a solution requires players predict the physical outcomes of possible actions. These puzzles only exist in a video game format. Physics puzzles have an element of unpredictability as the game environment changes in real-time according to the laws of physics. No precise rules are programmed for the effects of player actions, making this a challenging puzzle type to generate in terms of solvability.

In contrast to many other puzzle types, the solutions to these puzzles may require precise timing and/or execution of actions on the player's part. In some cases this means the player has only a limited time window before an outcome becomes unachievable; for example, shooting a new portal while falling in a *Portal* puzzle room. This speaks to the executing of a puzzle solution; the solution itself can generally be described by a precise sequence of actions.

Shaker et al. focused on the generation of levels for physics-based puzzle games, using a clone of the mobile game *Cut The Rope* as a test ground [3]. The goal of this game is to make the candy drop in such a way that it reaches a frog monster placed

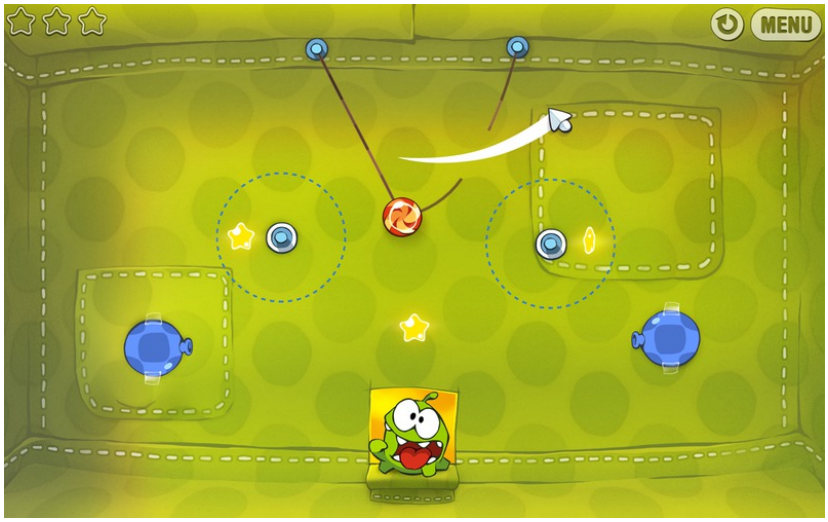


Figure 2.13: A screenshot from *Cut The Rope* [3]

at a fixed position. There are different game object that can be used to help change the movement direction of the candy, including: ropes, air cushions, bubbles, bumpers, and rockets. All these objects obey the laws of Newtonian physics, implemented in the CRUST 2D physics engine using XNA.

The game generator described in [3] evolves levels based on a context-free grammar, which is a set of recursive rewriting rules. Design grammars offer a concise way of describing a huge variety of possible level structures. Grammatical Evolution (GE) is a technique that combines an evolutionary algorithm with a grammatical representation. Shaker et al. started by analyzing the original levels for distance patterns of design - which can be represented in design grammars.

Levels, which are the phenotypes, are represented by lists of objects that can be placed anywhere in the map and may have some properties. As mentioned, the structure of a level is described by a design grammar, which is what is used to evolve levels. The GE process includes a genotype-to-phenotype mapping using production rules from the design grammar. This results in phenotypic programs that are syntactically correct - these programs are then evaluated for fitness.

A fitness function based on heuristic measures (based on prior knowledge about the game) is used to rate and consequently evolve levels. However, a heuristic alone is not sufficient to guarantee the playability of a level, so simulation-based playability test is done using an autonomous agent that acts randomly. The level is tested 10 times with random players to reach the final fitness score.

In later research, Shaker et al. replaced the agent with a more intelligent one in an effort to improve the quality of the generated levels [19]. They used two different rule sets for the agents; the first focused on the all game objects' properties and their placement in the level while the second contains only objects the candy can reach while in a given position, direction and velocity.

### 2.2.10 Time Manipulation

Time manipulation, or temporal puzzles, require the solver to use time as a puzzle solving mechanic (as mentioned in the introduction of this chapter). This could involve using a recording or rewinding mechanic to alter the linear time line. One of the puzzle solving tools provided in *The Talos Principle (2014)* is a recorder with which you can record some actions that can then be played back. This created a clone of the player as well as items the player moves during the recording.

*Braid (2008)* is perhaps the most well-known game to use time manipulation as a core mechanic. The puzzles in *Braid* are physical; the player takes on the role of a character navigating a platformer-style environment. Different worlds (sections) of the game have different time-based game mechanics including moving time forwards and backwards and at different speeds.

Procedural generation for a time manipulation puzzle would be challenging due to the need to define structures for parallel timelines and the non-linear causality relationships between events.

### 2.2.11 Logic Puzzles

Logic puzzles are solved through deductive reasoning; the player arrives at a solution through a series of deductions made from some given premises. The first logic puzzles appeared in Lewis Carroll's book *The Game Of Logic* [20] and were akin to syllogisms. Popular puzzles in this category are *Sudoku*, in which the player makes deductions about placements of numbers, and logic grid puzzles. The latter is so named because the solver is provided with a grid on which to fill in information obtained from clues.

*Shinro* is a Japanese logic puzzle in an 8x8 grid similar to *Sudoku*. To solve it, players must determine the locations of 12 stones using two types of clues; the number of stones located in each row and column, and arrows placed in the grid that point to at least one stone. As for other logic puzzles, the player uses elimination to reach one of two conclusions; a square must contain a stone, or it absolutely cannot contain a stone.

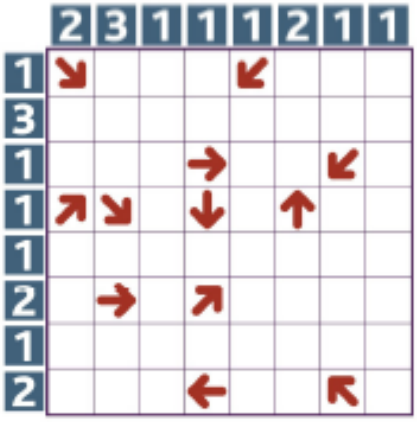


Figure 2.14: An example *Shinro* puzzle.

Oranchak used a genetic algorithm to automatically generate *Shinro* puzzles with desired qualities [21]. A genetic algorithm was also used by Mantere and Koljonen to generate *Sudoku* puzzles [22], which have a similarly large search space. Genetic algorithms fall into the generate and test category, where the fitness function represents the test. As such, the basis of the evolutionary algorithm is an automated solver that uses a list of deduction types to solve *Shinro* puzzles. Among others, deductions are made

based on the following observations: row or column count is satisfied, arrow points to only one free square, one stone remains to be placed and there is a horizontal or vertical arrow, and locations that can be excluded based on the pigeonhole principle. Some deductions are easier to make than others so it is reasonable to say that the difficulty of a *Shinro* puzzle, and a *Sudoku* puzzle, can be measured by the number of difficulty deductions (also referred to as moves).

The automated solver tries to solve the generated puzzles using a greedy search - it looks for easy moves first. During the search, the number of times each type of move is executed is tracked, and these numbers are then used to compute fitness, according to a variety of fitness functions.

The evolutionary algorithm is initialized with a small population of genomes because the fitness evaluation has a long run-time [21]. Genomes are matrices of integer values representing different grid square types. It is also possible to specify a target pattern which will restrict the possible values for some grid squares; a way for designers to influence the output of the generator. Tournament selection is applied to create a new population for each generation, but crossover was not implemented. The designers assumed it would have destructive effects on required *Shinro* characteristics, e.g. ensuring exactly 12 stones. Evolution occurs through mutation which probabilistically changes the values of randomly selected squares, and randomly decides whether to employ symmetry. When the algorithm stops, due to no more improvements in fitness, a brute force search is used to make sure the resulting puzzle only has one unique solution.

Oranchak found that optimizing puzzles according to the number of moves required to solve it did not necessarily lead to difficult puzzles. As mentioned, the important factor is more how challenging the moves themselves are, so concentrating on maximizing specific types of moves made more sense. However this is challenging due to the nature of the greedy search. Aesthetics such as symmetry can make a puzzle more appealing to a player - introducing the symmetry mutator greatly contributed to generating puzzle with symmetry in the stone and arrows positions. Overall, the genetic algorithm managed to effectively generate a variety of interesting and challenging puzzles.

### 2.2.12 Word Puzzles

This is a broad category that incorporates any puzzle that is built using words. Some puzzles only exist as word-based puzzles, e.g. riddles, crossword puzzles and anagrams, but many well-known word puzzle formats also have numerical or symbolic equivalents, e.g. analogies. Different sub-categories of word puzzles that are based on the semantic similarities, or dissimilarities, of words pairs have a lot in common. For example, odd one out and next in sequence word puzzles both require the player to discover the most plausible concept that relates a set of words [23].
Word puzzles have a long history on paper, but digital adaptations and novel types of these puzzles are now a popular category of mobile games. Procedural generation is useful for word puzzles due to specific word-related issues: new words get created, old words go out of common use, and existing words get new meaning [23].

Pinter et al. developed an automated world puzzle generator using: a corpus, an unstructured and un-annotated document collection; a topic model, which creates a topic dictionary from the input corpus; and a semantic similarity measure of word pairs [23]. This method would allow for domain specific puzzles, e.g. to fit into a narrative, by using a relevant corpus. Puzzles are produced by generating consistent sets of words and combining them with a weakly related, or unrelated, word.

Colton investigated three types of word puzzles: odd one out, analogy and next in sequence, also referred to as complete the sequence in this taxonomy [24]. These puzzles have a common characterization: each puzzles consists of a question, a set of choices including the answer, and an explanation. Like for riddles, the explanation refers to the single plausible solution. Finding a solution is generally based on player's previous knowledge about the characteristics of the objects, but obtained by systematically exploring the solution space.

Colton proposes that the difficulty of these puzzles can be influenced by the number of choices, the complexity of the solution concept, and the number of disguising concept - concepts that may look like a possible solution at first glance. He extended the HR theory formation system to generate puzzles [24]. A set of objects of interests with

characteristics were given as input, and production rules are used to generate a (solution) concept. The motivation behind using HR for puzzles generation is the ability of HR to form a new theory about a set of given objects. The negation and disjunction production rules were not used because they lead to concepts that are rarely found as a solution to puzzles.

The generating process is largely similar for the three different kinds of puzzles and while generally constructive, and check is required after generating each puzzle to ensure that the solution is the simplest one that exists. Colton's approach relies on highly structured data sets - which requires a lot of human annotation effort in comparison to the system in [23].

While the described generation methods output word puzzles, the words could be replaced by objects to place the puzzle in a 3D environment. This would only impose restrictions on the types of words that can be used as the choices.

### 2.2.13 Riddles

Riddles are puzzles that are solved by finding a plausible explanation for an unusual description, and require the solver to use lateral thinking. The solution is rarely immediately obvious, but should make sense to the solver upon its discovery. It is uncommon to see riddles in video games because parsing a natural language explanation is difficult for a computer. Exceptions are cases in which the answer to a riddle does not need to be stated but rather it gives players a hint as to an object they need or a direction they should travel in.

Riddles are difficult to generate because they often rely on a play of words - a difficult concept for computers. The authors of [25] generated riddles of the format "What is as hot as soup?", stringing several such comparisons together to make the riddle solvable. Only one comparison leads to too many possible answers, and would be impossible to solve without turning into a lengthy guessing game. The most well-designed riddles are based on nuanced meanings of words, and the generator in [25] does not possess that level of semantic power.

Galvan et al. used the Thesaurus Rex, a database of word associations which assigns words categories and attributes, according to their use in everyday language. Each category and attribute has an associated weight for each concept (word). The riddle generation process involves finding links between the target concept and other concepts using the categories and attributes. Difficulty of the riddles can be adjusted by changing the threshold of how similar concepts must be to be used as one of the comparisons.

The randomly generated riddles sometimes suffered from comparisons that did not add new information, and new comparisons that were contradictory to old comparisons due to the polysemy of some concepts. These issues feed into the problem of ambiguity and having one best plausible solution (a satisfying answer), a challenge that also exists for other categories of word puzzles.

Guerrero et al. developed a Twitter bot that generates riddles about celebrities and well-known characters by extracting information from sources such as Wikipedia zciteguerrero2015r. The riddles are composed of vague descriptions of the attributes of the person or character in question. Sometimes the riddles do include non-literal meanings of words.

### 2.2.14 Odd One Out

Puzzle characteristics: a question, a set of choices (including the answer), and an explanation which consists of a single, positively stated concept. Several papers look at these puzzles as word puzzles, but they also exist as symbolic or numeric puzzles.

### 2.2.15 Complete the Pattern

Patterns can be visual, maths or letter-based and are taught to the player by means of examples. In some cases this means that the player must comprehend an underlying system before having the knowledge to fill in a blank spot. This category has been discussed as a word puzzle, but as a visual puzzle it could easily lend itself to a 3D representation.

### 2.2.16 Analogies

Analogies can fall under the word puzzle category, but like the odd one out puzzles, analogies can also be symbol or number based. Some adventure game puzzles use an analogy structure for deduction type logical puzzles. Generation of this type of puzzle differs depending on the concepts being compared.

## 2.3 Summary

There is a trade off between generality of the solution and the amount of input required; more general solutions will mean that more work will be required by the designer when using the generation tool. However, most generators are based on the principle that the initial high cost for designers, i.e. defining the rules and inputs of a generator, are outweighed by the benefits of then being able to quickly generate a large variety of puzzles.

The most important challenge in puzzle generation is: every generated puzzle must be solvable. However, guaranteeing at least one solution is not synonymous to a good quality puzzle. Generally puzzles should not have too many or undesired solutions, nor should they be littered with pointless items and/or information. The issues that exist for puzzle generation are directly linked to the characteristics of good puzzle design, and thus vary for each puzzle type.

Controlling difficulty is a major topic in puzzle generation that comes up in much of the past research. A comprehensive generation tool should be able to return puzzles of varying levels of difficulty. The main challenge associated with this is often in the development of a good evaluation metric; for many puzzle types it is difficult to concisely define the factors of difficulty.

# Chapter 3

# Design

This chapter details the design of a system for procedurally generating narrative puzzles. As outlined in Chapter 1, the specific goals for this system are: the creation of solvable puzzles, expressive power, dynamic difficulty, the ability to integrate with existing games, and usability. After a brief overview of the core concepts of the system, the inputs to the generator are described in detail. Following that, the algorithm for puzzle generation is presented. Finally, the in-game presentation and solving of the created puzzles is discussed.

## 3.1 Core Concepts

The developed system for generating narrative puzzles is based on a context-free type grammar that defines possible behaviors of game items. The puzzle generator integrates with a game world to create puzzles on the fly based on the current state of the world.

There are three components that feed into the generator: a database of all items that can be used in puzzles, a collection of grammar rules that describe the space of all possible puzzles, and a comprehensive list of the game areas. Several core concepts form the basis of these component; they are briefly define in the list below:

- Items: Correspond to game objects and are defined by their types and an optional set of properties.

- Properties: Have a name, which identifies them, a value type, and a value.
- Rules: Composed of an output term, a set of by-product terms, an action and a set of input terms.
- Terms: The main units out of which rules are composed, each is defined by a single type and an optional list of properties.
- Action: Describes the player's action in carrying out a rule.
- Area: A single connected space that forms part of the game world; used to compartmentalize the puzzle generation.

The structure of these components is flexible in terms of the designer-defined content it can support, allowing the generator to be applied to a range of game types. This chapter will refer to a potential user of this system as the game designer.

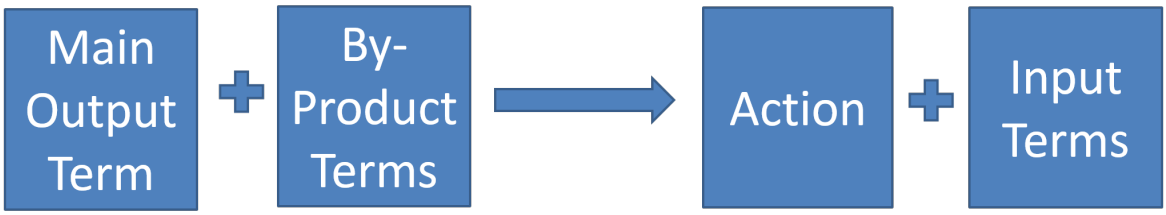


Figure 3.1: Abstract representation of the general structure of a rule.

The generator uses the set of production rules that constitute the grammar in a left to right direction to generate a puzzle backwards from an end goal, a process that ensures the puzzle is solvable. In a game that incorporates the generated puzzles, the same rules, but used in the right to left direction, function as game logic.

The inspiration for this puzzle generator came from the Puzzle-Dice system, which was described in Chapter 2. The design of this generation system attempts to improve upon the Puzzle-Dice system, specifically in terms of expressivity, usability and scalability, while maintaining the guarantee of solvability.

## 3.2 The Puzzle Items

A puzzle item is a conceptual representation of a tangible object that can be used as part of a puzzle generated by the puzzle generator. Each puzzle item has a unique name, which is also referred to as the item's specific type, an optional list of properties, a description to show in-game, and an associated prefab. There may be more than one puzzle item for an object that is conceptually the same but has multiple states; for example a tree in summer and that same tree in autumn. However, these specificities in item definitions are left open to the game designer.

Item properties are defined by their name and type; the type determines the legal values for the property. There are three types: string properties, boolean properties and integer properties. Properties are freely defined by the game designer, and as such they can be tailored to the needs of the puzzle genre. There are no required properties; if a property is not defined for an item, the generator assumes it does not have this property. In the case of a boolean, this is equivalent to assigning the value false for a property, e.g. not specifying the *carryable* property as equivalent to marking an item as "not carryable". Properties are compared based on equality between their names and values.

There are however several special properties which have explicit logic attached to them. One of these is the aforementioned *carryable* property; an item is queried in-game for this specific property to determine whether it can be added to a player's inventory.

Another special property is the *isa* property, which can be used to define all the categories a certain item belongs to; for example, a *PineTree* might have the *isa* properties *Tree* and *Plant*. Generally the property values correspond to the super-classes or types associated with an item. The name of an item is also considered to be an *isa* property of that item, as it defines what an item *is*. In other words, an item's name defines the most specific category it belongs to.

The value associated with an *isa* property may or may not be the name of another

item in the puzzle items database. Often they are not because the defined items represent distinct instances of objects, whereas the *isa* properties are likely to be generic types, e.g. *Tree*. Every puzzle item is automatically considered to be a sub-type of the type *Item* (so this does not need to be explicitly define). Notably the *isa* property allows for the creation of hierarchies among puzzle items. As will be seen in the next section, this hierarchy is central to the functioning of the grammar rules.

The *contains* property is technically a string property but logically it refers to an item that is contained inside the item for which it is defined. This property is not frequently used for puzzle items because most containers can exist independently in the game world and may contain many different types of items. For example, a treasure chest should not be defined as containing gold, because it could also contain other items, or none at all. As will discussed in the next section, the contains property is more important for the rules, which can refer to transient item states.

There are two special properties that can be used to restrict the possible locations of puzzle items. The *notSpawnable* property, which is defined in the negative so it can be left out by default, indicates that a puzzle item can only be used if it is already part of the game world, or if it is inside another item. The *area* property can be used to specify the legal areas for spawning and/or using an item, allowing the game designer to control which items may be included as part of a puzzle on a per area basis. For example, a lake should probably not be considered as a possible puzzle item when generating a puzzle for an indoor area. Multiple *area* property entries result in multiple legal locations.

## 3.3 The Grammar

The grammar, which comprises of a set of production rules, describes the space of all possible puzzles. Each rule describes a relationship between a set of inputs and a set of outputs. The format of the rules is loosely based on the format of rules that make up a context-free grammar. The grammar rules serve a dual purpose; they are used by the generator to create puzzles, and they are executed in-game as a result of player actions. The second purpose is a mirror of the first as it pertains to the player using

the rules in the opposite direction (as during generation) to solve the created puzzle.

The general format of a rule is as follows:

$$itemType[properties_{0...n}]_{1...n} ::= action\ itemType[properties_{0...n}]_{1...n} \tag{3.1}$$

In a context-free grammar all the productions are one-to-one, one-to-many or one-to-none, and for the purposes of generation, this holds true for the puzzle grammar. Production rules are read from left to right, and can be seen as describing breaking down an output into its input(s). A puzzle is created by iteratively decomposing outputs, starting from an end goal, as described in the Puzzle Generation section further on in this chapter.

In practice, for this application, the rules can, and often do, have multiple outputs because the right and left hand sides of the rule describes which items exist, and in what state, before and after the rule is applied, respectively. For generating a puzzle, in the form of a tree structure, only the first output is important, the others are considered to be by-products. The by-products are items that were not destroyed by the application of the rule, but which are also not the main outcome. For example, in rule 3.2, that expresses chopping down a tree, the axe is not an outcome, but it is important to account for the fact that it was not consumed as part of the execution of the rule. Overall, each input, or right-hand side, term is considered to be destroyed if it does not appear as an output, or left-hand side, term. The exception to this is container behavior; inputs that appear as the value of the *contains* property for an output are additionally not considered destroyed; rule 3.3 shows an example of this type of behavior.

$$TreeStump\ Axe ::= ChopDown\ Tree\ Axe \tag{3.2}$$

$$Container[contains : Eggs] ::= Gather\ Eggs\ Container \tag{3.3}$$

The output of the each rule is a list of terms, while the input is composed of an action and at least one term, as shown in Figure 3.1. Terms represent the non-terminals of the grammar while an action is a terminal that refers to what the player does to execute the rule. The left hand side of a production rule is always a non-terminal in a context-free grammar, which holds true for these rules. While the terms represent the non-terminals of the grammar, the puzzle items, described in the previous section, represent the terminals. In this way, terms can be seen as boxes with descriptions of what kind of puzzle item could be placed inside.

There is theoretically no limit on the number of inputs, not including the action, of which there is always only one, and outputs, those that are by-products. However, typically the number of inputs would be one or two. The rules should always have at least one output (the main output) and one input. Normally a context-free grammar rule can be one-to-none, as stated, but that is not applicable here as it would be equivalent to an item created out of nothing. Rules need some form of input state to check; if that state is blank it will always be satisfied and thus a rule would keep executing indefinitely. Rule execution is discussed in depth in the Puzzle Solving section of this chapter.

The terminals (puzzle items) are not directly used in the authoring of the grammar rules; a designer only looks at linking terms (non-terminals) to other terms. Internally, the puzzle generation system contains logic for which non-terminals may be replaced by terminals from the database. Some of the terms can be matched directly to puzzle items while others must first be broken down to other terms according to the production rules. While a designer only makes use of non-terminals in writing the rules, they should be aware of the need to include non-terminals that can be matched to terminals. The grammar is only valid if each input term can be matched to at least one output term in a different rule, or at least one puzzle item.

Terms are composed of an item type and an optional list of properties; this makes up a description of possible puzzle items that could be matched to it. The term's item type corresponds to the previously described *isa* property, which includes item names. The type can be specific, such as *PineTree* or general, like *Plant*. The more general the type, the more puzzle items have the potential to be matched to a term, and the more

different puzzles may be generated using that rule. Since any puzzle item is a sub-type of the special type *Item*, that type could be used for terms that could be filled in by any item.

The properties associated with a term are fundamentally the same as those for a puzzle items. For an item to match a term it must be of the same type or a sub-type as the term's type, and it must include all properties of the term. A puzzle item can of course have many more properties than those required by the term.

Terms cannot have *isa* properties because they are defined by a single type. They also would not have *area* properties because area considerations are not taken into account by the rules; the rules run at a lower level in the generator than the per area progression. Generally, it can be said that the properties associated with a term are those that are mutable, e.g. temperature is mutable while the fact that a pine tree is a tree is not. However some generally immutable properties, such as *carryable* may also be used to specify a term that can match any carryable item.

As discussed previously, the *contains* property is needed to represent scenarios in which items can be inserted into containers, and later removed again. Container insertion is important for making items that are not carryable, e.g. water, carryable. This property is also the only way to merge two items temporarily; this is important in-game where two inputs leading to one output would otherwise destroy one of the inputs.

Besides inputs and outputs each rule must also have an action, which can be considered a terminal. This action is only used as part of the second purpose of the rules, as game logic; it has no bearing on the puzzle generation. Mainly the action is a designer-targeted feature; it allows designers to specify a specific player action that must be performed to execute a rule. The actions that are available for an item can be displayed in-game as an action menu, but this is not the only possibility. For example, a designer could also use the action field to trigger a certain animation.

The action appears as the first symbol on the right hand side. It is associated with the first input non-terminal, and thus only with the item that is matched to that

non-terminal in the generation process. As such, it is important to consider the order of the inputs; for example in rule 3.2, the action *ChopDown* should appear attached to the *Tree* term, rather than the *Axe* term.

## 3.4 The Puzzle Areas

Each puzzle area corresponds to a connected area in the game world and must have an associated goal. The goal is used by the generator as the starting point for generating a puzzle for that area.

A designer can associate multiple possible goals with each area in order to increase the possibility space of puzzles that can be generated for that area. In the current design, only one goal is chosen, and thus one puzzle is generated per area. In this context, puzzle refers to all of the actions a player must execute to reach the area goal; a single puzzle has no prescribed size or difficulty.

The format of an area goal is the same as that of a single term in a rule of the grammar. Each goal specifies a type of item that must be obtained, and an optional list of properties that must be fulfilled for that item. The generator checks that the goal cannot be satisfied by any intermediate items that are chosen as part of the puzzle, as this would practically result in a player pre-maturely completing a puzzle.

Besides a goal, a puzzle area has a unique name, a list of connected areas, and maximum puzzle depth. The maximum depth refers to the depth of the tree structure representation of the puzzle that is created by the generator; discussed in the next section.

Puzzle areas can be predefined, or in the case of a procedurally generated game, they could also be automatically defined at run-time based on environmental attributes. The player's current in-game area is tracked by the generator and a puzzle solving manager that tracks the player's solving process. The generator needs this information to spawn puzzle items, while the puzzle manager makes area appropriate rules available and checks whether the goal for an area has been completed.

## 3.5 Puzzle Generation

The puzzle generator works by recursively generating inputs for outputs using the set of rules that make up the puzzle grammar. The rules are used in the left to right direction, as production rules, and do not take into account the by-product terms. Puzzle generation is done live, that is, while the game is being played, on the basis of currently accessible areas. This means that at a high level generation is running forwards throughout the game, but at a lower level, i.e. per area, generation runs backwards. This forward-backwards combination ensures solvability and quality, through lack or repetition, for the generation puzzles.

At the start of the game, a puzzle is generated for the area that has been designated as the start area. Finishing a puzzle for one area, i.e. achieving the area's goal, causes all its connected areas to become unlocked, and triggers the generation of puzzles for those newly available areas. The forwards part of the algorithm can branch off into different tracks depending on the specified connections between areas. The system maintains each of the available areas, along with their puzzles, so multiple puzzles, can be in progress at the same time.

The general forward direction of generation for the areas is important because it prevents unsolvable puzzles resulting from inaccessibility to needed items. Specifically, the scenario in which an item that is needed to solve a puzzle for one area is locked off in a next area, that cannot be opened until the puzzle in the current area is complete.

Per puzzle, the algorithm starts by finding a rule with a left hand side term that matches the current area's goal; the area goal is analogous to the grammar's start symbol. From that starting rule, it continues trying to substitute right hand side terms for other terms until no suitable rule can be found to perform such a substitution, or the area's depth limit is reached. At that point, the generator adds the puzzle item (terminal) that matches the last term to the game world, if it is not there already.

The rules used for the substitutions are recursively chained together into a tree structure that defines the entirety of the created puzzles.The items spawned in the

world correspond to the inputs for the rules that make up the leaves of that tree. One tree is created per area and is used to track a player's progress in solving the puzzle represented by the tree.

An example of a generated puzzle is given in figure 3.2, followed by the rules that would be chained together to create that puzzle. Practically it is the rules that make up each node of the tree, rather than the terms, but the terms make for a clearer representation of the structure. The narrative of this example is as follows: first the player must assemble a disguise out of glasses and a fake moustache and set of a car alarm to distract the security guard; these events can happen in either order. Then the player can steal the distracted security's badge, and proceed to unlock the safe with it. Finally, once the safe is unlocked, the player can open it and access the desired gold.

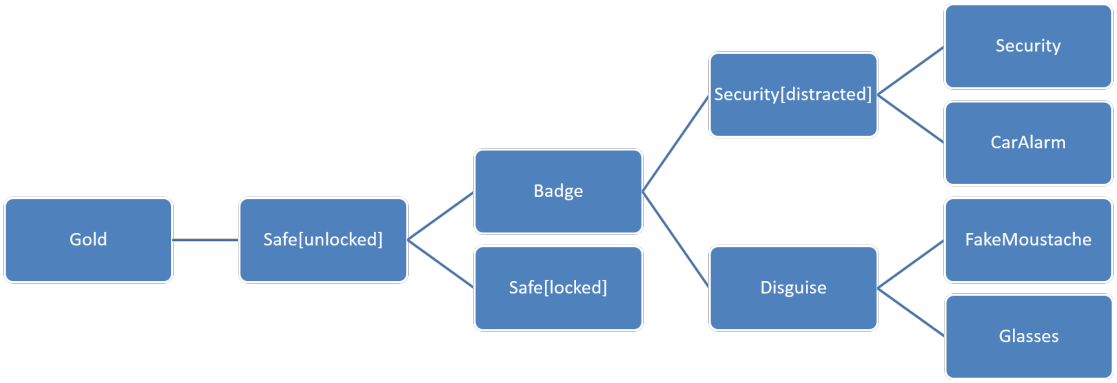


Figure 3.2: An example puzzle tree.

$$Gold\ Safe ::= Open\ Safe[locked : False] \tag{3.4}$$

$$Safe[locked : False]\ Badge ::= Unlock\ Safe[locked : True]\ Badge \tag{3.5}$$

$$Badge\ Security ::= Steal\ Security[distracted : True]\ Disguise \tag{3.6}$$

$$Security[distracted : True] ::= Trigger\ CarAlarm\ Security[distracted : False] \tag{3.7}$$

$$Disguise ::= CreateDisguise\ Glasses\ FakeMoustache \tag{3.8}$$

### 3.5.1 Matching Terms

Terms can be matched to other terms according to their types and properties. The properties must be an exact match, neither term can have properties not present for the other. While finding an appropriate output term for an input term, the type of the output term must be the same or more general than the type of the input term. For example, an input term of type *Tree* could be replaced by a rule with an output term of type *Tree* or *Plant* but not by one with an output term of type *PineTree*.

An important to note design choice is that the generation algorithm does not wait until it reaches a terminal to pick a matching puzzle items for a term. In fact, it attempts to find a matching item to assign to a term as early as possible. The reason for this is that it allows for the use of more specific rules, widening the scope of possible puzzles. Puzzle items are more specific than terms by definition, so once a term becomes more specific as a result of an attached item, it can be matched to more output terms in other rules. If a term has an assigned item, that item's type will override the term's type. For example, a rule with an input term with type *Tree*, as in the previous example, might pick a *PineTree* item as the matching puzzle item and change its type accordingly.

Of course terms cannot always be matched right away, due to no puzzle items having the properties of the term. However, the type of a term should always correspond to a type that exists for puzzle items in the database; otherwise a match will never be possible. This type check is performed for every term in the used rules; if it fails, that means the grammar is invalid.

When an item match is found for a term, that item is passed up the tree to previously visited rules, and attached to corresponding terms. In this way, each term in each rule that makes up the puzzle tree structure will have an associated puzzle item when generation completes. These puzzle item references can be used during puzzle solving to know which new items to spawn as the result of the execution of a rule.

### 3.5.2 Generation per Game Area

The game areas are modular but conscious of their context. New puzzles are created on a per area basis, with the generation algorithm being aware of all currently accessible areas, all items currently in the world, and all items in the player's inventory. Anytime an item is chosen for a term, the generator ensures that the chosen item is accessible. This check makes use of the items' aforementioned *area* and *notSpawnable* properties. If a terminal item already exists in the world it is not added to the world again, unless multiple copies of one item are required by the leaf rules.

Additionally, the generation algorithm will terminate upon reaching an intermediate puzzle item that already exists in the world as the result of a puzzle generated for a previously traversed area. This early termination prevents recreating a puzzle that the player has already solved, or creating a futile puzzle because the player already has the item he/she would be trying to make.

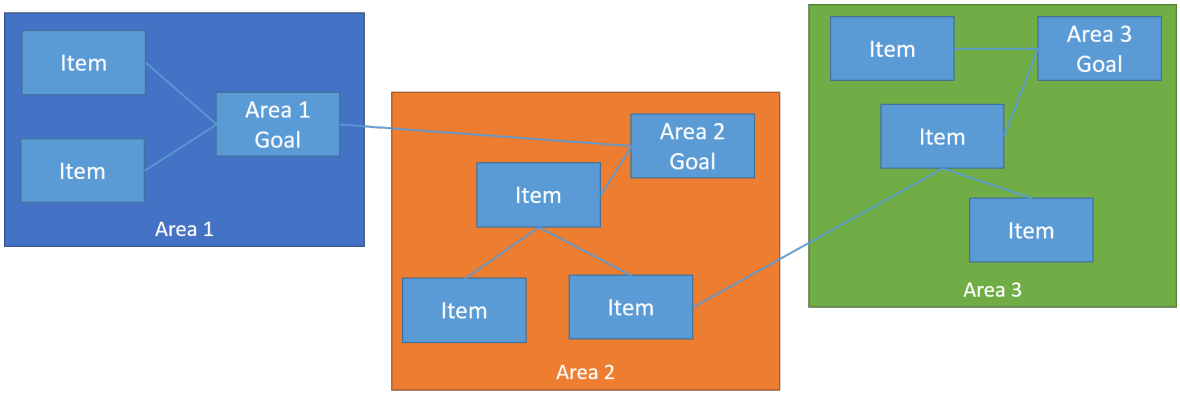


Figure 3.3: A layout of how puzzles in different game areas can be interconnected.

Figure 3.3 shows how puzzles in each area can re-use items from previously visited areas. For example, the goal for area 1 is re-used as one of the input items needed to acquire the goal for area 2, and one of the items from area 2, which is assumed to not be destroyed in this example, can be re-used as an input to a puzzle in area 3. Puzzles are generated per area in a linear order for this example, with the puzzles for area 2 being created after the goal for area 1 has been achieved. Within each area generation

runs backwards from its respective goal, but takes into account the items in the areas before it.

Unlike the Puzzle-Dice system, which also terminates generation with a list of items to be spawned, this system does not make the assumption that the world is empty at the start. It is possible for existing objects in the scene to be included in the puzzles, if they have components that identify them as puzzle items.

This is an important design choice for integrating puzzles into an environment. Puzzle items could correspond to environmental phenomenon, such as a lake, or large static structures, which are more logically placed in the game world as part of scene design. Allowing for the use of pre-existing objects as puzzle items gives the designer freedom in the construction of the game world.

One reason for this choice came from considering the application of this puzzle generator in a game with a procedurally generated environment, such as *Minecraft (2011)* or *No Man's Sky (2016)*. In such a game, the puzzle generator could run as a separate layer on top of the existing generator if the puzzles are constructed partially from already spawned items such as environmental phenomena and NPCs. Typically these types of items would correspond to those found at the leaves of the puzzle structure; it is rare that a lake or an NPC could be broken down into elementary parts.

The puzzle generator also tracks the depth of the tree that represents the current puzzle; this prevents a possible infinite loops scenario, and allows for designer influence on the length of the sequence of actions needed to solve a puzzle in a given area. As stated in the previous section, the designer can specify the max depth on an area by area basis. The number of actions needed to solve a puzzle is also determined by the breadth of the tree but due to a low average branching factor (most rules will have one or two inputs), depth influences the length of the solution sequence more than breadth.

## 3.6 Puzzle Solving

Next to puzzle generation, the grammar rules also provide the in-game logic that allows a player to solve a generated puzzle. For this purpose the rules are used from right to left; the inputs on the right hand side must be satisfied in order to produce the output(s) on the left hand side. Inputs are satisfied when they are co-located and have all of the appropriate properties. Co-location can be achieved through use of an inventory system, for NPCs it is automatically achieved when they are in the same game area.

When the inputs for a rule are satisfied, the action to execute that rule is provided to the player. Only when the player chooses that action is the rule actually executed, i.e. are its inputs replaced by its outputs. While the generator only really looked at the first output, each output is equally important in-game because they indicate which items should be created and/or destroyed.

Two basic rules are added by default for puzzle solving; the "pick up" and "put down" rules. These, self-explanatory, rules allow a player to pick up, and put down, any item that has a carryable property set to true. These rules are not needed during generation because the use of accessible areas and containers ensures that it will be possible to co-locate items in-game.

Primarily, the grammar is used to define logic that can be combined to form a narrative puzzle. However, additional rules can be defined to express possible fail states, i.e. scenarios in which the player fails to solve the puzzle and must start over (either with the same puzzle or a newly generated one). The outputs of these rules would correspond to game over scenarios. For example, the player may pick the action Eat on a mushroom that is poisonous, and die as a result. These rules would not be used for puzzle generation, as their outputs would, by definition, represent incorrect solutions.

# Chapter 4

# Implementation

The described system was implemented for the Unity game engine, and can be used in-engine as a tool to add generated narrative puzzles to a wide variety of games. Puzzle items, grammar rules and game areas are all implemented as different Unity assets, through defining them as Scriptable Objects, which can be created and edited using custom Unity editor windows. Assets can also be edited using the built-in Unity inspector, but these are tedious to use, so the custom editors were created with the aim of making the puzzle generation system user-friendly. Implementing these to be assets means that they can be easily reused between different projects.

The collections of the three asset types are managed by a database class that provides functions to operate on the entire collection. The database class is generic, with a specialized version that provides some specific functionality used for each asset type. The provided functions are all static so no instance of the database object has to be created, and the assets can be accessed anywhere in the program.

A small demo game was built as a proof of concept for the puzzle generation system. The code for the puzzle generation part of the game is mostly self-contained, but does need to interface with the player character. The system was integrated with small pre-designed game world.

This chapter describes the implementation of the three components that feed into

the generator, before delving into the generator class itself. Overall, the puzzle generation for a game is managed by the puzzle manager, whose functionality is explained in the last section of this chapter.

## 4.1 The Puzzle Items

Puzzle item assets are implemented using the `PuzzleItem` class which have fields for the item's name, prefab, user-friendly description and list of properties. The `Property` class is used to define properties across all of the different assets, i.e. for the puzzle items but also for the terms of the grammar rules and the goals of the puzzle areas. Property matches are endeavored between each of those at different stages in the puzzle generation and solving processes.

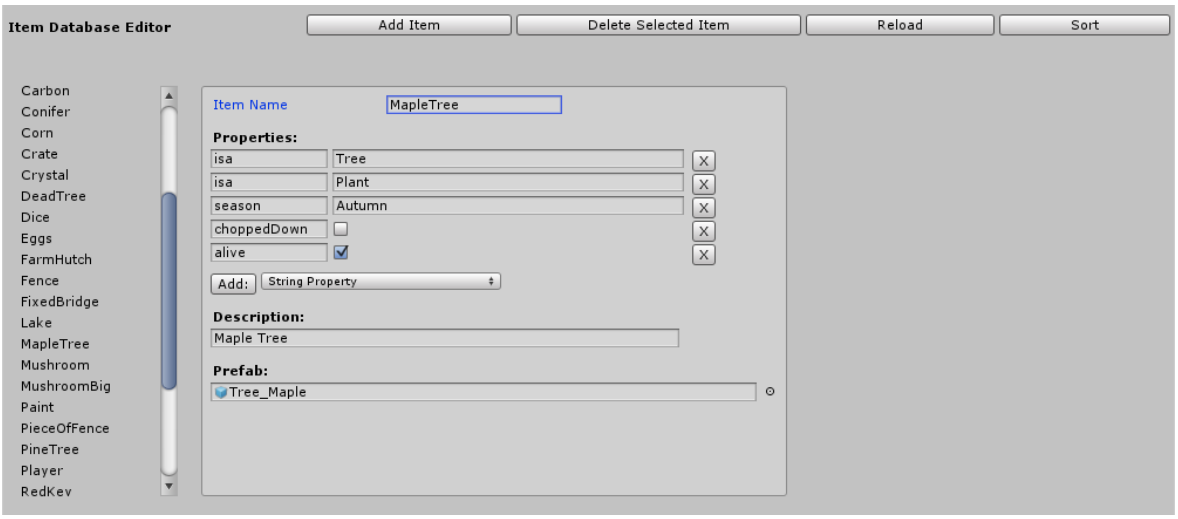


Figure 4.1: The custom editor used for creating and editing puzzle items.

The logical representation of a puzzle item is separate from its game object representation. The properties of a puzzle item do not change unless the asset is edited directly; that is, they cannot be changed in-game as a result of puzzle solving. Instead, a copy of the puzzle item, referred to as a game item, is used for this purpose. Specifically, the game item creates a copy of all of the properties of the puzzle item; these will be its properties when it is spawned in the world, but those properties could be

altered as the result of player actions.

The game item stores a reference to the puzzle item it is derived from, this is particularly important when checking if a copy of a certain puzzle item already exists in a given scene. For example, the generator may determine that one of the terminals is a tree item, but if the accessible areas already contain at least one tree, it is not necessary to spawn another one. The generator determines which items already exist by searching the accessible areas for game objects with the `GameItem` component, and checking which puzzle item is referenced by that component.

Each property name is identified by a string; the designer must enter in two identical strings if he/she wishes to refer to the same property for two different puzzle items, or terms. While this leaves room to some human error, it does allow for a quick work flow when defining a broad range of properties. Depending on the game being created, properties may or may not be frequently re-used between items. All property types are saved as strings for the purposes of serialization, but they are converted back and forth for the editor interface, so this is not noticeable from a user point of view.

As mentioned when discussing the design of the grammar, the action of a rule pertains to the first input term. Each game item queries the puzzle manager for currently available rules in which it appears as the first output. As such, the game item knows which actions could be performed on it. The criteria used by the puzzle manager for matching a game item (or puzzle item) to a rule input is item type; properties are only checked by the game item itself, as it contains the logic for determining whether the inputs for a rule have been fulfilled.

When a player hovers over an item, the item's description is displayed. Upon clicking on the item, an action menu pops up with all available actions for that item. An action is made available when the input criteria for a rule have been satisfied. This means that the game item must have all the properties (if any) required by the input term it matches, and any other input items must also be fulfilled, as well as be co-located, i.e. in the player's inventory, or in the same area in the case of NPCs. Game items also carry out the execution of a rule, but that will be described in the Puzzle

Solving section.

In order to implement the *contains* property for an item, the GameItem class has a field in which it can stored a reference to a contained game item. This field is used to keep track of a game object while it is inside a container, so that it can later be removed again.

## 4.2 The Grammar

The grammar rules are implemented in an object-oriented fashion; each rule and term corresponds to a C# class. A rule object contains a list of output terms and a list of input terms, as well as fields for referencing parent and children rules, which are used during puzzle generation.

The screenshot below shows how a rule can be created using the custom rule asset editor. At the top, a string representation of the rule is displayed to the designer as a quick overview. Each rule in the list can be expanded, as the one in the screenshot, in order for it to be edited. Outputs and inputs terms, and their associated properties can be added and deleted using the appropriate buttons. The action should be written in plain English as it is what gets displayed to the user. The rule in the example describes obtaining water for a water source, which could be a well, lake, etc. by using a container. The output is that container filled with water, as well as the water source as a by product because it would not be destroyed as the result of this action.

For this action to appear as a possibility to the player, when clicking on the water source, the player would need to have an item of the type *Container* in their inventory. After executing this action, the container inventory item will update to indicate that it now contains water.

During generation only the first output term, i.e. the *Container* term, would be considered. The rule could be used to break down any rule that had an input term of the same format as that *Container* term.

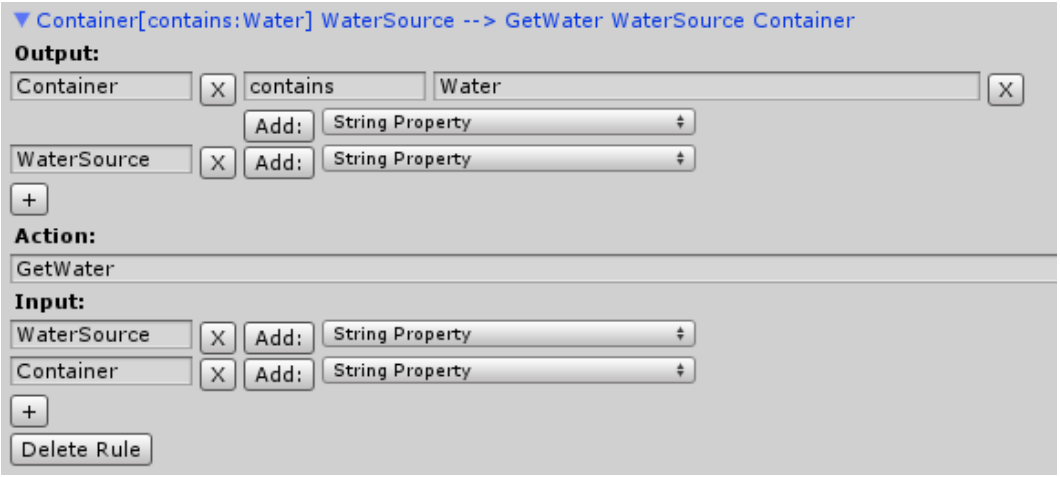


Figure 4.2: The custom editor used for creating and editing rules.

The main output of a rule should be different than either of the inputs, either by type or by property. If any of the inputs matched the output exactly, the rule could be infinitely chained together (practically this would be prevented by the depth limit), which is probably not a desired behavior. Each rule should affect the state of the game world in some way, and progress the player's journey. If the rule's main output matched one of its inputs, this essential component of narrative puzzles, for which the grammar is designed, would not be satisfied.

While the structure of a rule is strictly defined in terms of a main output, output by-products, and inputs, the terms can be flexibly defined. The open-endedness of term type and property definitions means that the rules can express a wide variety of behaviors. While the term type of the main output and the inputs must match what can be found in the database, as discussed in the design, the by-products can be used for any type of side effect. The values of the terms of a rule can be queried in game at the time of rule execution, so the game designer could extend the behavior of the rules to trigger other events. For example, one of the by-products might be a timed explosion, or a phrase uttered by one of the game's NPCs.

## 4.3 The Areas

The game areas are unique assets like the puzzle items and the grammar rules and can be similarly defined using a custom editor, of which a screenshot is shown below. Puzzle areas use the Term object to define their goal(s). As a result, area goals can be easily passed into the generator as the starting term for which to generate inputs.

For the example area shown in the screenshot, a puzzle will be generated that leads to a tree stump, meaning that the player will have to chop down a tree. The designer could choose to explicitly tell the player this goal, or provide a hint to steer in the player in the general direction of the puzzle solution.

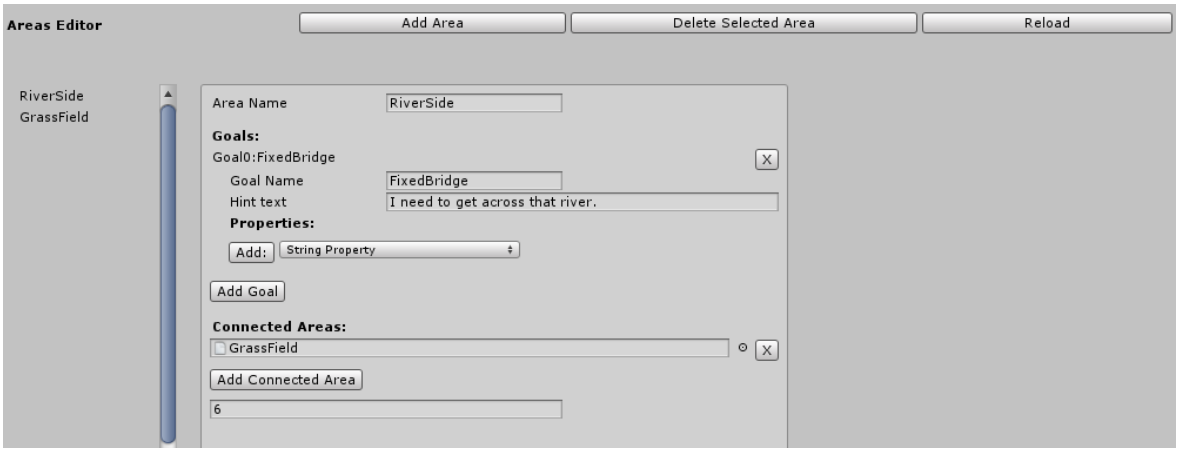


Figure 4.3: The custom editor used for creating and editing puzzle areas.

Similarly to the way in which puzzle items correspond to game item components, the puzzle areas correspond to game area components that can be attached to transforms in the scene. The game area script is mainly used by the generator to find appropriate spawn points for the terminal puzzle items. All items in the world that are, or could be, part of a puzzle for a given area should be children of the game object that has a game area component linking to that area. In this way, each game item knows what area it belongs to, and can pass that information back to the puzzle manager when asking it for rules.

## 4.4 Puzzle Generation

The puzzle generation algorithm is mainly comprised of a recursive method called `GenerateInputs()` that takes in a term and tries to find a rule to produce inputs for that term. In addition to a term, this method also takes in a parent rule, i.e. the production rule used to obtain the term, the current depth and area, all other accessible areas, and all (accessible) items that currently exist in the world. The first time this method is called for a certain area, it is prefaced by a search for all game items in the scene. The first term that is passed in, analogous to a start state of the grammar is the area goal, and the first rule is a blank rule that is the root of the tree structure.

When trying to generate inputs for a term, the first step is a search in the item database for all puzzle items that match the term. As mentioned in the Design chapter, an attempt is made to find a puzzle item match for a term as early as possible. For a puzzle item to match a term it must have the same type or a sub-type; this check is implemented by looking at the *isa* properties for each item and determining whether the value of one of those properties is that same as the term type. If this check succeeds, each term property, if there are any, must be found as a property for that puzzle item. Additionally, the system must determine whether the possible match is actually accessible; for that to be the case the item must either be spawnable and legal for the current area, based on the *area* property, or it must already exist in the scene. If no *area* property is found for an item, the system assumes that it can be legally placed anywhere.

If multiple matching items are found, one is chosen at random and assigned to the puzzle item field in the term object. If the chosen item is already found in the same, the `GenerateInputs()` method returns early. If no matches are found, an additional check is executed to ensure that the type of the term can be found for at least one accessible item in the database. If this is not the case, the generator returns an *invalid grammar* error because it means there is a non-terminal that could never reached a terminal.

The next step in generating inputs is finding all the rules that have an output term that matches the term passed into the method; if a puzzle item was chosen for the

term, that term now has a type that is as specific as that puzzle item's type. For a term to match the (main) output of a rule, it must be of the same type as or a sub-type of the output. Additionally, the lists of properties for each term must match exactly.

If multiple possible rules are found that match these conditions, and the area search depth limit has not been reached, one will be chosen at random. Again, type checks are performed; if the start term is more specific than the output term in the chosen rule, and the output term corresponds to any input terms in the rule, those input terms are update to be more specific. Each of the input terms of the chosen rule are then considered outputs, for which new inputs must be generated by calling the `GenerateInputs()` method for them. When the recursive call to the method returns, it passes puzzles item matches for terms back up the tree.

The final possible step of the `GenerateInputs()` method occurs when no rule match was found for the start term. In this case the puzzle item attached to the start term will be spawned in the current area, if it is not yet found in that area. If no puzzle item has been attached to the term by this stage, the generator returns an *invalid grammar* error as well, since it means there is a term that cannot be matched by either a different non-terminal or a terminal symbol of the grammar.

## 4.5 Puzzle Solving

Puzzle solving involves the sequential execution of rules that are part of the puzzle tree created by the generator. A player must explore the game world and consider which actions may lead to the fulfillment of the area goal, which can be displayed as a hint when the player enters a new area.

Game items communicate with the puzzle manager to learn which rules pertain to them at different points during the game. As mentioned, it is the game items themselves that contain the logic for deeming a rule executable. Besides checking the rules obtained from the puzzle manager, there are two other possible actions. The first is the *PickUp* action, which gets displayed for any item, regardless of its inclusion in a puzzle,

if it has the *carryable* property with the value set to true. Allowing interactions with game items in this way introduces some red herrings into the puzzle, i.e. the player may think he/she will have a use for an item when that item is not actually part of the puzzle solution in any way.

The converse action, *PutDown* can be executed for any game item that is currently marked is being contained in the player's inventory. In the demo game, items appear by their description in a small menu whose display can be toggled. Items are removed from the inventory by clicking on their name in this list. Practically, this causes them to be placed in front of the player. In the demo game, actions can only appear for items in the world, not while they are in the player's inventory, but this is an implementation detail that can be chosen by the game designer.

For the normal rules, the execution of a rule is composed of three steps that correspond to: item creation, item destruction and item property changes. First, items that appear as an input but not as an output are marked for destruction, with the exception of items whose type appears as the value of the contains property of another output item. The comparison between input and output items is done on the basis of exact matches between types. Note that the inputs and outputs of a rule here are discussed as items instead of terms; at the point of rule execution, all terms will have corresponding items assigned.

Items that appear both on the left and the right may have an associated property change; this change is applied to the game item's properties, as opposed to the puzzle item's properties. Finally, items that appear only on the left must be spawned in the world. If there are item in the *to be deleted* list, the locations of those items are used as the locations for spawning new items. The last step is the actual destruction of game items; this is not done till the end as it may involve destroying the game object that is leading the execution of the rule.

## 4.6 The Puzzle Manager

The puzzle manager is the point of entry for the puzzle generation system and manages a lot of the behavior described previously in this chapter. It tracks the areas that currently have puzzles in progress, and regulates the execution of rules, in the right-to-left direction, in conjunction with the puzzle item objects. As part of this, the puzzle manager monitors whether the goal for an area has been achieved; if it has, it makes a call to the generator to create a new puzzle for each area connected to the area that was just completed.

When the generator finishes running, the puzzle manager iterates through the rule tree in order to find out which rules are the current leaves of that tree. At any point in time, the leaves of the tree represent the rules that must be used (by the player) to move towards the puzzle solution. When a rule is executed, the puzzle manager removes it from the list of leaves, while adding its parent rule to that list.

The system presents two options for which grammar rules can be used during game play; allowing all the rules to be used, or only those that are specifically present in the puzzle tree. The puzzle manager sends rules to the game items depending on which option has been checked. For the first option, the puzzle manager also finds matching puzzle items for the rules' terms. Only rules that are part of the puzzle tree output by the generator will have terms tagged with puzzle items. In fact these rule assets are instantiated so that the rules can be amended to reflect that information without updating the assets themselves.

Allowing all rules to be part of the game logic, including those that are not part of the solution sequence, may lead to the game reaching an unsolvable state. However, using all the rules, regardless of the created puzzles, can lead to a more consistent game world, and to greater challenge in figuring out the puzzle solutions.

# Chapter 5

# Evaluation

The design and implementation of the system for procedural puzzle generation focused on achieving the objectives outlined in Chapter 1; creating an expressive, user-friendly system for the creation of solvable puzzles that can be integrated into a game. The system was not strongly concerned about computational efficiency, but, for the small demo game, the puzzles were easily generated on the fly as new areas became unlocked. The evaluation made in this chapter is qualitative; it focuses on describing the proof of concept game, and making comparisons with the Puzzle-Dice system, especially in terms of the objectives, which is the current state of the art for narrative puzzle generation.

## 5.1 Proof of Concept Game

A small proof of concept game was developed in Unity using the implementation of the narrative puzzle generation tool described in this dissertation. The game was made with free 3D assets and set in an environment with two areas; a grass field, and a river bank. These areas were designed by hand (as opposed to procedurally generated) and contained some game objects with `GameItem` components, such as some trees, corn stalks and a well. On a given playthrough, each of these may or may not be used, depending on the puzzle that's created. However, it is always possible to interact with the items; the corn can be added to the player's inventory regardless of whether it's part of the puzzle. This adds consistency to the world, and can throw the player off in

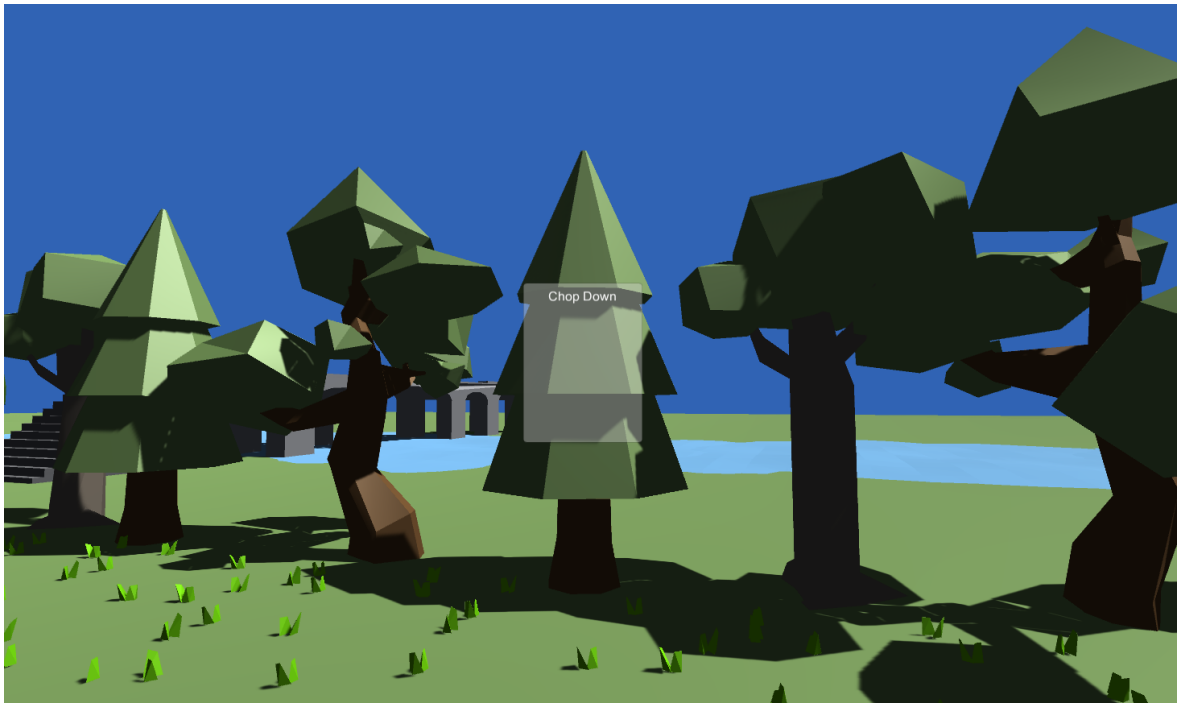


Figure 5.1: Opening the action menu on a tree, when there is an axe in the player's inventory.

terms of what items he/she needs to complete the puzzles for an area.

Figures 5.1 and 5.2 show screenshots from the game; the player clicks on a tree and it presented with the "Chop Down" action because he/she has in axe in their inventory. The inventory can be displayed, as a list of items, using the tab key. Executing the rule tied to the "Chop Down" action causes the tree to be replaced by a tree stump, as according to rule 3.2.

For this game, 25 rules and 38 puzzle items were created. The goal of the first area is to chop down a tree, which means that the player will have to acquire an axe. The rules can lead to several unique puzzle trees for obtaining this axe; besides unique puzzle trees, a single puzzle tree may also result in a slightly different playthrough when some different items have been picked for the same terms in the rules. The puzzle generation is integrated into this game by tagging some of the items in the world as game items, including the trees. As a result, trees never need to be spawned, when one is needed as an input to a puzzle, the player can use one of the ones that is already in the world.

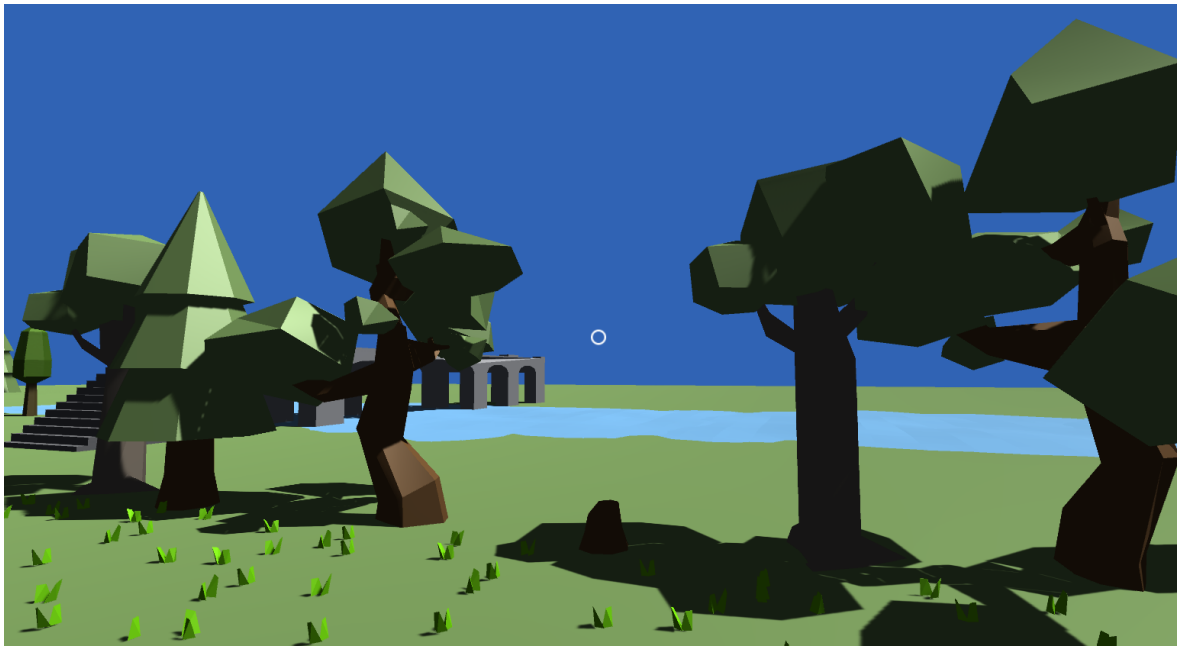

Figure 5.2: After having executed the rule attached to the "Chop Down" action.

## 5.2 Comparison with the Puzzle-Dice System

This system uses a similar but simplified item database as the Puzzle-Dice system. The simplification partially results from moving away from the single database table design that dictates that all existing item properties have to be defined for all items. In the new system, properties only have to be defined where relative, greatly reducing unnecessary work on the part of the game designer. Another factor in simplification is that some properties from the Puzzle-Dice system, such as the *madeby* property, are no longer needed because such a relationship between items is expressed by the grammar rules. Overall, this system places more emphasis on defining relationships between types of items, as opposed to defining the items as isolated components.

The system described in this dissertation integrates the designer-targeted editors into Unity, whereas they were previously stand alone. This makes the system more user-friendly as they puzzle generation components can be edited alongside the other game components.

The grammar replaces Puzzle-Dice's puzzle map, and a large number of puzzle item properties, and is the focus of designer-input. The format of the components of a rule is general enough to allow for a greater variety in puzzles than can be expressed by the Puzzle-Dice system's puzzle building blocks. These building blocks, which make up designer defined puzzle maps, are tied to specific patterns, such as *combination* or *insertion*, while each rule can describe the function of any such building block, and more. Additionally, rules allow for the inclusion of item properties in their specification, e.g. a rule can require that the iron combined with a hammer to produce a blade must be hot. The building blocks could place no such restrictions on the properties of its inputs.

The generator uses the grammar rules to create a puzzle as a tree structure of rules, which is similar to the structure of the puzzle maps in the Puzzle-Dice system. The creation of a puzzle map and the subsequent process of fitting items into that map is merged in the new puzzle generation algorithm.

The puzzle tree is created automatically at runtime, and is thus different each time. Items are filled into this structure while it is being generated, which also alleviates the algorithmic difficulty of creating a puzzle for a given puzzle map; this was cited as a limitation of the Puzzle-Dice system. In theory, this new design can create a wider variety of puzzles with less algorithmic difficulty and a comparable amount of designer input. Puzzle maps are implicitly defined by the generator instead of having to be explicitly designed, which speaks to stated future work for the Puzzle-Dice system.

Unlike the Puzzle-Dice system, which also terminates generation with a list of items to be spawned in the world, this system does not make the assumption that the world is empty at the start. Any game object in the world can be given a game item component that links it to one of the puzzle items in the asset database. This allows puzzles to be integrated into an existing world; for example, an appropriately tagged lake, even one that is part of a procedurally generated environment, could be used as a water source in a puzzle that required one.

## 5.3 Solvability

The first main objective for the puzzle generation system, i.e. creating puzzles that are guaranteed to be solvable, is satisfied through the use of the forwards-backwards generation combination. Moving forwards through the areas ensures locked door type puzzles will be solvable, while going backwards guarantees that the puzzle goal can be reached from the starting items.

The one requirement for the creation of solvable puzzles is that the grammar must be valid; this means all its non-terminal must match either other non-terminals or terminals (the puzzle items). The game designer is responsible for creating a valid grammar; the flexibility in the rule definitions means a lot can be expressed, but also leaves room for error. This issue is discussed further in a the next section.

## 5.4 Dynamic Difficulty

The difficulty of a puzzle can be influenced by multiple factors including the length of the solution sequence, i.e. the number of rules that need to be executed to achieve a goal, or the complexity of the logic written into the rules by the game designer. As mentioned in the Design Chapter, the game designer can specify the maximum depth of a puzzle tree on a per area basis. Of course, in order to use this depth, there have to be enough rules to chain together a puzzle of that length. This can be referred to as adding a lot of depth to the rules. The converse is adding breath to the rules, which is synonymous to high branching, or creating many rules with the same output term. Having breadth in the rules will lead to more variety in the generated puzzles on each playthrough of the game.

Additionally, high branching can lead to there being many start items in the world; which could also be artificially created through just placing a lot of game item tagged objects into the world. This tends to make a puzzle more challenging as well because it gives the player more options from which to figure out what needs to be done.

## 5.5 Expressivity vs. Usability

As is the case for many puzzle generation techniques, there is a direct trade off between the expressivity and the usability of the system. The more general, and expressive, a system is, the more work is required on the part of the designer. The system described in this dissertation defines appropriate structures, but those depend on the creativity of the designer to be filled in in such a way that they will lead to interesting puzzles. Part of the task of creating emergent behavior in the world falls on the shoulders of the designer in this way; the generator is limited by the breadth and depth of the designer-defined rules.

Generally, grammar breadth will lead to more puzzle variety, while depth will lead to higher per puzzle complexity in terms of the number of steps required for reaching a goal.

The flexibility of the term and puzzle item definitions means that the system could be used for a variety of narrative puzzle types. For example, the grammar could be used to define dialog trees as part of a puzzle that requires saying the right thing to an NPC. However, the drawback is that the designer needs to be careful about matching the spelling of equivalent properties and types, and the general grammar validity.

# Chapter 6

# Conclusion

This chapter concludes the dissertation by summarizing the contributions made by the proposed system for procedural generation of narrative puzzles, and discussing its limitations as well as possible future improvements.

## 6.1 Main Contributions

This dissertation presents a novel way of procedurally generating narrative puzzles, that builds onto what was achieved with the Puzzle-Dice system. This new approach is based on an extended type of context free grammar that provides expressive power, to a greater extent than the Puzzle-Dice system, as well as guaranteed puzzle solvability. The developed system for generating narrative puzzles can be integrated into existing games, given that the game designer defines puzzle items, rules and game areas as they pertain to his/her game. The difficulty of the puzzles is determined by the designer.

An implementation of the system was developed for Unity, and used to create a small proof-of-concept game that presents the player with different puzzles during different playthroughs. The system is mainly composed three custom editors that can be used to create all of the inputs required by the puzzle generator.

## 6.2 Limitations & Future Work

Game designers have to ensure that the grammar they create is valid; this poses a limitation, especially when the grammar contains many rules. Future work could include a grammar validator that would run through all the created rules and notify the designer of any issues with any of the terms. Generally, as the number of rules scales up, a better grammar management tool would be needed in order to facilitate designer work flow.

Another limitation that has been touched upon is the issue of entering into an unsolvable state when solving a puzzle in a game that has allowed all of the rules to be used (as opposed to just those that are part of the puzzle tree). This issue of reaching an unsolvable state is separate from that of generating solvable puzzles. It will always be possible to solve a puzzle, but an incorrect sequence of actions may lead to a failure on the player's part to do so. Using all of the rules thus makes the puzzle very challenging, because the player would need to determine themselves whether or not they have gotten themselves into an unsolvable state.

Future work could include checking for the occurrence of such a state, but this may be an expensive check. Even if the current state of the game world cannot directly lead to a solution, it may still be possible to reach such a state again. Determining that possibility would be nearly equivalent to running the puzzle generator again.

## 6.3 Final Thoughts

The increasing prominence of procedurally generated worlds in video games has brought up the question of how to make those worlds intriguing. Some of the recent commentary on *No Man's Sky* has indicated that players are not always satisfied with pure exploration, which is the hallmark of PCG worlds. *Minecraft*'s unique selling point is in it's endless, crafting-fueled sanbox, but even in that world, there may be room for more content variety. Developing systems for procedurally adding interesting content and story to procedurally generated worlds could be a promising avenue for game designer to explore. This dissertation proposes just one idea for such a system.

# Bibliography


[1] D. Ashlock, "Automatic generation of game elements via evolution," in *Proceedings of the 2010 IEEE Conference on Computational Intelligence and Games*, pp. 289–296, IEEE, 2010.

[2] C. Fernández-Vara and A. Thomson, "Procedural generation of narrative puzzles in adventure games: The puzzle-dice system," in *Proceedings of the The third workshop on Procedural Content Generation in Games*, p. 12, ACM, 2012.

[3] M. Shaker, M. H. Sarhan, O. Al Naameh, N. Shaker, and J. Togelius, "Automatic generation and analysis of physics-based puzzle games," in *Computational Intelligence in Games (CIG), 2013 IEEE Conference on*, pp. 1–8, IEEE, 2013.

[4] J. Togelius, G. N. Yannakakis, K. O. Stanley, and C. Browne, "Search-based procedural content generation: A taxonomy and survey," *IEEE Transactions on Computational Intelligence and AI in Games*, vol. 3, no. 3, pp. 172–186, 2011.

[5] A. M. Smith, E. Andersen, M. Mateas, and Z. Popović, "A case study of expressively constrainable level design automation tools for a puzzle game," in *Proceedings of the International Conference on the Foundations of Digital Games*, pp. 156–163, ACM, 2012.

[6] "increpare/puzzlescript: Open source html5 puzzle game engine." `https://github.com/increpare/PuzzleScript`. (Accessed on 08/29/2016).

[7] J. Taylor and I. Parberry, "Procedural generation of sokoban levels," in *Proceedings of the International North American Conference on Intelligent Games and Simulation*, pp. 5–12, 2011.

[8] B. Kartal, N. Sohre, and S. Guy, "Generating sokoban puzzle game levels with monte carlo tree search," 2016.

[9] Y. Murase, H. Matsubara, and Y. Hiraga, "Automatic making of sokoban problems," in *Pacific Rim International Conference on Artificial Intelligence*, pp. 592–600, Springer, 1996.

[10] J. Taylor, T. D. Parsons, and I. Parberry, "Comparing player attention on procedurally generated vs. hand crafted sokoban levels with an auditory stroop test," in *Proceedings of the 2015 Conference on the Foundations of Digital Games*, 2015.

[11] N. Sturtevant, "An argument for large-scale breadthfirst search for game design and content generation via a case study of fling," in *AI in the Game Design Process (AIIDE workshop)*, Citeseer, 2013.

[12] G. W. Flake and E. B. Baum, "Rush hour is pspace-complete, or why you should generously tip parking lot attendants," *Theoretical Computer Science*, vol. 270, no. 1, pp. 895–911, 2002.

[13] F. Servais, *Finding hard initial configurations of Rush Hour with binary decision diagrams*. PhD thesis, M. Sc. thesis, Université libre de Bruxelles, Faculté des sciences, 2005.

[14] J. Juul, "Swap adjacent gems to make sets of three: A history of matching tile games," *Artifact*, vol. 1, no. 4, pp. 205–216, 2007.

[15] "Article: Procedural generation of puzzle game levels - gamedev.net - your game development resource." `http://www.gamedev.net/page/resources/_/technical/game-programming/procedural-generation-of-puzzle-game-levels-r3862`. (Accessed on 08/30/2016).

[16] J. Dormans, "Adventures in level design: generating missions and spaces for action adventure games," in *Proceedings of the 2010 workshop on procedural content generation in games*, p. 1, ACM, 2010.

[17] A. M. Smith, E. Butler, and Z. Popovic, "Quantifying over play: Constraining undesirable solutions in puzzle design.," in *FDG*, pp. 221–228, 2013.

[18] I. Dart and M. J. Nelson, "Smart terrain causality chains for adventure-game puzzle generation," in *2012 IEEE Conference on Computational Intelligence and Games (CIG)*, pp. 328–334, IEEE, 2012.

[19] N. Shaker, M. Shaker, and J. Togelius, "Evolving playable content for cut the rope through a simulation-based approach.," in *AIIDE*, 2013.

[20] S. LOGIC, "The game of logic, lewis carroll," *Two delightful puzzle books by*.

[21] D. Oranchak, "Evolutionary algorithm for generation of entertaining shinro logic puzzles," in *European Conference on the Applications of Evolutionary Computation*, pp. 181–190, Springer, 2010.

[22] T. Mantere and J. Koljonen, "Solving, rating and generating sudoku puzzles with ga," in *2007 IEEE Congress on Evolutionary Computation*, pp. 1382–1389, IEEE, 2007.

[23] B. Pintér, G. Voros, Z. Szabó, and A. Lorincz, "Automated word puzzle generation via topic dictionaries," *arXiv preprint arXiv:1206.0377*, 2012.

[24] S. Colton, "Automated puzzle generation," in *Proceedings of the AISB02 Symposium on AI and Creativity in the Arts and Science*, 2002.

[25] P. Galván, V. Francisco, R. Hervás, and G. Méndez, "Riddle generation using word associations,"